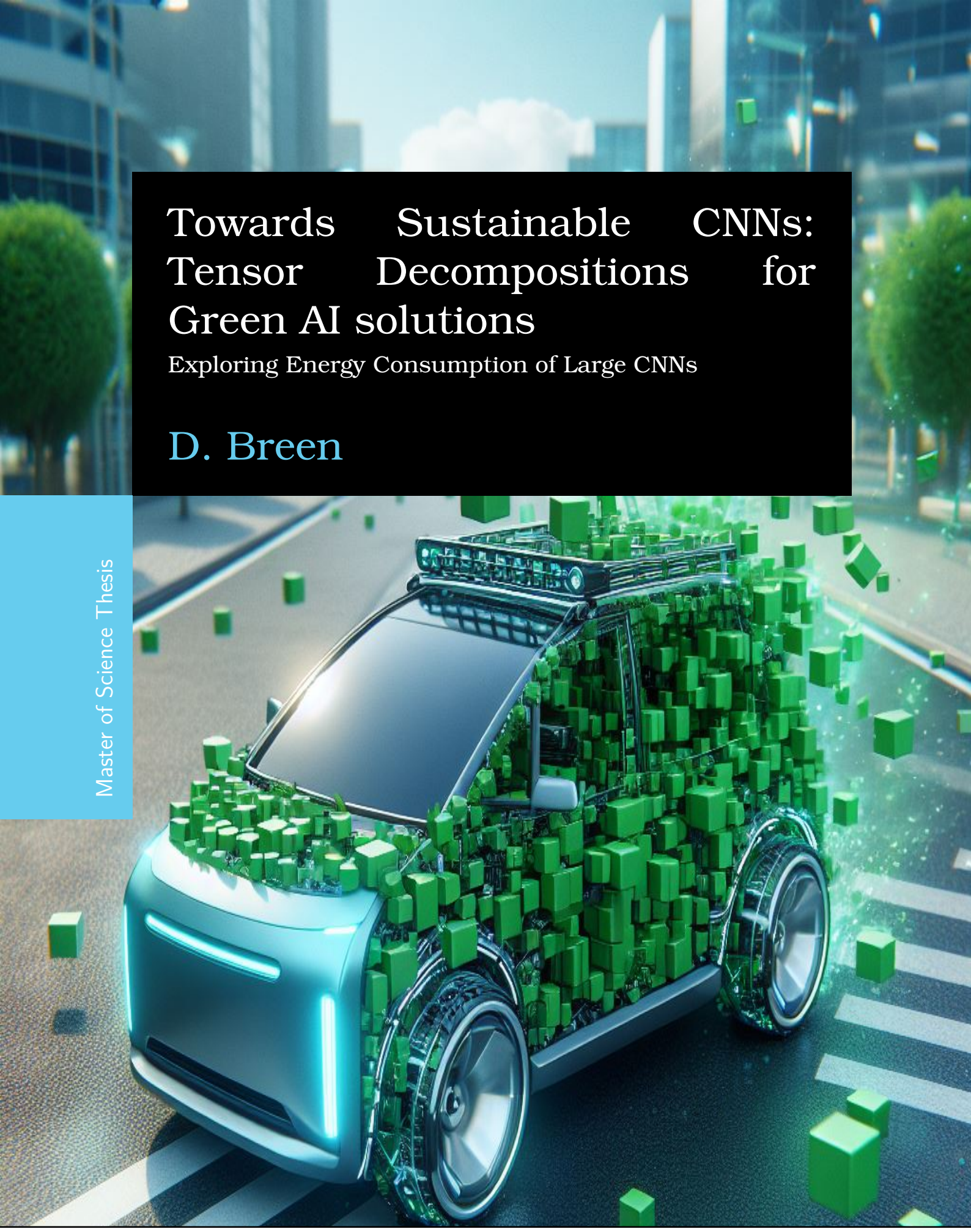
Towards Sustainable CNNs: Tensor Decompositions for Green AI solutions

Exploring Energy Consumption of Large CNNs

D. Breen

Master of Science Thesis



TU Delft Delft University of Technology

Delft Center for Systems and Control

Towards Sustainable CNNs: Tensor Decompositions for Green AI solutions

Exploring Energy Consumption of Large CNNs

Master of Science Thesis

For the degree of Master of Science in Systems and Control & Robotics
at Delft University of Technology

D. Breen

August 19, 2024

Faculty of Mechanical Engineering (ME) · Delft University of Technology

Abstract

The ever-increasing complexity of Artificial Intelligence (AI) models has led to environmental challenges due to high computation and energy demands. This thesis explores the application of tensor decomposition methods—CP, Tucker, and TT—to improve the energy efficiency of large Convolutional Neural Networks (CNNs) during inference by reducing energy consumption. The energy consumption of several convolution layers was measured using a watt meter across various CNN configurations and different hardware architectures (Central Processing Unit (CPU) and Graphics Processing Unit (GPU)). In addition, several regression models were fitted to estimate energy savings, incorporating memory usage. It was found that TT decomposition consistently provided the most significant energy savings across various compression ratios, influenced by CNN hyperparameters such as input/output channels, feature sizes, and kernel sizes, whereas CP decomposition was the least effective in reducing energy. The GPU implementations generally resulted in additional energy consumption, and the GPU regression models suggested a need for more complex relationships. The thesis also revealed that the efficiency of tensor decompositions might be highly dependent on the implementation details of software libraries, such as TensorlyTorch, which can significantly impact the computation and memory complexities. These findings underscore the importance of both hardware-specific considerations and careful software implementation in achieving energy-efficient CNNs, providing a foundation for further research in energy-constrained environments.

Table of Contents

Preface & Acknowlegdements

I have learned so much throughout my studies and this thesis marks the end of my double degree in Systems and Control and Robotics at TU Delft. It explores tensor decomposition methods for improving the energy efficiency of AI systems, a topic driven by the growing need to balance technological advancements with environmental sustainability. I enjoyed doing a thesis that also included some form of sustainability because I believe that this is important and not always considered in projects and courses in the Masters and Bachelors.

The creation of this thesis involved a lot of energy measurements, training of networks and the cover page was created using DALL-E, a large text-to-image AI model, which might contradict the research into how energy can be saved. Still, I hope that this work can make a small contribution to this research field, even though the thesis itself was not as sustainable.

I am very grateful to my supervisors, Dr. Ir. Kim Batselier and Dr. Julian Kooij, for their guidance and support throughout this thesis. Their expertise in tensor methods and CNNs has been invaluable.

I also want to thank my parents for their tireless support and encouragement, which have been crucial to my academic journey. To my friends, thank you for your motivation and understanding during this process.

Thank you to everyone who has supported me along the way.

List of Figures

List of Tables

Chapter 1

Introduction

The rise of intelligent applications, devices, and transportation has led to an increased dependence on Artificial Intelligence (AI) inducing a demand for faster and more advanced AI models. While AI is being promoted as an innovation contributing to our welfare and shaping our future [1], there is another less-highlighted side. The query for larger and better-performing AI models came along with some challenges [2]. Besides requiring substantial financial costs, these models also require extensive computational resources and energy, imposing great environmental impact and retaining the future of AI. It was found that training state-of-the-art natural language processing models resulted in up to 3 million dollars of financial costs but more importantly, up to 280 000 kg of CO2 emissions, which is equivalent to a person flying 300 times between Amsterdam and New York [3]. Additionally, it was established that the approximated energy consumption of U.S. data centres increased by 33 billion kWh from 2000 to 2006 and on a global scale the estimations increased by 65 billion kWh over five years [4]. Figure 1-1 contributes by showing the estimated energy consumption growth of milestone Deep learning (DL) models, which form a big part of the current AI research field. These examples give a clear image of the needed change in the current AI research that should not solely pursue better accuracy but also consider the consequences of ever-increasing compute and complexity.

This shift towards acknowledging the importance of more sustainability-driven metrics instead of solely pursuing accuracy was first encouraged by [2] and persisted under the umbrella term Green AI. Green AI is defined as AI research that pursues a high accuracy and performance but in doing so actively avoids increasing the complexity and number of computations and potentially even decreasing them [6]. It is the opposite of the current mentality of Red AI which is defined as the state-of-the-art AI that only seeks to result in the best accuracy and performance without taking into account the environmental, economic and social consequences [6]. To measure the 'greenness' or 'sustainability' of AI, several metrics have been proposed, such as the amount of carbon emitted, elapsed real time, the number of parameters, the number of floating point operations (FPO) and the amount of energy consumed [6]. However, these metrics lack universality making model comparison difficult and alternative metrics need to be developed as proposed by [7].

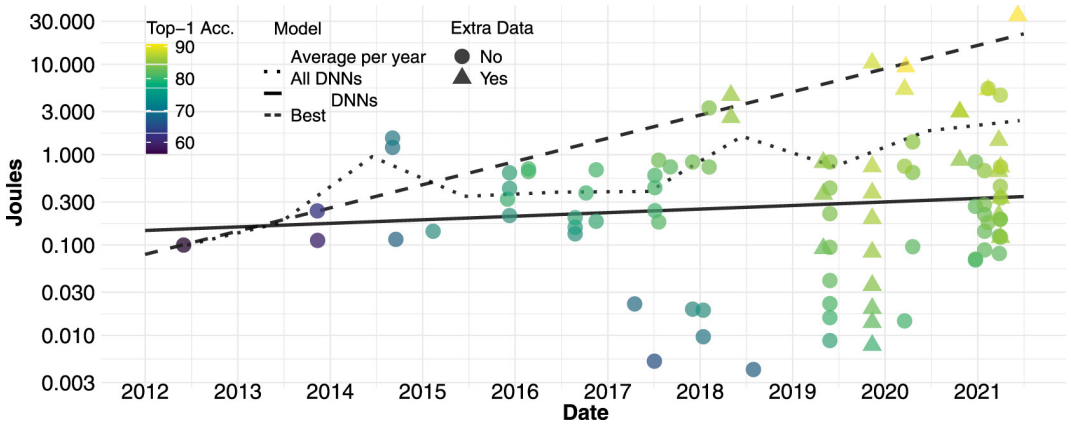


Figure 1-1: Estimated Joules of a forward pass for different Top-1 accuracy Deep Neural Networks (DNN). The dashed line is a linear fit on a logarithmic-scale, for the models with the highest accuracy per year. The solid line fits all models' average energy (retrieved from [5]).

Current Green AI research focuses primarily on large neural networks [8]. To increase the accuracy and performance of neural networks, deeper and wider networks are developed consisting of a growing number of parameters. These large networks with high accuracy will consume more energy than others as seen in Figure 1-1. Besides, the environmental implications another challenge arises. The current desire for autonomy and computation of resource-limited applications, such as mobile phones or autonomous vehicles (AVs), demands new technologies allowing energy-efficient computation and memory [5]. Object detection in AVs requires high computational power and the capacity to process 21.6 billion inferences daily without compromising critical safety requirements [9]. They are now even referred to as 'driving data centres' or 'supercomputers on wheels' [10], emitting the same amount of carbon emissions as data centres based on their expected power consumption as is shown in Figure 1-2. Therefore, it is essential to consider accuracy and energy consumption for these resource-constrained applications, in line with the Green AI mentality.

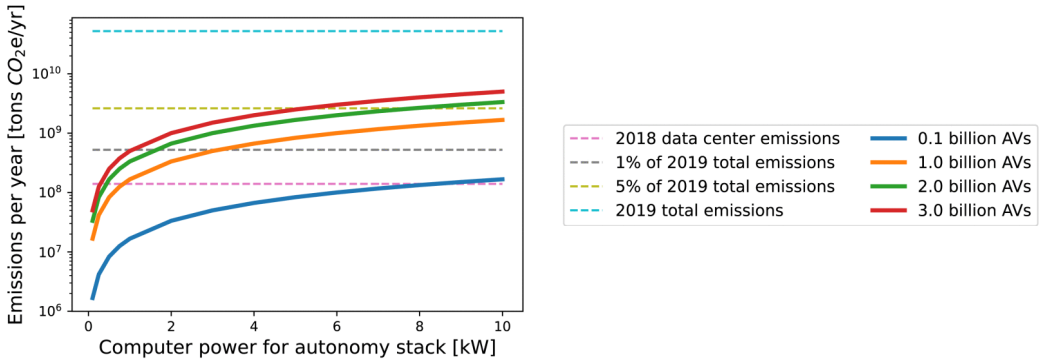


Figure 1-2: Emissions from computing onboard AVs driving 1 h/day: With one billion AVs, each using an average of 0.84 kW of computing power, the total emissions would be comparable to those produced by all data centres combined (reprinted from [10]).

To facilitate the integration of large and accurate Convolutional Neural Networks (CNNs) into the AVs without facing these resource, computation and sustainability challenges it is

important to find a way to reduce their energy consumption and computational workload. The development of more energy-efficient CNNs has been extensively researched, and methods such as pruning, quantization and knowledge distillation are widely used [11]–[15]. A promising state-of-the-art strategy is to use tensor decompositions. The tensor decomposition methods can be used to decompose the large kernels of the CNNs, breaking down the 'regular' convolution layers and replacing them with separate smaller ones. Especially for deep CNNs for which the kernels can become very large, the use of tensor decomposition shows great potential. This potential has been widely researched and in [16], a compact but thorough overview is presented consisting of the current advancements in this research field, addressing the energy efficiency problem through the use of tensor decompositions, especially in the context of CNNs.

Tensor decompositions offer a solution by approximating large CNNs with low-rank approximations while maintaining the desired accuracy and by extracting physically meaningful variables from the data in an accurate and computationally conserving way [17]–[20]. Three main tensor decompositions are used to decompose the layers of the CNN: Parallel Factors (PARAFAC) Canonical Decomposition (CANDECOMP) (CP) decomposition, Tucker decomposition and Tensor Train (TT) decomposition [16]. To illustrate their capabilities and accuracy preservation, another application of tensor decomposition, namely image compressions, is used in Figure 1-3. Here, the left image is decomposed into smaller components, consisting of only 2% of the original number of parameters. After reconstructing, based on only 2% of the parameters, the main features are still present and interpretable. This example can give an intuitive understanding, that when large CNN kernels are decomposed to a minimal number of parameters, the approximation could still achieve well enough performance.

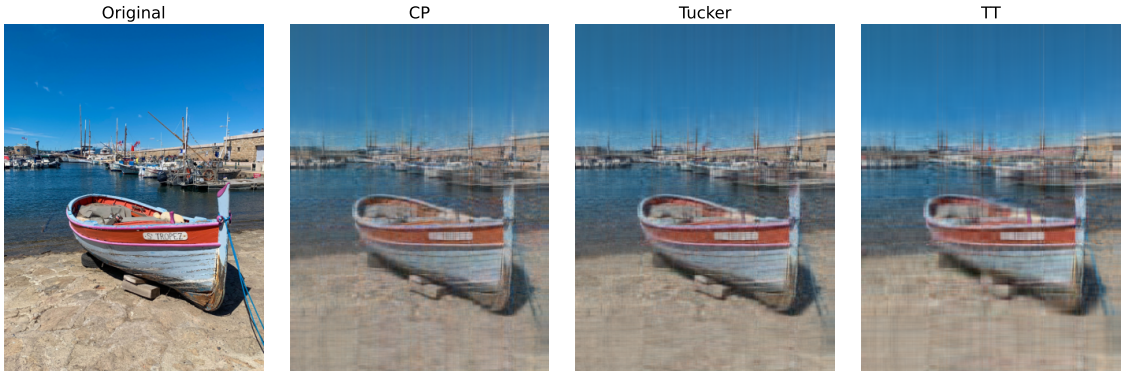


Figure 1-3: Example of tensor decomposition methods, CP, Tucker and TT, for image compression. It visualizes that the approximation of the tensor decomposition can keep accuracy even with only 2% of the original number of parameters.

Decomposing layers into smaller components reduces the number of parameters in large kernels, affecting both computational and memory complexity. This reduction is expected to decrease the number of operations required, lowering the computational load, and decreasing the memory usage compared to the original larger kernels. A lower computational workload and decreased memory movements suggest efficient and fast processing, theoretically reducing the energy consumption of large CNNs. However, despite the progress of implementing tensor decompositions in convolutional layers, further exploration is needed to fully realize

their potential for energy-efficient processing, particularly in analysing the real-time energy reduction of using tensor decompositions as network compression of CNNs.

This thesis aims to investigate the potential of implementing these tensor decompositions in CNNs for energy reduction, allowing for more energy-efficient inference. That is why the following research question will be answered:

How does the implementation of tensor decompositions (CP, Tucker, and TT) affect the energy efficiency of large CNNs during inference, with a focus on predicting the resulting energy savings?

This will be done by breaking this question into smaller sub-questions, presented below.

SQ1 Which type of tensor decomposition (CP, Tucker, or TT) and compression ratio yields the best performance for energy efficiency in CNNs?

- **Objective**: Identify the most efficient tensor decomposition method and optimal compression ratios for maximum energy savings.

SQ2 Which convolutional hyperparameters, such as input/output channels, input feature size, and kernel size, yield the greatest energy savings for decomposed convolutions using tensor decompositions?

- **Objective**: Determine which convolutional hyperparameters affect the saved energy consumption most.

SQ3 To what extent is the energy consumption of tensor-decomposed CNNs influenced by the use of hardware, specifically Graphics Processing Unit (GPU) compared to Central Processing Unit (CPU)?

- **Objective**: Evaluate the effect of the implementation of different types of hardware, in particular the differences between CPU and GPU?

SQ4 To what extent can energy savings in tensor-decomposed CNNs be accurately predicted based on pre-implementation data, such as calculated computation and memory usage, compared to models requiring additional empirical measurements?

- **Objective**: Determine the accuracy of predicting energy savings in tensor-decomposed CNNs using pre-implementation data versus empirical measurements, and compare the effectiveness of these predictive models.

This thesis consists of four chapters. First, Chapter 2 will outline the the state-of-the-art research in this field, which forms a basis for the next chapters. Second, Chapter 3 gives an elaboration on the used methodology. Third, Chapter 4 will present all results and key findings. Lastly, Chapter 5 will present the final conclusions to the above-presented question including a discussion which will also present the steps for future work.

Chapter 2

Related work

This chapter gives a concise overview of the existing literature and state-of-the-art research in the research field. It shows the current foundation on which this thesis will build its contributions. The current state of the Green AI perspective will be outlined, followed by different measurement techniques of energy consumption. After that the preliminaries and basic concepts of tensor decompositions are outlined, showing the theoretical background of decomposed convolutions. Lastly, the decomposed convolutions and current research will be presented.

2-1 Sustainable AI

The trends in current AI research are monitored closely [5], [21]–[25]. The exponentially growing demand for large models has initiated a large research field looking into 'sustainable' AI. This section will present state-of-the-art research that tries to define 'sustainable' AI and tries to come up with new 'sustainability' metrics. In addition, the current integration of this new view on AI into current autonomous driving research will be shortly outlined, showing the impact on current applications.

2-1-1 Trends in Green AI

Early awareness of the increasing sustainability impact of ICT was uncertain about whether the energy saved using ICT would outweigh the energy required for ICT resources. The GREENSOFT model was proposed to address this, which includes energy-efficient metrics, life-cycle design considerations, and sustainable software criteria [4]. It was suggested to closely monitor these sustainability concerns.

Concrete policy considerations were proposed in [2], highlighting that training large natural language models could emit up to 600.000 pounds of CO2. Authors were encouraged to report the resources used, including finances and computation, alongside model accuracy.

Sustainable AI was proposed by [26] and defined as "a movement to foster change in the entire lifecycle of AI products (i.e. idea generation, training, re-tuning, implementation, governance) towards greater ecological integrity and social justice". This term highlights the distinction between 'AI for sustainability' and 'sustainability of AI', reflecting the balance discussed in [4]. Another term is Green AI. It is defined as "AI research that is more environmentally friendly and inclusive" [6], contrasting with the common Red AI mentality. Although sustainable AI and Green AI are used interchangeably, Green AI has become a more established term. [8] provides an extensive overview of the Green AI research field.

Efficiency as an evaluation metric for AI was proposed by [6], while [1] argued that focusing solely on efficiency is insufficient, particularly in terms of compute, energy and carbon. Improvements in AI efficiency can lead to unexpected outcomes [1]. For instance, there are discrepancies between compute efficiency, energy efficiency and carbon efficiency and not all expected gains from increased energy efficiency are necessarily realized, also known as the rebound effect [27]. This effect, where increased efficiency leads to more usage, may counteract some benefits of new implementations [3].

2-1-2 Green autonomous driving

In autonomous driving, addressing the resource limitations while meeting the computational demands for safe and fast processing has led to the integration of Green AI. The contrast between Green AI and Red AI perspectives is noticeable in current research [28]–[31]. Solutions like edge inference, which combines cloud connectivity and external data processing, offer potential solutions. However, they merely shift the computational workload rather than reduce it, making sustainability still a concern. A combination of onboard computer and cloud connectivity is researched in [32].

Another downside of leveraging network connections for external computation and storage is that it introduces cybersecurity and safety challenges [33]. [34] provides an overview of edge AI for autonomous vehicles (AVs), involving opportunities of model optimization. The main research areas are model compression, consisting of parameter reduction, layer/node reduction, neural architecture search, and model approximation, which includes quantization, sparsification, low-rank approximations, and knowledge distillation [34]. Additionally, data-centric Green AI offers other opportunities to improve energy efficiency [35], [36].

2-1-3 Evaluating Green AI

To effectively implement the Green AI perspective and motivate researchers, new metrics are needed that embody the 'sustainability' of AI, focusing on the energy efficiency of a network. These new metrics require universality, enabling direct comparison between different AI models. In [37] an overview of metrics is presented, divided into compute, energy, carbon emission and runtime metrics.

The number of floating point operations (FPO)s can give an insight into the computational speed of an algorithm, typically calculated for one forward pass through a network [37]. An alternative is the number of Multiply-Accumulate operations (MACs) or Multiply-Add operations (MADs) [38], which is twice the number of FPOs and can be of substantial amount

in large Convolutional Neural Networks (CNNs) [37], [39]–[41]. These metrics perform well in showing the computational complexity of a model, however are not hardware-dependent. Different types of hardware can show a difference in energy efficiency for the same number of operations [42].

Another metric is the number of parameters in the network [43], which can correlate with training and inference time [41]. However, this metric is not universal, as different model architectures with a similar number of parameters, can show different energy profiles, and are not directly comparable [6], [44].

In contrast, more hardware-focused metrics such as hardware-utilization of Central Processing Unit (CPU) and Graphics Processing Unit (GPU) are crucial for designing resource-efficient Deep learning (DL)-frameworks [45], [46]. CPU-GPU utilization-aware energy-efficient strategies can reduce energy consumption significantly [47]. Related metrics include GPU- and CPU-hours, which can be seen as the utilization times the runtime [48]. However, these metrics do not consider the efficiency of the network and code that is running.

Runtime metrics, divided into the inference time and training time, are commonly used to compare models on the same hardware and software settings, despite their lack of universality [49]–[51].

Energy consumption metrics provide a direct measure of a model's energy efficiency, using measurements or estimates in Joules, kWh, or Watts for training and inference [37], [52]–[54]. Similar to other metrics, these metrics are not hardware-independent. Carbon emission estimation, another direct sustainability measure, considers energy consumption, location-specific carbon intensity, and Power Usage Effectiveness (PUE). [55]–[58].

Integrating energy consumption into the Red AI versus Green AI perspectives introduces new metrics that balance sustainability and accuracy. For example, in [59], accuracy per Joule is proposed and in [7] several energy-based efficiency metrics are given, including an energy-precision ratio and a recognition efficiency. The GreenQuotientIndex allows researchers to prioritize energy consumption over accuracy by tunable parameters.

Carbon footprint is another metric, which focuses more on the whole picture of the models' sustainability. As a basis for this lies the energy consumed by a model. Then, based on the location and time, either inference, training or both are performed, and the carbon intensity (the division of how the energy is produced) is used to calculate the estimated carbon emissions. However, this implies that the sustainability of a model relies on the location and time, which is not a general metric.

While these metrics focus on the training and inference phases of the model, the full lifecycle also includes infrastructure, recycling and development. To address this, the full-cycle energy consumption metric [60] and the deep learning lifecycle efficiency metric [7] were proposed.

2-2 Measuring the energy consumption

Most of the metrics above can be profiled quite straightforward, for example by using timestamps for runtime metrics, calculating the number of parameters or monitoring hardware utilization using integrated Application Programming Interfaces (APIs). However, accurately

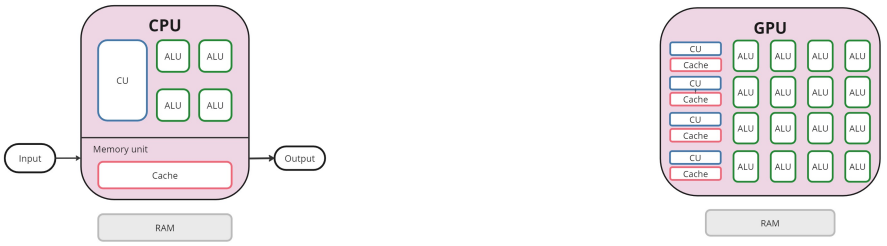
profiling energy consumption in specific processes is challenging [52]. This section will elaborate on the current research in energy measurement, consisting of what energy to measure and existing tools and estimators.

2-2-1 Energy to measure

Before diving deeper into the state-of-the-art measuring techniques several concepts need some clarification. This section provides a basic explanation of several hardware components for which it is important to measure their energy. Also, the full-life-cycle assessment will be discussed.

CPU and GPU computation

CPUs and GPUs are crucial processors for computationally demanding tasks, each designed for different but complementary purposes [61]. CPUs execute sequences of stored instructions and manage overall system operations, processing inputs, storing data, and outputting results through a fetch, decode, and execute cycle. Modern CPUs include an Arithmetic Logic Unit (ALU) for arithmetic operations, a Control Unit (CU) for control, and a cache to reduce memory retrieval costs [62] as shown in Figure 2-1a. However, their lack of extensive parallel processing limits performance in complex computations [63].



(a) CPU architecture **(b)** GPU architecture

Figure 2-1: The CPU and GPU architectures, showing their differences in sequential and parallel processing.

In contrast, GPUs, initially designed for graphics processing, excel in parallel processing and have expanded to general-purpose computing (General-Purpose Graphics Processing Unit (GPGPU)) [64]. They handle computationally intensive tasks like scientific simulations, data processing, and DL [65]. Unlike CPUs, GPUs operate many control units in parallel, enhancing efficiency in tasks like tensor computations [66]. This parallelization is visualized in Figure 2-1b, especially in comparison to the single CU structure of the CPU. This increased performance efficiency, however, can offset the energy efficiency of a GPU application [67] and requires considerations in tuning code and GPU configurations [68]. Another drawback of using GPU for performance efficiency is that it is more expensive than the CPU.

In the context of tensor decompositions, involving complex computations, the use of GPGPU still presents a potential option. Among others, research on using GPGPU for tensor decompositions includes an optimization algorithm for non-negative tensor factorization [66] and higher-order tensor computations in Google Colab [66]. The terms GPGPU and accelerated GPU are used interchangeably in tensor network research, however, they differ slightly. Besides focusing on general-purpose tasks, accelerated GPU includes accelerating other tasks

including graphics and communication. Accelerated GPU has been used for tensor computations such as the Hadamard product [63] and for a scalable C++ library for tensor network processing [69].

Memory
Another important component inside the processing hardware is Random-Access Memory (RAM) and can be divided into either Static Random-Access Memory (SRAM) or Dynamic Random-Access Memory (DRAM). Since DRAM is the most used component in current computer hardware the focus will lie on DRAM [70]. Despite its popularity, DRAM contributes heavily to energy consumption due to the movement of memory [71] and is almost 200 times more energy consuming than other operations, such as addition or multiplication [72].

Since the energy consumed for the memory allocation can be extensive, it is thus an important factor to consider when designing an energy-efficient CNN. The memory allocation and management can differ between the GPU and CPU, whereas for GPU the memory performance can be a key efficiency limitation [73]. Due to the parallel architecture of the GPU, handling multiple memory operations simultaneously, compared to the sequential processing of the CPU, efficient management and optimization of the memory access patterns is required to prevent bottlenecks and ensure all control units remain busy. It is thus more difficult to optimize the memory performance of this GPUs, in particular for energy efficiency where a lack of research is present, since research focuses more on the computational efficiency of the GPUs [73], [74]. In addition, the heterogeneous and distributed nature of the GPU subsystems adds to the complexity, with various types of caches and memory banks, showing different power usage and energy efficiency [75], [76]. These factors might make GPU memory management more complicated and energy-intensive compared to CPUs, highlighting the importance of extending the research into efficient memory management in the GPU, e.g. creating energy models based solely on memory usage [77].

Life cycle assessment
As mentioned before, current research mainly focuses on the training phase of developing new AI models. However, NVIDIA estimated that 80 to 90 percent of the total expected energy consumption is consumed during the inference phase [78]. Especially models that will be used extensively during deployment would have a great benefit in research into inference energy. However, the current research mainly focuses on the training of Machine learning (ML) models and sometimes even neglects to mention the inference phase. The fact that measuring the training energy is more isolated and controlled contributes to the lack of research in inference energy [78].

Another part of the energy consumption of new ML models is the full life-cycle assessment. The energy consumed by manufacturing and end-of-life stages of the hardware, data centres, and other parts is hard to estimate or predict and is thus frequently not included. Besides the research in [57], there is still a big research gap and there are great opportunities to look into these sources of energy consumption. [57] highlights the need for research and finds that probably most of the energy related to mobile devices and data centres comes from manufacturing and infrastructure. Due to the lack of research in the field of life-cycle assessment and the difficulty of requiring information about this topic, this part will not be taken into account further. However, the opportunities in the research around the energy consumption of running inference are addressed and will be discussed.

2-2-2 Methods to measure the energy

Following the taxonomy in [54], this section will discuss methods for measuring the energy consumption of the previously mentioned factors (CPU, GPU and DRAM), focusing on the software-level methods relevant to the energy consumption of the software implementations. These estimators will further be divided into internal (e.g. Python libraries) and external (e.g. watt meters) methods, following [79], focused on external tools, and [44], focused on internal tools.

External measuring
External measurements are commonly performed by power or watt meters with alternative options like Intelligent Platform Management Interfaces (IPMI) sensors, although these sensors lack accuracy [80]. Watt meters, used to measure the power in circuits, come in several types, such as electrodynamic watt meters, electronic watt meters and sensor-based watt meters (thermal, radiation, hall and diode)[79]. In contrast, energy meters, which measure the energy supplied to a circuit over time, include electric motor energy, induction-type watt-hour, electrolytic energy, static energy and all-digital energy meters, with the latter being the most modern and common [79].

All digital energy meters, like smart plugs, plug load meters and energy monitoring adapters, are typically placed between the wall socket and the power supply of the hardware [81], such as Kill a Watt and WattsUp [82], and can give information about voltage, power and current with adjustable sampling rates.

Recent smart meters proceeded on this allowing users to calculate, monitor, measure and transmission energy, enabling remote management of energy consumption [79]. However, the liability and performance of these meters can vary depending on the device, giving unrepresentable power measurements [81].

Internal measuring
One big limitation of external measuring is the inability to break down the energy consumption into component-specific energy. Internal measuring, relying on hardware sensors and software interfaces, does allow for this breakdown. These internal measuring tools, embedded by manufacturers, measure the power consumption for main components such as CPU, GPU and DRAM [81].

One of these embedded profiling APIs is Intel's Running Average Power Limit (RAPL) interface, developed to monitor and control the system's power consumption [81]. RAPL provides fine-grained and high-sampled measurements of the energy consumption and power limits for each domain, accessible through model-specific registers, read directly or through *sysfs* interface, *perf* events or the Performance API (PAPI) library [80]. However, detailed calculations by RAPL are not documented [81]. The RAPL data is mostly used for the measurements of the CPU, however, it is limited to Intel processors, it cannot provide data for virtual machines and the data is only accessible using Linux, since the Windows interface was discontinued.

For GPU measurements, Nvidia's C-based API library, Nvidia Management Library (NVML), monitors and manages GPU, providing real-time data on temperatures, utilization, power draw and other relevant metrics via *nvidia-smi* [83]. There are few details about the actual calculations of NVML. The library *pynvml* integrates NVML with Python.

Both RAPL and NVML measure the energy of the whole processor. To measure consumed energy at the software level, per processor energy consumption is necessary, which is retrieved

from real-time utilization information obtained from the cross-platform *psutil* library. If real-time utilization is unavailable, CPU utilization can be estimated using usage-based modelling, since the CPU utilization is correlated to the computing power [81]. The Thermal Design Power (TDP), provided by the manufacturer, can be a good approximation of the maximum power usage under steady workload [81]. Unlike CPU utilization, determining the exact GPU utilization is harder so commonly the full workload is assumed.

2-2-3 Internal measurement tools

Combining all these different libraries and interfaces to measure the full energy consumption of all three CPU, GPU and DRAM can be a complicated task and requires some knowledge of hardware and electrical engineering. To strive for more Green AI-driven metric reporting, keeping measurements accessible to all researchers, several tools and Python libraries have been developed. These tools are unique and can differ in some important parts. Below an overview will be given of a variety of state-of-the-art tools, their limitations and assumptions. Tools that are included are all still under development or updated. The tools will be divided into two different categories: the online calculators and the real-time energy monitors.

Online Calculators
Online calculators estimate energy consumption based on user-input hardware details and process information, without providing real-time data or life-cycle assessments.

Green algorithms [84] is a tool that calculates the carbon footprint and energy consumption of an algorithm, based on user-input such as running time (hours), number of cores, core model types, memory size (GigaBytes (GB)), core usage factor (0-1), location of running and PUE of the data centre. The calculator uses a database of known CPU and GPU models and their TDP values to find the power of the computing core. If the model is unknown, the user can provide the TDP and utilization or full utilization and a TDP of 12 W is assumed. For the power drawn by the memory, full memory utilization is assumed at an average of 0.3725 W/GB. Unsubstantiated assumptions can affect the accuracy and reliability of this tool.

Similarly, the ML Emissions Calculator (CO2 impact tracker) [85], estimates the CO2 emissions and energy consumption based on user input such as model type and training time. Unlike the Green algorithms, it focuses solely on GPU and does not explicitly display energy consumption. This methodology is less transparent, but further investigation reveals it uses the TDP to estimate the energy consumption which may not be entirely accurate [81].

Real-time estimators
The online calculators lack accuracy and are estimations of the energy consumption of algorithms after they have been run. Real-time information can increase the accuracy of the estimations and allow the researchers to monitor the energy consumption on the software level. Several tools have been developed, which vary greatly in approach and functionalities.

Codecarbon [86], developed by the same researchers as the ML emissions calculator, is a Python library that tracks the carbon footprint and energy consumption of algorithms during a predefined time interval of all three CPU, GPU and DRAM with the option of full machine or per process monitoring. Utilization is determined using *psutil* for DRAM, NVML for GPU (assuming zero if no NVIDIA GPU is found) and RAPL for CPU. If the RAPL files are unavailable, a non-substantiated 'global average' of 85 W is used as the TDP. Energy by

memory is not measured but is estimated at 0.375 W/GB based on Crucial DRAM [1]. The Codecarbon library is used often in literature. It is used to estimate the carbon footprint of large models, such as the Open Pretrained Transformer [87] and BLOOM [88]. In the context of autonomous driving Codecarbon is used to estimate the CO2 impact of a CNN trained for AVs [89]. Lastly, Codecarbon is used to perform a hardware-level comparison [90].

Merged into the current CodeCarbon library is the Energy Usage Reports [91] that calculates the energy consumption of a given function and publishes an energy usage report that gives more details about the calculated results and presents suggestions for locations with lower carbon footprint.

Another Python library available through PyPi is CarbonTracker [92], which estimates the energy consumption of DL models over a defined number of epochs, gathering real-time power samples from RAPL for CPU and DRAM, and from NVML for GPU. Specific processor utilization is not considered, so 100% is used. Also, if RAPL and NVML are unavailable, no alternatives are presented and zero energy consumption is reported. The Carbontracker is often used in literature, estimating the carbon footprint of large models, e.g. medical image analysis [93], and for 'common' data science [94]. In [94] it is found that the measurements of Carbontracker are very different from actual RAPL data. Besides that, also the influence of network setups on energy consumption has been researched using the data of the Carbontracker [95].

Tracarbon [96] is a python package hosted by PyPi, initially designed for Apple Mac but now also able to track RAPL files on Linux. There is no paper substantiating the methods used and it is unclear what the algorithm measures specifically and which processors are measured. The GitHub states that when implemented on non-Apple products it only uses RAPL data.

Contrary to the other tools, the Eco2AI library, focuses only on processes related to the training phase of ML models [97]. The total GPU power is retrieved from *pynvml*. If no NVIDIA GPU is available, then the GPU energy is not taken into account. The Python libraries *os* and *psutil* are used to track the utilization. The total CPU energy is based on the TDP of the CPU and when it is unknown, it is assumed to be 100 W based on [98]. RAM energy is proportional to allocated memory, with an estimate of 0.375 W/GB for DDR3 and DDR4 memory, for which utilization is tracked using *psutil*. Other memory models are not considered. Eco2AI has been used to measure the energy consumption of compressed transformers using matrix decomposition [99].

To summarize, these tools and their assumptions are presented in Table 2-1. It can be seen that these tools have large variability in methods and estimates, showing that the tools might not be as reliable as needed.

Other tools are present in the literature that try to estimate energy consumption, however, these tools are no longer under development and haven't been updated. Tools like the experiment impact tracker [37], EnergyVis [100], Pyjoules[2], Cumulator and pTop are mentioned in literature surveys and propose similar methods as the tools presented above.

[1] https://www.crucial.com/support/articles-faq-memory/how-much-power-does-memory-use
[2] https://powerapi.org/

Table 2-1: Overview of tools still under development, containing their measurement methods (main/alternatives) and utilization.

Tools	Methods	Utilization
Green algorithms	**CPU**: TDP/12 W **GPU**: TDP/12 W **RAM**: 0.375 W/GB	given or 100%
ML emissions calculator	**CPU**: - **GPU**: TDP **RAM**: -	not computed, so 100%
Codecarbon	**CPU**: RAPL/TDP/ 85 W **GPU**: NVML **RAM**: 0.375 W/GB	not computed, so 100%
Carbontracker	**CPU**: RAPL **GPU**: NVML **RAM**: RAPL	psutil, nvidia-smi or 100%
Tracarbon	**CPU**: RAPL **GPU**: - **RAM**: -	not computed, so 100%
Eco2AI	**CPU**: TDP **GPU**: NVML **RAM**: 0.375 W/GB	psutil, os or 100%

2-3 Tensor decompositions

As presented in the previous section, the complexity of both memory and computation can influence the energy consumption of a model and this can differ for different hardware architectures. As presented before the tensor decompositions show great potential for energy-efficient models, reducing the number of parameters in a model, with a great influence on the number of operations and memory usage. In this section, the preliminaries of tensors and three commonly used tensor decompositions will be elaborated by outlining the different methods, hyper-parameters, benefits and drawbacks. This provides a basic understanding of the theory behind this compression method and enables more intuitive understanding of the actual implementation in CNNs.

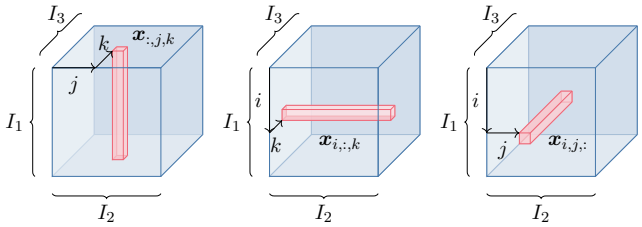
2-3-1 Preliminaries

To keep a consistent nomenclature, this section will be based on the terminology and theories used in [101] and [17]. The definition of a ***tensor*** used is different from the familiar stress tensor and tensor fields known from physics and engineering. In the context of this research, the tensor is defined as an N-dimensional array and is given as an Nth-order or N-way tensor. The ***order*** of a tensor is given by the number of dimensions also known as ***modes***. For clarity, a vector is a 1st-order tensor and a matrix is a 2nd-order tensor. To make a clear distinction between a higher-order tensor $\boldsymbol{\mathcal{X}}$, a matrix $\mathbf{X}$, a vector $\boldsymbol{x}$ and a scalar x, they have different notations. A special tensor is a ***diagonal*** tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ which only has entries on the superdiagonal, so $x_{i_1 i_2 \cdots i_N} \neq 0$ only if $i_1 = i_2 = \cdots = i_N$. Visualizing higher-order tensors is challenging so ***tensor diagrams***, also known as the Penrose graphical notations [102], were developed. Figure 2-4 shows five different tensor diagrams.

To talk about specific elements in tensors subscripts are used, also known as ***indices***. The indices of the different dimensions of tensors typically range from one to their uppercase

version, $i = 1, \ldots, I$. Using superscripts the elements in sequences are denoted, e.g. $\mathbf{X}^{(n)}$ which represents the nth matrix in a sequence.

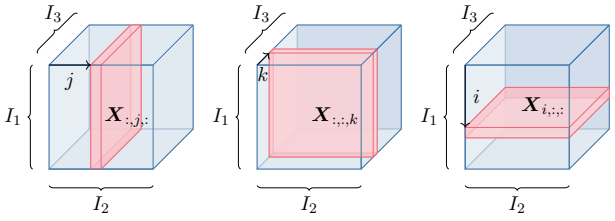
Similar to the concept of rows and columns in matrices, fixing all but one index in a tensor, is called a ***fiber***. Figure 2-2 shows a 3rd-order tensor where two out of the three indices are kept fixed resulting in column, row and tube fibres.



a) mode-1 fibers: $\boldsymbol{x}_{:,j,k}$ b) mode-2 fibers: $\boldsymbol{x}_{i,:,k}$ c) mode-3 fibers: $\boldsymbol{x}_{i,j,:}$

Figure 2-2: Mode-1 (column), mode-2 (row) and mode-3 (tube) fibers of a 3rd-order tensor (adapted from [103]).

Fixing all but two indices in a tensor is called a ***slice***. The lateral, frontal and horizontal slices of a 3rd-order tensor are visualized in Figure 2-3. For example, a 4th-order tensor has six different slices.



a) Lateral slices: $\boldsymbol{X}_{:,j,:}$ b) Frontal slices: $\boldsymbol{X}_{:,:,k}$ c) Horizontal slices: $\boldsymbol{X}_{i,:,:}$

Figure 2-3: Lateral, frontal and horizontal slices of a 3rd-order tensor (adapted from [103]).

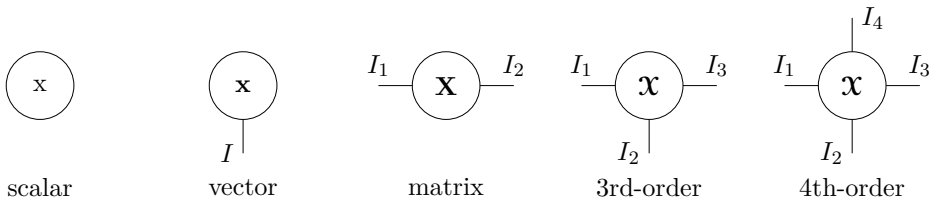


Figure 2-4: Different tensor diagrams for a scalar, a vector, a matrix, a 3rd-order tensor and 4th-order tensor.

An Nth order tensor is of ***rank one*** if it can be written as the outer product of N vectors, i.e.

$$\mathcal{X} = \mathbf{x}^{(1)} \circ \mathbf{x}^{(2)} \circ \cdots \circ \mathbf{x}^{(N)}. \tag{2-1}$$

The '$\circ$' symbol represents the ***outer product***. This means that each element of tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4}$ is given by a product of the corresponding vector elements, i.e.

$$x_{i_1 i_2 i_3 i_4} = x^{(1)}_{i_1} x^{(2)}_{i_2} x^{(3)}_{i_3} x^{(4)}_{i_4} \quad \text{for all } 1 \leq i_n \leq I_n. \tag{2-2}$$

The ***inner product***, of two same-sized tensors $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, is given by the sum of the product of their entries [101], as visualised in Figure 2-5, i.e.

$$\langle \mathcal{X}, \mathcal{Y} \rangle = \sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} x_{i_1 i_2 \cdots i_N} y_{i_1 i_2 \cdots i_N}. \tag{2-3}$$

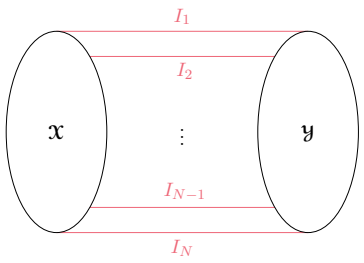


Figure 2-5: Tensor diagram of the inner product between $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ summing over all indices indicated by the red contracting lines.

Transforming an Nth-order tensor into a matrix is called ***unfolding***, ***flattening*** or mode-n ***matricization*** as visualized in Figure 2-6. Matricization can be done over each mode of a tensor. A mode-n unfolding of tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is denoted by $\mathbf{X}_{(n)}$. Unfolding rearranges the mode-n fibres to be the columns of the resulting matrix, called grouping, where tensor elements $(i_1, i_2, \ldots, i_N)$ map to matrix elements (i_n, j) where

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) J_k \quad \text{with} \quad J_k = \prod_{\substack{m=1 \\ m \neq n}}^{k-1} I_m \quad [101]. \tag{2-4}$$

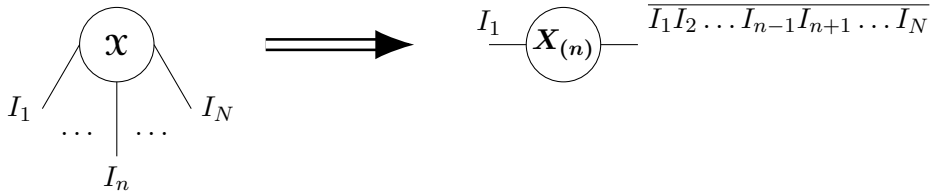


Figure 2-6: Tensor diagram of the mode-n matricization. Left the original Nth-order tensor is shown and on the right the mode-n matricization of the Nth-order tensor is shown, by grouping indices $I_1 I_2 \ldots I_{n-1} I_{n+1} \ldots I_N$.

The ***mode-n product*** entails the multiplication of the mode-n fibre with a matrix, replacing the n-mode dimension with the matrix dimension. The mode-n product between $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_n}$ and matrix $\mathbf{S} \in \mathbb{R}^{J \times I_n}$ is given as $\mathcal{Y} = \mathcal{X} \times_n \mathbf{S} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$. Element-wise this gives,

$$(\mathcal{X} \times_n \mathbf{S})_{i_1 \cdots i_{n-1} j i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \cdots i_N} s_{j i_n}. \tag{2-5}$$

Figure 2-7 visualizes the mode-n product between tensor $\mathcal{X}$ and matrix $\mathbf{S}$. The order of multiplication is irrelevant as long as the products are not along the same mode.

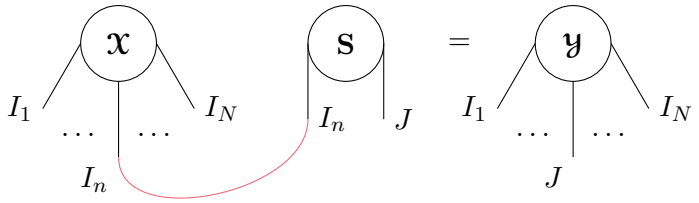


Figure 2-7: Mode-n product between tensor $\mathcal{X}$ and matrix $\mathbf{S}$ by contracting over I_n given by the red connecting line.

The ***Kronecker product*** between matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$ is given by $\mathbf{A} \otimes \mathbf{B}$ and it results in

$$\begin{aligned} \mathbf{C} = \mathbf{A} \otimes \mathbf{B} &= \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1J}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & a_{I2}\mathbf{B} & \cdots & a_{IJ}\mathbf{B} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_1 \otimes \mathbf{b}_2 & \mathbf{a}_1 \otimes \mathbf{b}_3 & \cdots & \mathbf{a}_J \otimes \mathbf{b}_{L-1} & \mathbf{a}_J \otimes \mathbf{b}_L \end{bmatrix}, \end{aligned} \tag{2-6}$$

with $C \in \mathbb{R}^{IK \times JL}$.

One important property of the Kronecker product is that each mode of tensor $\mathcal{Y}$ can be written as

$$\begin{aligned} \mathcal{Y} &= \mathcal{X} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)} \\ &\Updownarrow \\ \mathbf{Y}_{(n)} &= \mathbf{A}^{(n)} \mathbf{X}_{(n)} (\mathbf{A}^{(N)} \otimes \cdots \otimes \mathbf{A}^{(n+1)} \otimes \mathbf{A}^{(n-1)} \otimes \cdots \otimes \mathbf{A}^{(1)})^\top. \end{aligned} \tag{2-7}$$

The ***Kathri Rao product*** is the column-wise version of the Kronecker product. For matrices $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{J \times K}$ the Katrhi-Rao product $(\mathbf{A} \odot \mathbf{B})$ results in

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_2 \otimes \mathbf{b}_2 & \ldots & \mathbf{a}_K \otimes \mathbf{b}_K \end{bmatrix}, \tag{2-8}$$

with $\mathbf{C} \in \mathbb{R}^{IJ \times K}$.

The ***Hadamard product*** is an element-wise matrix product. For matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{I \times J}$ the Hadamard product $(\mathbf{A} \circledast \mathbf{B})$ results in

$$\mathbf{C} = \mathbf{A} \circledast \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1J}b_{1J} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2J}b_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \cdots & a_{IJ}b_{IJ} \end{bmatrix}, \tag{2-9}$$

with matrix $\mathbf{C} \in \mathbb{R}^{I \times J}$. The Hadamard notation '$\circledast$' is taken from [17] and is different from the notation of [101].

The Kronecker, Khatri-Rao and Hadamard products have different properties which are useful later on in the tensor decompositions [101] and are shown below. Here $\mathbf{A}^\dagger$ denotes the Moore-Penrose pseudo-inverse of $\mathbf{A}$ [104].

$$
\begin{aligned}
(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) &= \mathbf{AC} \otimes \mathbf{BD}, \\
(\mathbf{A} \otimes \mathbf{B})^{\dagger} &= \mathbf{A}^{\dagger} \otimes \mathbf{B}^{\dagger}, \\
\mathbf{A} \odot \mathbf{B} \odot \mathbf{C} &= (\mathbf{A} \odot \mathbf{B}) \odot \mathbf{C} = \mathbf{A} \odot (\mathbf{B} \odot \mathbf{C}), \\
(\mathbf{A} \odot \mathbf{B})^{\top}(\mathbf{A} \odot \mathbf{B}) &= \mathbf{A}^{\top}\mathbf{A} \circledast \mathbf{B}^{\top}\mathbf{B}, \\
(\mathbf{A} \odot \mathbf{B})^{\dagger} &= ((\mathbf{A}^{\top}\mathbf{A}) \circledast (\mathbf{B}^{\top}\mathbf{B}))^{\dagger}(\mathbf{A} \odot \mathbf{B})^{\top}.
\end{aligned} \tag{2-10}
$$

A summarized nomenclature of the previously presented concepts is given in Table 2-2.

Table 2-2: Nomenclature of the different tensor concepts and operations. On the left the notation is given with the definition on the right.

x	Scalar	$\circ$	Outer product
$\mathbf{x}$	Vector	$\otimes$	Kronecker product
$\mathbf{X}$	Matrix	$\odot$	Khatri-Rao product
$\mathcal{X}$	Tensor	$\circledast$	Hadamard product
i_n	Index of the nth dimension	$\times_n$	Mode-n product
I_n	Size of the nth dimension	$\|\mathbf{X}\|_F$	Frobenius norm
$x_{i_1,i_2,\ldots,i_N}$	An element of $\mathcal{X}$	$\dagger$	Pseudo-inverse

2-3-2 CP decomposition

The Parallel Factors (PARAFAC) decomposition, also known as the polyadic form of a tensor, topographic components model or PARAFAC Canonical Decomposition (CANDECOMP) (CP) [101], involves decomposing the tensor into a sum of component rank-one tensors, which is visualized in Figure 2-8. For higher-order CP decompositions, visualisation becomes challenging so tensor diagrams become essential as illustrated in Figure 2-9. Here, tensor $\mathcal{X}$ is approximated by a superdiagonal core tensor and factor matrices $A^{(1)}, A^{(2)}, \ldots, A^{(N)}$.

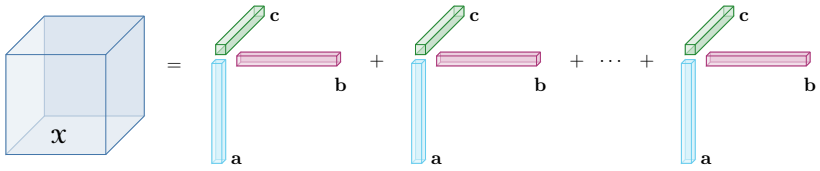


Figure 2-8: CP decomposition of a 3rd-tensor $\mathcal{X}$ into the sum of rank-one tensors (adapted from [101]).

The Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is approximated by

$$
\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{a}_r^{(1)} \circ \mathbf{a}_r^{(2)} \circ \cdots \circ \mathbf{a}_r^{(N)} = [\![\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!], \tag{2-11}
$$

consisting of R vectors $\boldsymbol{a}_r^{(1)}, \ldots, \boldsymbol{a}_r^{(N)}$ for $r = 1, \ldots, R$ with R a positive integer. On the right, an alternative shorthand notation for the CP is given. The matrices $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$ con-

tain the rank one vectors and are called the ***factor matrices***, i.e. $\mathbf{A}^{(1)} = \begin{bmatrix} \mathbf{a}_1^{(1)} & \mathbf{a}_2^{(1)} & \dots & \mathbf{a}_R^{(1)} \end{bmatrix}$ and similarly for all other modes.

To improve numerical stability or computational efficiency, it is common to normalize the factor matrices to length one. These weights are then contracted into vector $\lambda \in \mathbb{R}^R$ and from this the approximation changes into

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r \mathbf{a}_r^{(1)} \circ \mathbf{a}_r^{(2)} \circ \cdots \circ \mathbf{a}_r^{(N)} \equiv [\![\boldsymbol{\lambda}; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)}]\!]. \tag{2-12}$$

Each mode-n of the tensor can be approximated using the Khatri-Rao product between all other factor matrices and is given by

$$\mathcal{X}_{(n)} \approx \mathbf{A}^{(n)} \mathbf{\Lambda} \left(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)} \right)^{\top} \tag{2-13}$$

where $\mathbf{\Lambda} = diag(\lambda)$.

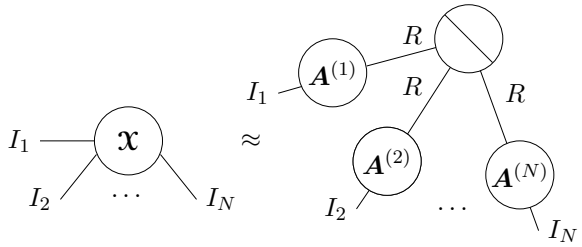


Figure 2-9: The tensor diagram of the CP decomposition of an Nth-order tensor $\mathcal{X}$ into N factor matrices $\mathbf{A}^{(1)} \dots \mathbf{A}^{(N)}$ (adapted from [16]).

Tensor rank

The ***rank*** of a tensor $\mathcal{X}$ is defined as the smallest number of rank-one tensors necessary for an exact representation. Unlike matrix rank, the tensor rank of a real-valued tensor can differ over $\mathbb{R}$ or over $\mathbb{C}$. There is no straightforward method to determine it. Finding the exact rank is NP-hard and an incorrect choice of rank affects the approximation: too low a rank can cause underfitting, while too high can cause overfitting.

Uniqueness

The CP decomposition of a tensor of rank R is ***essentially unique*** if the R rank-1 terms are unique, meaning no alternative decomposition exists for a given number of components. However, this uniqueness is subject to two indeterminacies: permutation and scaling. The CP decomposition is ***generally unique*** if the components are sufficiently different and the number of components is not unreasonably large. Additionally, a more deterministic condition, the Kruskal condition, and a generic condition exist and are further elaborated in Appendix A Section A-1.

Computing the CP decomposition

There are different algorithms to compute the CP decomposition, but one particular algorithm is used often in research, namely the Alternating Least Squares (ALS) method [101]. The main idea of the ALS is to iteratively minimize a cost function by optimizing each factor matrix individually while keeping the others fixed [17]. This results in a linear least squares

problem. The convex cost function used for the CP is commonly chosen to be the Frobenius norm which can be denoted as

$$J\left(\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}\right) = \|\mathcal{X} - [\![\mathbf{\Lambda}; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]\|_F^2. \tag{2-14}$$

The full CP-ALS algorithm can be found in Appendix A Algorithm 1. Besides the ALS algorithm, other techniques are suggested [105], [106].

2-3-3 Tucker decomposition

The CP decomposes a tensor into factor matrices, each representing a different mode, making it suitable for categorical and sparse data [107]. For dense data with structure and patterns spread over more than one mode, the CP decomposition is insufficient and cannot grasp the multi-mode complexity. An alternative decomposition is the Tucker decomposition, also known as Three-mode factor analysis (3MFA), N-mode Principle component analysis (PCA), Three-mode PCA (3MPCA), N-mode Singular Value Decomposition (SVD) and Higher-order SVD (HOSVD) [101]. The fundamental idea of the Tucker decomposition is that the tensor is decomposed into a core tensor, capturing interactions across all modes and factor matrices capturing all mode-specific information.

This idea generalizes to an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ that is approximated by

$$\mathcal{X} = \mathcal{G} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \cdots \times_N \mathbf{U}^{(N)} = [\![\mathcal{G}; \mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)}]\!], \tag{2-15}$$

consisting of a core tensor $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$ and factor matrices $\mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)} \in \mathbb{R}^{I_n \times R_n}$, for $n = 1, \ldots, N$, which can be thought of as principal components of each mode. Here the R_n depicts the number of components (i.e. columns) in the factor matrices. When $R_n < I_n$ then the Tucker decomposition is a compressed (i.e. truncated) version of the tensor $\mathcal{X}$. On the right side, an alternative shorthand notation is given for the Tucker decomposition.

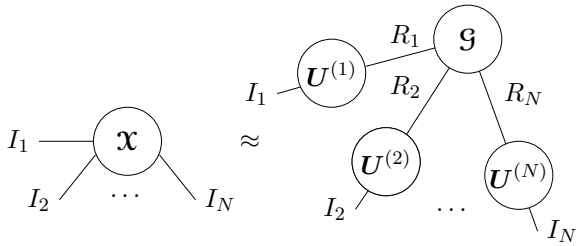


Figure 2-10: Tensor diagram of the Tucker decomposition (adapted from [16]).

The CP decomposition is a special form of the Tucker decomposition, where the core tensor is superdiagonal and the truncation R_n of each factor matrix is the same. For mode-n matricization, using the Kronecker product, the Tucker decomposition is given by

$$\mathbf{X}_{(n)} = \mathbf{U}^{(n)} \mathbf{G}_{(n)} \left(\mathbf{U}^{(N)} \otimes \cdots \otimes \mathbf{U}^{(n+1)} \otimes \mathbf{U}^{(n-1)} \otimes \cdots \otimes \mathbf{U}^{(1)}\right)^\top. \tag{2-16}$$

Computing the Tucker decomposition

The well-known and commonly used method to compute the Tucker decomposition of a

tensor is a form of SVD, generalized to higher order tensors, also known as Multilinear SVD (MLSVD) or HOSVD. For a 3rd-order tensor, the visualization of this approach is given in Figure 2-11, which shows the three factor matrices along each mode of the core tensor. The truncation parameters $R_1, R_2, \ldots, R_N$ can be determined by removing low left

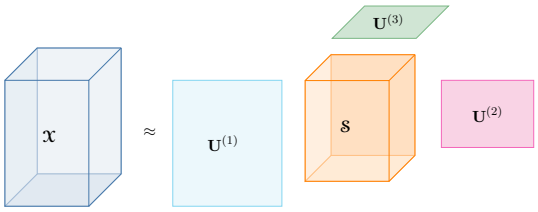


Figure 2-11: The MLSVD of tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ into core tensor $\mathcal{S} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$ and factor matrices $\mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)} \in \mathbb{R}^{I_n \times R_n}$ for $n = 1, \ldots, N$ (adapted from [101]).

singular values until the decomposition approximation is sufficient. Alternatively, a method can be used based on the Eckart-Young theorem [108]. This algorithm is common and is known as sequentially truncated HOSVD (st-HOSVD) [109] given in Appendix A Algorithm 2. Combining HOSVD with an initialization using ALS, presents the full algorithm known as Higher-order orthogonal iteration (HOOI) or HOSVD-ALS, which is given in Appendix A Algorithm 3. Besides HOOI, there are also alternative algorithms to compute the Tucker decomposition [110]–[112].

Uniqueness

Tucker decompositions are not unique but non-uniqueness can also bring some opportunities. It is possible to apply conditions on the factor matrices like non-negativity, and sparsity. In addition, with some constraints, the solution can inherently be unique, such as the HOSVD. Implementing the HOSVD to find the Tucker decomposition, can inherit the uniqueness properties of the SVD.

Multilinear rank

Contrary to matrix rank, the different mode-n ranks $R_1, R_2, \ldots, R_N$ are not all the same and in addition differ from the tensor rank. The rank corresponding to the N-tuple $R_1, R_2, \ldots, R_N$ is called the ***multilinear rank*** or ***n-rank*** and consists of the dimensions of the vector space spanned by its mode-n fibres, i.e.

$$\text{rank}_{ML}(\mathcal{X}) = \{\text{rank}(\boldsymbol{X}_{(1)}), \text{rank}(\boldsymbol{X}_{(2)}), \ldots, \text{rank}(\boldsymbol{X}_{(N)})\}. \tag{2-17}$$

2-3-4 TT decomposition

For higher-order tensors, the Tucker core can become large, potentially negating the benefits of decomposition due to the increased storage requirements for these large cores. As a solution, the Tensor Train (TT) represents the tensor as a structured low-rank approximation. Similar to CP and Tucker, TT decomposes the Nth-order tensor into N 3rd-order tensor cores. In practice, the ranks of the TT are lower than those of the CP making it a great alternative for high dimensional tensors.

The tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is approximated by the TT decomposition given by

$$\mathcal{X}(i_1, i_2, \ldots, i_N) = \mathcal{G}^{(1)}(:, i_1, :)\mathcal{G}^{(2)}(:, i_2, :) \cdots \mathcal{G}^{(N)}(:, i_N, :) = \langle\langle \mathcal{G}^{(1)}, \mathcal{G}^{(2)}, \ldots, \mathcal{G}^{(N)} \rangle\rangle \tag{2-18}$$

and consists of the multilinear product of 3rd-order TT-cores $\mathcal{G} \in \mathbb{R}^{R_{n-1} \times I_n \times R_{n+1}}$ ($n = 1, 2, \ldots, N$), shown in Figure 2-12. On the right side, the alternative shorthand notation is given.

In addition to dimensionality reduction, the TT is computationally more efficient due to its structure. Adding, the inner product, computation of tensor norms, Hadamard and Kronecker product, matrix-by-vector and matrix-by-matrix products can be efficiently performed using slice matrices of the individual core tensors.

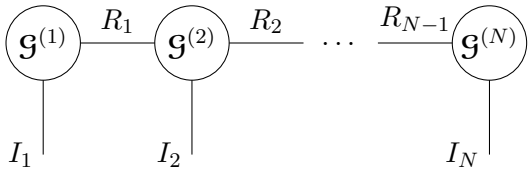


Figure 2-12: TT consisting of multiple tensor cores $\mathcal{G}^{(1)}, \mathcal{G}^{(2)}, \ldots, \mathcal{G}^{(N)}$ and tensor ranks $R_1, R_2, \ldots, R_{N-1}$.

Computing the TT decomposition

There are several methods to compute the TT approximation [113]–[115] but the TT-SVD algorithm is the most common. The full process of TT-SVD is given in Appendix A Algorithm 4 and is visualized in Figure 2-13. The TT-SVD makes use of the standard matrix SVD using truncation based on the Eckart-Young theorem [108]. The TT decomposition is not unique and so tensor train cores can be transformed for convenience.

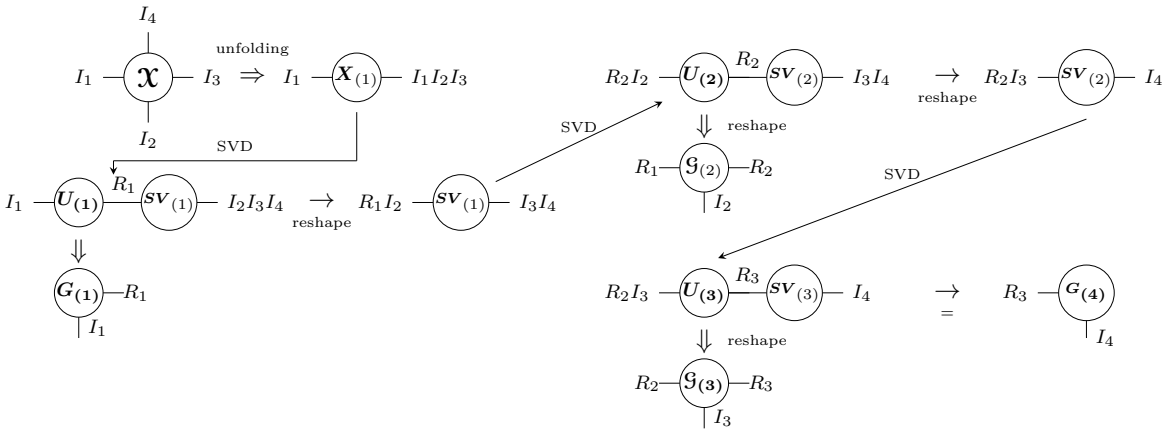


Figure 2-13: The TT-SVD process decomposing a 4th-order tensor (adapted from [116]).

TT-ranks

The truncation ranks in the TT format are defined as ***TT-ranks***. For each unfolded matrix $\mathbf{X}_n$ of tensor $\mathcal{X}$ with $\text{rank}(\mathbf{X}_n) = R_n$ there exists a decomposition with TT-ranks not higher than R_n [117]. The TT-ranks are thus bounded by the following rule

$$R_n = \text{rank}(\mathbf{X}_{(n)}) \leq \min(\prod_{k=1}^{n} I_k, \prod_{k=n+1}^{N} I_k). \tag{2-19}$$

The TT format is not unique so there might be situations in which the TT-ranks are not optimal for the representation of tensor $\mathcal{X}$. For example, basic linear algebra operations,

such as adding, can result in higher TT-ranks. There is a technique to find lower TT-ranks and a more optimal TT format without losing accuracy which is called ***TT-rounding***. The full algorithm is given in Appendix A Algorithm 5. Especially for higher-order tensors, it is important to find a low-rank approximation.

2-4 Energy-efficient CNNs

The presented tensor decompositions are not the only options for efficient CNNs. This research field has been growing, focusing on two main strategies: acceleration using specialized hardware and compression techniques. Hardware architecture greatly impacts code efficiency, with alternatives to general GPUs and CPUs, such as Field-Programmable Gate Array (FPGA) and Tensor Processing Unit (TPU), being explored [118], [119]. On the other side, model compression could create more energy efficiency and in the scope of this thesis, this section will focus on energy-efficient CNNs through model compression.

2-4-1 Model compression methods

Promising compression methods for energy-efficient CNNs include quantization, pruning, knowledge distillation (KD), and low-rank approximations, including the tensor decompositions, which will be shortly introduced to give a comprehensive and thorough overview of the current state-of-the-art.

Quantization reduces computational and memory costs by storing full-precision values in low bid-width precision. It can greatly compress a model, however, it can reduce the accuracy and requires hardware with low precision properties [15], [120], [121]. This trade-off between accuracy and onboard latency is of importance in object detection in AVs [122], [123].

Pruning [11], [13] involves zeroing out less influential parameters, e.g. gradients or weights, or removing entire channels, with various strategies like evolutionary algorithms [124], iterative L1 pruning [125], combinations with other methods like Knowledge distillation (KD) [126] or hardware-aware pruning [127] which are commonly used in object detection. Despite the compression abilities, this method can also reduce accuracy [128], [129], requiring retraining or fine-tuning and usually starts with an over-parameterized pre-trained model [128].

KD transfers knowledge from larger teacher models to lightweight student models [12], [14]. The performance of the student model depends on the architecture of the teacher [130] with limits on the distillable knowledge [131]. In object detection, different techniques are present [12] including instance-aware KD [122] and feature-based KD [129].

Unlike the other compression methods, focusing on training modifications, low-rank approximations also compress during training, enforcing low-rank structures in the network [132]. For instance, self-attention with low-rank approximation reduces the complexity in deep metric learning [133]. Additionally, low-rank structures in layers such as depth-wise separable convolutional layers [40] and learning low-rank structured sparsity in language models, have shown great promise in reducing parameters and accelerating models.

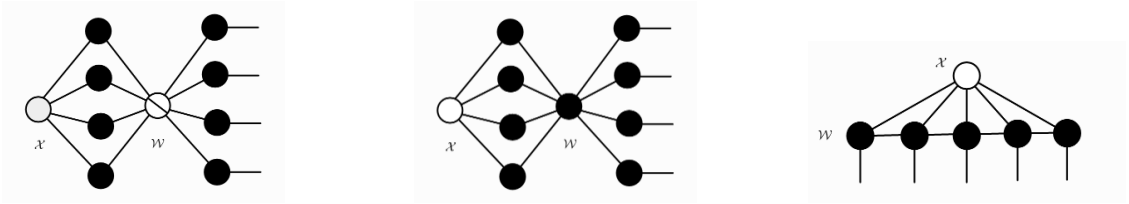
Low-rank tensor structures are especially relevant in visual data due to nonlocal similarities and underlying patterns [134]. These low-rank tensor structures are used in image completion

and denoising [135]–[137] and are suitable for tensor decomposition techniques. In recognition tasks in AVs, these decompositions can compress the model while still persevering underlying patterns in the data [138]. Unlike flattening, tensor decompositions maintain underlying patterns and exploit interactions of latent factors [138], as seen in TensorFaces [139] and multilinear Active Appearance Models (AAM's) [140].

A special form of low-rank approximation is tensor decomposition. These low-rank approximations can be used to approximate CNNs by decomposing convolution, pooling or fully connected layers [141], [142]. The pooling or sub-sampling layer, generates a smaller-sized output vector, compressing the number of trainable parameters [143]. Pooling layers inherently compress data, so tensor decompositions are less common there. Most research focuses on compressing the convolution layers, which can be more parameter-heavy and energy-consuming than fully connected layers, especially for deeper networks, making them ideal for tensor decompositions [144]. In addition, the redundancy among convolutional kernels, resulting from the difference in size between input/output channels and the kernel size, makes them very suitable for tensor decompositions [18].

2-4-2 Tensor decomposed convolutions

The focus of this thesis lies on the tensor decompositions and their implementation in convolution layers. This section will elaborate extensively on the current state-of-the-art of these decomposed convolution layers, consisting of the three popular tensor decomposition methods: CP decomposed convolutions, Tucker decomposed convolutions and TT decomposed convolutions, as presented in the tensor diagrams in Figure 2-14.



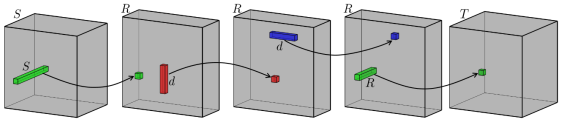
(a) CP-CNN **(b)** Tucker-CNN **(c)** TT-CNN

Figure 2-14: Tensor diagrams for CNN convolutional layers decomposed by either CP decomposition, Tucker decomposition and TT decomposition.

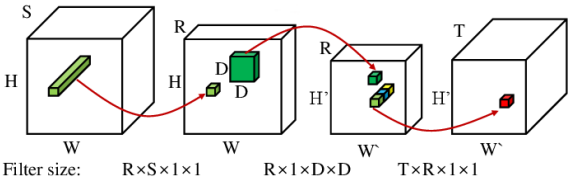
CP convolution

Using CP decomposition, the convolutional layers are approximated by four [19] or three [18] (combining the spatial dimensions) smaller convolutions, as shown in Figure 2-15. Another method is also suggested, that uses CP but does not break the original structure of the network [72].

Several other papers use CP decomposition to compress the kernels of CNNs [5], [145]–[148]. Each method contributes different insights. [149] was one of the first to use a low-rank approximation of the kernels and used a greedy CP decomposition.



(a) Decomposing the kernel into four factor matrices (reprinted from [19]).



(b) Decomposing the kernel into three factor matrices (reprinted from [150]).

Figure 2-15: Decomposing the kernel into factor matrices using the CP decmposition.

It was found that using this greedy update method was worse than alternative methods such as nonlinear least squares (NLS) [19] or ALS and using NLS led to better performance with or without fine-tuning. Alternatively, [149] used a bi-clustering method, exploiting the use of redundancy of the kernels in higher convolutional layers, which could also be a useful addition to the NLS [19]. In addition, the greedy method does not guarantee to decrease the size of the tensor and compresses the kernel sufficiently [18]. Using numerical optimization algorithms such as the NLS or ALS can furthermore result in diverging components and degeneracy (non-uniqueness) [18].

Balancing the estimation stability and accurate approximation is a challenge and asks for additional constraints for increased stability, such as column-wise orthogonality, positivity and nonnegativity. Unfortunately, these constraints are not always feasible to apply in certain data sets [18]. Moreover, the 'standard' CP decompositions, show sensitivity problems with increased rank and are not robust to small perturbations in the factor matrices [18]. To solve these sensitivity and stability problems [151] introduced the Error Preserving Correction (EPC) to the approximation which in addition to minimizing the Frobenius norm also bounds the approximation error ζ. Based on the EPC, the sensitivity of the decomposition can be minimized while still preserving a good approximation error, enhancing the convergence of the CP decomposition [18]. Another approach to enhance the convergence of the CP decomposition is based on the scaling factor. In [152], an extension of this normalization is used to enable separate optimization for global and per-rank scaling.

Tucker convolution

As mentioned in Subsection 2-3-3, the Tucker decomposition can take into account the interaction between different modes and factor matrices through a core tensor. Additionally, the use of the Tucker decomposition removes the challenge of finding a fitting rank and compression ratio. The Tucker ranks can be selected based on the left singular values as mentioned in Subsection 2-3-3 or by using Variational Bayesian Matrix Factorization (VBMF) as proposed by [153]. [154] proposes a method that learns the Tucker ranks during compressing and training, by optimizing the loss function of the CNN and the Tucker cost function simultaneously. Furthermore, utilizing the Tucker decomposition results in better convergence stability [154].

Due to small kernel sizes, the redundancy in those dimensions does not need reduction. That is why, several papers [18], [153]–[156] focus more on the compression of the third and fourth dimension of the kernels. This form of Tucker decomposition is also known as Tucker-2 decomposition. A sequence of three convolutions now substitutes the full convolution, as shown in Figure 2-16.

The Tucker decomposition is less researched in the context of CNNs. This is likely because the Tucker core tensor remains a 4th-order tensor and can consist of many parameters. However,

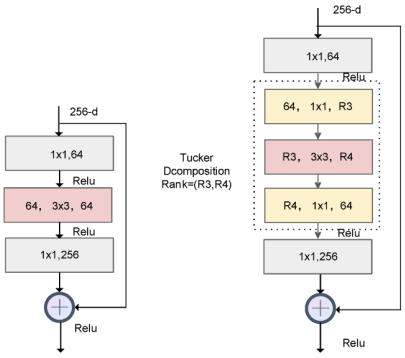


Figure 2-16: Tucker-2 decomposition of a convolution layer of a CNN (reprinted from [157]).

[155] uses Hierarchical Tucker-2 decompositions which show significant drops in the amount of FPO needed during training. This is achieved by also compressing the Tucker core using a binary-tree model which reduces the complexity of storage. The combination of using Tucker and CP decomposition can furthermore reduce the complexity of the Tucker core. [18] first uses the Tucker decomposition to decompose the convolution layers and then uses the CP decomposition to decompose the relatively large Tucker core.

These methods are all based on 4th-order kernels. In addition, [156] proposes two methods that can deal with higher order tensors, namely Adaptive Dimension Adjustment Tucker decomposition (ADA-Tucker), that performs Tucker decompositions on arbitrary-order tensors and Shared Core ADA-Tucker (SCADA-Tucker) that proposes a shared tensor core for all layers. Finally, to introduce non-linear instead of linear activation functions combined with the Tucker decomposition, [153] proposes an optimization algorithm that optimises both the multi-linear rank and the nonlinear cost function simultaneously. These frameworks can also be applied to fully connected layers.

Tensor Train convolutions

One of the first to apply TT decomposition in a Deep Neural Networks (DNN), applied the TT directly to the fully connected layers [148]. Subsequently, [158] was one of the first to apply TT in convolution layers. In this method the TT decomposition was applied on a matricized kernel and the convolution operations were rewritten into matrix-by-matrix multiplications. Alternatively, it is proposed to apply the TT directly to a permuted version of the kernel [20], [159]. [20] proposes a method that decomposes the convolution layer into a sequence of four layers consisting of the TT-cores as shown in Figure 2-17. Alternatively, similar to CP, combining the spatial dimensions allows for the substation of three convolutions [159].

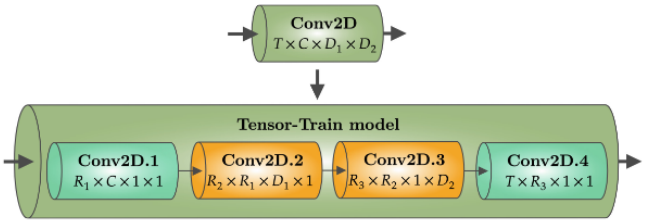


Figure 2-17: Decomposed convolution layer using TT into several smaller layers (reprinted from [20]).

As mentioned in Subsection 2-3-4, the ranks of the TT decomposition are not always optimal. A solution to this can be TT-rounding. Alternatively, [160] proposes a method to reduce these high TT-ranks by using a Riemannian Gradient Descent Method.

2-4-3 Training vs inference

There is an imbalance in research looking at the benefits of tensor decompositions in training and inference of CNNs. All of the already mentioned papers, except [153], [20] and [148], do not take into account inference and focus mainly on training. This is for the three main types of tensor decompositions CP, Tucker and TT and specifically in CNNs. Other research outside of these bounds can contain insights on running inference. [146] takes into account the benefits of implementing TT decomposition, however, focuses on implementing it in Recurrent neural network (RNN)s. Only one specific paper was found that showed a detailed outline of a TT-based inference engine [147].

2-5 Contributions

The previous sections have given a comprehensive overview of the current state-of-the-art research for efficient CNNs and the theory behind tensor decompositions. The objective of this thesis is to further investigate the potential of tensor decompositions for making large CNNs more energy efficient. In doing so it will build upon the foundation of the current research field presented in the previous sections.

In Section 2-1, the current state of Green AI is presented, highlighting the growing research into the 'sustainability' of large DL models. Traditional Red AI focuses on performance and accuracy, but new metrics are needed to capture the environmental impact represented by efficiency. Several metrics, summarized in Figure 2-18, focus on different aspects of energy efficiency but lack generality.

As shown in Figure 2-18, parameters, runtime, and FPO primarily allow for comparison based on code efficiency, often neglecting hardware discrepancies. Metrics such as CPU/GPU hours or utilization emphasize hardware efficiencies, however, lack generalization in algorithmic optimization. This thesis will focus on the energy consumed by a model, offering a holistic approach, encompassing hardware, computation and code efficiency.

Subsection 2-2-2 presents methods to measure or estimate energy consumption, using external watt meters or software implementations based on API data such as RAPL and NVML. Table 2-1, highlights the variability and reliability issues of these tools, whose techniques are not always well substantiated. Subsection 2-2-3 further indicates that these tools are often used when researching energy consumption, however, fail to check the reliability. There is a gap in the research, where the energy consumption is measured in real-time using an external device. Combined, this thesis has the following contribution:

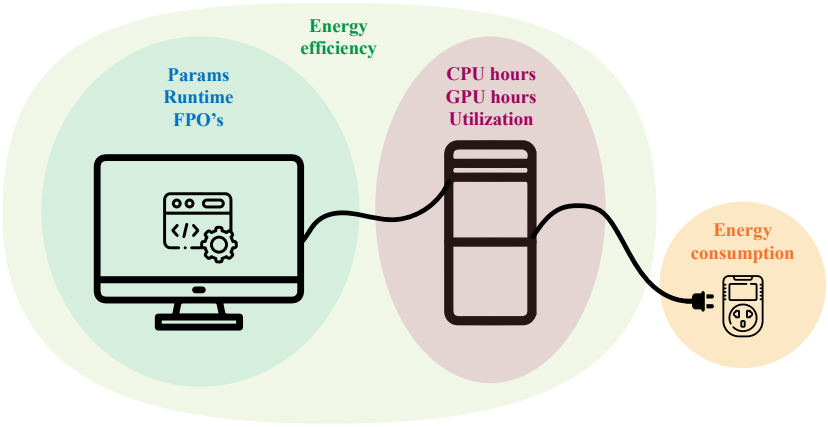


Figure 2-18: Overview of 'sustainability' metrics proposed in literature resembling the energy efficiency of a model. Some focus more on code implementation, whereas others focus on the impact of hardware. The energy consumption presents a more holistic approach, directly measured by an external device.

Contribution 1: Emperical real-time energy measurements

This thesis will contribute an empirical study design, where the energy efficiency of the model is presented by the energy consumption measured in real-time using an external watt meter. This allows for fully controllable energy measurements, increased reliability, and holistic comparison between different models.

As discussed in Subsection 2-4-2, there are several ways to compress a model, making it more energy efficient, with a particular focus on tensor decompositions. However, the focus lies on theory and measuring the actual energy consumption of implementing these methods is not done. Especially, looking at tensor decompositions, there is research that shows the potential of tensor decomposition for energy efficiency, however, no research measures the actual energy saved during implementation. In addition, most research looks into the benefits during training and not during inference. Combined another contribution of this research is:

Contribution 2: Inference energy savings of tensor-decomposed convolutions

This thesis will directly measure the energy saved by running inference on tensor-decomposed convolutional layers and identify key parameters that influence the amount of energy saved. This will be done by decomposing convolution layers with different configurations using different tensor decomposition methods and compression ratios, enabling not only theoretical comparison of energy efficiency but also practical.

Subsection 2-2-1 shows that current research has identified that GPU and CPU can result in different amounts of power usage due to architecture differences. However, when looking at tensor decomposition, there is a lack of research that presents the effects of implementing the decompositions on different types of hardware looking also at what this does for the energy consumption. This thesis will consider different hardware types resulting in the following contribution:

Contribution 3: CPU vs. GPU

This thesis will look at the differences in energy savings between running inference on tensor-decomposed convolution layers on both a CPU and a GPU. In this way, it will be possible to identify differences between hardware implementations comparing a sequential and parallel processor.

Current research mainly focuses on the theoretical reduction of parameters and computational complexity by using tensor decompositions. As was mentioned in Subsection 2-1-3, the reduced number of parameters might not directly correlate to the energy efficiency. Subsection 2-2-1 has shown that key aspects of energy consumption are computational complexity, e.g. MAC operations, and in addition memory usage. However, there is a gap in the research looking into whether the energy saved by these tensor decompositions can be predicted based on the expected computational complexity and especially on memory usage. So the final contribution of this thesis is:

Contribution 4: Energy modelling

This thesis will fit several models to predict the energy saved by implementing these tensor decompositions in convolutional layers, based on the relation, either linear or polynomial, between computational complexity, i.e. the number of MAC operations, and in particular the memory usage.

Chapter 3

Methodology

This chapter will build on the current state of the art presented in Chapter 2 by presenting the methodology used to answer the research questions and elaborating on the contributions of this thesis.

Figure 3-1 shows the different steps of this methodology. First, an analysis was done on the current implementation of tensor decompositions in convolutions, highlighting the important parameters. Based on these findings, experiments were designed, followed by the data acquisition, consisting of the energy consumption measurements. After that, the acquired dataset is processed and lastly, the findings have been used to model the energy consumption based on important parameters. The last four steps, bounded in the box, are where the contribution of this thesis comes through most since the analysis contains current state-of-the-art research too. The next sections will outline these different steps in more detail.

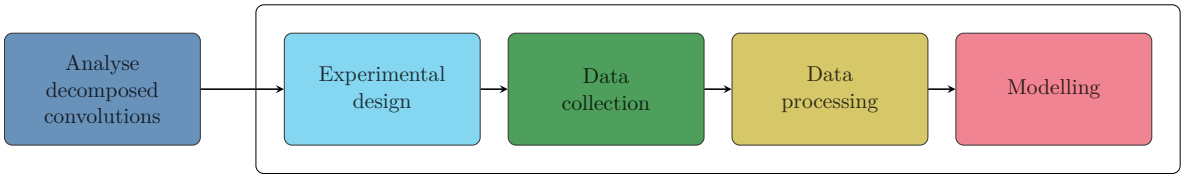


Figure 3-1: The methodological process of this thesis consists of five steps, for which the last four, bounded in the box, lie as a basis for this thesis contribution.

3-1 Decomposed convolutions

Several state-of-the-art papers highlight the theoretical benefits and challenges of decomposed convolutions, as mentioned in Subsection 2-4-2. However, practical implementation relies on hardware and software. The code implementation can have a great influence on the resulting efficiency. Since this thesis is focused on the potential efficiency increase of these decomposition methods, it will analyse a practical implementation of these decompositions instead of focusing on theoretical benefits. A popular state-of-the-art library TensorlyTorch

was used providing a combination of the PyTorch library and the Tensorly library, facilitating tensor learning through tools including tensor operations and tensor manipulations. It was chosen to use an existing library, used by current research, to show the current state of energy-efficient decomposed convolutions.

The documentation of TensorlyTorch lacks a clear explanation of how it is implemented and on which theory the implementation is based. In the next sections, the implementation will be elaborated to create a more intuitive understanding of the library's back end and ensure that further analysis of the results can be more extensive. Through this analysis, it is possible to compare theory and implementation and highlight any bottlenecks involving efficiency.

As mentioned in Subsection 2-2-1, two aspects are great contributors to the energy consumption of implemented software: the computational complexity and the memory usage. In addition to clarifying the TensorlyTorch decomposed convolutions, the analysis also allows for the identification of important parameters influencing energy savings by decomposing, based on these two aspects. To highlight any discrepancies between theoretical computation complexity, the calculated Multiply-Accumulate operations (MACs) based on the analysis of TensorlyTorch will be compared to the operations presented in the literature. Since literature does not involve memory considerations, it will be extended by this analysis, presenting a contribution to the current field.

3-1-1 Convolutional neural networks

First, a small introduction to 'regular' convolutions implemented in PyTorch is given, establishing the important parameters, design choices and their basic function. This allows comparison to the decomposed convolutions.

The structure of a Convolutional Neural Network (CNN) consists of one input layer and one output layer with multiple hidden layers in between. One specific neuron takes input vector $\mathbf{x}$ and produces output vector $\mathbf{y}$ following a given function with a certain weight vector $\mathbf{w}$. This weight vector represents the interconnection between neurons of two adjacent layers [143] and can then be used to extract certain features from images. An example of the CNN structure, in particular of ResNet18, is given in Figure 3-2 and clearly shows these types of hidden layers.

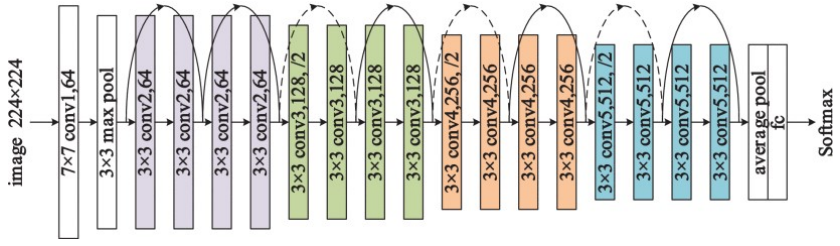


Figure 3-2: Architecture of Resnet18 (reprinted from [161]).

The convolution layer is an important hidden layer in the context of object detection. Within this layer, the weight vector, referred to as kernel or filter, slides over the input image and generates a feature map. Each neuron in the network is connected to other neurons in adjacent layers and this correlation is called the receptive field which builds up the kernel. Through weight-sharing similar features in different locations of the image can be detected [143]. The

sliding over the image is called the convolution operation and is a time-consuming operation. It maps the input tensor $\mathcal{U} \in \mathbb{R}^{W \times H \times S}$ into an output tensor $\mathcal{V} \in \mathbb{R}^{W' \times H' \times T}$ using the linear mapping

$$v_{w',h',t} = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{s=1}^{S} k_{i,j,s,t} u_{w(i),h(j),s} \quad [19]. \tag{3-1}$$

Here, the kernel $\mathcal{K} \in \mathbb{R}^{d \times d \times S \times T}$ is a 4th-order tensor with the spatial dimensions or filter size given by d and input and output channels S and T. Note that the output feature map $u_{w(i),h(j),s}$ is a function of i and j dependent on the padding and stride as given in Equation 3-2. Visualization of a full convolution is shown in Figure 3-3. The number of MAC operations in these convolutions is given by $STd^2W'H'$, where W' and H' are the width and height of the output feature map.

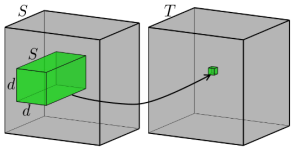


Figure 3-3: Full convolution operation (reprinted from [19]).

These weight kernels can become rather substantial and consist of many parameters, especially in deep or wide networks. As mentioned in Subsection 2-4-1, the redundancy in these convolutional kernels, due to the difference in size between the spatial dimensions and the number of input and output channels, makes them very suitable for tensor decompositions [18].

In addition to the kernel size, influenced by the number of input and output channels, other parameters influence the efficiency of the convolution operations. Looking at the MAC operations, the output feature maps are also of importance, which is given by

$$W' = \frac{W + 2P - d}{l}, \quad H' = \frac{H + 2P - d}{l}, \tag{3-2}$$

where W and H are the input feature map sizes, P is the padding and l is the stride. Padding is a technique to keep spatial dimensions after convolution, by adding extra entries around the original feature map. The stride is the number of 'steps' or pixels the sliding window makes.

3-1-2 CP convolution

For Parallel Factors (PARAFAC) Canonical Decomposition (CANDECOMP) (CP) TensorlyTorch uses the method proposed by [19], described in Subsection 2-3-2. The element-wise rank-R CP decomposition of a 4th-order tensor kernel $\mathcal{K} \in \mathbb{R}^{d \times d \times S \times T}$ is given by

$$k_{i,j,s,t} = \sum_{r=1}^{R} k^{(x)}_{i-x+\delta,r} k^{(y)}_{j-y+\delta,r} k^{(s)}_{s,r} k^{(t)}_{t,r} \quad [19], \tag{3-3}$$

where $\delta = \frac{d-1}{2}$. It shows that the convolution from Equation 3-1 is decomposed into four 2nd-order factor matrices $\boldsymbol{K}^{(s)} \in \mathbb{R}^{S \times R}, \boldsymbol{K}^{(t)} \in \mathbb{R}^{T \times R}, \boldsymbol{K}^{(x)} \in \mathbb{R}^{d \times R}, \boldsymbol{K}^{(y)} \in \mathbb{R}^{d \times R}$. The CP

decomposition is shown in Figure 2-15 and shows a clear difference to the full convolutional layer shown in Figure 3-3.

The complete convolution operation $\boldsymbol{\mathcal{V}}$, using a 4th-order tensor kernel $\boldsymbol{\mathcal{K}}$, is now substituted by a sequence of four convolutions with intermediate tensors $\boldsymbol{\mathcal{U}}^{(s)}, \boldsymbol{\mathcal{U}}^{(sy)}$ and $\boldsymbol{\mathcal{U}}^{(syx)}$ consisting of factor matrices $\boldsymbol{K}^{(s)}, \boldsymbol{K}^{(t)}, \boldsymbol{K}^{(x)}, \boldsymbol{K}^{(y)}$. Element-wise this is given by

$$\begin{aligned} u^{(s)}_{i,j,r} &= \sum_{s=1}^{S} k^{(s)}_{s,r} u_{i,j,s} \\ u^{(sy)}_{i,y,r} &= \sum_{j=i-\delta}^{y+\delta} k^{(y)}_{j-y+\delta,r} u^{(s)}_{i,j,r} \\ u^{(syx)}_{x,y,r} &= \sum_{i=x-\delta}^{x+\delta} k^{(x)}_{i-x+\delta,r} u^{(sy)}_{i,y,r} \\ v_{x,y,t} &= \sum_{r-1}^{R} k^{(t)}_{t,r} u^{(syx)}_{x,y,r}. \end{aligned} \tag{3-4}$$

The analysis of the TensorlyTorch pseudo code, actually showed that the implementation is slightly different than the theory shown above. Similar to the theory the convolution is substituted by four smaller 1D convolutions. However, the input and output sizes of the intermediate results are different, where the spatial dimensions are combined and reshaping is performed. This results in different intermediate tensors, influencing the memory and computational complexity, outlined below:

- The first layer, convolves over the number of input channels S, which involves a mode-n product over mode S. The input feature map $\boldsymbol{\mathcal{U}} \in \mathbb{R}^{B \times S \times WH}$, combining the spatial dimensions convolves with the first factor matrix $\boldsymbol{K}^{(s)} \in \mathbb{R}^{R \times S \times 1}$ into output feature map $\boldsymbol{\mathcal{U}}^{(s)} \in \mathbb{R}^{B \times R \times WH}$. The MACs of this convolution thus come to $BRSWH$.
- Before the next convolution some reshaping is performed, where the output feature map $\boldsymbol{\mathcal{U}}^{(s)} \in \mathbb{R}^{B \times R \times WH}$ of the previous convolution reshapes to input feature map $\boldsymbol{\mathcal{U}}^{(s)} \in \mathbb{R}^{BH \times R \times W}$. Then this feature map is convolved with the third factor matrix $\boldsymbol{K}^{(x)} \in \mathbb{R}^{d,1,R}$ to output feature map $\boldsymbol{\mathcal{U}}^{(sy)} \in \mathbb{R}^{BH \times R \times W'}$, resulting in $BHdRW'$ MAC operations.
- After permutation and reshaping the new input feature map $\boldsymbol{\mathcal{U}}^{(sx)} \in \mathbb{R}^{BW' \times R \times H}$ is convolved with the fourth factor matrix $\boldsymbol{K}^{(y)} \in \mathbb{R}^{d,1,R}$ into output feature map $\boldsymbol{\mathcal{U}}^{(sxy)} \in \mathbb{R}^{BW' \times R \times H'}$, resulting in $BW'dRH'$ MAC operations.
- Lastly, another mode-n product over mode-R of input feature map $\boldsymbol{\mathcal{U}}^{(sy)} \in \mathbb{R}^{B \times R \times W'H'}$ and factor matrix $\boldsymbol{K}^{(t)} \in \mathbb{R}^{T \times R \times 1}$ is performed resulting in the output feature map $\boldsymbol{\mathcal{V}} \in \mathbb{R}^{B \times T \times W'H'}$. When the last two dimensions are then separated, the original output size of the 'regular' convolution is reached. The MACs of this convolution thus come to $BRTW'H'$

In [19], it is stated that the complexity after decomposing becomes $R(S + 2d + T)$ for both the number of parameters and the number of operations per output pixels. This would come

to $BR(S+2d+T)W'H'$ MAC operations, which is different from the $BR(SWH+dW'H+dW'H'+TW'H')$ operations found in the analysis of TensorlyTorch.

As mentioned in Section 2-3, rank R is an important parameter. In addition to influencing the accuracy of the approximation, it also greatly influences the computational and memory complexity. For the CP, the rank is equal for all modes and as mentioned in Subsection 2-3-2, finding an optimal rank is NP-hard. Tensorly uses the desired reduction of parameters to calculate the rank, defined as the compression. The analysed Tensorly pseudo-code can be found in Section B-1. The compression is the amount of parameters left after compressing the convolutional layer, given as a fraction. So if the compression is $c = 0.1$, the result consists of 10% of the original number of parameters and the layer is thus compressed to 10%.

Based on the principles found in Section 2-3, the compression ratio is defined as

$$c = \frac{\text{number of params after compression}}{\text{number of params original tensor}} = \frac{R\sum_{k=1}^{N} I_k}{\prod_{k=1}^{N} I_k}, \tag{3-5}$$

where c is the compression rate, R is the CP rank, $I_1 = T, I_2 = S, I_3 = d$ and $I_4 = d$. As a result, R is thus given by

$$R = \frac{c\prod_{k=1}^{N} I_k}{\sum_{k=1}^{N} I_k}. \tag{3-6}$$

Tensorly uses this formula to determine the rank corresponding to the predefined compression rate, which can then be rounded, capped or floored according to the programmer's preference.

3-1-3 Tucker convolution

The TensorlyTorch library uses the Higher-order orthogonal iteration (HOOI) algorithm to compute the Tucker decomposition of kernel $\mathcal{K} \in \mathbb{R}^{T\times S\times d\times d}$ as defined in Subsection 2-3-3 and Algorithm 3 in Appendix A. Contrary to the efficient Tucker-2 method, omitting spatial modes from the decomposition [18], [153]–[156], TensorlyTorch decomposes the full kernel into four factor matrices $\mathbf{K}^{(t)} \in \mathbb{R}^{T\times R_1}, \mathbf{K}^{(s)} \in \mathbb{R}^{S\times R_2}, \mathbf{K}^{(x)} \in \mathbb{R}^{d\times R_3}, \mathbf{K}^{(y)} \in \mathbb{R}^{d\times R_4}$ and a tensor core $\mathcal{G}$ given by

$$\mathcal{K} \approx \mathcal{G} \times_1 \mathbf{K}^{(t)} \times_2 \mathbf{K}^{(s)} \times_3 \mathbf{K}^{(x)} \times_4 \mathbf{K}^{(y)}. \tag{3-7}$$

The complete convolution operation $\mathcal{V}$, using a 4th-order tensor kernel $\mathcal{K}$, is now substituted by three convolutions with intermediate tensors $\mathcal{U}^{(s)}$ and $\mathcal{U}^{(syx)}$ consisting of factor matrices $\boldsymbol{K}^{(t)}, \boldsymbol{K}^{(s)}, \boldsymbol{K}^{(x)}, \boldsymbol{K}^{(y)}$. Element-wise this is given by

$$u^{(s)}_{r_2,ij} = \sum_{s=1}^{S} k^{(s)}_{s,r_2,1} u_{s,ij} \tag{3-8}$$

$$u^{(syx)}_{r_1,i,j} = \sum_{j-\delta}^{j+\delta}\sum_{i-\delta}^{i+\delta}\sum_{r_2=1}^{R_2} \tilde{k}_{r_2,r_1,i,j} u^{(s)}_{r_2,i,j} \tag{3-9}$$

$$v_{t,xy} = \sum_{r_1=1}^{R_1} k^{(t)}_{t,r_1,1} u^{(syx)}_{r_1,ij}, \tag{3-10}$$

with $\tilde{\mathcal{K}} = \mathcal{G} \times_3 \boldsymbol{K}^{(x)} \times_4 \boldsymbol{K}^{(y)}$.

After analysing the TensorlyTorch pseudo-code to see the actual intermediate steps and implementation, highlighting any additional number of MACs or memory, these convolutions can be broken down in more detail. This decomposition results in two one-dimensional convolutions and one two-dimensional convolution, contrary to the three two-dimensional convolutions from the Tucker-2 implementation [1]:

- The first layer, convolves over the number of input channels S, which involves a mode-n product over mode S. The input feature map $\boldsymbol{\mathcal{U}} \in \mathbb{R}^{B \times S \times WH}$ convolves with the first factor matrix $\boldsymbol{K}^{(s)} \in \mathbb{R}^{R_2 \times S \times 1}$ into output feature map $\boldsymbol{\mathcal{U}^{(s)}} \in \mathbb{R}^{B \times R_2 \times WH}$. The MACs of this convolution thus comes to BR_2SWH.
- Before the next convolution some reshaping is performed separating the spatial dimensions to input feature map $\boldsymbol{\mathcal{U}^{(s)}} \in \mathbb{R}^{B \times R_2 \times W \times H}$. To get the new kernel $\tilde{\mathcal{K}}$, the mode-n products between the core tensor and the third and fourth-factor matrices is performed using Tensorly's *tenalg.multimode* function. Then a two-dimensional convolution is applied, convolving in both spatial dimensions, resulting in the output feature map $\boldsymbol{\mathcal{U}^{(syx)}} \in \mathbb{R}^{B \times R_1 \times W' \times H'}$. The number of MAC operations thus comes to the operations from the two mode-n products $2R_1R_2R_3R_4$ and the two-dimensional convolution $BR_1R_2W'H'd^2$.
- Lastly, another mode-n product over mode-R_1 of input feature map $\boldsymbol{\mathcal{U}^{(syx)}} \in \mathbb{R}^{B \times R_1 \times W'H'}$ and factor matrix $\boldsymbol{K}^{(t)} \in \mathbb{R}^{T \times R_1 \times 1}$ is performed resulting in the output feature map $\boldsymbol{\mathcal{V}} \in \mathbb{R}^{B \times T \times W'H'}$. When the last two dimensions are then separated, the original output size of the 'regular' convolution is reached. The MACs of this convolution thus comes to $BR_1TW'H'$.

For the popular Tucker-2 decomposed convolutions, the number of operations would come to $B(SR_2HW+d^2R_1R_2H'W'+TR_1H'W')$, which is different from the $B(WHSR_2+2(R_1R_2R_3R_4)+R_1R_2W'H'd^2+R_1TW'H')$ operations found in the analysis of TensorlyTorch, with additional intermediate results.

In Subsection 2-3-3, the multi-linear ranks were determined using the Higher-order Singular Value Decomposition (SVD) (HOSVD), however in TensorlyTorch the multi-linear ranks are determined based on the desired compression of the number of parameters, similar to CP. The analysed Tensorly pseudo-code can be found in Section B-1. Each factor matrix $U^{(n)}$ is truncated by rank R_n, so $R_n = t_r I_n$ with truncation factor t_r. The multilinear ranks that need to be found are thus defined as $(t_rI_1, t_rI_2, \ldots, t_rI_n)$. In the same way as CP the compression ratio is defined as shown in Equation 3-1-2 and rewritten this is given by

$$\text{number of params after compression} = c * \text{number of params original tensor.} \tag{3-11}$$

The number of parameters after compression is given by the number of parameters in the factor matrices plus the number of parameters in the Tucker core given by

$$\text{number of params after compression} = \underbrace{t_r^N \prod_{k=1}^{N} I_k}_{\text{Tucker core}} + \underbrace{t_r \prod_{k=1}^{N} I_k^2}_{\text{factor matrices}} . \tag{3-12}$$

[1] https://github.com/jacobgil/pytorch-tensor-decompositions/tree/master

To find the truncation factor the Brent optimization method [162] is used after which the ranks are calculated based on a combination of Equation 3-1-3 and Equation 3-1-3 given by

$$f(t_r) = t_r^N \prod_{k=1}^{N} I_k + t_r \prod_{k=1}^{N} I_k^2 - c \sum_{k=1}^{N} I_k, \tag{3-13}$$

where $I_1 = T, I_2 = S, I_3 = d$ and $I_4 = d$. Due to the use of these optimization algorithms the influence of the different CNN parameters, such as number of input channels, number of output channels, kernel size and feature size becomes less interpretable and intuitive.

3-1-4 TT convolution

For the Tensor Train (TT) convolution the implementation is similar to [20] but different from [158]. The TensorlyTorch uses the TT-SVD [117] method presented in Subsection 2-3-4 and Algorithm 4 in Appendix A to decompose the permuted kernel $\mathcal{K} \in \mathbb{R}^{S \times d \times d \times T}$ into four core tensors $\mathbf{G}^{(s)} \in \mathbb{R}^{S \times R_1}$, $\mathcal{G}^{(x)} \in \mathbb{R}^{R_1 \times d \times R_2}$, $\mathcal{G}^{(y)} \in \mathbb{R}^{R_2 \times d \times R_3}$, $\mathbf{G}^{(t)} \in \mathbb{R}^{R_3 \times T}$. Element-wise this is given by

$$k_{s,x,y,t} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} g^{(s)}_{s,r_1} g^{(y)}_{r_1,x,r_2} g^{(x)}_{r_2,y,r_3} g^{(t)}_{r_3,t} \qquad [20]. \tag{3-14}$$

The complete convolution operation $\mathcal{V}$, using a 4th-order tensor kernel $\mathcal{K}$, is now substituted by a sequence of four convolutions with intermediate tensors $\mathcal{U}^{(s)}, \mathcal{U}^{(sy)}$ and $\mathcal{U}^{(syx)}$. Element-wise this is given by

$$u^{(s)}_{i,j,r_1} = \sum_{s=1}^{S} g^{(s)}_{s,r_1} u_{i,j,s} \tag{3-15}$$

$$u^{(sy)}_{i,y,r_2} = \sum_{j=i-\delta}^{y+\delta} \sum_{r_1=1}^{R_1} g^{(y)}_{r_1,j-y+\delta,r_2} u^{(s)}_{i,j,r_1} \tag{3-16}$$

$$u^{(syx)}_{x,y,r_3} = \sum_{i=x-\delta}^{x+\delta} \sum_{r_1=2}^{R_2} g^{(x)}_{r_2,i-x+\delta,r_3} u^{(sy)}_{i,y,r_2} \tag{3-17}$$

$$v_{x,y,t} = \sum_{r-1}^{R_3} k^{(t)}_{r_3,t} u^{(syx)}_{x,y,r_3}. \tag{3-18}$$

The analysis of the TensorlyTorch pseudo-code showed that the implementation is slightly different from the theory. Similar to the theory the convolution is substituted by four one-dimensional convolutions, however, additional reshaping is performed to fit the convolutions, as presented below in more detail.

- The first layer, convolves over the number of input channels S, which involves a mode-n product over mode S. The input feature map $\mathcal{U} \in \mathbb{R}^{B \times S \times WH}$ convolves with the tensor core $\boldsymbol{G}^{(s)} \in \mathbb{R}^{R_1 \times S \times 1}$ into output feature map $\mathcal{U}^{(s)} \in \mathbb{R}^{B \times R_1 \times WH}$. The MACs of this convolution thus come to BR_1SWH.

- Before the next convolution the spatial dimensions are again unfolded to input feature map $\mathcal{U}^{(s)} \in \mathbb{R}^{B \times R \times W \times H}$. Then this feature map is convolved with the reshaped core tensor $\mathcal{G}^{(y)} \in \mathbb{R}^{R_2 \times R_1 \times d}$ to output feature map $\mathcal{U}^{(sy)} \in \mathbb{R}^{B \times R_2 \times W \times H'}$, resulting in BR_2R_1dWH' MAC operations.
- The input feature map $\mathcal{U}^{(sy)} \in \mathbb{R}^{B \times R_2 \times W \times H'}$ is then convolved with the reshaped core tensor $\mathcal{G}^{(x)} \in \mathbb{R}^{R_3 \times R_2 \times d}$ to output feature map $\mathcal{U}^{(syx)} \in \mathbb{R}^{B \times R_3 \times W' \times H'}$, resulting in $BR_2R_3dW'H'$ MAC operations.
- Lastly, another mode-n product over mode-R_3 of input feature map $\mathcal{U}^{(syx)} \in \mathbb{R}^{B \times R_3 \times W'H'}$ and factor matrix $\boldsymbol{G}^{(t)} \in \mathbb{R}^{R_3 \times T \times 1}$ is performed resulting in the output feature map $\mathcal{V} \in \mathbb{R}^{B \times T \times W'H'}$. When the last two dimensions are then separated, the original output size of the 'regular' convolution is reached. The MACs of this convolution thus come to $BR_3TW'H'$.

In [20], it is stated that the complexity after decomposing is $B(R_1WH + R_2d(R_1WH + R_3W'H) + R_3TW'H')$, which is slightly different from the $BWHSR_1 + BR_2(R_1dWH' + dR_3W'H') + BW'H'TR_3$ operations found in the analysis of TensorlyTorch.

As mentioned in Subsection 2-3-4, the TT-SVD algorithm might result in higher and non-optimal ranks. In addition to using the compression ratio to find the TT-ranks, the implementation of Tensorly finds the ranks proportional to the size of the dimensions. The analysed Tensorly pseudo-code can be found in Section B-1.

The number of parameters in the first and final TT-cores is given by R_1I_1 and R_NI_N respectively. For the in-between TT-cores, the number of parameters is given by $R_kI_kR_{k+1}$ for $k = 1, \ldots, N-1$. If the rank was chosen to be the same this would result in R^2I_k. Now, similar to the Tucker ranks, a truncation factor $t_r^{(k)} = \frac{I_k + I_{k+1}}{2}$ is chosen proportional to the dimension size and this results in the number of parameters being $(t_r^k R)I_k(t_r^{k+1}R)$. Now the compression ratio can be defined as

$$c = \frac{\sum_{k=1}^{N} t_r^k I_k t_r^{k+1} R^2 + (t_r^1 I_1 + t_r^N I_N)R}{\prod_{k=1}^{N} I_k}, \tag{3-19}$$

which results in the following quadratic formula

$$\sum_{k=1}^{N} t_r^k I_k t_r^{k+1} R^2 + (t_r^1 I_1 + t_r^N I_N)R - c \prod_{k=1}^{N} I_k = 0, \tag{3-20}$$

from which the rank $R_k = t_r^k R$ is calculated using $I_1 = S, I_2 = d, I_3 = d$ and $I_4 = T$.

3-2 Experimental design

The previous analysis described the modified convolutions, revealing differences from the current theoretical state-of-the-art implementation. Based on this analysis two types of experiments were designed: one set isolates a single convolution layer to map the influence of various parameters on energy consumption, and another set applies tensor decompositions to a CNN, specifically ResNet18 using the CIFAR10 dataset.

3-2-1 Single convolution

Table 3-1 summarizes the found computational and memory complexity. The memory complexity is chosen to be the added memory by the 'new' intermediate results since this is what sets the methods apart from each other. To allow a fair comparison between the methods, the amount of parameters in the kernels has been excluded since each method has the same amount of parameters due to the rank selection of Tensorly based on the compression ratio. In addition, the input and output feature maps were not included as they are the same for each method and the original convolution.

Table 3-1: Intermediate storage complexity and computational complexity for different tensor decomposition methods.

Method	Complexity
CP	Memory: $BRWH + BRW'H + BRW'H'$ Computation: $BR(SWH + dW'H + dW'H' + TW'H)$
Tucker	Memory: $BR_2WH + BR_1W'H' + 2R_1R_2R_3R_4$ Computation: $B(WHSR_2 + 2(R_1R_2R_3R_4) + R_1R_2W'H'd^2 + R_1TW'H')$
TT	Memory: $BR_1WH + BR_2W'H + BR_3W'H'$ Computation: $B(WHSR_1 + R_2(R_1dW'H' + dR_3W'H') + W'HTR_3)$
Regular	Computation: $Bd^2STW'H'$

For each experiment, several independent variables were used, based on the identification of key parameters from Table 3-1, to find the correlation between the energy consumption and the different variables. This resulted in the following experimental set $F \in \{S, T, d, W, M, c, Ha\}$:

- The number of input channels $S \in \{192, 256, 320, 384, 448\}$. The specific number is chosen based on the practical example of ResNet18, presented in Subsection 3-1-1, showing input channels as a multiple of 64.
- The number of output channels $T \in \{192, 256, 320, 384, 512\}$. The specific number is chosen based on the practical example of ResNet18, presented in Subsection 3-1-1, showing output channels as a multiple of 64, where $T \geq S$.
- The input feature size, chosen to be square, $W \in \{2, 4, 6, 8\}$. Smaller input feature sizes combined with larger input and output channels were selected to represent the deeper layers of convolutional networks, that are expected to benefit most from decompositions.
- The kernel size, chosen to be square, $d \in \{1, 3, 5\}$, as these are common in popular architectures such as ResNet[2].
- The tensor decompositions method $M \in \{cp, tucker, tt, not\ decomposed\ (nd)\}$.
- The compression ratio $c \in \{0.1, 0.25, 0.5, 0.75, 0.9\}$, defined as the ratio of parameters kept, not the number of parameters removed. For example, a compression ratio of ten per cent means, that the decomposed kernel consists of 10% of the original kernel and $c = 0.1$ is thus the most compression. This definition was chosen to be the same as

[2]Resnet18 consists of kernels 1×1 and 3×3. In addition, larger ResNets can also contain kernels of 5×5

defined by TensorlyTorch. A multitude of compression ratios was chosen to show the range and influence of small changes in compression ratio.

- The hardware used, Ha ∈ {Intel Core i7-6700HQ Central Processing Unit (CPU),NVIDIA GeForce GTX Titan X Graphics Processing Unit (GPU)}. The first being a laptop and the second a desktop, which were available for this thesis.

Separate experiments to see the influence of padding and stride were not designed, as Table 3-1 already suggests that a higher stride leads to more MAC and larger intermediate results. Padding was kept constant at one, a common size that keeps spatial dimensions intact.

The common dependent variables are the energy consumption, the calculated number of MAC operations, the calculated additional memory based on the intermediate output feature maps and the measured memory using a profiling tool. Calculation of the intermediate results, next to real-time measurements, was chosen to determine whether pre-calculated memory complexity could accurately represent actual memory complexity. If this is confirmed, estimating energy savings before actual measurements would be possible, as discussed in SQ4.

The experiments run can be divided into four 'themes', each showing the influence of one CNN parameter (S, T, d, W) and together showing the influence of the different tensor decomposition parameters (M, c). This division will also be used in Subsection 3-2-2. Each experiment in this section has a basic experimental design. For these experiments, a controlled environment was desired to focus on the specific gains of decomposing. That is why it was chosen to run inference on one isolated convolution layer, by performing multiple forward passes. Using PyTorch, one convolution layer was made using *Conv2d*, based on the independent variables defined in each experiment. The layer is decomposed using the previously mentioned TensorlyTorch library. Since only the inference energy is of interest the *with torch.no_grad* mode of PyTorch is used during each run, disabling any gradient calculations. To decompose the layer, the *tltorch.FactorizedConv.from_conv* function is used from TensorlyTorch, for which the analysis was already given in Section 3-1. It is important to note that the *FactorizedConv.from_conv* library on default reconstructs the kernel when performing the convolution, which means only benefiting from less memory. Changing the parameter '*implementation*' is crucial for efficient decomposed convolutions.

3-2-2 Decomposed Resnet18

To provide a broader perspective, including a real implementation instead of a single convolution, energy and memory measurements of running inference on a commonly used CNN were performed. A similar method to [163] was used, where multiple layers of ResNet18 were decomposed using different tensor decompositions with various compressions. Unlike [163], this thesis focuses on energy and memory consumption rather than the approximation error.

The pre-trained ResNet18 from [163] was used as a starting point for decomposition. The open-source GitHub code [3] [163] was adapted by adding the data collection code (Section 3-3). The *FactorizedConv* implementation was adjusted to be 'factorized' instead of using reconstructed convolutions. [163] focused on approximation error and accuracy, saving models during the best epochs, which caused random energy drops in the energy measurements.

[3] https://github.com/JSchuurmans/tddl

Removing this part stabilized the measurements, crucial for reliable results. Using a similar methodology as the previous isolated experiments, only the forward pass was performed, using *torch.no_grad()*.

This experiment was run only on GPU due to computational constraints of the CPU. The configurations of the chosen convolution layers are detailed in Section B-4 in the appendix. Not all layers were decomposed as some have identical hyperparameters. All three tensor decompositions were used, with various compression ratios to measure smaller differences. The same seed (*torch.manual_seed(), random.seed(), np.random()*) was used for consistency across all runs.

The reduced energy consumption of decomposing the separate layers will be presented as the ratio between the energy saved (measured based on the energy consumption of running inference on the whole CNN) and the original baseline energy when not decomposing, i.e. $R_{\text{energy}} = \frac{E_{\text{baseline}} - E_{\text{decomposed}}}{E_{\text{baseline}}}$.

3-3 Data collection

This section discusses the methodology used for collecting the energy and memory measurements. Two extraneous effects were taken into account for all forms of data collection to ensure the reliability of the results:

- Background processes: Measurements were done on both the GPU and CPU with no visible processes running in the background. Invisible processes and Ubuntu updates could not be controlled, so three runs were performed per experiment at different times to account for variability.
- Software configurations: Consistent software configurations were used across all experiments, by using the same libraries and versions on both the laptop and desktop (specific requirements can be found on GitHub). This allows for direct comparison.

First, an overview of the energy measurement methodology will be presented, followed by memory measurements, and finally the logging of the data.

3-3-1 Energy measurement

One key contribution of this thesis is measuring the real-time energy consumption of running code. As discussed in Section 2-2, existing tools for monitoring energy consumption use various methods and libraries, often without substantiated methodologies. For this thesis, careful control over measurements was important. Additionally, one of the objectives was to model energy consumption based on memory usage and MAC operations. The state-of-the-art monitoring tools typically estimate memory energy consumption based on a global average, not the energy specific to actual memory usage. Therefore, a direct measuring tool, the watt meter, was chosen instead of software tools.

Another benefit of using a watt meter is its hardware independence. Many software tools require specific software and library versions, which can interfere with current code implementations or other software packages. The watt meter avoids conflicting packages and

provides a simple, effective way to measure energy consumption. Additionally, running these tools in the background adds an extra time cost, possibly interfering with the measurements and this could add up for many experiments.

A requirement for the watt meter was the ability to log measurements frequently and not only show the measurements on a display. If the energy is logged, then the meter logs data at low frequencies, such as daily averages. Industrial watt meters offer high-frequency logging and precision but are expensive and sometimes require integration into the circuit, which was not feasible for the personal laptop and desktop. Therefore, the VoltCraft SEM5000 [4] was chosen, depicted in Figure 3-4. This watt meter is placed between the power adapter and the wall socket, logging voltage, current, and power every minute to an insertable and removable SD card. It logs the current value at the start of each minute and does not average over the minute, providing the necessary frequency. Table 3-2, shows the technical details of the measurement tool, including tolerance (deviation in measurements), resolution (number of decimals) and the frequency of the measurements.

Table 3-2: Technical Details of the VoltCraft SEM-5000

Parameter	Range	Specification
Tolerance	$< \pm 2\%$	> 10 W
	$< \pm 10\%$	3 - 10 W
	$< \pm 0.03$ W	< 3 W
Resolution	0.001 W	0.3 - 9.999 W
	0.01 W	10.00 - 99.99 W
	0.1 W	100.0 - 999.9 W
	1 W	1000 - 3680 W
Frequency	Every minute	

Figure 3-4: The VoltCraft SEM-5000

Because the energy measurements had a frequency of one minute, the number of epochs was chosen in such a way that each measurement would run a sufficient amount of time, around four minutes, so enough data points were gathered per experiment.

Measuring power through wall sockets introduces challenges due to power fluctuations. These fluctuations can result from grid capacity differences (e.g. night vs. day) or from internal power management in GPUs and CPUs, which adjust power consumption based on workload. To address this, three runs of each experiment will be measured and run at varying times rather than consecutively.

The steps used to start the measurements using the watt meter are as follows:

1. Charge the used hardware, if possible (laptop), completely so that all energy drawn from the wall socket is assumed to be for the processes running on the device. For a desktop, the energy drawn for idle consumption is assumed to be consistent, whereas the energy drawn for charging laptops is not consistent over time.

[4] https://www.conrad.nl/nl/p/voltcraft-sem5000-energiekostenmeter-kostenprognose-alarmfunctie-instelbaar-stroomtarief-datalogger-2587314.html

2. Install the watt meter between the power plug and the wall socket, by setting the precise time and date using the *SET*-button as seen in Figure 3-4. Then define the precise measurement period using the *F*-button. When the circle light is green it means the watt meter is initialized correctly and is ready, otherwise, it is coloured red.
3. Start running the code which needs to be measured.
4. After the code is finished, read out the SD card and process the data which methodology is further described in detail in Subsection 3-3-3.
5. Repeat this for three runs of each experiment.

3-3-2 Memory profiling

To measure memory usage, memory profiling tools were used in addition to the calculated estimates based on intermediate results from Table 3-1. For both CPU and GPU, memory profiling was executed during a single forward pass to ensure direct comparability with the calculated memory and avoiding the influence of PyTorch's internal memory management, which uncontrollably releases and frees up memory after epoch runs. Memory usage was recorded in bytes and then converted to megabytes (MB) by dividing it by 1024^2.

In more detail, the CPU memory was measured using the integrated PyTorch memory profiling tool [5], profiling the memory consumption of the model operators. The profiler variables are set in 12 in Listing 3.3.

For the GPU memory consumption, the PyTorch profiler tool did not manage to make use of the *pynvml*, due to version conflicts, so another method was used. As shown in Listing 3.1, in each epoch the PyTorcyh CUDA peak memory stats are cleared and the peak memory allocated is then used to represent the memory consumption of the GPU.

To connect the energy savings to the measured memory usage, the additional memory was calculated by subtracting the memory used in each baseline run (without decomposition) from the memory used in each experimental run.

```
with torch.no_grad():
    torch.cuda.reset_peak_memory_stats()
    for _ in tqdm(range(m), desc="Forward Iterations"):
        output = model(Variable(x))
    peak_memory = torch.cuda.max_memory_allocated()
```

Listing 3.1: Memory Profiling of the GPU memory consumption

3-3-3 Logging

Since an external device was used to measure the energy consumption it was important to find a method to log the data coherently on two different platforms: both the watt meter and Python. The watt-meter logs the data every minute and saves it as a single CSV file on an SD card. After the experiments, the SD card is removed and read out using *Pandas*. Note

[5] https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

that before reading the CSV, the top lines of the description need to be removed from the CSV before processing.

To log which experiment corresponds to each time period and energy measurement, the Python *logging* library was used. This Application Programming Interface (API) allows each Python module to log to the same file. For each log entry the date (DD/MM/YYYY) and time (h:m:s) are recorded. Before any code has been run the log handler needs to be initialized, as long as no other exists. The logging level is set to INFO to provide relevant information without interfering with ERROR, WARNING, or FATAL logs. The file handler level is set using *fh.setLevel()*. More details of the code can be found in Listing 3.2.

```
import logging
logger = logging.getLogger('Layer_fin')
logger.setLevel(logging.INFO)

# Check if the logger already has a FileHandler
if not any(isinstance(handler, logging.FileHandler) for handler in logger.
    ↪ handlers):
    fh = logging.FileHandler('Kernel.log')
    fh.setLevel(logging.INFO)
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(
    ↪ message)s')
    fh.setFormatter(formatter)
    logger.addHandler(fh)
```

Listing 3.2: Logging setup for Layer_fin

Before the forward passes are run, the logger records the starting time, logging the experiment name (consisting of the important parameters S, T, W, d, M, and c) prefixed with 'start' and ending with '-ind' followed by the index of the current run (1-3). Since the watt meter logs every minute, the code is synchronized to start at the beginning of a minute using Python's *time* and *datetime* libraries. After the forward passes are completed, the logger records the end time, again logging the experiment name but now prefixed with 'end', as shown in Listing 3.3. Once all experiments are finished, each log file is converted to a CSV using the code given in Section B-1 Listing B.4.

```
. . .
model.train()
timers.sleep(60)
now = datetime.now()
sec_wait = 60 - now.second
timers.sleep(sec_wait)
start_training = perf_counter()
if decompose:
    logger.info(f"dec-start-outch{out_channels}-inch320-fact{factorization}-r{
    ↪ rank}-wh{img_w}-ind{ind}s")
else:
    logger.info(f"bas-start-outch{out_channels}-inch{in_channels}-wh{img_w}-
    ↪ ind{ind}s")
with torch.no_grad():
    with profile(activities=[ProfilerActivity.CPU], record_shapes=False,
    ↪ profile_memory=True) as prof:
        for _ in tqdm(range(m), desc="Forward Iterations"):
```

```
            output = model(Variable(x))
        if decompose:
            logger.info(f"dec-end-outch{out_channels}-inch320-fact{
    ↪ factorization}-r{rank}-wh{img_w}-ind{ind}s")
        else:
            logger.info(f"bas-end-outch{out_channels}-inch{in_channels}-wh{
    ↪ img_w}-ind{ind}s")
end_training = perf_counter()
training_time = start_training - end_training
key_averages = prof.key_averages()
peak_memory = sum([item.cpu_memory_usage for item in key_averages])
. . .
```

Listing 3.3: Code snippet of inference logging and memory profiling on CPU

3-4 Data processing

After all data was logged, it needed to be processed before it could be used. The various log files of the Python logger contained the different measurement periods for the different experiments. The watt meter logs files were saved per month in CSV files and consisted of the power measurements per minute. It was key to combine the information of these two log files into coherent results.

To separate the power measurements into different periods based on the Python logger files, the 'start' prefix assigned to each experiment was used. After locating the first 'start' in the log, the corresponding DateTime was collected and matched with the DateTime in the power measurements CSV. The end time was then found by moving one index down in the logger file and compared with the CSV. This process was repeated to extract all measurement periods from the power CSV. Once the individual measurement periods were identified, the different runs of the same experiment were combined using the 'ind' value, into a single measurement ID.

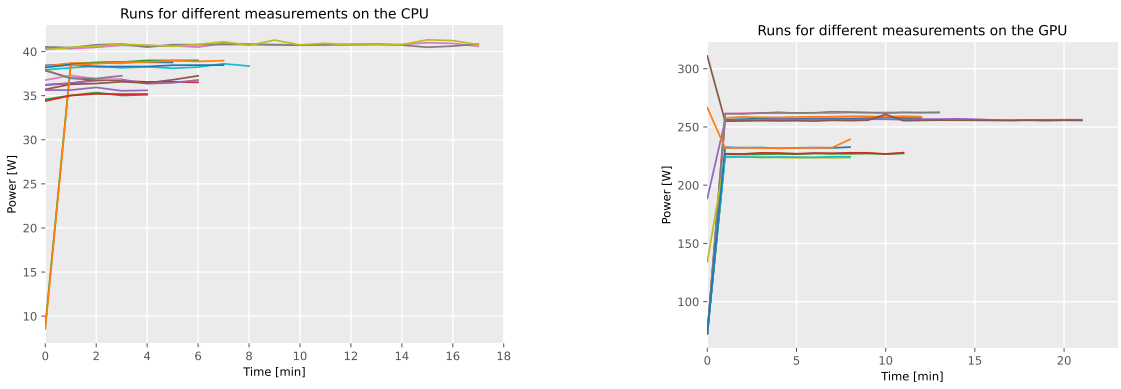


Figure 3-5: Several power measurement sequences for different experiments run both on the CPU and GPU.

Figure 3-5 shows different power measurements from various experiments. Occasionally, the measurements display a large outlier in the first minute. This may be due to the prior measurement ending close to the next minute, resulting in a shorter waiting time. Instead of starting from the idle consumption, some peak or dip in energy might still be present Therefore, the first measurement was treated as an outlier and removed. The duration of the first minute was kept to accurately reflect its influence on final energy consumption (E=power*time). The remaining measurements in the period were used and the median was taken to reflect the power over that time period, removing any other outliers and power grid fluctuations. Then the energy was calculated by multiplying the median with the total time period (including the first minute). Energy consumption was converted into kWh, a standard unit in research. For each experiment, run multiple times, the average energy consumption was calculated by taking the mean over all runs.

The full pseudo-code of the data processing is given in Listing B.5 in Section B-1.

3-5 Modelling the expected energy savings of decomposed convolutions

For this thesis, a simple energy complexity model was used, assuming that the two main contributors to total energy consumption are memory usage and MAC operations. Previous research suggests that memory operations typically consume more energy compared to MAC operations [72]. Modelling both the MAC and memory can give insight into the influence ratio between these variables on the energy saved.

To establish a baseline for comparison, a benchmark model was first developed using a simple linear regression with the MAC operations as the sole predictor. This benchmark reflects the traditional focus on the computational complexity of tensor decomposed convolutions established in Chapter 2, where energy efficiency is often discussed primarily in terms of MAC operations and reduced number of parameters. However, in the TensorlyTorch implementation used in this thesis, CP, Tucker, and TT decomposed layers result in the same number of parameters, despite leading to different energy reductions. Therefore, the number of parameters was not chosen as a representative variable for the benchmark. By establishing the benchmark, it is possible to quantify the basic predictive power of computational complexity alone and use it as a reference point to evaluate the improvements offered by more complex models that incorporate memory usage.

Following up on that another aspect is considered: whether the memory is measured or calculated. The sizes of the intermediate results of the separate decomposed convolutions can be calculated based on the analysis of the chosen CNN hyperparameters and decomposition parameters following Subsection 2-4-2. It would be convenient to predict the energy saved before actually measuring anything, i.e. measuring memory usage using a profiling tool. Therefore, it was decided to not only model the energy savings based on the number of reduced MAC operations and the measured memory usage but also create models based on the in-advance calculated memory from the intermediate results. The results of these models could indicate whether the calculated memory could make valid predictions on the energy saved by decomposing before actual implementation.

Both multivariate linear and polynomial regression models were employed to analyze the relationships between the dependent variable (energy saved) and the independent variables (calculated memory, measured memory, and MAC operations). It was chosen to fit a multivariate linear regression model as a simple approach to understanding the linear relationships in the data. Additionally, a polynomial regression model was applied to capture potential non-linear patterns, providing a more complex analysis. This additional complexity might particularly be relevant in the context of complex memory management scenarios, such as those encountered in GPU platforms, where non-linear relationships might better explain the data.

For each hardware implementation (CPU and GPU), a model was fitted since it is expected that different models are necessary, especially for the memory allocations that can differ between the different hardware, which was derived from literature is Subsection 2-2-1.

In summary, eight different models were fitted—four for each type of hardware—either linear or polynomial, with the measured or calculated memory. More details are provided below, starting with processing the data.

3-5-1 Data augmentation before fitting

The data used for these models was obtained from measurements taken during the single-layer convolution experiments, as detailed in earlier sections. The dependent variable in these models is the energy savings (ΔE) and the independent variables are the reduction in MAC operations (ΔMAC) and the additional memory required (Δmemory). The memory variable can be either calculated or measured, depending on the model variant being used. These variables represent the differences in operations and memory before and after decomposition compared to the original baseline convolution.

Given the difference in the order of magnitude between the energy saved, the MAC operations and the memory usage, it was necessary to standardize the independent variables. Standardization ensures that the coefficients derived from the regression models are on the same scale, allowing for a more meaningful comparison of their relative influence on energy savings. The *StandardScalar* from the *scikit-learn* library was used for this standardization process.

After standardization, the datasets were split into training and test sets based on an 80%-20% ratio, so that the models could be evaluated on unseen data. The total dataset contained 200 measurements, resulting in a training set of 180 measurements and a test set of 20 measurements. The energy measurements obtained from the watt meter were processed according to the procedure described in Section 3-4, and no further outlier removal was performed for the regressions.

3-5-2 Fitting several regression models

Building upon the established benchmark model, more complex regression models were developed to explore the multivariate relationships between energy savings, MAC operations, and memory usage. Both linear and polynomial regression models were fitted, exploring different complexities to capture potential linear and non-linear patterns.

To find these relationships, Linear Regression was used with either linear or polynomial features, under the assumptions of linearity, normality of the residuals, homoscedasticity, no multicollinearity and independence of the measurements. For both hardware types and different memory conditions, these properties were investigated, and a detailed description is provided in Section B-3 in the appendix. The data distribution showed some deviation from a normal distribution, although it largely resembled one. Additionally, there were indications of heteroscedasticity.

Linear regression model
The multivariate linear regression model incorporates both MAC operations and memory usage (calculated or measured) as independent variables. The model assumes a linear relationship between the predictors and the energy saving, as presented by

$$\Delta E = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon, \tag{3-21}$$

where β_0 is the intercept, β_1, β_2 are the least squares estimators and x_1 and x_2 are the independent variables Δmemory and ΔMAC respectively, with error ϵ, explaining any variance not explained by the model.

The linear regression was performed using the *Ordinary Least Squares (OLS)* method from the *statsmodels* library. It was chosen to use the *statsmodels* linear regression over the other commonly used *scikit-learn* because it allows for better statistical comparison between models through the available statistical model summary. The slight indication of heteroscedasticity was addressed by using robust Standard Errors (SEs) (*cov='HC1'*) to mitigate a biased SE. This adjustment results in more reliable p-values, test statistics, and confidence intervals [164].

Polynomial regression model
To capture more complex relationships that might exist between the variables, a second-order polynomial was applied. This model includes squared and interaction terms to account for potential non-linear effects, given by

$$\Delta E = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 + \beta_5 x_1 x_2 + \epsilon, \tag{3-22}$$

where β_0 is the intercept, β_1, β_2, β_3, β_4, and β_5 are the least squares estimators, and x_1 and x_2 are the independent variables Δmemory and ΔMAC respectively, with some additional noise ϵ. Now additional squared (x_1^2, x_2^2) and interaction $(x_1 x_2)$ terms can account for the potential non-linear relations.

Similar to the linear regression, the *OLS* from *statsmodels* was used, however, now polynomial features were included using the *scikit-learn* PolynomialFeatures. Using PolynomialFeatures, the intercept is added by default so it is important to note not to do this (include_bias=False) since the intercept is added through *statsmodels*. For the polynomial regression, again robust SEs are used.

3-5-3 Evaluation metrics

To compare and evaluate the fitted models several evaluation metrics are used to assess the performance and robustness. Below an overview is given of the different metrics.

Root-mean-square error (RMSE)
To measure the performance of the models, in terms of accuracy, the RMSE was used. An advantage of the RMSE is that this metric has the same unit as the dependent variable, allowing for direct comparison. A lower RMSE indicates that the predicted values are closer to the actual values, evaluating the model accuracy.

R-squared and Adjusted R-squared
The R-squared metric is used to see the proportion of the variance in the dependent variables that can be explained by the independent variables. When the R-squared value is closer to one, it suggests that the model can account for a significant part of the variance. When additional predictors are added, the R-squared value will either remain constant or will increase, even when the new predictor does not add extra predicting power. Comparing the benchmark (single independent variable) with the other models (multivariate) solely based on the R-squared value might not give a representable result. That is why, in addition to the R-squared, also the adjusted R-squared is used, which only increases when the additional dependent variable adds additional prediction power. Using the adjusted R-squared, it is thus possible to evaluate whether incorporating the memory usage and polynomial terms leads to meaningful improvements.

Model robustness
As mentioned before, a train-test split will be used to show the results of the model on unseen data. In that way, the robustness of the model can be evaluated on reliability and generalizability.

Chapter 4

Experiments

In this section, all experiments will be discussed and results will be presented. The analysis will help to derive conclusions presented in Chapter 5. Initially, the isolated experiments for both Central Processing Unit (CPU), Graphics Processing Unit (GPU) and the ResNet18 experiments will be discussed addressing SQ1, SQ2 and SQ3, followed by the regression models, addressing SQ4.

The analysis will focus on the assumed contributors to energy consumption: Multiply-Accumulate operation (MAC) operations and memory consumption, both measured and calculated. Given the interest in reducing energy consumption, the comparison between parameters and methods will be based on the difference in energy saved $\Delta E = E_{baseline} - E_{decomposed}$, where negative numbers indicate additional energy consumption and positive numbers indicate energy savings. As a reference, all baseline energy consumptions are presented in Appendix C Section C-1. In addition, it was also of interest whether the calculated memory was a good indicator of the real additional memory, enabling prediction before actual measurements are needed. The differences between the measured and calculated memory will also be outlined.

For each isolated experiment, random input features are generated in the sizes needed for the experiments, sampled from a normal distribution with the mean and standard deviation of the CIFAR10 dataset. Each batch is sampled once and used for all experiments, to allow for fair comparison. For experiments run on both GPU and CPU, input features are reformatted to float32 from float64, optimized for GPU and commonly used in Convolutional Neural Networks (CNNs), using *x.float()*. This allows for more efficient processing. The code explicitly moves the model and variables to either CPU or GPU using *torch.device("cuda" if torch.cuda.is_available() else "cpu")*. For running the inference on the pre-trained ResNet18 the CIFAR10 train dataset was used as input, consisting of 50000 images with a batch size of 128.

4-1 Influence of input channels

This section elaborates on the results of experiment one where the effect of the input channels combined with the different tensor decompositions and compression ratios is investigated. Table 4-1 shows the different hyperparameters used for this single convolution experiment.

Table 4-1: Control, independent and dependent variables of experiment 1.

Control Variables	Independent Variables	Dependent Variables
$T = 512$	$S \in \{192, 256, 320, 384\}$	Energy consumption
$W \times H = 4 \times 4$	$\mathrm{M} \in \{\mathrm{cp}, \mathrm{tucker}, \mathrm{tt}, \mathrm{nd}\}$	Memory
$d = 3$	$c \in \{0.1, 0.25, 0.5, 0.75, 0.9\}$	MAC
epochs $= 7.0 \times 10^3$(CPU), 5.8×10^5(GPU)	Hardware $\in \{\mathrm{CPU}, \mathrm{GPU}\}$	

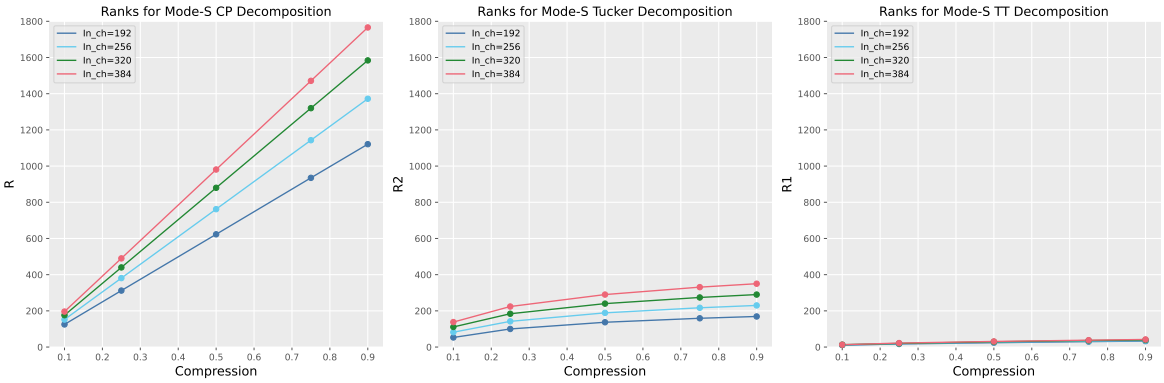


Figure 4-1: The mode-S ranks of all three tensor decomposition methods across all compression ratios.

Figure 4-1, show the different ranks, for the different number of input channels. It shows that with the increase of S and decrease of compression ratio, the Parallel Factors (PARAFAC) Canonical Decomposition (CANDECOMP) (CP) rank is significantly higher than the mode-S specific Tucker and Tensor Train (TT) ranks. It is also visible, that with less compression, the CP ranks are growing more rapidly than the mode-specific ranks. The rank has a great influence on the reduction of MAC, but especially on the intermediate tensors that need to be stored. Figure 4-1 might indicate that the expected energy saved will be less for the CP compared to Tucker and TT.

4-1-1 CPU energy savings

Figure 4-2 illustrates the energy reduction achieved by different tensor decomposition methods, varying compression rates, and the number of input channels for the CPU using a line chart. On the y-axis, the saved energy (positive values) and added energy (negative values) in kWh are displayed, while the x-axis presents the different compression rates. The standard deviation of the measurements is shown by the hue.

For CP decomposition, higher compression results in higher energy savings. The most energy is saved, $E = 0.008$ kWh (67.23% of baseline E), for $S = 384$ at $c = 0.1$. At compression of $c = 0.25$ energy savings cross zero, indicating that from there additional energy is consumed. With less compression, all configurations show negative energy savings. Notably at around $c = 0.5$, a transition takes place where configurations with a larger number of input channels, begin to consume more energy compared to those with fewer channels.

The Tucker decomposition method shows similar patterns. Higher compression ratios lead to increased energy savings, with a maximum of $\Delta E = 0.0097$ kWh (81.51% of baseline E). Between $c = 0.5$ and $c = 0.75$, a similar transition occurs, where a larger amount of input channels consume more energy than those with fewer channels, also crossing the zero-energy savings line.

Among the three decomposition methods, the TT shows the most consistent energy savings, remaining positive for all input channel configurations, with a maximum energy saving of $\Delta E = 0.0099$ kWh (83.19% of baseline E). For all compression ratios, the higher the number of input channels shows slight variations in the energy savings.

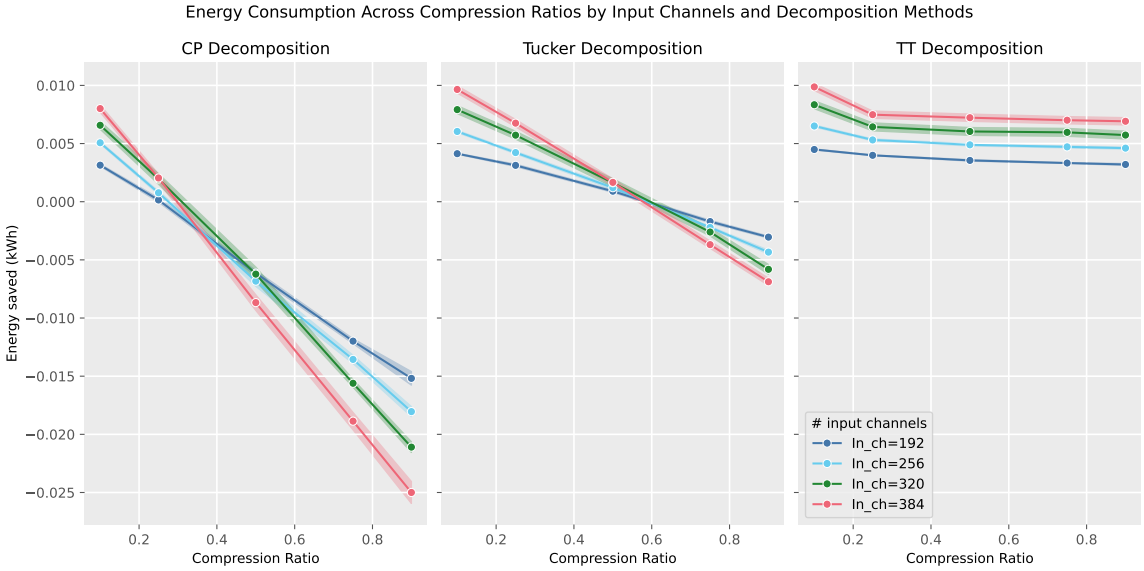


Figure 4-2: The energy saved for different methods, CP, Tucker and TT, for a different number of input channels, given by different colours, run on a CPU. The hue presents the standard deviation of the measurements.

In Figure 4-3, CP decomposition shows visible variation in MAC reduction across different compression ratios, particularly in lower compression ($c \geq 0.5$), where an increase in MAC operations is visible. Both additional measured and calculated memory show an increase with increasing input channels. This pattern matches the pattern in Figure 4-2, where the energy savings drop to zero for compression rates $c > 0.25$. The spike in memory usage and MAC operations at lower compressions likely contributes to this drop in energy savings.

Similarly, Tucker has lower MAC reduction at lower compression ratios ($c > 0.5$) for all input channels. Combined with the increase of memory, particularly at lower compression with more input channels, this can explain the negative energy savings observed in Figure 4-2.

In contrast, TT shows a substantial MAC reduction and low memory usage, consistent for all compression rates and input channels. This consistency matches well with the consistent energy savings and minimal differences in energy consumption between the increasing number of input channels.

Figure 4-3 shows that the calculated memory has similar patterns as the measured memory for all measurements and input channels. However, a slight difference can be found in the Tucker decomposition, where the calculated memory shows more distinct spikes of memory for lower compression ratios $c > 0.5$.

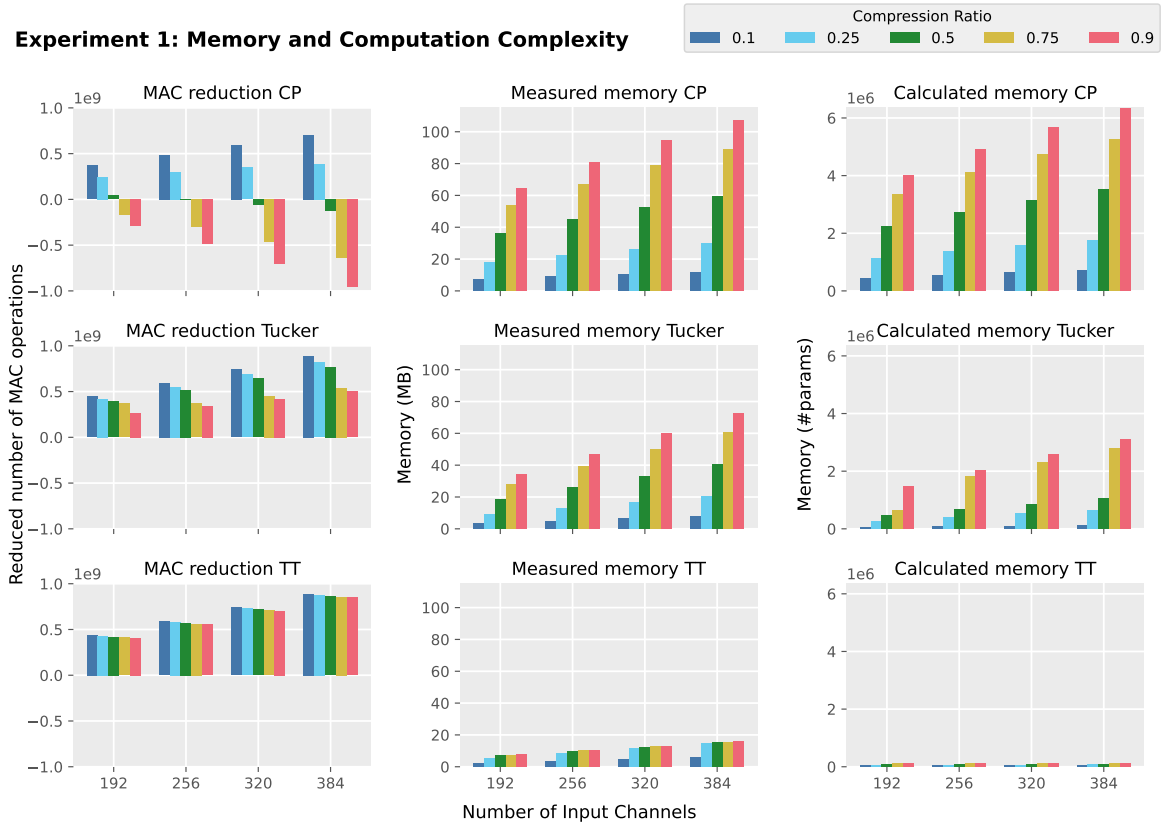


Figure 4-3: The MAC operations and memory, both calculated and measured on the CPU, for the different methods and input channels. On the left the MAC operations reduced by decomposing, in the middle the measured memory and on the right the calculated memory based on the intermediate results.

4-1-2 GPU energy savings

Figure 4-4 illustrates the energy reduction achieved by different tensor decomposition methods, varying compression rates, and the number of input channels for the GPU using a line chart. On the y-axis, the saved energy (positive values) and added energy (negative values) in kWh are displayed, while the x-axis presents the different compression rates. The standard deviation of the measurements is shown by the hue.

In contrast to the CPU energy savings, the CP decomposition has no energy savings for the decomposed layers and even has a large amount of additional energy, with the least additional energy of $\Delta E = -0.030$ kWh (575% of baseline E). In addition, a different pattern is visible where the large number of input channels $S = 384$ performs less for most compression. When the compression is $c = 0.5$ it can be seen that the middle number of input channels $S = 256$ and $S = 192$ perform even worse but then spike up at $c = 0.75$ and $c = 0.9$. For the least amount of compression, a similar result to the CPU can be seen where the lowest number of input channels adds less energy than the largest number of input channels.

The energy savings from the Tucker decomposition are all below zero, with the least additional energy of $\Delta E = -0.011$ kWh (92.44% of baseline E), and show consistent overall compression ratios. Figure 4-4 indicates that with increasing input channels increased added energy is consumed. There is no longer a transition as seen in the CPU results. In contrast, the TT decomposition shows some, but very subtle, energy savings, with the largest $\Delta E = 0.006$ kWh (50.42% of baseline E), for the largest number of input channels $S = 384$ for high compressions $c \leq 0.25$.

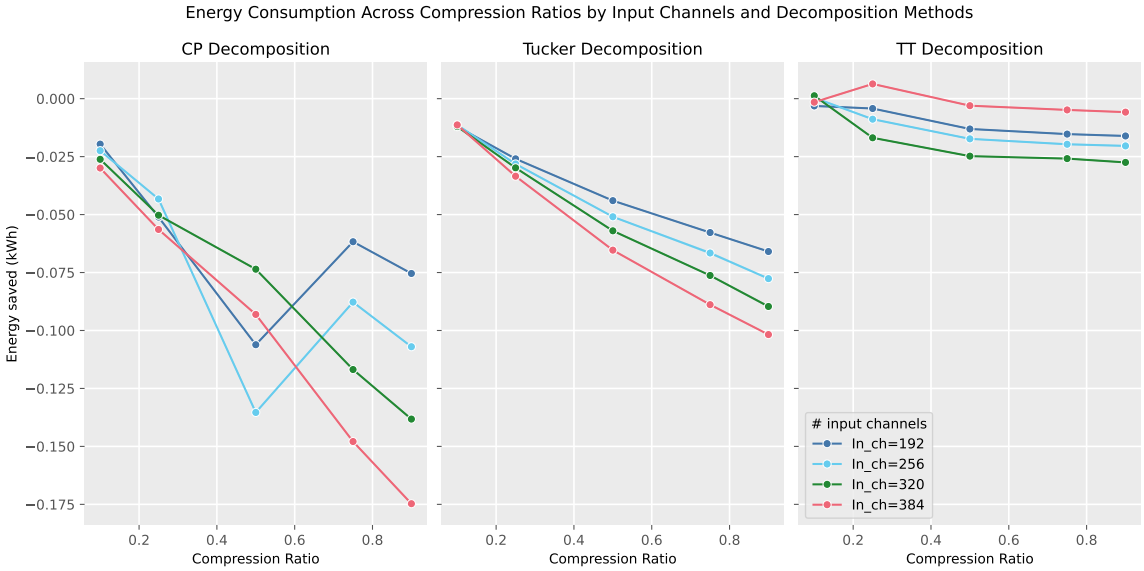


Figure 4-4: The energy saved for different methods, CP, Tucker and TT, for a different number of input channels, given by different colours, run on a GPU. The hue presents the standard deviation of the measurements.

The calculated memory and reduced MAC operations shown in Figure 4-3 also apply to the GPU and cannot explain the observed fluctuation and differences in the energy savings. Figure 4-5 shows the measured memory on the GPU, which gives more insight into the memory usage on the GPU. Especially since it is known from the literature that specialized GPU consist of more complex memory structures, the memory usage may be more influential, whereas the MAC operations may not be due to efficient parallel computing [165].

Looking at CP memory usage, it can be seen that for $S = 192$ and $S = 256$ a drop is visible at compression $c = 0.75$. This matches the spikes in energy savings in the CP in Figure 4-4. For Tucker it can be seen that the memory usage increases with increased input size and

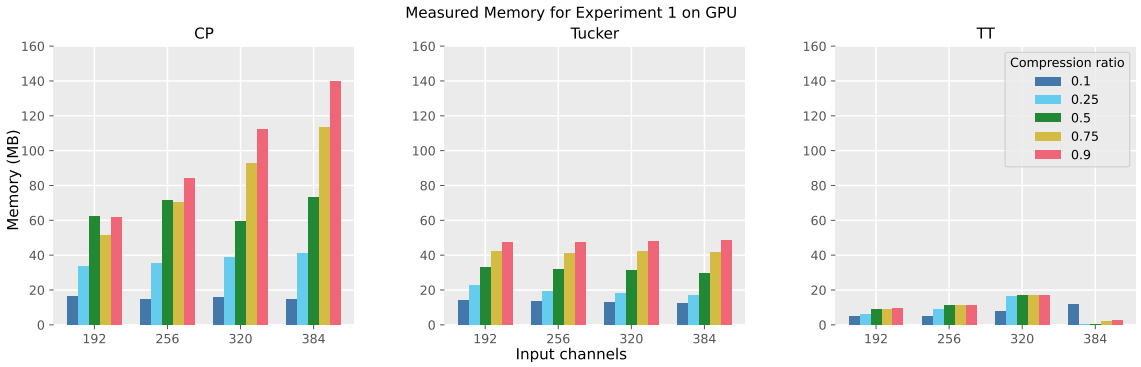


Figure 4-5: The measured memory usage on the GPU, for the different methods and input channels.

compression ratio, however, it doesn't increase much, which could match the small variation in Figure 4-4. The energy savings of the Tucker and CP are below zero and this might be so due to too much added memory. Looking at TT the memory added is lower, especially for a large number of input channels and compression 0.25 which shows a similar pattern to Figure 4-4 for which the energy savings of $S = 384$ are the highest and the other number of input channels show added memory similarly to Tucker and TT.

Figure 4-5 shows that for the memory usage of the GPU, the calculated intermediate results from Figure 4-3, might not be a good predictor for the actual memory consumption.

4-1-3 Key findings

This experiment was designed to look into whether the number of input channels of a layer had a noticeable impact on the inference energy savings by decomposing the layer using either CP, Tucker or TT decomposition across different compression levels. Below the key findings of this experiment are outlined.

- MAC operations and memory: The results suggest that indeed the reduced MAC operations and the additional memory usage play an important role in energy savings. For both the CPU and GPU, it was seen that the memory and MAC operations showed patterns that could explain certain changes in energy consumption. In addition, especially for GPU, it could seen that the additional memory may be the most prominent factor and that the calculated memory, especially for CPU, shows similar patterns to the measured memory, which might make it a good predictor.
- Number of input channels: The additional memory and reduced MAC operations are dependent on the number of input channels. For high compression, the larger the number of input channels the more benefit is gained. However, for lower compression levels $c \geq 0.5$, the smaller number of input channels performs better. So this suggests that the number of input channels is important when decomposing a convolutional layer and there is a trade-off present.
- Decomposition methods: TT shows the most potential for decomposing as it shows consistency in overall compression levels and the number of input channels. For the

higher compression levels ($c = 0.1, c = 0.25$), all three methods showed similar energy savings, however for both Tucker and CP the lower compression levels $c \geq 0.5$ showed additional energy and especially the CP decomposition performed worse.

- Hardware: The energy savings of the CPU were higher than of the GPU for all input channels and the GPU mostly even presented additional energy needed after decomposing. The differences presented in the reduced MAC operations and memory might indicate similar patters and might align with the energy savings.

4-2 Influence of output channels

This section elaborates on the results of experiment two where the effect of the number of output channels combined with the different tensor decompositions and compression ratios is investigated. Table 4-2 shows the variables used in this experiment. First, the results of the CPU experiment will be outlined, after which a comparison will be made with the GPU experiment.

Table 4-2: Control, independent and dependent variables of experiment 2.

Control Variables	Independent Variables	Dependent Variables
$S = 192$	$T \in \{192, 256, 320, 384\}$	Energy consumption
$W \times H = 4 \times 4$	M $\in \{\text{cp}, \text{tucker}, \text{tt}, \text{nd}\}$	Memory
$d = 3$	$c \in \{0.1, 0.25, 0.5, 0.75, 0.9\}$	MAC
epochs $= 9.0 \times 10^3$(CPU), 5.8×10^5 (GPU)	Hardware $\in \{\text{CPU}, \text{GPU}\}$	

Figure 4-6, show the different ranks, for the different number of output channels. Similar to the mode-S ranks, the difference between the CP rank and mode T specific Tucker and TT rank is visible. The ranks are smaller than the mode-S ranks. Again, it would be expected that the balance between MAC reduction and memory allocation will result in the CP decomposed convolution saving less energy than the other methods, based on these differences in the ranks.

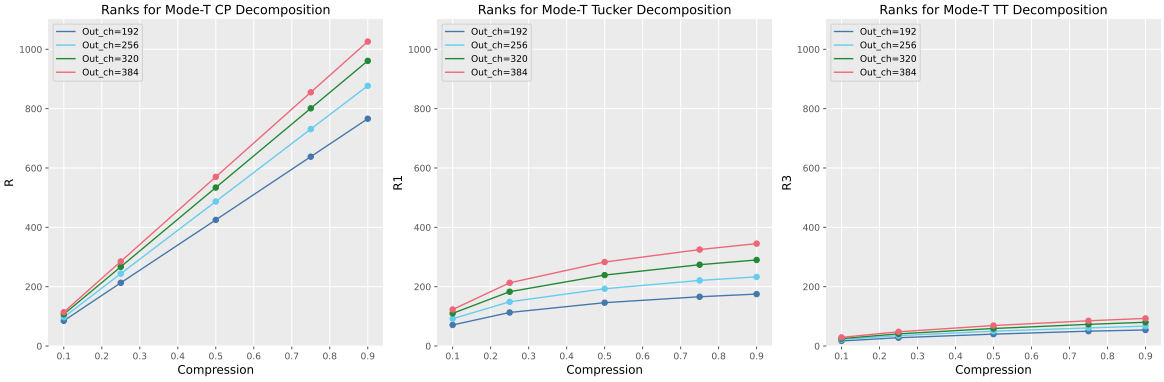


Figure 4-6: The mode-T ranks of all three tensor decomposition methods across all compression ratios.

4-2-1 CPU energy savings

Figure 4-7 illustrates the energy reduction achieved by different tensor decomposition methods, varying compression rates, and the number of output channels for the CPU using a line chart. On the y-axis, the saved energy (positive values) and added energy (negative values) in kWh are displayed, while the x-axis presents the different compression rates. The standard deviation of the measurements is shown by the hue.

Generally, the CP decomposition shows an increase in energy consumption rather than savings, except at a high compression ratio of $c = 0.1$, across all output channels, with a maximum of $\Delta E = 0.0026$ kWh (51.38% of baseline E). Similar to the pattern seen in the number of input channels, there is a noticeable transition between $c = 0.25$ and $c = 0.5$, where the energy added for larger output channels $T = 384$ grows stronger than that of the smaller output channels. This suggests that compression in convolution layers consisting of a large number of output channels is only beneficial at high compression levels. Additionally, the standard deviation for the smaller compressions (c=0.75 and c=0.9) appears to be larger, indicating more variability in the energy measurements.

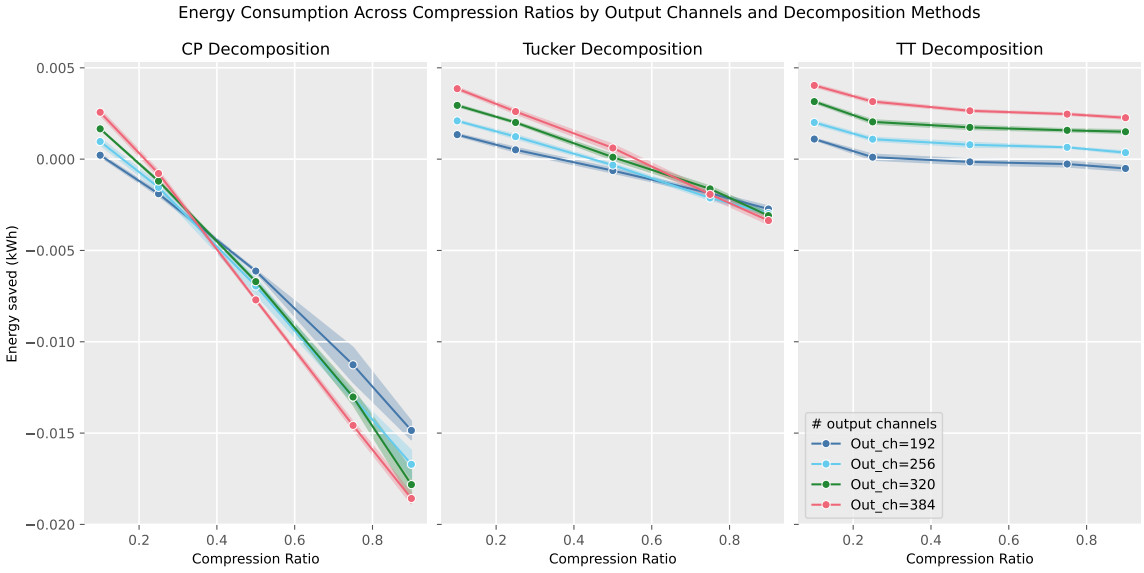


Figure 4-7: The energy saved for different methods, CP, Tucker and TT, for a different number of output channels, given by different colours, run on a CPU. The hue presents the standard deviation of the measurements.

For the Tucker decomposition, layers with a larger number of output channels ($T = 384$) induce more energy savings than those with smaller output channels until $c = 0.75$, with the largest savings of $\Delta E = 0.0039$ kWh (77% of baseline E). At this compression level, a transition occurs, and all output channels show similar energy consumption. At high compression levels $c < 0.5$, the Tucker decomposition shows energy savings, however beyond this point the benefit of decomposing is reduced, resulting in additional energy consumption.

The TT decomposition shows consistent energy savings for all compression ratios, increasing with the number of output channels, with a maximum energy savings of $\Delta E = 0.004$ kWh (79.05% of baseline E). However, the smallest number of output channels $T = 192$, lies on

the zero energy savings line, suggesting that decomposing layers with fewer output channels might result in additional energy consumption.

Overall, the TT decomposition shows the most consistent energy savings and performs best at low compression levels. The Tucker decomposition presents similar energy savings at higher compression levels, but at lower compression levels it results in additional energy consumption. Lastly, the CP decomposition performs the worst even at high compression levels. This trend is consistent across all number of output channels

Figure 4-8 shows the MAC operations reduced by decomposing and the additional memory both calculated and measured. In Figure 4-8 CP decomposition shows visible variation in MAC reduction across different compression ratios. In particular, for each number of output channels, there is a clear transition between MAC reductions ($\Delta E > 0$) and additional MAC operations ($\Delta E < 0$) between compression ratios $c = 0.25$ and $c = 0.5$. Differently from all other compression ratios, less MAC operations are added with $c = 0.5$ with increased T. In combination with, the consistently increasing memory with a higher number of output channels, both calculated and measured, it might explain the transition shown in Figure 4-7. It suggests that for higher output channels, more memory is required, especially in the lower compression levels, which negates the benefit from the MAC reductions.

For the Tucker decomposition, Figure 4-8 shows with more output channels $T = 384$, the most MAC reduction occurs, especially at high compression levels. However, the differences between high and low levels of compression become more noticeable for $T = 384$. Additionally, there is a consistent increase in memory usage, both calculated and measured, across all compression rates, with the highest usage at a higher number of output channels. This might explain the transition observed in Figure 4-7, where lower MAC reduction at low compression levels, in combination with increased memory, might balance out and result in additional energy consumption.

Figure 4-7 showed consistent positive energy savings across all compression rates and increased savings with a higher number of output channels. This pattern is also visible in Figure 4-8, where the TT decomposition demonstrates minimal memory usage and similar MAC reduction compared to the other methods, with little difference across different compression ratios except for $c = 0.1$. The MAC reduction increases significantly with the number of output channels, whereas memory usage shows minimal growth, explaining the increased savings for a higher number of output channels.

Similar to the previous experiment, the calculated memory seems to be a good predictor for the actual memory usage, for the CPU. However, also in Figure 4-8 it can be seen that the calculated memory shows more peaks in the memory for the Tucker decomposition.

4-2-2 GPU energy savings

Figure 4-9 illustrates the energy reduction achieved by different tensor decomposition methods, varying compression rates, and the number of output channels for the GPU using a line chart. On the y-axis, the saved energy (positive values) and added energy (negative values) in kWh are displayed, while the x-axis presents the different compression rates. The standard deviation of the measurements is shown by the hue. The measured energy savings differ from those on the CPU, demonstrating specific patterns. Notably, all energy savings are negative for all three methods, indicating additional energy consumption.

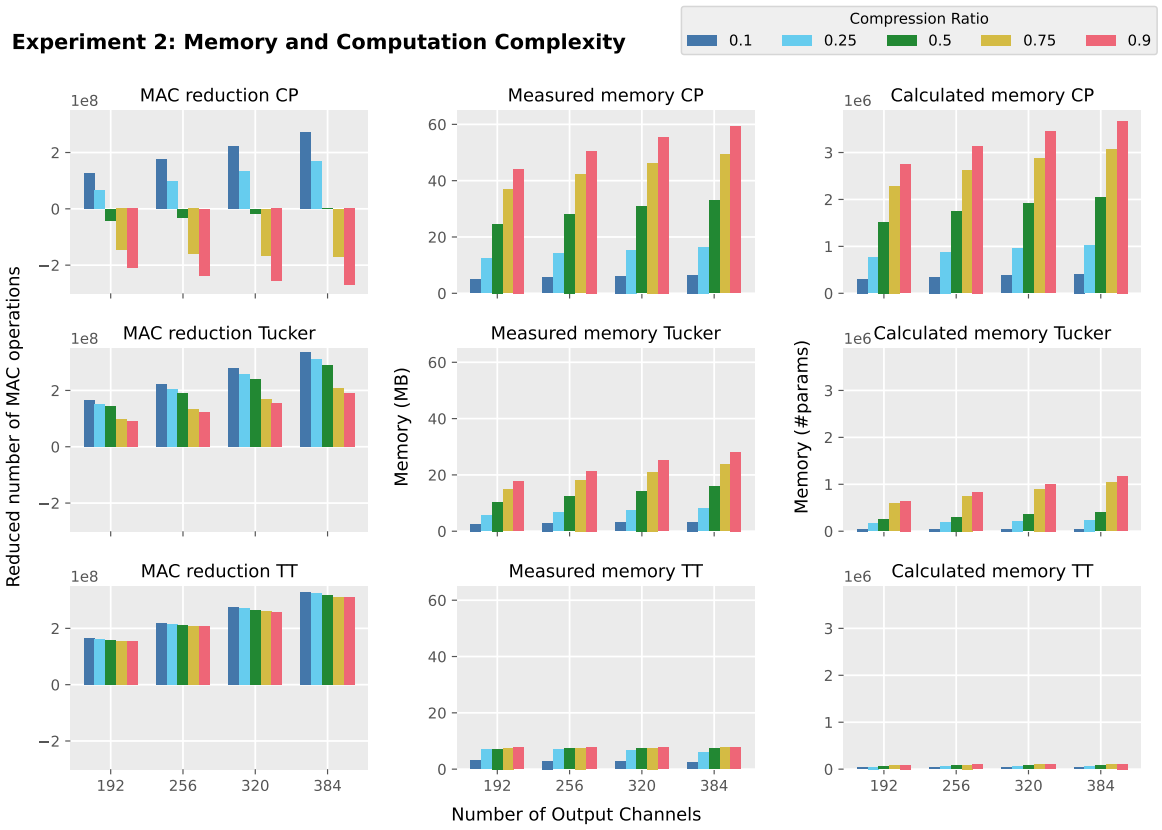


Figure 4-8: The MAC operations and memory, both calculated and measured on the CPU, for the different methods and input channels. On the left the MAC operations reduced by decomposing, in the middle the measured memory and on the right the calculated memory based on the intermediate results.

Looking at Figure 4-9 for the CP it can be seen that generally the highest $T = 384$ and lowest $T = 192$ number of output channels outperform the others, except for low compression level $c = 0.9$ where these number of output channels show a jump. Over all the added energy is present at $T = 320$ of $\Delta E = -0.0025$ kWh (60.38% of baseline E).

For both the Tucker and TT decompositions, the additional energy consumption is much lower compared to CP across all methods and compression levels, with a minimum of $\Delta E = -0.0018$ kWh (43.48% of baseline E). TT decomposition even shows consistent results across different compression ratios and several output channels, suggesting a minimal influence of these parameters on energy consumption. Tucker decomposition shows a similar pattern, with minimal additional energy of $\Delta E = -0.0007$ kWh (16.91% of baseline E), but there is a clear separation between the middle number of output channels ($T = 256$, $T = 320$) and the extreme output channels ($T = 192$, $T = 384$).

Again, the calculated memory and MAC operations from Figure 4-8 are the same for the GPU experiments. However, the additional measured memory of the GPU implementation, as shown in Figure 4-10, differs visibly from the calculated memory. This discrepancy suggests that the calculated memory may not be a reliable predictor for actual GPU memory usage.

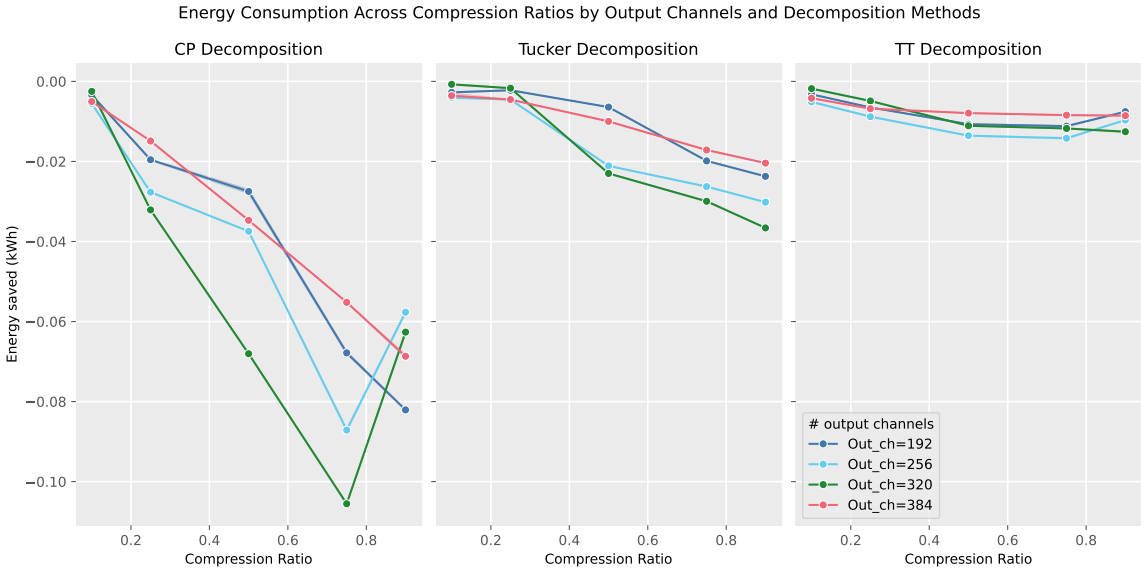


Figure 4-9: The energy saved for different methods, CP, Tucker and TT, for a different number of input channels, given by different colours, run on a GPU. The hue presents the standard deviation of the measurements

For the CP decomposition, it can be seen that for each of the output channels, the memory increases with the compression rates, except for $T = 320$. Here, there is a peak memory at compression $c = 0.75$, which can be lead back to the additional energy consumption in Figure 4-9. Also, the increase in memory between compression level $c = 0.75$ and $c = 0.9$ for $T = 256$ is less prominent than for $T = 192$ and $T = 384$. In combination with more reduction of MAC operations for these output channels, it might be the reason that between $c = 0.75$ and $c = 0.9$ spikes up above the other output channels.

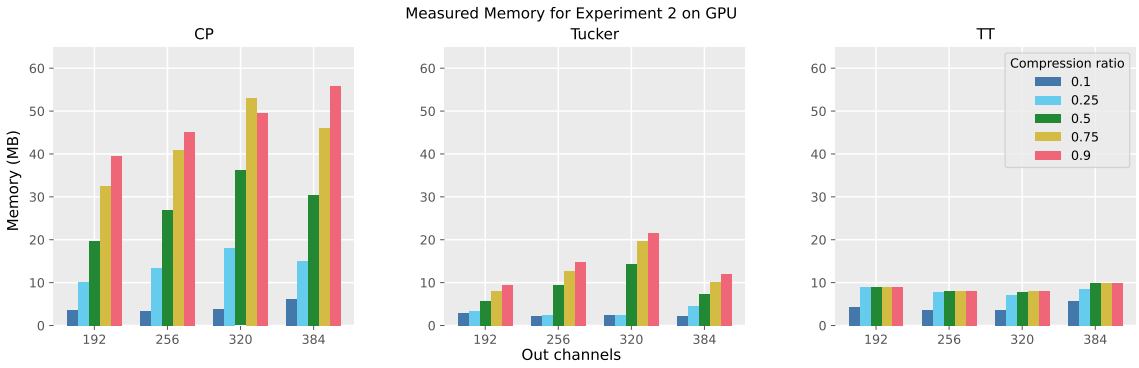


Figure 4-10: The measured memory usage on the GPU, for the different methods and output channels.

For Tucker, the memory usage shown in Figure 4-10, indicates that both $T = 384$ and $T = 192$ have similar memory patterns with a minimal increase between them. In contrast, $T = 256$ and 320 show more increased memory at all compression ratios and between each other. This pattern resembles the one observed in Figure 4-9, though more differentiation would be

expected between $T = 256$ and $T = 320$ based on the memory. The similarity between the energy savings may be attributed to the difference in MAC reduction, negating the memory influence.

4-2-3 Key findings

This experiment was designed to look into whether the number of output channels of a layer had a prominent impact on the inference energy savings by decomposing the layer using either CP, Tucker or TT decomposition across different compression levels. Below the key findings of this experiment are outlined.

- MAC operations and memory: Similar to experiment 1, it was found that the reduction in MAC operations and additional memory, could explain the patterns seen in the energy savings. Again, the calculated memory seems to show the same patterns as the measured memory, except for the GPU. With a high compression ratio, the reduction of the MAC is optimal, with less additional memory. This is where the most benefit can be found.
- Number of output channels: It can be seen that for all three decompositions, the layers with the highest number of output channels show the most benefit. However, for lower compression, this flips around and the larger layers perform the worst.
- Decomposition methods: In this experiment again TT has the most consistent energy savings. Tucker decomposition shows similar energy savings for high compression ratios after which it decreases. However, CP performs worse on all compression ratios and the number of output channels.
- Hardware: Similar to experiment 2, there is a difference in energy savings between CPU and GPU. The GPU measurements show no energy savings with a lower amount of memory and the same MAC reduction. This might suggest that the memory plays a more prominent role in the energy consumption of the GPU.

4-3 Influence of the feature size

This section will give an outline of the found results, comparing the energy savings over different feature sizes, methods and compression ratios. All variables are presented in Table 4-3.

4-3-1 CPU energy savings

Figure 4-11 illustrates the energy reduction achieved by different tensor decomposition methods, varying compression rates, and feature sizes for the CPU using a line chart. On the y-axis, the saved energy (positive values) and added energy (negative values) in kWh are displayed, while the x-axis presents the different compression rates. The standard deviation of the measurements is shown by the hue.

Table 4-3: Control, independent and dependent variables of experiment 3.

Control Variables	Independent Variables	Dependent Variables
$S = 448$	$W, H \in \{2, 4, 6, 8\}$	Energy consumption
$T = 512$	$\mathrm{M} \in \{\mathrm{cp}, \mathrm{tucker}, \mathrm{tt}, \mathrm{nd}\}$	Memory
$d = 3$	$c \in \{0.1, 0.25, 0.5, 0.75, 0.9\}$	MAC
epochs $= 9.5 \times 10^3$(CPU), 3.3×10^5 (GPU)	Hardware $\in \{\mathrm{CPU}, \mathrm{GPU}\}$	

Starting with the CP decomposition, a big variety in energy savings is shown. As can be seen for high level of compression $c = 0.1$, all feature sizes give energy savings, where most energy is saved $\Delta E = 0.035$ kWh (62.28% of baseline E), using the largest feature sizes. Then with lower compression levels $c \geq 0.25$, the energy savings become negative indicating additional energy consumption. Especially for larger feature size (8×8) the additional energy grows extensively. For the smallest feature size (2×2), there is no energy gained but also not much energy added. This suggests that decomposing large feature sizes gives the most benefit only for large compression levels.

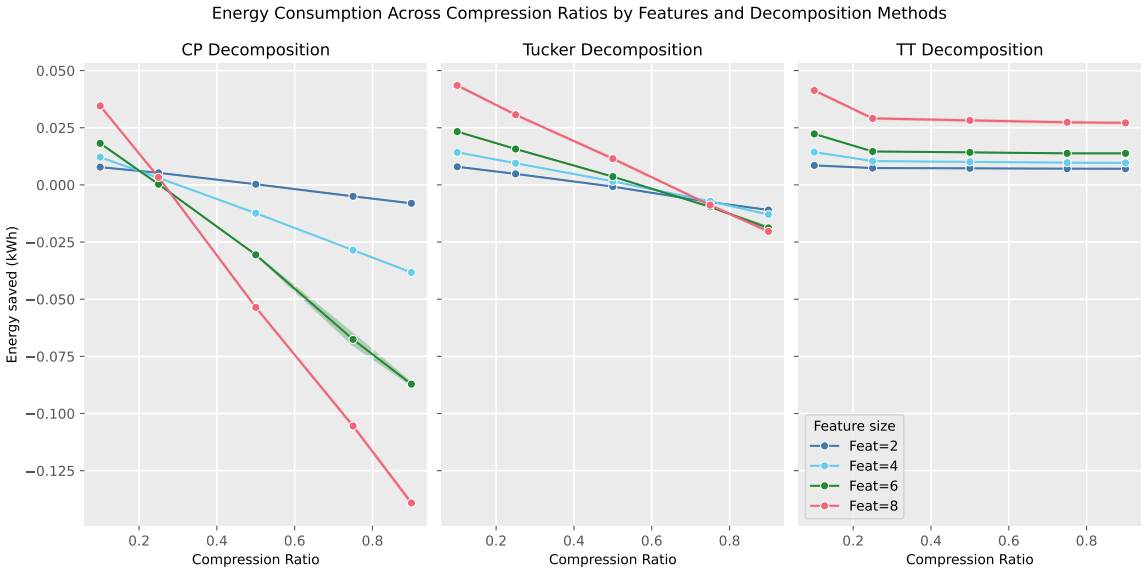


Figure 4-11: The energy saved for different methods, CP, Tucker and TT, for different feature sizes, given by different colours, run on a CPU. The hue presents the standard deviation of the measurements.

For the Tucker decomposition, the energy savings are positive for a larger set of compression levels $c \leq 0.5$, with a maximum of $\Delta E = 0.044$ kWh (78.29% of baseline E). Similar to the CP decomposition, the larger the feature size the more energy is saved, until a transition point between $c = 0.5$ and $c = 0.75$. For the low compression rates, the smaller feature sizes seem to add less energy than the larger ones.

Similar to the previous experiments, the TT decomposition shows consistent energy savings for all feature sizes across all compression levels, with a maximum of $\Delta E = 0.041$ kWh

(72.95% of baseline E). Here the larger the feature size the larger the energy savings. There is a small decrease visible when compressing less than $c = 0.1$.

Overall, Figure 4-11 shows that again the TT shows the most benefits from decomposing overall feature sizes. Following that the Tucker shows energy savings for all feature sizes with sufficient compression levels $c \leq 0.5$. In contrast, the CP shows the least potential for using tensor decompositions to save energy consumption. Only for a high compression level $c = 0.1$ energy savings are present.

Looking at Figure 4-12, the general differences between feature sizes are more prominent than for the other parameters. Also, the MAC reduction is minimal for all methods for the small 2×2 feature size.

Looking at CP, a similar pattern is visible where there is a reduction in MAC operations up to a certain compression level ($c = 0.5$), after which additional MAC operations occur. This is particularly noticeable with larger feature sizes, where the increase in MAC operations is significant. This may explain the observed transition to increased energy consumption in Figure 4-11 after $c = 0.25$. Furthermore, memory usage increases with larger feature sizes and lower compression levels, showing signs of quadratic growth for both measured and calculated. This growth could explain why the energy savings for different feature sizes diverge more in the lower compression levels than previous experiments, which showed more linear growth.

For the Tucker decomposition, Figure 4-12 shows that the MAC reductions increase with the increased feature size, but the reduction over all feature sizes reduces for smaller compressions. Also, the memory consumption, both measured and calculated, grows for increased feature sizes, but also with the compression levels. Combined the memory and MAC operations can explain the reduction in energy savings over compression ratios, but also show that due to memory growth, the larger feature sizes will decrease faster.

The MAC reductions of the TT are consistent over all compression rates, as also seen before at the number of input channels and output channels. However, the MAC reduction does show an impactful increase with the increase in feature size. This can explain why the largest feature size of 8×8 has the most energy savings. Looking at the memory usage, it is more than for the Tucker decomposition, which is different than seen in previous experiments. This might explain that for this experiment the Tucker decomposition has more energy savings in Figure 4-11 for the largest feature size since they have similar MAC reduction.

This difference in memory usage between the Tucker and TT is also not fully captured when looking solely at the calculated memory, which shows lower and minimal memory usage for the TT. Again for the Tucker decomposition spikes are noticeable in the calculated memory across lower compression levels.

4-3-2 GPU energy savings

Figure 4-13 illustrates the energy reduction achieved by different tensor decomposition methods, varying compression rates, and feature sizes for the GPU using a line chart. On the y-axis, the saved energy (positive values) and added energy (negative values) in kWh are displayed, while the x-axis presents the different compression rates. The standard deviation of the measurements is shown by the hue.

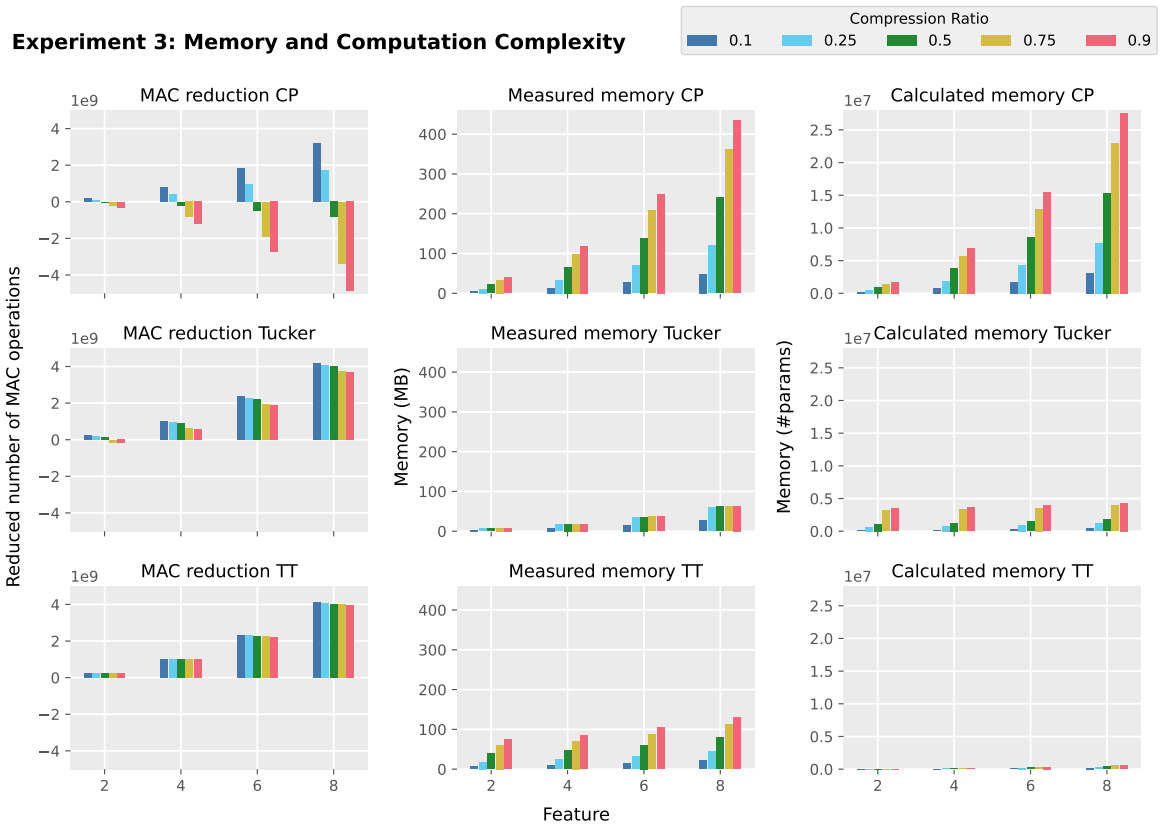


Figure 4-12: The MAC operations and memory, both calculated and measured on the CPU, for the different methods and feature sizes. On the left the MAC operations reduced by decomposing, in the middle the measured memory and on the right the calculated memory based on the intermediate results.

For the CP decomposition, similar patterns are seen as on the CPU, however now none of the feature sizes at any compression level present energy savings, only for 8×8 at $c = 0.1$, where $\Delta E = 0.0042$ kWh (7.12% of baseline E). Also, a division is visible, where both the features 4×4 and 2×2 show similar energy consumption and the features 6×6 and 8×8 show similar energy savings, where the 6×6 shows less additional energy at low compressions $c \geq 0.75$.

The Tucker decomposition also indicates an increase in energy consumption, except for the largest feature size at high compression levels ($c \geq 0.5$), with maximum savings of $\Delta E = 0.028$ (49.82% of baseline E). For all other feature sizes and compression ratios, additional energy is required. The energy consumption for feature sizes 2×2, 4×4 and 6×6 shows minimal differences, particularly at higher compression level, with a maximum energy saved $\Delta E = 0.025$ kWh (44.48% of baseline E). A similar trend is observed in the TT decomposition; however, almost no additional energy is needed. The smaller feature sizes lie close to the zero-energy line, with only the largest feature size demonstrating some energy savings.

The memory usage, presented in Figure 4-14 shows a general increase in memory usage, compared to the memory used for the CPU, for all feature sizes, compression ratios and method, except for the CP decomposition. The additional memory needed for the CP decomposition is lower on the GPU than on the CPU, which might explain similar energy savings

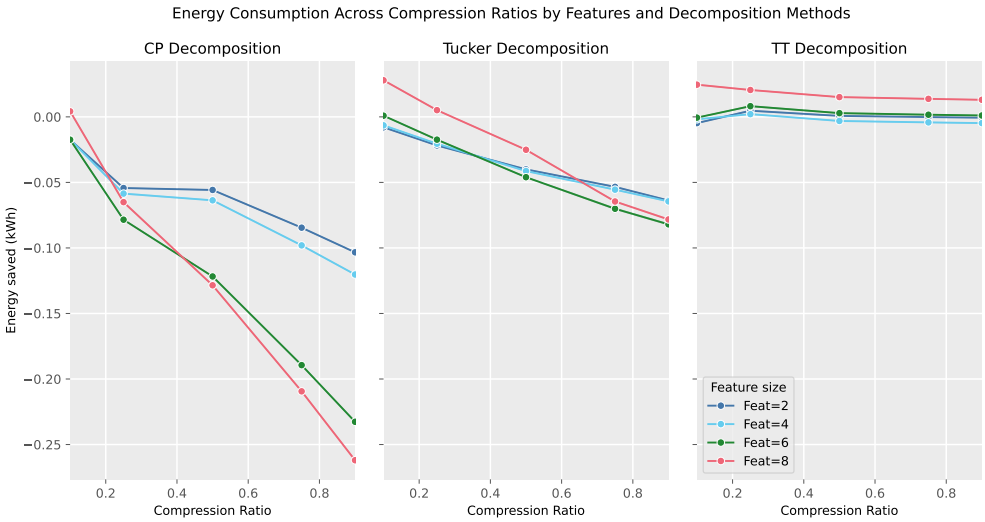


Figure 4-13: The energy saved for different methods, CP, Tucker and TT, for different feature sizes, given by different colours, run on a GPU. The hue presents the standard deviation of the measurements.

despite the complicated GPU memory management. For the Tucker and TT, the higher memory consumption could explain the lower energy savings for GPU than for CPU. Especially the memory usage of the TT for large compressions is much higher than for the CPU.

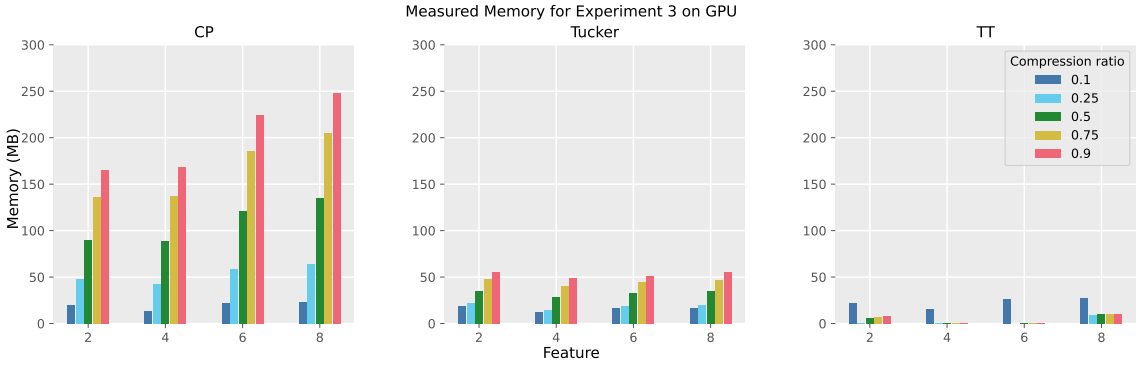


Figure 4-14: The measured memory usage on the GPU, for the different methods and feature sizes.

4-3-3 Key findings

This experiment was designed to look into whether the feature size of a layer had a prominent impact on the inference energy savings by decomposing the layer using either CP, Tucker or TT decomposition across different compression levels. Below the key findings of this experiment are outlined.

- MAC operations and memory: The difference in MAC operations and memory between the different feature sizes is more present than for the other parameters, which follows

the quadratic relation from the analysis in Table 3-1, also in the measured memory. The higher decrease of MAC and addition memory, again shows a trade-off. The measured memory shows the same pattern which would suggest that the calculated memory is compliant to the measured memory.

- Feature size: The results suggest that the feature size has a more prominent effect on energy savings. For larger feature sizes the energy saved is large for high compression $c = 0.1$, however, the additional energy for low compressions and certain methods suggests that layers with larger feature sizes need a more careful selection of compression ratio.

- Decomposition methods: For high compression $c = 0.1$ all methods show promising results, however with increased feature size and low compression, the CP shows undesired additional energy. Tucker shows energy savings also for smaller compression levels until $c \geq 0.75$, but the most consistent energy savings are again from decomposing with TT. Careful selection of the method is thus required for larger feature sizes.

- Hardware: There is a great difference between the energy savings of the CPU and the GPU, which might be explained by the difference in memory usage between the different feature sizes. Especially for CP where the intermediate results are very large coinciding with the more prominent decrease in energy savings.

4-4 Influence of the kernel size

This section will give an outline of the found results, comparing the energy savings over different kernel sizes, methods and compression ratios. All variables used are given in Table 4-4.

Table 4-4: Control, independent and dependent variables of experiment 4.

Control Variables	**Independent Variables**	**Dependent Variables**
$S = 384$	$d \in \{1, 3, 5\}$	Energy consumption
$T = 512$	$\mathrm{M} \in \{\mathrm{cp}, \mathrm{tucker}, \mathrm{tt}, \mathrm{nd}\}$	Memory
$W \times H = 4 \times 4$	$c \in \{0.1, 0.25, 0.5, 0.75, 0.9\}$	MAC
epochs $= 9.0 \times 10^3$(CPU), 6.0×10^5 (GPU)	Hardware $\in \{\mathrm{CPU}, \mathrm{GPU}\}$	

Figure 4-15, show the different ranks, for the different kernel sizes. It can be seen that again the CP ranks are very large compared to the Tucker and TT. Also, the CP rank grows extensively between the different kernel sizes, where $d = 5 \times 5$, shows more than twice as large a rank as the 3×3 kernel. Notably, the 1×1 kernel shows similar ranks to the TT and Tucker mode-specific ranks. Again, the expected savings from the CP will be lower, however for the 1×1 kernels similar results might be expected.

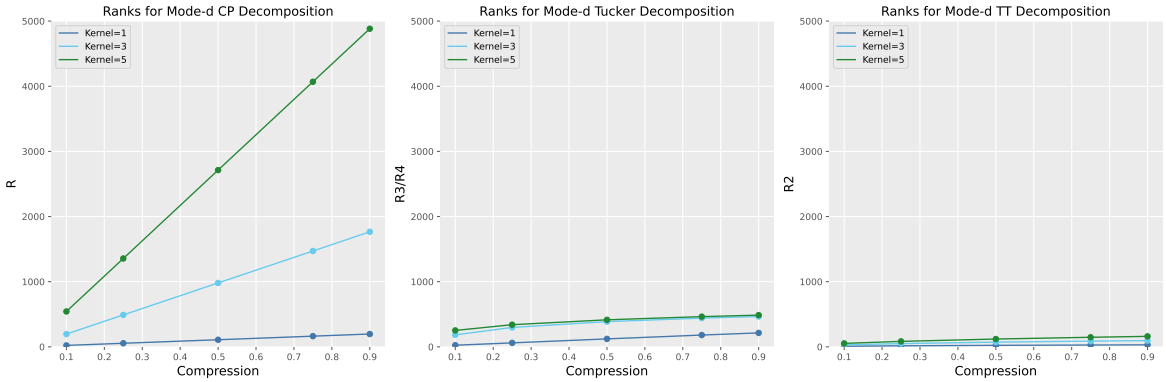


Figure 4-15: The mode-d ranks of all three tensor decomposition methods across all compression ratios.

4-4-1 CPU energy savings

Figure 4-16 illustrates the energy reduction achieved by various tensor decomposition methods, compression rates, and kernel sizes for the CPU illustrated using a line chart. On the y-axis, the saved energy (positive values) and added energy (negative values) in kWh are displayed, while the x-axis presents the different compression rates. The standard deviation of the measurements is shown by the hue.

In the case of the CP decomposition, energy savings are only observed for larger kernel sizes and higher compression ratios with notable savings at a high compression of $c = 0.1$, with a maximum of $\Delta E = 0.015$ (60.24% of baseline E). For lower compression ratios, energy savings drop below zero meaning additional energy is required for these decomposed convolutions. This becomes especially significant for the larger 5×5 kernel at low compression levels. For the smaller kernels (1×1 and 3×3), the energy drop is less prominent but still present. At low compression levels, both the 1×1 and the 3×3 kernel show similar energy addition.

For the Tucker decomposition, the largest 5×5 kernel shows the most initial energy savings for high compression, with a maximum of $\Delta = 0.020$ kWh (80.32% of baseline E). However, between compression ratios $c = 0.25$ and $c = 0.5$, there is a transition where energy addition begins for all kernel sizes, except the 1×1 kernel. The 5×5 kernel shows the most significant addition for low compression levels, while the 1×1 kernel remains constant across all compression levels, showing zero energy savings or addition.

For the TT decomposition, consistent energy savings are observed across all compression ratios, with savings increasing with larger kernel sizes to a maximum of $\Delta = 0.019$ kWh (76.31% of baseline E). Especially for the largest 5×5 kernel, TT shows the most consistent energy savings across all compression levels. In addition, a clear pattern is seen where the energy savings grow with the increased kernel size. Again, the 1×1 kernel shows no effect after decomposition, neither adding nor saving energy.

Looking at Figure 4-17, for all methods, there is minimal MAC reduction for the smallest 1×1 kernel, which could explain why decomposing would not result in energy savings for all methods. Looking specifically at the CP decomposition, for larger kernels, there is more MAC reduction, especially at high compression ratios. After $c > 0.25$, there are additional

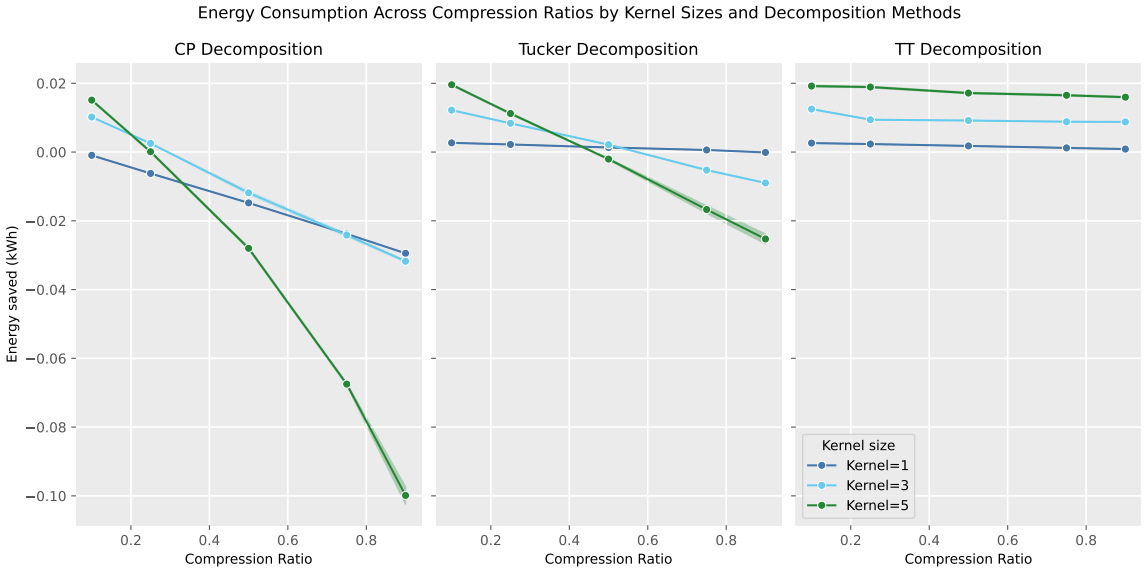


Figure 4-16: The energy saved for different methods, CP, Tucker and TT, for different kernel sizes, given by different colours, run on a CPU. The hue presents the standard deviation of the measurements.

MAC operations, particularly for the largest 5×5 kernel. In addition, the memory usage between the different kernel sizes seems to grow quadratically, also in the measured memory, with increasing kernel size for all compression ratios, which follows the relations found in the TensorlyTorch analysis from Table 3-1. In combination with the additional MAC, this might explain the increased additional energy consumption for the largest 5×5 kernel size compared to the other kernels shown in Figure 4-17. For the 1×1 kernel, the CP decomposition shows growing memory consumption for lower compression ratios, which might explain why for CP, the 1×1 kernel shows changing energy consumption.

For the Tucker decomposition, the MAC reduction increases significantly with the larger kernel size. However, across compression levels, each kernel shows a decrease in MAC reduction, especially for lower compression ratios $c > 0.5$. For the largest 5×5 kernel the difference between the MAC reduction for high compression and low compression becomes more present. For the memory, there is minimal change between the different kernel sizes. Combined, this can explain why lower compressions $c > 0$ will show additional energy usage, especially for the largest 5×5 kernel.

4-4-2 GPU energy savings

Figure 4-18 illustrates the energy reduction achieved by different tensor decomposition methods, varying compression rates, and kernel sizes for the GPU using a line chart. On the y-axis, the saved energy (positive values) and added energy (negative values) in kWh are displayed, while the x-axis presents the different compression rates. The standard deviation of the measurements is shown by the hue.

For the CP decomposition, a different pattern can be seen for the GPU energy savings. For all kernel sizes, additional energy is required, with a minimum of $\Delta = -0.014$ kWh (354% of

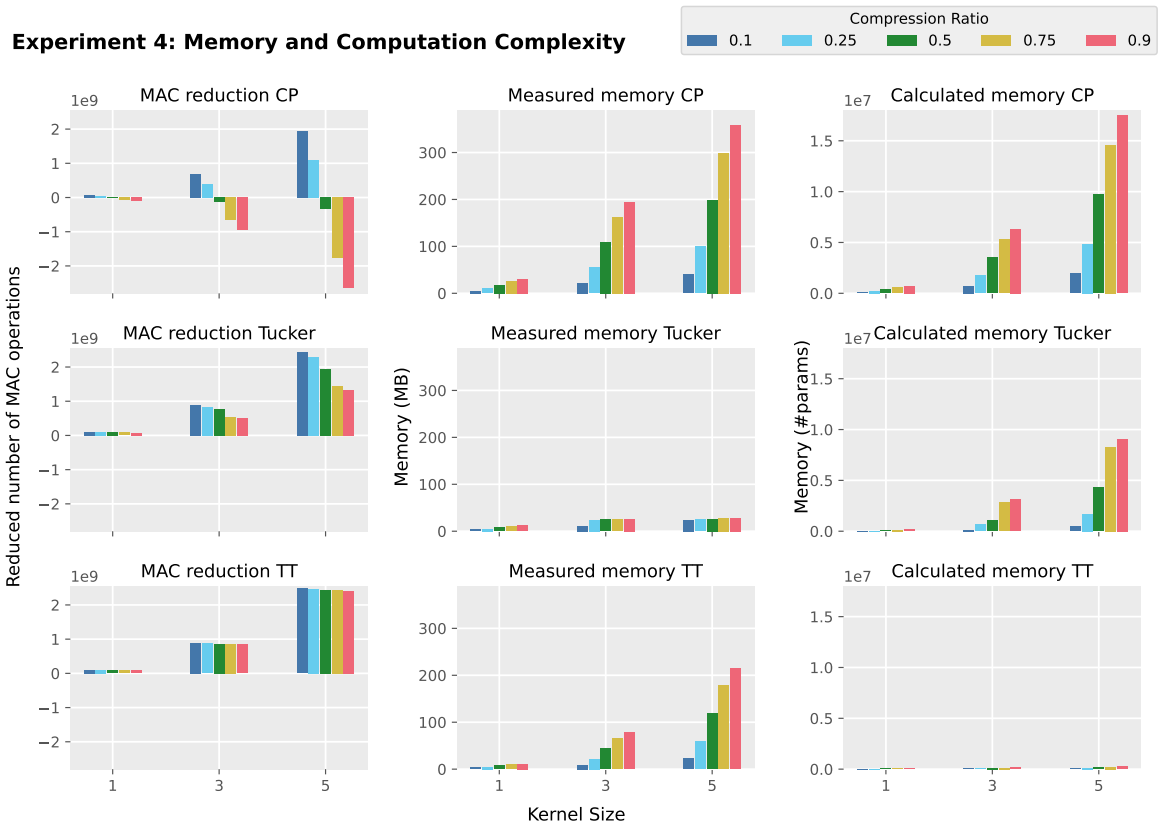


Figure 4-17: The MAC operations and memory, both calculated and measured on the CPU, for the different methods and kernel sizes. On the left the MAC operations reduced by decomposing, in the middle the measured memory and on the right the calculated memory based on the intermediate results.

baseline E), growing with the increased kernel size and with compression levels. Especially the largest 5×5 kernel shows a high amount of additional energy.

The Tucker decomposition shows a little less energy addition, even for the largest kernel 5×5 but for all levels no energy savings are present, except at $c = 0.1$ with $\Delta E = 0.0062$ kWh (24.90% of baseline E). After $c = 0.1$ all three kernel sizes show increased additional energy for decreased compression.

In contrast, the TT decomposition shows energy savings for the largest 5×5 kernel overall compressions, with a maximum of $\Delta E = 0.035$ (140% of baseline E). This energy savings seems to be similar to the TT decomposition run on the CPU. Again the TT shows consistency over all compression rates for all kernel sizes.

To explain these differences between CPU and GPU, Figure 4-19 shows the measured memory from the GPU. The memory of the CP decomposition shows even more differences and a more prominent quadratic increase for larger kernel sizes. This might explain why for the GPU, the energy consumption differences between the kernel sizes are more pronounced than for the CPU.

The Tucker memory seems very similar to the memory measured on the CPU. However, the

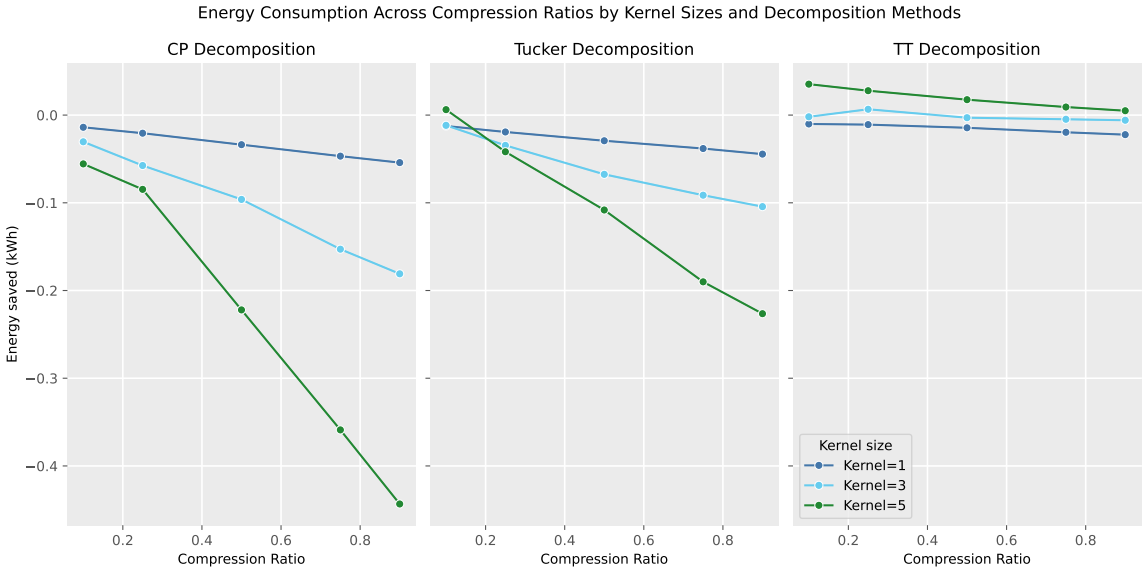


Figure 4-18: The energy saved for different methods, CP, Tucker and TT, for different kernel sizes, given by different colours, run on a GPU. The hue presents the standard deviation of the measurements

energy savings are lower than those of the CPU. This might be the case since memory plays a more important role for the GPU, and thus the MAC reduction does not counterbalance the memory loss as much as in the CPU.

For the TT decompositions, the memory shows a similar pattern to the CPU memory, however, the values seem a little lower, particularly for the large 5×5 kernel. This might explain why the energy saved on the GPU appears to be slightly higher than that of the CPU. For the other kernel sizes, the TT decomposition also performs a little worse on the GPU.

4-4-3 Key findings

This experiment was designed to look into whether the kernel size had a prominent impact on the inference energy savings by decomposing the layer using either CP, Tucker or TT decomposition across different compression levels. Below the key findings of this experiment are outlined.

- MAC operations and memory: For this experiment, the MAC reductions and memory seem to explain the patterns in energy savings. For a larger kernel size, the reduction of MAC operations increases rapidly. For the memory, this difference is also present where the increase seems to be quadratic similar to the different feature sizes. For a really small kernel 1×1 almost no reduction of MAC operations and additional energy is present for all methods. Different to the other experiments the Tucker method seems to have less measured memory than the other methods, which cannot seem to explain the energy saving and is not compliant to the calculated energy.
- Kernel size: The kernel size seems to have a great influence on the energy savings and for high compression the energy saved is quite high. With the increase of the kernel size,

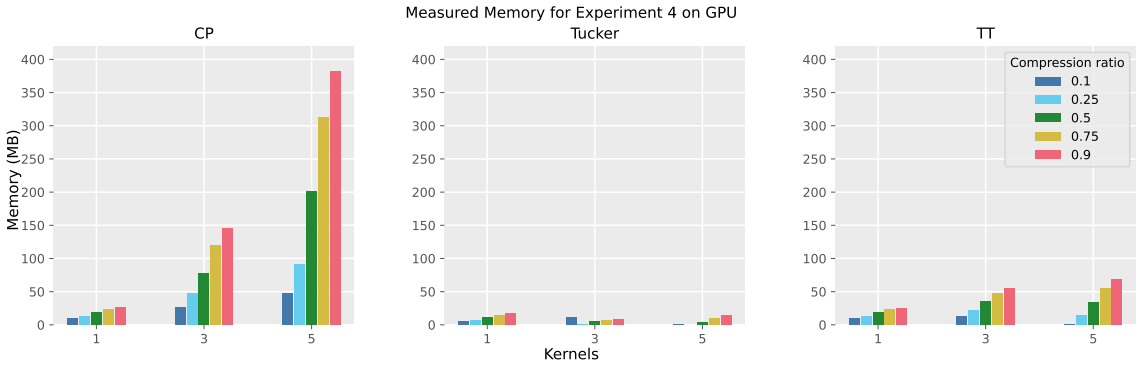


Figure 4-19: The measured memory usage on the GPU, for the different methods and kernel sizes.

more energy is saved, however again a trade-off is present where for lower compressions, $c \geq 0.25$ for CP and $c \geq 0.5$ for Tucker, present additional energy.

- Decomposition methods: Again, the TT decomposition provides consistent energy savings over all compression levels which increases with the kernel size. CP performs the worst especially for low compressions and high kernel size, followed by Tucker.
- Hardware: The energy saved on the GPU is much lower and except for TT, only shows additional energy consumption. For GPU also the Tucker memory is the lowest. The trends in energy savings are similar to the CPU.

4-5 Energy savings decomposed ResNet18

In this section, the result of the energy measurements to a commonly used CNN, ResNet18, will be outlined. For such a network the different layers all have different configurations where more than one of the parameters from previous experiments change per layer. It is interesting to see whether the effects per parameter can be translated to a network with combined connections.

The variables used for this experiment are given in Table 4-5. As input data, the CIFAR10 dataset was used.

Table 4-5: Control, independent and dependent variables for training and inference Resnet18 experiment.

Control Variables	Independent Variables	Dependent Variables
Input image=$32 \times 32 \times 3$	$L \in \{63, 57, 51, 47, 41, 35, 28, 25, 19, 6\}$	Energy consumption
seed=1	Method $\in \{\text{cp}, \text{tucker}, \text{tt}, \text{not decomposed}\}$	Memory
lr=1×10^{-5} (only for training)	$c \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$	MAC
epochs = 100	Hardware $\in \{\text{GPU}\}$	

Figure 4-20 shows the percentage of inference energy saved for each layer, compression level and decomposition method. As also mentioned in Subsection 3-2-2, the metric is the ratio of

energy saved compared to the original baseline energy. The gradient shows the energy saved in blue and the additional energy in red.

The CP decomposition shows that lower compressions generally increase the energy, except for layers 57, 41, and 25. This energy increase is particularly noticeable in the largest layer (63), which aligns with observations from isolated experiments. Layers 57, 41, and 25 are downsampling layers with a 1×1 kernel, possibly explaining the minimal energy change. At higher compressions, the additional energy decreases slightly, consistent with isolated experiment results. The most energy saved in the CP decomposition is 0.4% in layer 28 ($c = 0.1$), while the most energy added is -33.34% in layer 63 ($c = 0.9$).

For the Tucker decompositions, similar patterns are seen, where the lower compressions add more energy than the higher compression levels, however for the largest layers it can be seen that this does not result in energy savings, but it gets close to zero. For the middle layers, Tucker does show some slight energy savings and compared to CP shows overall better results, with a maximum of 2.24% savings for layer 28 ($c = 0.3$) and the most additional energy of -15.83% for layer 63 ($c = 0.7$).

The best results are found for the TT decomposition, however not many differences can be found between TT and Tucker. For the larger layers $63, 57$ and 51 there is less profounding energy addition, however still no savings. At high compression on the largest layer, however some blue can be seen where there is a suggestion that for large layers with high compression, there is potential for energy savings, with a maximum of 7.24% for layer 63 ($c = 0.75$). Also for the other layers, at high compression, a hue of blue can be found, however, most layers show additional energy with the most added energy of -7.4% at layer 63 ($c = 0.9$).

As also observed in the previous isolated experiments, TT shows a more consistent and better result than the other methods, followed by Tucker and CP. However, there is still little energy saved and mostly energy addition. This is a similar situation as for the other experiments, where in general the GPU experiments showed mainly energy addition.

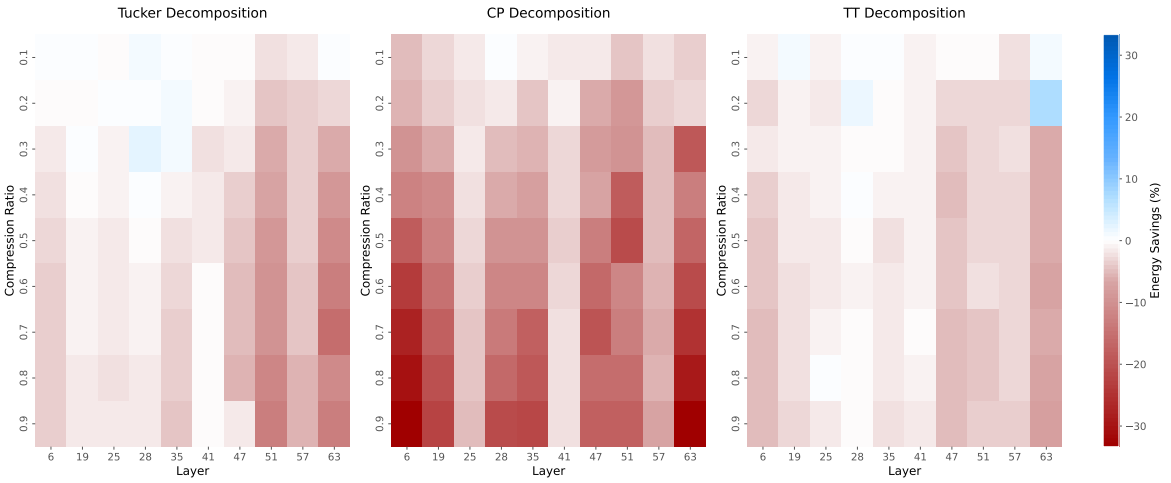


Figure 4-20: The inference energy saved by decomposing several layers of the ResNet18 with different decomposition methods, CP, Tucker and TT, for various compression levels. The colorbar shows, that the greener the more energy is saved and the more orange, the less energy is saved.

4-6 Modelling of the saved energy

This section will elaborate on the results from the regression models. The benchmark model performance will be presented, from which comparisons can be derived with the more complex multivariate models, either linear or polynomial, based on the memory usage and the reduced MAC operations. This will be done based on the R-squared, adjusted R-squared and the Root-mean-square error (RMSE). To look at the generalization of the models the dataset, consisting of the single convolution energy measurements, is split into a test and training set resulting in different RMSE.

4-6-1 Benchmark Model Performance

The benchmark model is based solely on the number of reduced MAC operations, representing the focus of current research on the computational complexity of tensor decomposed convolutions. For both hardware types, a separate benchmark model is fitted, taking into account the difference in compute (parallel vs. serial). The *statsmodels* summary of the fitted models are given in detail in Appendix C Subsection C-2-1.

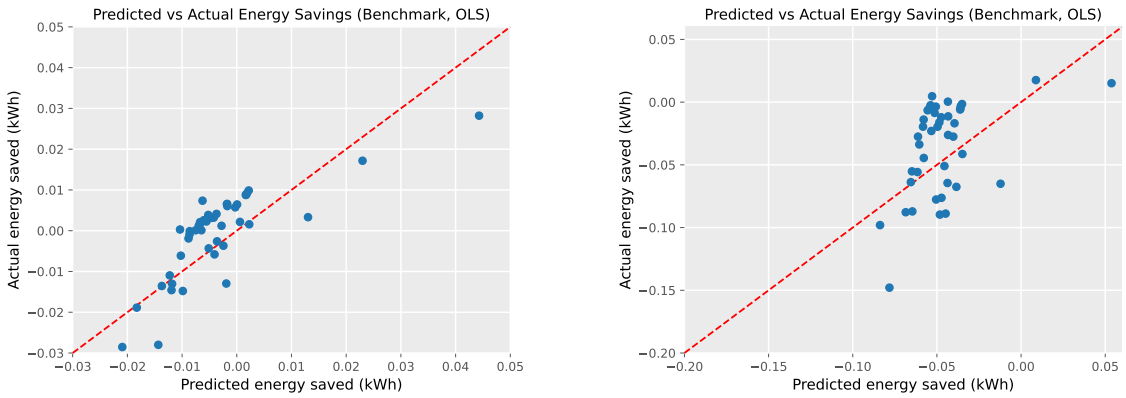
For the CPU, the benchmark model has an R-squared value of 0.586, with an adjusted R-squared of 0.584. These values are quite low, however larger than the GPU, indicating that the model is not capable of capturing all variances in the model. The coefficient for the reduced MAC operations is 1.68×10^{-2} (Standard Error (SE)= 2.52×10^{-3}, p<0.001), indicating a statistically significant positive relationship. The intercept -2.00×10^{-3} (SE= 1.06×10^{-3}, p-0.062) is not statistically significant, indicating that it does not contribute to the predictive power of the model.

For the GPU, the benchmark model has an R-squared value of 0.305, with an adjusted R-squared of 0.301. These values are quite low, indicating that the model is not capable of capturing all variances in the model. The coefficient for the reduced MAC operations is 3.55×10^{-2} (SE= 6.72×10^{-3}, p<0.001), again indicating a statistically significant positive relationship. The intercept -4.38×10^{-2} (SE= 4.00×10^{-3}, p<0.001) is more negative compared to the CPU model and significant, suggesting that the reduced MAC operations can not capture all variances and another negative contribution might be present.

Looking at Figure 4-21a and Figure 4-21b, it can be seen that the benchmark models do not perform very well.

Table 4-6: Comparison of RMSE values for different models on CPU and GPU.

Metric	Bench	Lin (Meas)	Lin (Calc)	Poly (Meas)	Poly (Calc)
CPU RMSE **(Train)**	0.0535	0.0087	0.0058	0.0079	0.0055
CPU RMSE **(Test)**	0.0075	0.0057	0.0045	0.0049	0.0046
GPU RMSE **(Train)**	0.0141	0.0302	0.0324	0.0294	0.0251
GPU RMSE **(Test)**	0.0415	0.0237	0.0227	0.0241	0.0206



(a) Measured memory **(b)** Calculated memory

Figure 4-21: Predicted vs actual energy from the linear benchmark model for CPU (left) and GPU (right).

4-6-2 Comparative analysis with linear and polynomial models including memory usage

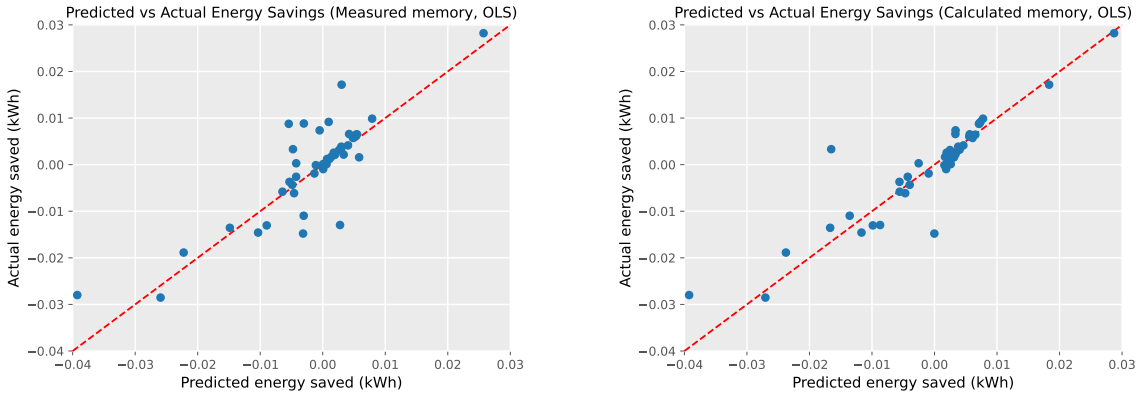
The more complex models also include the additional memory required when the convolution layers are decomposed. For memory usage, both calculated memory (based on the parameters in the intermediate tensors) and the memory measured with a profiling tool were used. To see whether additional complexity, capturing also non-linear relations, would increase the performance both linear and polynomial models were fitted on either the GPU or CPU data.

The results showed that in general the inclusion of memory as a predictor significantly improves the model's explanatory power. The results for both linear and polynomial models, incorporating either measured or calculated memory, are provided in detail in Appendix C Subsection C-2-2 and Subsection C-2-3.

CPU models

For the linear CPU model, based on the measured memory (Table C-7), the adjusted R-squared increases to 0.841 compared to the benchmark adjusted R-squared of 0.584, indicating that the addition of the measured memory captures more of the variance in the data. The coefficient for measured memory is -0.0119 (SE=0.0011) indicating a strong negative effect on the energy savings ($p < 0.001$). The MAC coefficient remains positive (0.0124, SE=0.0013), confirming that reduced operations contribute positively to energy savings and even with a similar amount to the memory. This is reflected in the RMSE values, where the train RMSE drops to 0.0087, and the test RMSE to 0.0057, improving from the benchmark RMSE (0.0535 and 0.0075).

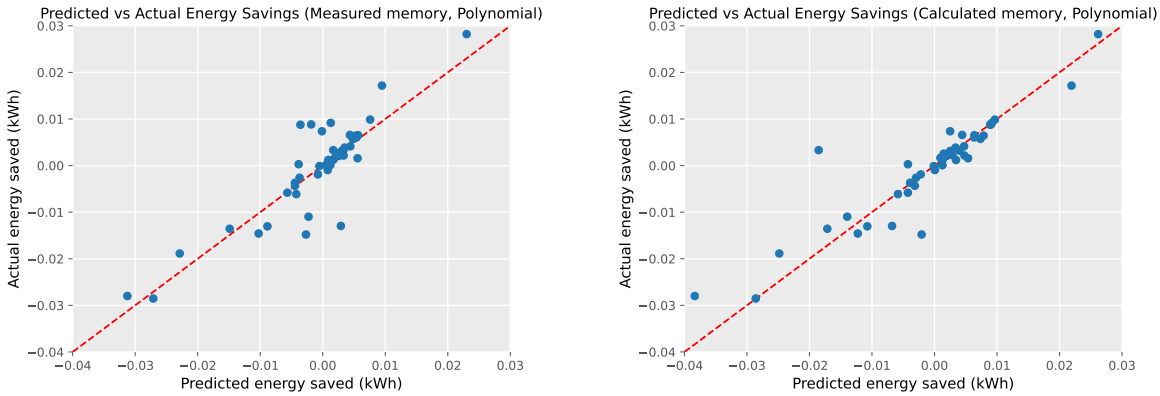
The linear CPU model using the calculated memory (Table C-9), shows a better predictive performance, with an adjusted R-squared of 0.928. Here the calculated memory coefficient shows a more negative influence (-0.0151, SE=0.0008) on the energy savings ($p<0.001$), compared to the measured energy. The MAC coefficient is 0.0090 (SE=0.0011), showing a slightly decreased influence ($p<0.001$), balanced out by the calculated memory. The lower RMSE values 0.0058 (train) and 0.0045 (test) confirm the better fit of the model, as also seen in Figure 4-22, where the calculated memory model aligns more closely with the diagonal.



(a) Measured memory **(b)** Calculated memory

Figure 4-22: Predicted vs actual energy saved from linear model on CPU.

The polynomial models show some minor improvements in the R-squared values (0.937 for calculated memory and 0.870 for measured memory). Compared to the linear models, the increased complexity only has subtle accuracy differences, with train RMSE of 0.0079 (measured) and 0.0055 (calculated), and test RMSE of 0.0049 (measured) and 0.0046 (calculated). In addition, it is important to note, that some of the higher-order coefficients, such as $\beta_3 = -9.30 \times 10^{-3}$ (SE= 4.85×10^{-3}, p=0.0598) from the calculated memory, are not statistically significant, indicating that the additional complexity does not significantly contribute to the predictive power. Combined, this suggests that the slight performance improvement may not fully justify the added complexity of the polynomial models, especially when using calculated memory. Figure 4-23, also visually confirms that the performance of the polynomial does not improve that much.



(a) Measured memory **(b)** Calculated memory

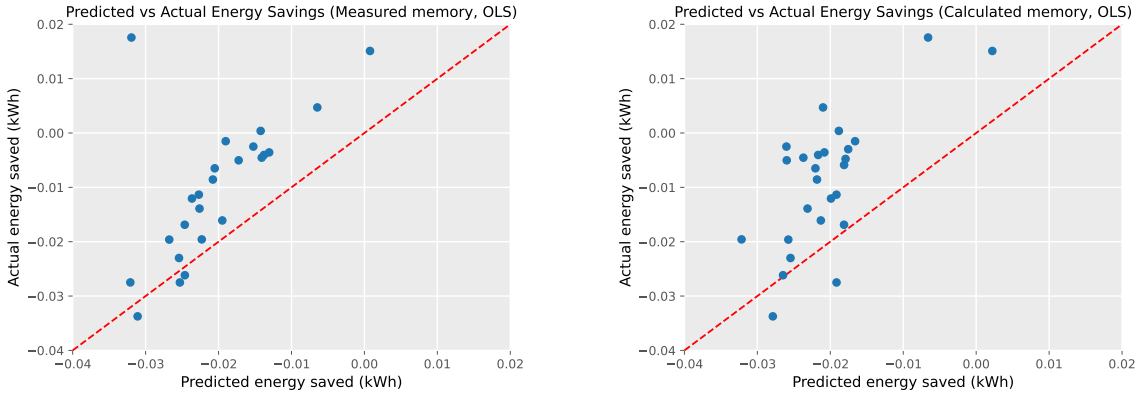
Figure 4-23: Predicted vs actual energy saved from the polynomial model on CPU.

GPU models

Compared to the linear benchmark model, the adjusted R-squared of the linear GPU model, with measured memory, improves from 0.301 to 0.776 (Table C-11). The coefficient for the

measured memory is -5.24×10^{-2} (SE=$3.12 \times 10^{-3}, p < 0.0001$), indicating that the memory usage has a significant negative impact on the energy savings. The MAC coefficient remains positive at 7.30×10^{-3} (SE=$2.58 \times 10^{-3}, p = 0.000457$) and is larger than the measured memory coefficient, indicating that the MAC operations might be more influential.

For the linear GPU model, using the calculated memory, the R-squared of 0.745 (adjusted R-squared of 0.742) is slightly lower than the measured memory model. The calculated memory coefficient is -4.99×10^{-2} (SE=$8.68 \times 10^{-3}, p < 0.001$), showing significant negative influence to the energy consumption and is more pronounced than for the CPU model. The MAC coefficient is similar to that in the measured model at 9.40×10^{-3} (SE=$2.68 \times 10^{-3}, p = 0.000432$). The RMSE values 0.0324 (train) and 0.0227 (test) suggest, in addition to the lower R-squared values compared to the CPU, that the complexity of GPU may require more complex models. This difference in performance is also visible in Figure 4-24 and Figure 4-22.



(a) Measured memory **(b)** Calculated memory

Figure 4-24: Predicted vs actual energy saved from linear model on GPU.

The polynomial models on the GPU data introduce some improvement in the R-squared values (0.790 for measured memory and 0.847 for calculated memory) and adjusted R-squared values (0.784 for measured memory and 0.843 for calculated) compared to the benchmark, however, the gains are subtle, with train RMSE of 0.0294 and test RMSE of 0.0241 for measured memory, and train RMSE of 0.0251 and test RMSE of 0.0206 calculated memory models. In addition, Figure 4-25, shows that the predictions for the polynomial models are not much better than from the linear models. The coefficients for the interaction terms and squared terms in these polynomial models, such as the squared term for calculated memory -9.83×10^{-2} (SE=$1.16 \times 10^{-2}, p < 0.001$), indicate potential non-linear relations, but their overall impact is limited.

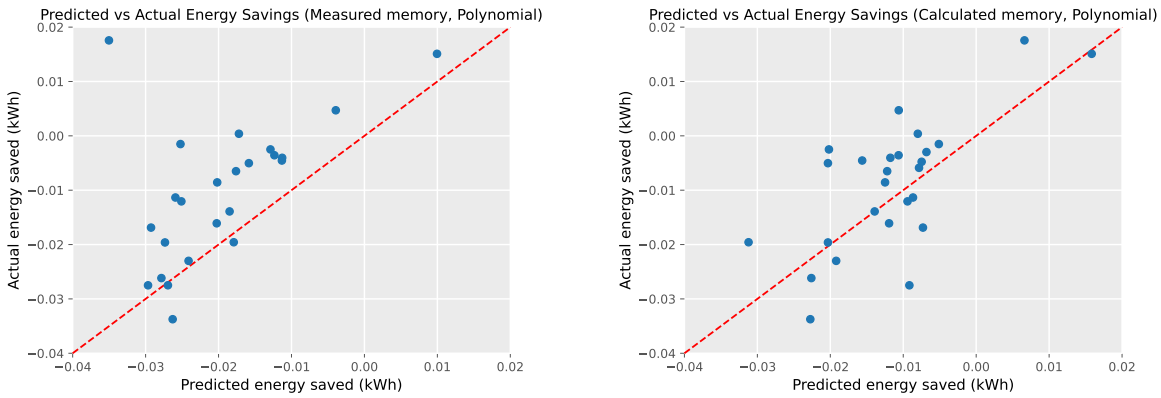
4-7 Key findings

The presented results of the eight different regression models have highlighted some interesting insights. Below a summary of the key findings will be given.

- Benchmark model limitations: The benchmark models, which focused only on MAC operations, had low adjusted R-squared values (0.584 for CPU and 0.301 for GPU),

indicating that MAC operations alone are insufficient to accurately predict energy savings.

- Impact of memory usage as an additional predictor: Adding memory usage as a predictor greatly improved model performance. For the CPU, the adjusted R-squared increased to 0.841 with measured memory and 0.928 with calculated memory. For the GPU, these values were 0.776 and 0.742, respectively, highlighting the strong influence of memory on energy savings on both types of hardware.
- Calculated versus measured memory: Models using calculated memory generally outperformed those using measured memory, however, the test RMSE was lower for calculated memory models, indicating better predictive accuracy, particularly in the CPU models.
- Additional complexity from polynomial features: Polynomial models offered only slight improvements over linear models, particularly in the GPU models. Although the adjusted R-squared values for the polynomial models were slightly higher (0.937 for calculated memory on CPU and 0.847 for GPU), this added complexity did not lead to a significant boost in predictive accuracy. Some of the higher-order coefficients, were not statistically significant, indicating that these polynomial terms did not meaningfully improve the model. Additionally, when comparing RMSE values between training and test sets, it became clear that linear models, especially those using calculated memory, perform better. This was particularly visible in the CPU data, where the simpler linear models performed nearly as well as the more complex polynomial models.



(a) Measured memory **(b)** Calculated memory

Figure 4-25: Predicted vs actual energy saved from the polynomial model on GPU.

Chapter 5

Conclusion & Discussion

This section summarizes the results from both the single convolution experiments and the regression models from Chapter 4 and uses them to address the initial research questions. Following this, a discussion will highlight the limitations and assumptions of the thesis. Lastly, a presentation of the possible future work, extending on this research, will be given.

5-1 Conclusion

As a foundation of this thesis the following research question was formulated:

How does the implementation of tensor decompositions (Parallel Factors (PARAFAC) Canonical Decomposition (CANDECOMP) (CP), Tucker, and Tensor Train (TT)) affect the energy efficiency of large Convolutional Neural Networks (CNNs) during inference, with a focus on predicting the resulting energy savings?

This research question was divided into smaller sub-questions, for which the findings will be given below before answering the main research question.

5-1-1 Tensor decomposition methods

The first sub-question, related to the decomposition methods is: *Which type of tensor decomposition (CP, Tucker, or TT) and compression ratio yields the best performance for energy efficiency in CNNs?*

The analysis of tensor decompositions (CP, Tucker, TT) for decomposed convolutions reveales that both the decomposition method and the compression ratio greatly affect the energy efficiency of the decomposed convolutions. CP decomposition generally provides the least energy savings, especially at lower compression ratios, due to increased memory demands. Tucker decomposition offers better energy efficiency across a wider range of compression ratios, but it still consumed more energy at lower compression ratios, especially for larger layers.

In contrast, the TT decomposition consistently yielded the best energy savings across all compression ratios, with minimal additional energy requirements. For the Tucker and CP sometimes additional energy was required for the decomposed convolutions at lower compression levels, which is contradicting the intended benefits of decomposing.

While this thesis focuses on energy efficiency, it's important to consider that lower compression ratios likely offer higher accuracy, which is essential in safety-critical systems like autonomous vehicles (AVs). For AVs, where safety constraints may require lower compression ratios, TT decomposition is recommended based on its consistent performance. If higher compression is acceptable, both Tucker and TT decompositions are viable options.

5-1-2 CNN configurations

The second sub-question, related to different convolution layer configurations, was: *Which convolutional hyperparameters, such as input/output channels, input feature size, and kernel size, yield the greatest energy savings for decomposed convolutions using tensor decompositions?*

The analysis of different convolutional configurations revealed that increased input/output channels, feature input sizes, and kernel sizes generally led to greater energy savings when using decomposed convolutions at high compression ratios. However, lower compression levels often resulted in additional energy consumption for larger convolutions. Thus, careful selection of which convolutional layers to decompose is crucial, especially with specific tensor decomposition and compression ratio combinations.

The most influential parameters were identified as the kernel size and feature size, showing the most impact on the reduction of Multiply-Accumulate operations (MACs) and additional memory required. Consequently, these parameters had the most effect on the energy savings. Notably, the 1×1 kernel generally shows no reduced energy but also no added energy consumption, balancing around zero energy saved. While the number of input and output channels also demonstrated increased energy savings with larger sizes, the differences were less prominent compared to kernel and feature sizes.

The experiments on the popular ResNet18, showed that the larger layers, consisting of larger kernels and input/output channels, combined with high compression can reduce the energy consumption, however, for lower compressions additional energy is required, especially for the CP.

These findings suggest that strategically choosing which layers to decompose, particularly considering kernel and feature sizes, is crucial for maximizing energy savings. This aligns with current research focused on balancing compression, energy efficiency, and computational performance. Therefore, informed architectural decisions are necessary for effective inference on decomposed CNNs, especially in resource-limited implementations such as AVs.

5-1-3 Hardware considerations

The third sub-question, related to the difference between implementing tensor decomposition on either Central Processing Unit (CPU) or Graphics Processing Unit (GPU), was: *To*

what extent is the energy consumption of tensor-decomposed CNNs influenced by the use of hardware, specifically GPU compared to CPU?

The analysis of all experiments performed on either CPU or GPU, has shown that using GPU has an important influence on the energy saved. Overall the energy savings of using the GPU were lower than that of the CPU. This was an unexpected result since GPU parallel processing was expected to decrease the computational demand. An explanation for the higher energy consumption might be the difference in memory management between the GPU and CPU. The found memory patterns of the GPU were different from the CPU and the results suggest that the additional memory allocation, necessary for the smaller intermediate tensors, plays a more prominent role in the specialized hardware such as GPU. Due to these memory fluctuations, it is more difficult to see patterns based on layer configurations and compressions and predict the effect.

In implementing tensor decomposition in AVs for efficient inference, these findings highlight the importance of considering hardware-specific characteristics. Although GPUs are known for their parallel processing powers, which should reduce computational demand, their possible memory management inefficiencies can lead to higher energy consumption. This insight stresses the need to optimise the implementation of tensor decompositions for computational efficiency and memory usage, particularly when deploying on large CNNs on specialized hardware in AVs.

5-1-4 Modelling of energy savings

The fourth sub-question, related to the modelling of the expected energy savings, was: *To what extent can energy savings in tensor-decomposed CNNs be accurately predicted based on pre-implementation data, such as calculated computation and memory usage, compared to models requiring additional empirical measurements?*

Eight different regression models were fitted to answer this sub-question, based on two predictors for energy saving: memory usage and MAC operations. These models were compared to a benchmark model that focused only on the computational complexity (reduced MACs) after decomposing.

The benchmark models, for both CPU and GPU, did not perform very well at predicting the energy savings. For the CPU, the benchmark model had an R-squared value of 0.586 and for the GPU, it was even lower at 0.305. The coefficients of the MAC operations were significant, indicating that it does contribute to the energy savings, however, they alone can not capture all influences on the energy savings.

When memory usage was added as an additional predictor, the performance increased significantly. The linear model, including either measured or calculated memory, had an increase in the adjusted R-squared values up to 0.928 for CPU models (calculated memory) and 0.776 for GPU models (measured memory). This indicates that memory usage is an important factor contributing to the energy savings of tensor-decomposed CNNs. The differences in performance between the CPU and GPU further suggest that the memory and computational complexity of the GPU might ask for more complex modelling techniques.

The comparison between models using calculated and measured memory showed that the calculated memory generally was better in predicting the energy savings, especially in CPU

models. This suggests that predictive models based on calculated memory, which can be determined prior to model implementation, can be sufficient for modelling the energy savings. This allows for convenient in-advance predictions of energy savings.

The additional complexity in the polynomial models did not show major improvements in the predicting performance compared to the linear models. The adjusted R-squared values for these models were slightly higher, but some of the higher-order terms were not statistically significant, indicating that the added complexity did not always result in better predictions. In particular, for the CPU models the simpler linear models performed almost the same as the polynomial models. For the GPU models, showing some increased accuracy, increased complexity might be necessary, maybe in the form of additional predictors.

In conclusion, while it is possible to predict energy savings in tensor-decomposed CNNs to some extent, particularly when additional memory is added to the models, the complexity of the energy savings, especially on specialized hardware like GPU, limits the accuracy of these predictions. For CPU, linear models, especially with the calculated memory, offer a promising approach for initial predictions, however, more sophisticated modelling might still be necessary to accurately and reliably predict. Also, since the sample size of the data used in these models (n_train=180, n_test=20) is not that large, more additional measurements could enhance the performance of these regression models.

5-1-5 Tensor decompositions for CNNs in AVs

Finally, the main question, which was presented again at the beginning of this chapter will be answered.

This thesis has looked further into the potential of tensor decomposition methods to increase the energy efficiency of large CNNs. It has shown that implementing these decompositions in CNNs can result in energy savings. However, to do so, careful selection of key parameters, such as decomposition method, CNN layers to decompose and hardware implementations, is required. These parameters can drastically affect energy savings, resulting in additional energy consumption.

In the context of implementation in AVs the first potential has been shown to increase the energy efficiency. However, this thesis has also shown that the expected results and theoretical benefits, might not always be present. Especially implementing these decomposition methods on specialized hardware such as GPU shows weak energy savings and most of the time even additional energy. This implies, that more research is still necessary before these decomposition methods can be implemented into real-life applications.

For this thesis, the accuracy of the resulting convolutions has not been taken into account, however, in the context of safety-critical AVs this is still important to keep in mind. The findings in this thesis show that for better energy savings the compression levels need to be high, probably affecting the accuracy. Before implementation, it is thus important to look into the balance between compression and accuracy, considering the safety implications.

Lastly, regression models showed that predicting energy savings in tensor-decomposed CNNs is feasible, particularly when memory usage is included as a predictor. Benchmark models based solely on MACs were less accurate, especially for GPU. However, adding memory usage significantly improved prediction accuracy, with calculated memory being particularly

effective for the CPU data, suggesting that pre-implementation predictions can be reliable. While polynomial models offered minimal improvements, the complexity of GPU energy savings may require more complex models. Additional empirical data could further enhance the prediction accuracy.

5-2 Discussion

During this research, several choices and assumptions were made that could have influenced the results presented. In this section, some of these decisions and assumptions will be highlighted. In addition, any unexpected outcomes will be pointed out.

One of the main design choices was to use a watt meter, instead of using existing profiling tools. Although the watt meter has several advantages, it also comes with some drawbacks. The energy can not be measured per process and the power fluctuations might introduce noise. The alternative tools also have their advantages and disadvantages but choosing one of those would have possibly changed the results. To get an image of the other tools compared to the watt meter some small experiments were done using the CarbonTracker tool and the online Green Algorithms calculator. Figure 5-1 shows the results of these tools for different experiments, where each experiment uses different CNN parameters and decomposition method, with a total of three experiments. It can be seen that the online tool substantially overestimates the energy consumption. Contrary, the CarbonTracker shows similar results as the watt meter. This suggests that the measurements and results in this thesis can be assumed relevant compared to the CarbonTracker and changing to another measurement tool might not change the results of this thesis that much.

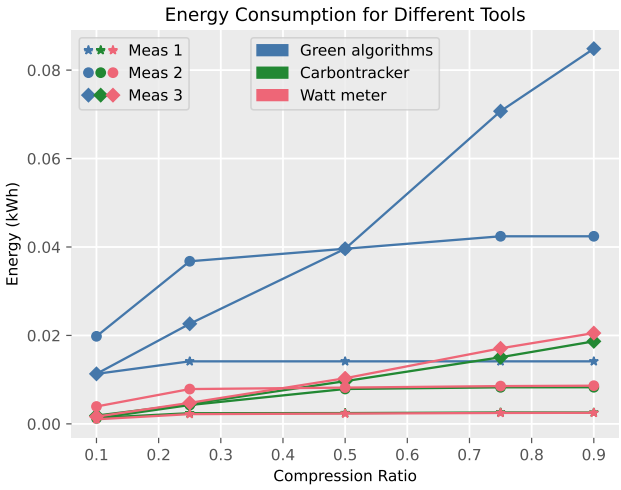


Figure 5-1: Energy consumption measured in three experiments, each with a different tensor decomposition across various compression ratios. For each experiment, the energy is measured by Green algorithms, CarbonTracker or the watt meter.

An unexpected result is that the use of the specialized GPU resulted in worse energy consumption than the CPU. The parallel computation was designed to be more efficient, however, this

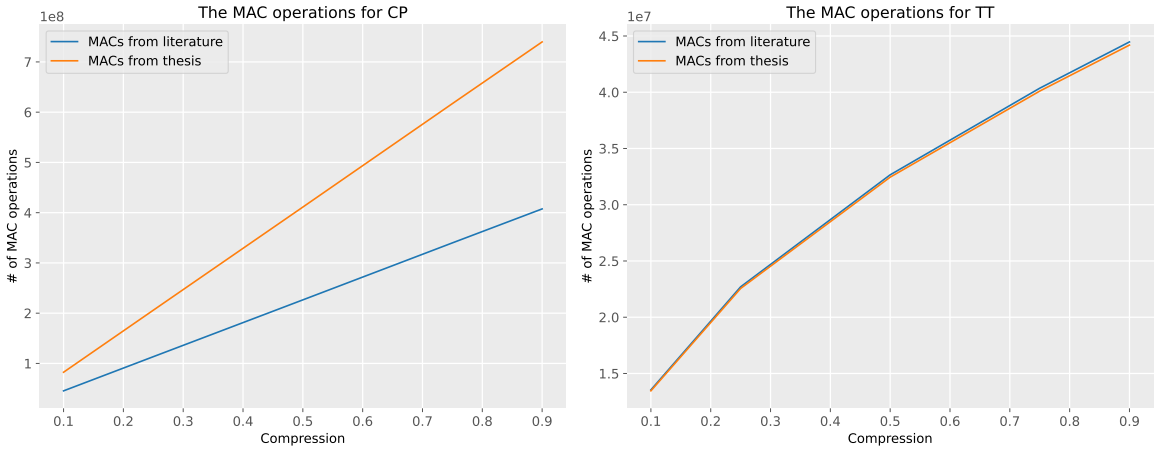


Figure 5-2: The difference in MAC operations for one experiment for the complexity in the literature and the found complexity in the TensorlyTorch analysis for both the CP and TT decomposition.

thesis has shown that for this implementation the bottleneck of the GPU, in the context of energy consumption, might be the memory allocation. Further investigating this phenomenon, by doing more experiments on other types of GPU or other specialized hardware such as Tensor Processing Unit (TPU), might give more insights into this bottleneck, particularly focusing on the memory management.

Talking about memory, another limitation of this thesis is worth mentioning. For the memory profiling, presented in Section 3-3, only the peak memory was measured compared to measuring the complete memory. It can be expected, that the peak memory is a representative measure since it presents the highest needed memory for that piece of code and thus shows the memory demand. However, to be sure it would be a good idea to look closer into the memory allocated on the GPU, especially since the results are disappointing.

Another possible cause for the found energy inefficiencies could be the badly documented TensorlyTorch library. It is possible that the library was not coded efficiently or that the programmers did not consider the memory implications when coding. The efficiency of the allocation of the memory might have a big impact on the energy savings from decomposing. It would thus be beneficial to look into how this library is constructed in the context of memory allocation and to see what the energy savings would do if this library was not used.

In addition to the influence on the memory, the TensorlyTorch implementation also results in different computational complexities, as was found in this thesis analysis. The number of MAC operations was used to calculate the reduced complexity and the derived reduction did match the patterns seen in the energy consumption. However, when we take the MACs for the CP decomposition as an example, Figure 5-2 shows that this thesis analysis of TensorlyTorch shows a substantially larger number of MAC operations compared to the number of operations from the literature. The TT decomposition however doesn't show a big difference, however, the large difference in the CP, compared to the complexities described in the literature, is an interesting observation and might require more research.

One part, of the thesis, was modelling the energy savings based on the reduced number

of MAC reductions and the measured or calculated additional memory. To use a linear regression, assumptions were made, about the data. It was assumed that the data was normally distributed, which it almost was, however, some skewness was present. This could have influenced the accuracy of the model. The number of data points used for the linear regression was also quite low, where only 180 data points were included in the training set. Collecting more data points might increase the accuracy of the models. In the dataset of the GPU there was also an imbalance between many negative energy savings and less positive energy savings. This data imbalance might also impact the accuracy of the model.

The linear regressions made a start in modelling the energy savings from decomposing. These models were based on the memory and MACs. To extent on these models, it might be nice to investigate other predictors, for example the different ranks that have a direct relation with the MAC and memory. In addition, different regressions or other modelling techniques could be used to see whether this can increase the accuracy.

As mentioned in Section 3-3, each experiment was run three times, from which at least one run was at another date and time. Running three runs instead of one increases the representation of the actual measurements and reduces the inclusion of noise. However, three runs might not be sufficient, and adding additional runs would be preferred. The number of epochs for each run was chosen based on some trial and error runs to make sure the measurements were sufficiently long, having at least four data points. However, in hindsight, it was found that not all measurements were sufficiently long and some might have been a bit too short.

5-3 Future work

In the previous section, several unexpected results but also limitations of this thesis were outlined. Combined these form a basis for future work, extending on this thesis. Below possible extensions will be summarized presenting an overview of the future work.

One of the main questions following this thesis is why the implementations on the GPU showed additional energy consumption in most of the experiments. Several explanations were thought off, in particular the memory allocation of the GPU. Future work should look into these memory allocation patterns and the efficiency of them.

Combined with that, future work should look into whether the popular TensorlyTorch library is efficiently implemented and whether the use of this library has a great impact on the energy efficiency of the decomposed convolutions.

Lastly, more extensive and longer experiments are necessary with more variations in independent variables. This thesis has presented a baseline, identified initial patterns and shown the first potential benefits of decomposed convolutions. Future research, could extent on this creating a larger dataset of energy savings and showing the more broad implications of tensor-decomposed convolutions. This could also include looking into other types of specialized hardware such as TPUs, specialized for tensor operations. More data could also encourage expanding the regression models and exploring other modelling techniques to find accurate and reliable models predicting the energy savings needed.

To conclude, this thesis has made a great start and has laid a foundation for future work looking into the great potential of tensor decomposition in large CNNs.

Appendix A

Tensor decompositions

The algorithms presented below form the basis for tensor decomposition methods such as Parallel Factors (PARAFAC) Canonical Decomposition (CANDECOMP) (CP), Tucker and Tensor Train (TT) decomposition. They are separated for each technique and are referenced in Section 2-3.

A-1 CP decomposition

Algorithm 1 gives the representation of finding the best low-rank approximation of the original tensor by implementing the CP using Alternating Least Squares (ALS).

Algorithm 1 CP-ALS [101]

function CP-ALS($\mathcal{X}, R$)
  Initialize $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$
  repeat
    for $n = 1, \ldots, N$ **do**
      $\mathbf{V} \leftarrow \mathbf{A}^{(1)\top}\mathbf{A}^{(1)} \circledast \cdots \circledast \mathbf{A}^{(n-1)\top}\mathbf{A}^{(n-1)} \circledast \mathbf{A}^{(n+1)\top}\mathbf{A}^{(n+1)} \circledast \cdots \circledast \mathbf{A}^{(N)\top}\mathbf{A}^{(N)}$
      $\mathbf{A}^{(n)} \leftarrow \mathbf{X}^{(n)} \left(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)}\right) \mathbf{V}^{\dagger 1}$
      normalize columns of $\mathbf{A}^{(n)}$ and store norms as λ
    end for
    return $\lambda, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$
  until convergence or maximum iterations
end function

When the factor matrices have been determined there is a deterministic condition to say something about the uniqueness of the CP decomposition, which is given below. There is also a general condition which can tell something about the uniqueness of the decomposition before computing the factor matrices, which is given below for a 4th-order tensor. This general condition can also be used to find an initial choice of the tensor rank.

Theorem A-1.1: A generic condition [166]

Consider a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K \times L}$. Then the CP decomposition is generally unique if

$$R \leq L \quad \text{and} \quad R(R-1) \leq IJK(3IJK - IJ - IK - JK - I - J - K + 3)/4. \quad \text{(A-1)}$$

Theorem A-1.2: Kruskal's condition [167]

Consider the CP decomposition $\mathcal{X} = [\![\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]$. If

$$\sum_{n=1}^{N} k_{A^{(n)}} \geq 2R + (N-1) \quad \text{(A-2)}$$

then $rank(\mathcal{X})$ =R and the CPD is unique.

$k_{\mathbf{A}^{(n)}}$ represent the Kruskal-rank of the factor matrix $\mathbf{A}^{(n)}$. The Kruskal rank is the maximum value of k such that any k columns of the factor matrix $\mathbf{A}^{(n)}$ are linearly independent.

A-2 Tucker decomposition

The well-known and commonly used method to compute the Tucker decomposition of a tensor is a form of Singular Value Decomposition (SVD), generalized to higher order tensors, also known as Multilinear SVD (MLSVD) or Higher-order SVD (HOSVD). The truncation parameters $R_1, R_2, \ldots, R_N$ can be determined by removing low left singular values until the decomposition is a sufficient approximation. Alternatively, a method can be used based on the Eckart-Young theorem [108]. This method truncates the factor matrices by thresholding the truncation error. The theorem states that this error is given by $\epsilon_n^2 = \frac{\epsilon^2 \|\mathcal{X}\|}{N} = \|\mathbf{\Sigma}^{(n)}\|_F^2$, where ϵ is the relative error between the original tensor and the approximation, ϵ_n is the truncation error of the current SVD and $\mathbf{\Sigma}$ is a matrix with the truncated singular values on the diagonal. The R_n is increased until the inequality does not hold. The method is common and is known as sequentially truncated HOSVD (st-HOSVD) [109] given in Algorithm 2.

As mentioned in Subsection 2-3-3 the MLSVD does not give the optimal approximation of the tensor. In addition, an ALS algorithm was proposed using the MLSVD as initialization and minimizing the cost function

$$\begin{aligned} &\min_{\mathcal{G}, \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}} \left\| \mathcal{X} - [\![\mathcal{G}; \mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)}]\!] \right\| \\ &\text{subject to } \mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}, \\ &\qquad \mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R_n} \text{ column-wise orthogonal for n=1, \ldots, N,} \end{aligned} \quad \text{(A-3)}$$

where $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ are the factor matrices and $\mathcal{G}$ the core tensor. The full algorithm known as Higher-order orthogonal iteration (HOOI) or HOSVD-ALS is given in Algorithm 3 and updates each factor matrix by keeping the others fixed.

Algorithm 2 st-HOSVD [109]

function ST-HOSVD($\mathcal{X}, \epsilon$)
  $\mathcal{Y} = \mathcal{X}$
  for $N = 0$ to $N - 1$ **do**
    $[\mathbf{U}, \mathbf{S}, \sim] = \mathsf{SVD}(\mathbf{Y}_{(n)})$
    $R_n = \min\left\{R \mid \sum_{i=R+1}^{I_n} \sigma_i^2 \leq \epsilon^2 \|\mathcal{X}\|^2 / N\right\}$
    $\mathbf{U}^{(n)} = \mathbf{U}(:, 1 : R_n)$
    $\mathcal{Y} = \mathcal{Y} \times_n \mathbf{U}^{(n)\top}$
  end for
  $\mathcal{G} = \mathcal{Y}$
  return $(\mathcal{G}, \mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)})$
end function

Algorithm 3 HOOI [101]

function HOOI($\mathcal{X}, R_1, R_2, \ldots, R_N$)
  initialize $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R}$ for $n = 1, \ldots, N$ using HOSVD
  repeat
    for $n = 1, \ldots, N$ **do**
      $\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{U}^{(1)\top} \cdots \times_{n-1} \mathbf{U}^{(n-1)\top} \times_{n+1} \mathbf{U}^{(n+1)\top} \cdots \times_N \mathbf{U}^{(N)\top}$
      $\mathbf{U}^{(n)} \leftarrow R_n$ leading left singular vectors of $\mathbf{Y}_{(n)}$
    end for
  until convergence or maximum iterations are surpassed
  $\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{U}^{(1)\top} \times_2 \mathbf{U}^{(2)\top} \cdots \times_N \mathbf{U}^{(N)\top}$
  return $\mathcal{G}, \mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)}$
end function

A-3 TT decomposition

The full process of TT-SVD is given in Algorithm 4 using the standard matrix SVD. First, the tensor is put into its mode-1 matricization. Then, the SVD of the new matrix is computed. The unitary matrix $\mathbf{U}$ is then truncated by TT-rank R_n again based on the Eckart-Young theorem [108]. Then the $\mathbf{SV}^\top$ is reshaped, grouping the current indices I_n and the tensor rank R_{n-1}, which will be the new input for the SVD. This process is repeated until the final index I_N.

The TT format is not unique so there might be situations in which the TT-ranks are not optimal for the representation of tensor $\mathcal{X}$. For example, basic linear algebra operations with TTs can result in higher TT-ranks. Lowering the TT-ranks and finding a more optimal TT format without losing accuracy is called *TT-rounding*. The TT-rounding method is usually implemented using either SVD or QR-factorization and is similar to the TT-SVD algorithm (Algorithm 4) but then on the TT-format. The full rounding process is given in Algorithm 5. First, the TT is brought into its site-N-mixed canonical form, putting the norm of the TT in the final tensor core. Then the same procedure as in the TT-SVD is applied where each tensor core is matricized and truncated to a lower rank.

Algorithm 4 TT-SVD [117]

Require: N-dimensional tensor $\mathcal{X}$, prescribed accuracy ϵ

Ensure: Cores $\mathcal{G}^{(1)}, \mathcal{G}^{(2)}, \ldots, \mathcal{G}^{(N)}$ of the TT approximation $\tilde{\mathcal{X}}$ to $\mathcal{X}$ in the TT format with TT ranks R_n equal to the δ-ranks of the unfoldings $\mathbf{X}_n$ of $\mathcal{X}$, where $\delta = \frac{\epsilon}{\sqrt{N-1}}\|\mathcal{X}\|_F$. The computed approximation satisfies $\|\mathcal{X} - \tilde{\mathcal{X}}\|_F \leq \epsilon\|\mathcal{X}\|_F$

initialization: compute truncation parameter $\delta = \frac{\epsilon}{\sqrt{N-1}}\|\mathcal{X}\|_F$
$\mathcal{C} = \mathcal{X}$, $R_0 = 1$
for $n = 1$ to $N-1$ **do**
    $\boldsymbol{C} := \text{reshape}(\mathcal{C}, [R_n I_n, \frac{\text{numel}(\mathcal{C})}{R_{n-1} I_n}]$
    Compute δ-truncated SVD: $\mathbf{C} = \mathbf{USV} + \mathbf{E}, \|\mathbf{E}\|_F \leq \delta, R_n = \text{rank}_\delta(\mathbf{C})$
    New core: $\mathcal{G}^{(n)} := \text{reshape}(\boldsymbol{U}, [R_{n-1}, I_n, R_n])$
    $\boldsymbol{C} := \boldsymbol{S}\boldsymbol{V}^\top$
end for
$\mathcal{G}^{(N)} = \boldsymbol{C}$
Return tensor $\tilde{\mathcal{X}}$ in TT format with cores $\mathcal{G}^{(1)}, \mathcal{G}^{(2)}, \ldots, \mathcal{G}^{(N)}$

Algorithm 5 TT-rounding [17]

Input: Nth-order tensor $\mathcal{X} = \langle\langle \mathcal{G}^{(1)}, \mathcal{G}^{(2)}, \ldots, \mathcal{G}^{(N)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, in a TT format with an overestimated TT rank, $R_{TT} = \{R_1, R_2, \ldots, R_{N-1}\}$, and TT-cores $\mathcal{G} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$, absolute tolerance ϵ and maximum rank R_{max}

Output: Nth-order tensor $\tilde{\mathcal{X}}$ with a reduced TT-rank; the cores are rounded according to the input tolerance ϵ and/or ranks bounded by R_{max} such that $\|\mathcal{X} - \tilde{\mathcal{X}}\|_F \leq \epsilon\|\mathcal{X}\|_F$

initialization $\tilde{\mathcal{X}} = \mathcal{X}$ and $\delta = \frac{\epsilon}{\sqrt{N-1}}$
put the TT $\mathcal{X}$ in site-N-mixed canonical form, placing the norm in the final tensor core:
for $n = 1$ to $N-1$ **do**
    QR decomposition $\mathbf{G}^{(n)}_{(2)} = \mathbf{Q}_n R$ with $\mathbf{G}^{(n)}_{(2)} \in \mathbb{R}^{R_{n-1} I_n \times R_n}$
    Replace cores $\mathbf{G}^{(n)}_{(2)} = \mathbf{Q}_n$ and $\mathbf{G}^{(n+1)}_{(1)} = \mathbf{Q}_n \leftarrow \mathbf{R}\mathbf{G}^{(n+1)}_{(1)}$ with $\mathbf{G}^{(n+1)}_{(1)} \in \mathbb{R}^{R_n \times I_{n+1} R_{n+1}}$
end for
start rounding process
for $n = N$ to 2 **do**
    Perform δ-truncated SVD $\mathbf{G}^{(n)}_{(1)} = \mathbf{U}\text{diag}\{\sigma\}\mathbf{V}^\top$
    Determine minimum rank $\tilde{R}_{n-1}$ such that $\boldsymbol{\Sigma}_{r > R_{n-1}} \sigma_r^2 \leq \delta^2 \|\sigma\|^2$
    Replace cores $\tilde{\mathbf{G}}^{(n-1)}_{(2)} \leftarrow \tilde{\mathbf{G}}^{(n-1)}_{(2)} \tilde{\mathbf{U}}\text{diag}\{\tilde{\sigma}\}$ and $\tilde{\mathbf{G}}^{(n)}_{(1)} = \tilde{\mathbf{V}}^\top$
end for
return Nth-order tensor $\tilde{\mathcal{X}} = \langle\langle \tilde{\mathcal{G}}^{(1)}, \tilde{\mathcal{G}}^{(2)}, \ldots, \tilde{\mathcal{G}}^{(N)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with reduced TT-cores $\tilde{\mathcal{G}}^{(n)} \in \mathbb{R}^{\tilde{R}_{n-1} \times I_n \times \tilde{R}_n}$

Appendix B

Methodology

In this chapter, more extensive elaborations will be given for several parts presented in Chapter 3.

B-1 Tensorly Torch decomposed convolutions

In Section 3-1, the analysis of the Tensorly Torch library was presented, showing key variables and differences between Tensorly Torch and theory. This analysis was done based on the pseudo-code of the Tensorly Torch library which are presented below.

The Parallel Factors (PARAFAC) Canonical Decomposition (CANDECOMP) (CP) decomposition pseudo-code is presented below, which shows that the CP decomposition replaces on convolution layer with four 1D convolutions.

```
def cp_conv(x, cp_tensor, bias=None, stride=1, padding=0, dilation=1):
    """Perform a factorized CP convolution

    Parameters
    ----------
    x : torch.tensor
        tensor of shape (batch_size, C, I_2, I_3, ..., I_N)

    Returns
    -------
    NDConv(x) with a CP kernel
    """
    shape = cp_tensor.shape
    rank = cp_tensor.rank

    batch_size = x.shape[0]
    order = len(shape) - 2

    if isinstance(padding, int):
```

```
        padding = (padding, )*order
    if isinstance(stride, int):
        stride = (stride, )*order
    if isinstance(dilation, int):
        dilation = (dilation, )*order

    # Change the number of channels to the rank
    x_shape = list(x.shape)
    x = x.reshape((batch_size, x_shape[1], -1)).contiguous()

    # First conv == tensor contraction
    # from (in_channels, rank) to (rank == out_channels, in_channels, 1)
    x = F.conv1d(x, tl.transpose(cp_tensor.factors[1]).unsqueeze(2))

    x_shape[1] = rank
    x = x.reshape(x_shape)

    # convolve over non-channels
    for i in range(order):
        # From (kernel_size, rank) to (rank, 1, kernel_size)
        kernel = tl.transpose(cp_tensor.factors[i+2]).unsqueeze(1)
        x = general_conv1d(x.contiguous(), kernel, i+2, stride=stride[i],
    ↪ padding=padding[i], groups=rank)

    # Revert back number of channels from rank to output_channels
    x_shape = list(x.shape)
    x = x.reshape((batch_size, x_shape[1], -1))
    # Last conv == tensor contraction
    # From (out_channels, rank) to (out_channels, in_channels == rank, 1)
    x = F.conv1d(x*cp_tensor.weights.unsqueeze(1).unsqueeze(0), cp_tensor.
    ↪ factors[0].unsqueeze(2), bias=bias)

    x_shape[1] = x.shape[1] # = out_channels
    x = x.reshape(x_shape)

    return x
```

Listing B.1: Factorized CP Convolution

The Tensor Train (TT) decomposition pseudo-code is presented below, which shows that the TT decomposition replaces on convolution layer with four 1D convolutions.

```
def tt_conv(x, tt_tensor, bias=None, stride=1, padding=0, dilation=1):
    """Perform a factorized tt convolution

    Parameters
    ----------
    x : torch.tensor
        tensor of shape (batch_size, C, I_2, I_3, ..., I_N)

    Returns
    -------
    NDConv(x) with an tt kernel
    """
    shape = tt_tensor.shape
    rank = tt_tensor.rank
```

```
15
16      batch_size = x.shape[0]
17      order = len(shape) - 2
18
19      if isinstance(padding, int):
20          padding = (padding, )*order
21      if isinstance(stride, int):
22          stride = (stride, )*order
23      if isinstance(dilation, int):
24          dilation = (dilation, )*order
25
26      # Change the number of channels to the rank
27      x_shape = list(x.shape)
28      x = x.reshape((batch_size, x_shape[1], -1)).contiguous()
29
30      # First conv == tensor contraction
31      # from (1, in_channels, rank) to (rank == out_channels, in_channels, 1)
32      x = F.conv1d(x, tl.transpose(tt_tensor.factors[0], [2, 1, 0]))
33
34      x_shape[1] = x.shape[1] #rank[1]
35      x = x.reshape(x_shape)
36
37      # convolve over non-channels
38      for i in range(order):
39          # From (in_rank, kernel_size, out_rank) to (out_rank, in_rank,
    ↪ kernel_size)
40          kernel = tl.transpose(tt_tensor.factors[i+1], [2, 0, 1])
41          x = general_conv1d(x.contiguous(), kernel, i+2, stride=stride[i],
    ↪ padding=padding[i])
42
43      # Revert back number of channels from rank to output_channels
44      x_shape = list(x.shape)
45      x = x.reshape((batch_size, x_shape[1], -1))
46      # Last conv == tensor contraction
47      # From (rank, out_channels, 1) to (out_channels, in_channels == rank, 1)
48      x = F.conv1d(x, tl.transpose(tt_tensor.factors[-1], [1, 0, 2]), bias=bias)
49
50      x_shape[1] = x.shape[1]
51      x = x.reshape(x_shape)
52
53      return x
```

Listing B.2: Factorized TT Convolution

The Tucker decomposition pseudo-code is presented below, which shows that the Tucker decomposition replaces one convolution layer with two 1D convolutions and one 2D convolution.

```
1  def tucker_conv(x, tucker_tensor, bias=None, stride=1, padding=0, dilation=1):
2      # Extract the rank from the actual decomposition in case it was changed by
   ↪ , e.g., dropout
3      rank = tucker_tensor.rank
4
5      batch_size = x.shape[0]
6      n_dim = tl.ndim(x)
7
8      # Change the number of channels to the rank
```

```
    x_shape = list(x.shape)
    x = x.reshape((batch_size, x_shape[1], -1)).contiguous()

    # This can be done with a tensor contraction
    # First conv == tensor contraction
    # from (in_channels, rank) to (rank == out_channels, in_channels, 1)
    x = F.conv1d(x, tl.transpose(tucker_tensor.factors[1]).unsqueeze(2))

    x_shape[1] = rank[1]
    x = x.reshape(x_shape)

    modes = list(range(2, n_dim+1))
    weight = tl.tenalg.multi_mode_dot(tucker_tensor.core, tucker_tensor.
    ↪ factors[2:], modes=modes)
    x = convolve(x, weight, bias=None, stride=stride, padding=padding)

    # Revert back number of channels from rank to output_channels
    x_shape = list(x.shape)
    x = x.reshape((batch_size, x_shape[1], -1))
    # Last conv == tensor contraction
    # From (out_channels, rank) to (out_channels, in_channels == rank, 1)
    x = F.conv1d(x, tucker_tensor.factors[0].unsqueeze(2), bias=bias)

    x_shape[1] = x.shape[1]
    x = x.reshape(x_shape)

    return x
```

Listing B.3: Factorized Tucker Convolution

B-2 Data processing

As mentioned in Section 3-4, it was discussed that the log file, consisting of the DateTime stamps of each experiment needed to be transferred to a CSV file. This was done by using the following pseudo-code:

```
import csv

# Open the logger log file
with open('Feat.log', 'r') as log_file:
    # Open a new csv file, which will be filled with data
    with open('testen.csv', 'w', newline='') as csv_file:
        # Define the CSV writer
        csv_writer = csv.writer(csv_file)

        # Give headers to the CSV file, with the Datetime stamp and the
    ↪ experiment name
        csv_writer.writerow(['Datetime', 'Name'])

        # Iterate over each line in the log file
        for line in log_file:
            # Based on the ' - ' split the lof file lines
            parts = line.strip().split(' - ')
```

```
17
18              # Check if the line has at least two parts
19              if len(parts) >= 2:
20                  datetime_str = parts[0]  # Get the datetime string
21                  name = parts[-1]  # Get the name of the experiment
22
23                  # Split the datetime string into seperate date and time to
    ↪ remove the milliseconds
24                  datetime_parts = datetime_str.split(' ')
25
26                  # Check if the datetime string has two parts
27                  if len(datetime_parts) == 2:
28                      date, time = datetime_parts
29                      # Split the time part by comma and remove the milliseconds
30                      time = time.split(',')[0]
31
32                      # Combine date and time into a single datetime stamp
33                      datetime_combined = f"{date} {time}"
34
35                      # Write datetime and info to the CSV file
36                      csv_writer.writerow([datetime_combined, name])
37                  else:
38                      print(f"Ignoring line: {line.strip()}")  # Print warning
    ↪ for unexpected line
39              else:
40                  print(f"Ignoring line: {line.strip()}")  # Print warning for
    ↪ unexpected line
```

Listing B.4: Creating a CSV file from the Python logger file.

After, combining the data from the watt meter and the logger file, a big dataset needed to be created with the different periods per run, different experiments with their runs combined under one name, the mean energy consumption, standard deviations and more. The pseudo-code to combine all information is given below and gives a good overview, including descriptions of the used code.

```
1  def get_dicts(testlog,decompose):
2      testlog=testlog
3      # Create an empty lists for the periods
4
5      periods = {}
6      # Find the different start and end pairs, by iterating over all the log
       ↪ rows
7      for index, row in testlog.iterrows():
8
9          if row['Info'].startswith(f'{decompose}-start'): #find a row that
       ↪ starts with start
10             #read out the DateTime stamp
11             start_time = pd.to_datetime(row['Datetime'])
12             # Find the end time corresponding to the start time
13             end_time = pd.to_datetime(testlog.loc[index + 1, 'Datetime'])
14             end_time_round=end_time.replace(second=0)
15
16             #filter the period out of the watt meter measurements between the
       ↪ start and the end time
```

```
            close_strt = data_struct.iloc[(data_struct['Datetime'] - start_time).abs().argsort()[:1]]
            period = data_struct[(data_struct['Datetime'] >=close_strt.iloc[0]['Datetime']) & (data_struct['Datetime'] < end_time_round)]

            #Extract the name of the measurement presented after 'start'
            exp_name = row['Info'].split('start-')[1]  # Extract the text following 'start-'

            # Add filtered data with period name as key to dictionary
            periods[exp_name] = period

            #Add the extra seconds, since measurements are per minute
            periods[exp_name]['extra_t']=(end_time-end_time_round).total_seconds()

    # Create a new dictionary to find different runs of the same measurement
    all_runs = {}

    # Now find different runs for the same measurement by iteratinf over all period dict items
    for exp_name, runs in periods.items():

        # Extract the name of the experiment by removing the ind
        name = exp_name.split('-ind')[0]

        # Check if the type of measurement was alreadt seen in the list of periods
        if name in all_runs:
            # If so append the new run of that experiment
            all_runs[name].append(runs)
        else:
            # If not start a new list under the name of the experiment
            all_runs[name] = [runs]

    # Create a new dictionary for the final results
    experiment_results = {}

    # Find all measurements for each experiment
    for experiment_name, experiment_runs in all_runs.items():
        # Create lists to store means, standard deviations, period lengths, and total energies
        means = []
        stds = []
        period_lengths = []
        total_energies = []
        total_energy_kwh=[]
        # Iterate over each different experiment, over the different runs
        for run in experiment_runs:
            # Calculate median over the power measurments
            power=run['Power (W)'].to_numpy()
            #To be sure check whether length of exp is larger than one, if so remove the first minute
            if len(power)>1:
                median=np.median(power[1:])
            else:
                median=np.median(power)
```

```
66
67              # Calculate period length, combining the seconds after full
    ↪ minutes
68              period_length =len(power)-1+np.mean(run['extra_t']) / 60  #
    ↪ Convert to minutes
69
70              # Calculate the totale energy based on the measured power and
    ↪ period length
71              total_energy=median*period_length*60
72              period_lengths.append(period_length)
73              total_energies.append(total_energy)
74              total_energy_kwh.append(total_energy/(3600*1000))
75
76          # Calculate mean and standard deviation of total energies
77          mean_total_energy = np.mean(total_energies)
78          std_total_energy = np.std(total_energies)
79
80          energy_kwh=np.mean(total_energy_kwh)
81          energy_std=np.std(total_energy_kwh)
82
83          # Add results to the experiment_results dictionary
84          experiment_results[experiment_name] = {
85              'period_lengths': period_lengths,
86              'total_energies': total_energies,
87              'total_energies_kwh':total_energy_kwh,
88              'mean_total_energy': mean_total_energy,
89              'std_total_energy': std_total_energy,
90              'energy(kWh)': energy_kwh,
91              'std_energy(kWh)': energy_std
92          }
93      return experiment_results, all_runs, periods
```

Listing B.5: Processing the data.

B-3 Regression

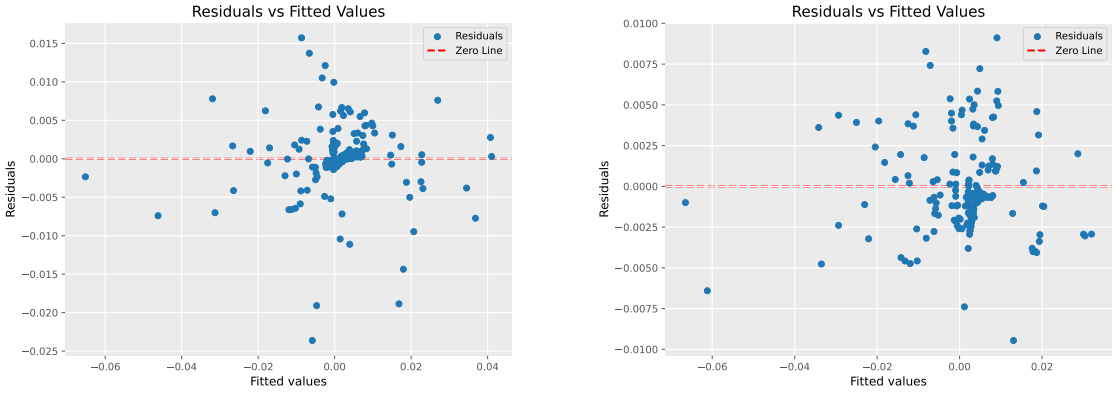
As mentioned in Section 3-5 the use of a regression model means that certain assumptions need to be met:

- Linear relationship: the dependent variables and both independent variables need to show a linear relationship.
- Multivariate normality: The regression model assumes that the residuals are normally distributed.
- No multicollinearity: The different independent variables can not correlate too much with each other. This can be checked by verifying that the VIF is lower than 1.5.
- Homoscedasticity: the variance in the residuals should be consistent across all levels of the independent variables, i.e. there should be no discernible increase in the scatter of the residuals versus the predicted variables.

The assumptions were checked for both Central Processing Unit (CPU) and Graphics Processing Unit (GPU) models and for both the measured and calculated memory.

B-3-1 CPU regression model

The first assumption was based on the linearity between the dependent and independent variables. For this assumption to hold, the residuals vs fitted values in Figure B-1 need to show no clear patterns and consist of spread out scatter. As can be seen, there is a scatter somewhat random, however some indication of a pattern is present, where there is a cluster visible. This might indicate that the relation between the dependent and independent variables is not fully linear and more complex models might be necessary. Figure B-1 indicates



(a) Data based on MAC and measured memory

(b) Data based on Multiply-Accumulate operation (MAC) and calculated memory

Figure B-1: Residuals versus the fitted values of the CPU models based on the measured memory (left) and on the calculated memory (right).

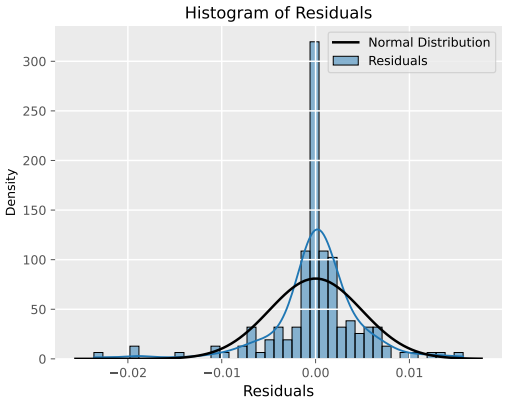
homoscedasticity in the measured memory plot since the variance is consistent across all levels and seems not to depend on the fitted values. For the calculated memory, however, it is less straightforward, since a slight indication of a tunnel is present, where the variance increases with increased fitted values. since this effect seems not heavily present, it is decided to assume homoscedasticity for both.

To check the multicollinearity, the VIF value was calculated using the *statsmodels* library. The VIF was found to be 1.0 for both the measured and calculated memory, which is lower than 1.5. So there is no multicollinearity present in the data.

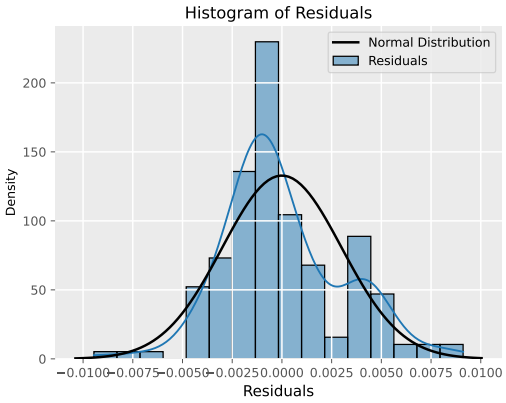
Lastly, the data is assumed to be normally distributed. Looking at the histograms, based on the data for both the memory types, it can be seen that the distribution has some skewness and shows some non-normalities. The distribution comes close to the normality distribution, however, the assumption of normality might influence the results of the regression models.

B-3-2 GPU regression model

The GPU model shows similar results. As can be seen in Figure B-3, the distribution of the data is not exactly normally distributed and shows some skewness and irregularities. For the GPU model, the mean of the residuals seems to be even further away from the zero mean than the CPU mode.



(a) Data based on MAC and measured memory

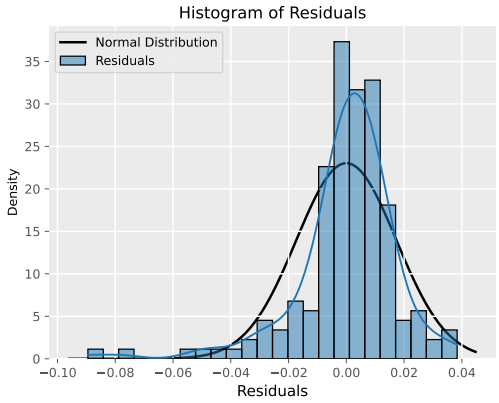


(b) Data based on MAC and calculated memory

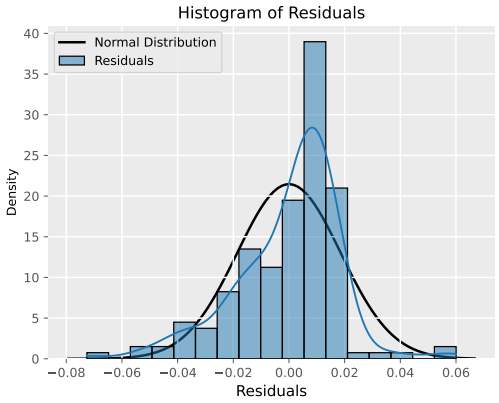
Figure B-2: Histogram of the residuals of the CPU models based on the measured memory (left) and on the calculated memory (right).

For the linear regression, the assumption was made of homoscedasticity and linearity. Looking at Figure B-4, it can be seen that for the GPU the values are not as scattered as for the CPU, showing more clustering. Besides the clustering, now also other more tunnel-shaped patterns are present indicating heteroscedasticity and non-linear behaviour. This might have an impact on the results of the linear and polynomial regression

Lastly, the Variance Inflation Factor (VIF) of the GPU model came to 1.043 which is smaller than 1.5 and thus indicates no multicollinearity.

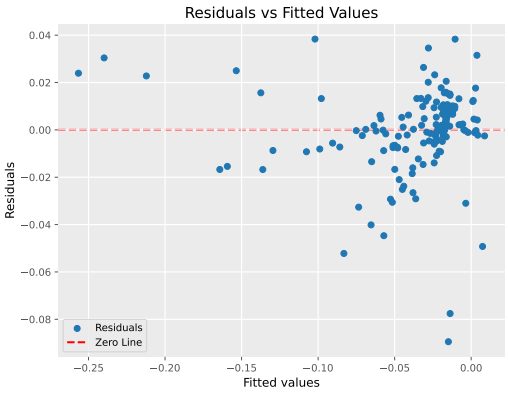


(a) Data based on MAC and measured memory

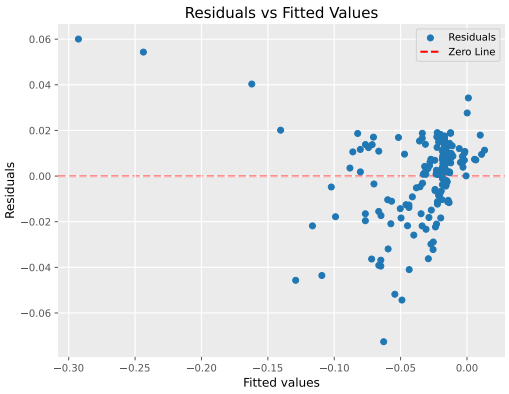


(b) Data based on MAC and calculated memory

Figure B-3: Histogram of the residuals of the GPU models based on the measured memory (left) and on the calculated memory (right).



(a) Data based on MAC and measured memory



(b) Data based on MAC and calculated memory

Figure B-4: Residuals versus the fitted values of the GPU models based on the measured memory (left) and on the calculated memory (right).

B-4 Resnet18 architecture

Table B-1: RESNET18 Layers and Parameters (Page 1)

Layer	Parameters	Description
0	1728	Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
1	128	BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
2	0	ReLU(inplace=True)
3	0	MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
6	36864	Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
7	128	BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
8	0	ReLU(inplace=True)
9	36864	Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
10	128	BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
12	36864	Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
13	128	BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
14	0	ReLU(inplace=True)
15	36864	Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
16	128	BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
19	73728	Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
20	256	BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
21	0	ReLU(inplace=True)
22	147456	Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
23	256	BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
25	8192	Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
26	256	BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
28	147456	Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
29	256	BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
30	0	ReLU(inplace=True)
31	147456	Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
32	256	BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
35	294912	Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
36	512	BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

Table B-2: RESNET18 Layers and Parameters (Page 2)

Layer	Parameters	Description
37	0	ReLU(inplace=True)
38	589824	Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
39	512	BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
41	32768	Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
42	512	BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
44	589824	Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
45	512	BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
46	0	ReLU(inplace=True)
47	589824	Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
48	512	BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
51	1179648	Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
52	1024	BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
53	0	ReLU(inplace=True)
54	2359296	Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
55	1024	BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
57	131072	Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
58	1024	BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
60	2359296	Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
61	1024	BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
62	0	ReLU(inplace=True)
63	2359296	Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
64	1024	BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
65	0	AdaptiveAvgPool2d(output_size=(1, 1))
66	5130	Linear(in_features=512, out_features=10, bias=True)

Appendix C

Experiments

Below the additional details are given for the single convolution experiments and the regression models, elaborating on Chapter 4.

C-1 Single convolution baseline

Below the baseline energy consumption, of the non-decomposed convolution layers are presented to which the measured energy of the decomposed layers is compared.

Table C-1: Baseline energy consumption of Central Processing Unit (CPU) and Graphics Processing Unit (GPU) at different values of S

S	E_{CPU} (kWh)	E_{GPU} (kWh)
192	$5.21 \times 10^{-3} \pm 2.90 \times 10^{-5}$	$9.72 \times 10^{-3} \pm 2.90 \times 10^{-5}$
256	$7.50 \times 10^{-3} \pm 9.00 \times 10^{-5}$	$1.38 \times 10^{-2} \pm 4.00 \times 10^{-6}$
320	$9.74 \times 10^{-3} \pm 3.36 \times 10^{-4}$	$1.81 \times 10^{-2} \pm 3.30 \times 10^{-5}$
384	$1.19 \times 10^{-2} \pm 3.10 \times 10^{-4}$	$2.24 \times 10^{-2} \pm 3.90 \times 10^{-5}$

Table C-2: Baseline energy consumption of CPU and GPU at different values of T

T	E_{CPU} (kWh)	E_{GPU} (kWh)
192	$2.15 \times 10^{-3} \pm 2.90 \times 10^{-5}$	$9.64 \times 10^{-3} \pm 1.3 \times 10^{-5}$
256	$3.11 \times 10^{-3} \pm 2.70 \times 10^{-5}$	$7.66 \times 10^{-3} \pm 3.0 \times 10^{-6}$
320	$4.14 \times 10^{-3} \pm 1.90 \times 10^{-5}$	$1.10 \times 10^{-2} \pm 5.1 \times 10^{-5}$
384	$5.06 \times 10^{-3} \pm 5.40 \times 10^{-5}$	$8.70 \times 10^{-3} \pm 7.0 \times 10^{-6}$

Table C-3: Baseline energy consumption of CPU and GPU at different values of W

W	E_{CPU} (kWh)	E_{GPU} (kWh)
2	$9.57 \times 10^{-3} \pm 4.40 \times 10^{-5}$	$1.05 \times 10^{-2} \pm 6.50 \times 10^{-5}$
4	$1.83 \times 10^{-2} \pm 0.80 \times 10^{-5}$	$1.39 \times 10^{-2} \pm 2.00 \times 10^{-5}$
6	$3.11 \times 10^{-2} \pm 4.00 \times 10^{-5}$	$2.91 \times 10^{-2} \pm 4.70 \times 10^{-5}$
8	$5.62 \times 10^{-2} \pm 4.07 \times 10^{-5}$	$5.75 \times 10^{-2} \pm 5.10 \times 10^{-5}$

Table C-4: Baseline energy consumption of CPU and GPU at different values of d

d	E_{CPU} (kWh)	E_{GPU} (kWh)
1	$3.96 \times 10^{-3} \pm 1.70 \times 10^{-5}$	$1.07 \times 10^{-3} \pm 1.20 \times 10^{-5}$
3	$1.54 \times 10^{-2} \pm 1.90 \times 10^{-5}$	$2.33 \times 10^{-2} \pm 0.50 \times 10^{-5}$
5	$2.49 \times 10^{-2} \pm 0.21 \times 10^{-5}$	$6.66 \times 10^{-2} \pm 9.40 \times 10^{-5}$

C-2 Regression

This section presents summary tables for the eight Ordinary Least Squares (OLS) regression models and the benchmark models, detailing the key statistical values and coefficients. These results provide a comprehensive analysis of the regression models discussed in Subsection 3-2-2, focusing on the comparison between measured and calculated memory, as well as linear versus polynomial models, across both CPU and GPU platforms.

For each table, the R-squared and Adjusted R-squared values indicate the proportion of variance explained by the model. The F-statistic and Prob (F-statistic) assess the overall significance of the model. The coefficients and their Standard Errors, z-values, and p-values provide insight into the importance and impact of each independent variable in the model.

C-2-1 Benchmark model results

This subsection provides the OLS summaries of the linear regression benchmark models, for both CPU (Table C-5) and GPU (Table C-6) data.

Table C-5: OLS Regression Results: CPU, Benchmark Model (MAC Operations Only)

Statistic	**Value**			
R-squared	5.86×10^{-1}			
Adjusted R-squared	5.84×10^{-1}			
F-statistic	4.47×10^{1}			
Prob (F-statistic)	2.89×10^{-10}			
Log-Likelihood	5.11×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-2.00×10^{-3}	1.06×10^{-3}	-1.87	6.16×10^{-2}
MAC Operations (x1)	1.68×10^{-2}	2.52×10^{-3}	6.68	2.34×10^{-11}

Table C-6: OLS Regression Results: GPU, Benchmark Model (MAC Operations Only)

Statistic	**Value**			
R-squared	3.05×10^{-1}			
Adjusted R-squared	3.01×10^{-1}			
F-statistic	2.80×10^{1}			
Prob (F-statistic)	3.59×10^{-7}			
Log-Likelihood	2.72×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-4.38×10^{-2}	4.01×10^{-3}	-1.09×10^{1}	8.48×10^{-28}
MAC Operations (x1)	3.55×10^{-2}	6.72×10^{-3}	5.29	1.23×10^{-7}

C-2-2 CPU results

This subsection provides all OLS summaries of the CPU models, both linear and polynomial, with measured and calculated memory.

Table C-7: OLS Regression Results: CPU, Measured Memory, Linear Model

Statistic	**Value**			
R-squared	8.41×10^{-1}			
Adjusted R-squared	8.40×10^{-1}			
F-statistic	2.92×10^{2}			
Prob (F-statistic)	7.98×10^{-57}			
Log-Likelihood	5.98×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-2.00×10^{-3}	6.58×10^{-4}	-3.01	3.00×10^{-3}
Additional Memory (x1)	-1.19×10^{-2}	1.09×10^{-3}	-1.10×10^{1}	4.90×10^{-28}
Reduced MAC (x2)	1.24×10^{-2}	1.32×10^{-3}	9.40	5.53×10^{-21}

Table C-8: OLS Regression Results: CPU, Measured Memory, Polynomial Model

Statistic	**Value**			
R-squared	8.70×10^{-1}			
Adjusted R-squared	8.66×10^{-1}			
F-statistic	2.08×10^{2}			
Prob (F-statistic)	2.01×10^{-71}			
Log-Likelihood	6.16×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-2.00×10^{-3}	6.00×10^{-4}	-3.30	9.72×10^{-4}
x1	-1.41×10^{-2}	2.01×10^{-3}	-7.03	2.14×10^{-12}
x2	1.15×10^{-2}	2.58×10^{-3}	4.44	8.93×10^{-6}
x3	8.80×10^{-3}	2.96×10^{-3}	2.99	2.83×10^{-3}
x4	7.90×10^{-3}	2.41×10^{-3}	3.28	1.03×10^{-3}
x5	-2.90×10^{-3}	2.34×10^{-3}	-1.25	2.10×10^{-1}

Table C-9: OLS Regression Results: CPU, Calculated Memory, Linear Model

Statistic	**Value**			
R-squared	9.29×10^{-1}			
Adjusted R-squared	9.28×10^{-1}			
F-statistic	8.13×10^{2}			
Prob (F-statistic)	6.36×10^{-90}			
Log-Likelihood	6.70×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-2.00×10^{-3}	4.40×10^{-4}	-4.50	6.75×10^{-6}
Additional Memory (x1)	-1.51×10^{-2}	8.47×10^{-4}	-1.78×10^{1}	8.91×10^{-71}
Reduced MAC (x2)	9.00×10^{-3}	1.07×10^{-3}	8.33	7.90×10^{-17}

Table C-10: OLS Regression Results: CPU, Calculated Memory, Polynomial Model

Statistic	**Value**			
R-squared	9.37×10^{-1}			
Adjusted R-squared	9.35×10^{-1}			
F-statistic	4.11×10^{2}			
Prob (F-statistic)	2.37×10^{-94}			
Log-Likelihood	6.81×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-2.00×10^{-3}	4.18×10^{-4}	-4.73	2.23×10^{-6}
x1	-1.12×10^{-2}	2.18×10^{-3}	-5.14	2.73×10^{-7}
x2	1.66×10^{-2}	2.70×10^{-3}	6.14	8.50×10^{-10}
x3	-5.60×10^{-3}	4.68×10^{-3}	-1.19	2.30×10^{-1}
x4	-9.30×10^{-3}	4.95×10^{-3}	-1.88	5.98×10^{-2}
x5	-6.10×10^{-3}	2.53×10^{-3}	-2.41	1.61×10^{-2}

C-2-3 GPU results

This subsection provides all OLS summaries of the GPU models, both linear and polynomial, with measured and calculated memory.

Table C-11: OLS Regression Results: GPU, Measured Memory, Linear Model

Statistic	**Value**			
R-squared	7.78×10^{-1}			
Adjusted R-squared	7.76×10^{-1}			
F-statistic	2.21×10^{2}			
Prob (F-statistic)	8.27×10^{-49}			
Log-Likelihood	3.74×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-4.38×10^{-2}	2.27×10^{-3}	-1.93×10^{1}	6.82×10^{-83}
Additional Memory (x1)	-5.24×10^{-2}	3.12×10^{-3}	-1.68×10^{1}	4.65×10^{-63}
Reduced MAC (x2)	7.30×10^{-3}	2.58×10^{-3}	2.84	4.57×10^{-3}

Table C-12: OLS Regression Results: GPU, Measured Memory, Polynomial Model

Statistic	**Value**			
R-squared	7.90×10^{-1}			
Adjusted R-squared	7.84×10^{-1}			
F-statistic	3.64×10^{2}			
Prob (F-statistic)	3.84×10^{-90}			
Log-Likelihood	3.79×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-4.40×10^{-2}	2.23×10^{-3}	-1.97×10^{1}	6.17×10^{-86}
x1	-5.90×10^{-2}	7.99×10^{-3}	-7.35	1.94×10^{-13}
x2	-1.60×10^{-2}	1.43×10^{-2}	-1.13	2.60×10^{-1}
x3	8.00×10^{-3}	1.09×10^{-2}	7.23×10^{-1}	4.70×10^{-1}
x4	2.30×10^{-2}	1.63×10^{-2}	1.38	1.70×10^{-1}
x5	1.90×10^{-2}	1.00×10^{-2}	1.86	6.29×10^{-2}

Table C-13: OLS Regression Results: GPU, Calculated Memory, Linear Model

Statistic	**Value**			
R-squared	7.45×10^{-1}			
Adjusted R-squared	7.42×10^{-1}			
F-statistic	2.01×10^{1}			
Prob (F-statistic)	1.36×10^{-8}			
Log-Likelihood	3.62×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-4.38×10^{-2}	2.44×10^{-3}	-1.80×10^{1}	2.86×10^{-72}
Additional Memory (x1)	-4.99×10^{-2}	8.68×10^{-3}	-5.75	9.10×10^{-9}
Reduced MAC (x2)	9.40×10^{-3}	2.68×10^{-3}	3.52	4.32×10^{-4}

Table C-14: OLS Regression Results: GPU, Calculated Memory, Polynomial Model

Statistic	**Value**			
R-squared	8.47×10^{-1}			
Adjusted R-squared	8.43×10^{-1}			
F-statistic	3.41×10^{1}			
Prob (F-statistic)	3.43×10^{-24}			
Log-Likelihood	4.08×10^{2}			
Variable	**Coefficient**	**Standard Error**	**z-value**	**p-value**
Constant	-4.38×10^{-2}	1.90×10^{-3}	-2.30×10^{1}	1.61×10^{-117}
x1	-9.83×10^{-2}	1.16×10^{-2}	-8.47	2.37×10^{-17}
x2	1.24×10^{-2}	4.76×10^{-3}	2.60	9.39×10^{-3}
x3	6.12×10^{-2}	2.70×10^{-2}	2.26	2.37×10^{-2}
x4	7.30×10^{-3}	1.90×10^{-2}	3.84×10^{-1}	7.01×10^{-1}
x5	-4.00×10^{-4}	4.75×10^{-3}	-9.20×10^{-2}	9.26×10^{-1}

Bibliography

[1] D. Wright, C. Igel, G. Samuel, and R. Selvan, *Efficiency is Not Enough: A Critical Perspective of Environmentally Sustainable AI*, Sep. 2023. DOI: 10.48550/arXiv.2309.02065. [Online]. Available: http://arxiv.org/abs/2309.02065.

[2] E. Strubell, A. Ganesh, and A. McCallum, *Energy and Policy Considerations for Deep Learning in NLP*, Jun. 2019. DOI: 10.48550/arXiv.1906.02243. [Online]. Available: http://arxiv.org/abs/1906.02243.

[3] E. Memmel, C. Menzen, J. Schuurmans, F. Wesel, and K. Batselier, *Position: Tensor Networks are a Valuable Asset for Green AI*, May 2024. [Online]. Available: http://arxiv.org/abs/2205.12961.

[4] S. Naumann, M. Dick, E. Kern, and T. Johann, "The GREENSOFT Model: A reference model for green and sustainable software and its engineering," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 4, pp. 294–304, Dec. 2011, ISSN: 2210-5379. DOI: 10.1016/j.suscom.2011.06.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537911000473.

[5] R. Desislavov, F. Martínez-Plumed, and J. Hernández-Orallo, "Trends in AI inference energy consumption: Beyond the performance-vs-parameter laws of deep learning," *Sustainable Computing: Informatics and Systems*, vol. 38, p. 100 857, Apr. 2023, ISSN: 2210-5379. DOI: 10.1016/j.suscom.2023.100857. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537923000124.

[6] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, *Green AI*, Aug. 2019. DOI: 10.48550/arXiv.1907.10597. [Online]. Available: http://arxiv.org/abs/1907.10597.

[7] N. Lenherr, R. Pawlitzek, and B. Michel, "New universal sustainability metrics to assess edge intelligence," *Sustainable Computing: Informatics and Systems*, vol. 31, Sep. 2021, ISSN: 2210-5379. DOI: 10.1016/j.suscom.2021.100580. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537921000718.

[8] R. Verdecchia, J. Sallou, and L. Cruz, "A systematic review of Green AI," en, *WIREs Data Mining and Knowledge Discovery*, vol. 13, no. 4, e1507, 2023, ISSN: 1942-4795. DOI: 10.1002/widm.1507. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1507.

[9] A. Zewe, *Computers that power self-driving cars could be a huge driver of global carbon emissions*, Jan. 2023. [Online]. Available: https://news.mit.edu/2023/autonomous-vehicles-carbon-emissions-0113.

[10] S. Sudhakar, V. Sze, and S. Karaman, "Data Centers on Wheels: Emissions From Computing Onboard Autonomous Vehicles," en, *IEEE Micro*, vol. 43, no. 1, pp. 29–39, Jan. 2023, ISSN: 0272-1732, 1937-4143. DOI: 10.1109/MM.2022.3219803. [Online]. Available: https://ieeexplore.ieee.org/document/9942310/.

[11] Y. He and L. Xiao, "Structured Pruning for Deep Convolutional Neural Networks: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 5, pp. 2900–2919, May 2024, ISSN: 1939-3539. DOI: 10.1109/TPAMI.2023.3334614. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10330640#full-text-section.

[12] ShengJieCheng, QiuxiaZhao, XinYunZhang, N. Yadikar, and K. Ubul, "A Review of Knowledge Distillation in Object Detection," *IEEE Access*, pp. 1–1, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3288692. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10159388.

[13] H. Cheng, M. Zhang, and J. Q. Shi, *A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations*, Aug. 2023. DOI: 10.48550/arXiv.2308.06767. [Online]. Available: http://arxiv.org/abs/2308.06767.

[14] S. M. Kaleem, T. Rouf, G. Habib, T. j. Saleem, and B. Lall, *A Comprehensive Review of Knowledge Distillation in Computer Vision*, Apr. 2024. DOI: 10.48550/arXiv.2404.00936. [Online]. Available: http://arxiv.org/abs/2404.00936.

[15] B. Rokh, A. Azarpeyvand, and A. Khanteymoori, "A Comprehensive Survey on Model Quantization for Deep Neural Networks in Image Classification," *ACM Transactions on Intelligent Systems and Technology*, vol. 14, no. 6, 97:1–97:50, Nov. 2023, ISSN: 2157-6904. DOI: 10.1145/3623402. [Online]. Available: https://dl.acm.org/doi/10.1145/3623402.

[16] M. Wang, Y. Pan, Z. Xu, X. Yang, G. Li, and A. Cichocki, *Tensor Networks Meet Neural Networks: A Survey and Future Perspectives*, May 2023. [Online]. Available: http://arxiv.org/abs/2302.09019.

[17] A. Cichocki, N. Lee, I. Oseledets, A.-H. Phan, Q. Zhao, and D. P. Mandic, "Tensor Networks for Dimensionality Reduction and Large-scale Optimization: Part 1 Low-Rank Tensor Decompositions," English, *Foundations and Trends® in Machine Learning*, vol. 9, no. 4-5, pp. 249–429, Dec. 2016, ISSN: 1935-8237, 1935-8245. DOI: 10.1561/2200000059. [Online]. Available: https://www.nowpublishers.com/article/Details/MAL-059.

[18] X. Zhang, J. Wu, and M. K. Ng, "Multilinear multitask learning by transformed tensor singular value decomposition," vol. 13, 2023, p. 100 479. DOI: https://doi.org/10.1016/j.mlwa.2023.100479. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666827023000324.

[19] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, *Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition*, Apr. 2015. [Online]. Available: http://arxiv.org/abs/1412.6553.

[20] M. Gabor and R. Zdunek, "Convolutional neural network compression via tensor-train decomposition on permuted weight tensor with automatic rank determination," in *Computational Science – ICCS 2022: 22nd International Conference, London, UK, June 21–23, 2022, Proceedings, Part III*, London, United Kingdom: Springer-Verlag, 2022, pp. 654–667, ISBN: 978-3-031-08756-1. DOI: 10.1007/978-3-031-08757-8_54. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1007/978-3-031-08757-8_54.

[21] S. N. Ajani, P. Khobragade, M. Dhone, B. Ganguly, N. Shelke, and N. Parati, "Advancements in Computing: Emerging Trends in Computational Science with Next-Generation Computing," en, *International Journal of Intelligent Systems and Applications in Engineering*, vol. 12, no. 7s, pp. 546–559, 2024, ISSN: 2147-6799. [Online]. Available: https://ijisae.org/index.php/IJISAE/article/view/4159.

[22] R. Desislavov, F. Martínez-Plumed, and J. Hernández-Orallo, "Compute and Energy Consumption Trends in Deep Learning Inference," *Sustainable Computing: Informatics and Systems*, vol. 38, p. 100 857, Apr. 2023, ISSN: 22105379. DOI: 10.1016/j.suscom.2023.100857. [Online]. Available: http://arxiv.org/abs/2109.05472.

[23] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos, "Compute Trends Across Three Eras of Machine Learning," in *2022 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2022, pp. 1–8. DOI: 10.1109/IJCNN55064.2022.9891914. [Online]. Available: https://ieeexplore.ieee.org/document/9891914.

[24] S. Yu, H. Jiang, S. Huang, X. Peng, and A. Lu, "Compute-in-Memory Chips for Deep Learning: Recent Trends and Prospects," *IEEE Circuits and Systems Magazine*, vol. 21, no. 3, pp. 31–56, 2021, ISSN: 1558-0830. DOI: 10.1109/MCAS.2021.3092533. [Online]. Available: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/9512855.

[25] M. B. Giles and I. Reguly, "Trends in high-performance computing for engineering calculations," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 372, no. 2022, p. 20 130 319, Aug. 2014. DOI: 10.1098/rsta.2013.0319. [Online]. Available: https://royalsocietypublishing.org/doi/full/10.1098/rsta.2013.0319.

[26] A. van Wynsberghe, "Sustainable AI: AI for sustainability and the sustainability of AI," en, *AI and Ethics*, vol. 1, no. 3, pp. 213–218, Aug. 2021, ISSN: 2730-5961. DOI: 10.1007/s43681-021-00043-6. [Online]. Available: https://doi.org/10.1007/s43681-021-00043-6.

[27] D. Font Vivanco, J. Freire-González, R. Galvin, *et al.*, "Rebound effect and sustainability science: A review," en, *Journal of Industrial Ecology*, vol. 26, no. 4, pp. 1543–1563, 2022, ISSN: 1530-9290. DOI: 10.1111/jiec.13295. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/jiec.13295.

[28] T. Yigitcanlar, R. Mehmood, and J. M. Corchado, "Green Artificial Intelligence: Towards an Efficient, Sustainable and Equitable Technology for Smart Cities and Futures," en, *Sustainability*, vol. 13, no. 16, p. 8952, Jan. 2021, ISSN: 2071-1050. DOI: 10.3390/su13168952. [Online]. Available: https://www.mdpi.com/2071-1050/13/16/8952.

[29] J. Wang, K. Zhu, and E. Hossain, "Green Internet of Vehicles (IoV) in the 6G Era: Toward Sustainable Vehicular Communications and Networking," *IEEE Transactions on Green Communications and Networking*, vol. 6, no. 1, pp. 391–423, Mar. 2022, ISSN: 2473-2400. DOI: 10.1109/TGCN.2021.3127923. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9614348.

[30] H. Kim, J. Ben-Othman, and L. Mokdad, "Intelligent Terrestrial and Non-Terrestrial Vehicular Networks with Green AI and Red AI Perspectives," en, *Sensors*, vol. 23, no. 2, p. 806, Jan. 2023, ISSN: 1424-8220. DOI: 10.3390/s23020806. [Online]. Available: https://www.mdpi.com/1424-8220/23/2/806.

[31] H. Chen, T. Zhao, C. Li, and Y. Guo, "Green Internet of Vehicles: Architecture, Enabling Technologies, and Applications," *IEEE Access*, vol. 7, pp. 179 185–179 198, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2958175. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8926352.

[32] P. Lv, W. Xu, J. Nie, *et al.*, "Edge Computing Task Offloading for Environmental Perception of Autonomous Vehicles in 6G Networks," *IEEE Transactions on Network Science and Engineering*, vol. 10, no. 3, pp. 1228–1245, May 2023, ISSN: 2327-4697. DOI: 10.1109/TNSE.2022.3211193. [Online]. Available: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/9906430.

[33] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge Computing for Autonomous Driving: Opportunities and Challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, Aug. 2019, ISSN: 1558-2256. DOI: 10.1109/JPROC.2019.2915983. [Online]. Available: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/8744265.

[34] D. Katare, D. Perino, J. Nurmi, M. Warnier, M. Janssen, and A. Y. Ding, "A Survey on Approximate Edge AI for Energy Efficient Autonomous Driving Services," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 4, pp. 2714–2754, 2023, ISSN: 1553-877X. DOI: 10.1109/COMST.2023.3302474. [Online]. Available: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/10213996.

[35] S. Salehi and A. Schmeink, "Data-Centric Green Artificial Intelligence: A Survey," *IEEE Transactions on Artificial Intelligence*, vol. 5, no. 5, pp. 1973–1989, May 2024, ISSN: 2691-4581. DOI: 10.1109/TAI.2023.3315272. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10251541?casa_token=w3cKnPt9w54AAAAA:uZtiNHoWIgfqQ9kE_rHmVO9ecbjWpiJwnA3tvhWrtxMrwg78nV41KNODP98aeWUhfTBwElXtKA.

[36] R. Verdecchia, L. Cruz, J. Sallou, M. Lin, J. Wickenden, and E. Hotellier, "Data-Centric Green AI An Exploratory Empirical Study," in *2022 International Conference on ICT for Sustainability (ICT4S)*, Jun. 2022, pp. 35–45. DOI: 10.1109/ICT4S55073.2022.00015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9830097.

[37] P. Henderson, J. Hu, J. Romoff, E. Brunskill, D. Jurafsky, and J. Pineau, *Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning*, Nov. 2022. DOI: 10.48550/arXiv.2002.05651. [Online]. Available: http://arxiv.org/abs/2002.05651.

[38] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 662–673. DOI: 10.1109/ISCA.2018.00061. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8416863?casa_token=hh3qCAczAQwAAAAA:_96YpO4FNMNZf9lwNyY8Dlslz9dWEA36ewtIcN9BgKlD2SvuBwEdflUaf67gCpggwQvUDcQ3EA.

[39] M. Tan and Q. V. Le, *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*, Sep. 2020. DOI: 10.48550/arXiv.1905.11946. [Online]. Available: http://arxiv.org/abs/1905.11946.

[40] A. G. Howard, M. Zhu, B. Chen, *et al.*, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, Apr. 2017. DOI: 10.48550/arXiv.1704.04861. [Online]. Available: http://arxiv.org/abs/1704.04861.

[41] J. Xu, W. Zhou, Z. Fu, H. Zhou, and L. Li, *A Survey on Green Deep Learning*, Nov. 2021. DOI: 10.48550/arXiv.2111.05193. [Online]. Available: http://arxiv.org/abs/2111.05193.

[42] A. Asperti, D. Evangelista, and M. Marzolla, *Dissecting FLOPs along input dimensions for GreenAI cost estimations*, Jul. 2021. DOI: 10.48550/arXiv.2107.11949. [Online]. Available: http://arxiv.org/abs/2107.11949.

[43] S. Hassantabar, Z. Wang, and N. K. Jha, "SCANN: Synthesis of Compact and Accurate Neural Networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 9, pp. 3012–3025, Sep. 2022, ISSN: 1937-4151. DOI: 10.1109/TCAD.2021.3116470. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9552199?casa_token=89Fs3zNcv8AAAAAA:9Sl9JthMgEYpLUII0fH3rItgJ0K4ycc0XR9qnn8azoWSEtYpGV1BUsVrVS-bYlcelsdg_MbKlw.

[44] V. Mehlin, S. Schacht, and C. Lanquillon, *Towards energy-efficient Deep Learning: An overview of energy-efficient approaches along the Deep Learning Lifecycle*, Feb. 2023. DOI: 10.48550/arXiv.2303.01980. [Online]. Available: http://arxiv.org/abs/2303.01980.

[45] G. Yeung, D. Borowiec, A. Friday, R. Harper, and P. Garraghan, "Towards GPU utilization prediction for cloud deep learning," in *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'20, USA: USENIX Association, Jul. 2020, p. 6.

[46] J. J. K. Park, Y. Park, and S. Mahlke, "Dynamic Resource Management for Efficient Utilization of Multitasking GPUs," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 527–540, ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037707. [Online]. Available: https://dl.acm.org/doi/10.1145/3037697.3037707.

[47] X. Tang and Z. Fu, "CPU–GPU Utilization Aware Energy-Efficient Scheduling Algorithm on Heterogeneous Computing Systems," *IEEE Access*, vol. 8, pp. 58 948–58 958, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2982956. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9045988.

[48] P. Krawczuk, G. Papadimitriou, R. Tanaka, *et al.*, "A Performance Characterization of Scientific Machine Learning Workflows," in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, Nov. 2021, pp. 58–65. DOI: 10.1109/WORKS54523.2021.00013. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9652609.

[49] S. Jiang and S.-G. Wang, "Fast Training Methods and Their Experiments for Deep Learning CNN Models," in *2021 40th Chinese Control Conference (CCC)*, Jul. 2021, pp. 8253–8260. DOI: 10.23919/CCC52363.2021.9549817. [Online]. Available: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/9549817.

[50] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, *Learning Efficient Convolutional Networks through Network Slimming*, Aug. 2017. DOI: 10.48550/arXiv.1708.06519. [Online]. Available: http://arxiv.org/abs/1708.06519.

[51] Z. Liu, T. Zheng, G. Xu, Z. Yang, H. Liu, and D. Cai, *Training-Time-Friendly Network for Real-Time Object Detection*, Nov. 2019. DOI: 10.48550/arXiv.1909.00700. [Online]. Available: http://arxiv.org/abs/1909.00700.

[52] A. Kruglov, G. Succi, and G. Dlamini, "System Energy Consumption Measurement," en, in *Developing Sustainable and Energy-Efficient Software Systems*, ser. SpringerBriefs in Computer Science, A. Kruglov and G. Succi, Eds., Cham: Springer International Publishing, 2023, pp. 27–38, ISBN: 978-3-031-11658-2. DOI: 10.1007/978-3-031-11658-2_3. [Online]. Available: https://doi.org/10.1007/978-3-031-11658-2_3.

[53] J. Getzner, B. Charpentier, and S. Günnemann, *Accuracy is not the only Metric that matters: Estimating the Energy Consumption of Deep Learning Models*, Apr. 2023. [Online]. Available: http://arxiv.org/abs/2304.00897.

[54] E. García-Martín, C. F. Rodrigues, G. Riley, and H. Grahn, "Estimation of energy consumption in machine learning," *Journal of Parallel and Distributed Computing*, vol. 134, pp. 75–88, Dec. 2019, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2019.07.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731518308773.

[55] A. S. Luccioni and A. Hernandez-Garcia, *Counting Carbon: A Survey of Factors Influencing the Emissions of Machine Learning*, Feb. 2023. DOI: 10.48550/arXiv.2302.08476. [Online]. Available: http://arxiv.org/abs/2302.08476.

[56] L. B. Heguerte, A. Bugeau, and L. Lannelongue, "How to estimate carbon footprint when training deep learning models? A guide and review," *Environmental Research Communications*, Sep. 2023, ISSN: 2515-7620. DOI: 10.1088/2515-7620/acf81b. [Online]. Available: http://arxiv.org/abs/2306.08323.

[57] U. Gupta, Y. G. Kim, S. Lee, *et al.*, "Chasing Carbon: The Elusive Environmental Footprint of Computing," *IEEE Micro*, vol. 42, no. 4, pp. 37–47, Jul. 2022, ISSN: 0272-1732. DOI: 10.1109/MM.2022.3163226. [Online]. Available: https://doi.org/10.1109/MM.2022.3163226.

[58] A. Lacoste, A. Luccioni, V. Schmidt, and T. Dandres, *Quantifying the Carbon Emissions of Machine Learning*, Nov. 2019. DOI: 10.48550/arXiv.1910.09700. [Online]. Available: http://arxiv.org/abs/1910.09700.

[59] J. Lee, L. Mukhanov, A. S. Molahosseini, *et al.*, "Resource-Efficient Deep Learning: A Survey on Model-, Arithmetic-, and Implementation-Level Techniques," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–36, Dec. 2023, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3587095. [Online]. Available: http://arxiv.org/abs/2112.15131.

[60] B. Li, X. Jiang, D. Bai, *et al.*, *Full-Cycle Energy Consumption Benchmark for Low-Carbon Computer Vision*, Oct. 2021. DOI: 10.48550/arXiv.2108.13465. [Online]. Available: http://arxiv.org/abs/2108.13465.

[61] S. Mittal and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Computing Surveys*, vol. 47, no. 4, 69:1–69:35, Jul. 2015, ISSN: 0360-0300. DOI: 10.1145/2788396. [Online]. Available: https://dl.acm.org/doi/10.1145/2788396.

[62] S. Han, X. Liu, H. Mao, *et al.*, *EIE: Efficient Inference Engine on Compressed Deep Neural Network*, May 2016. DOI: 10.48550/arXiv.1602.01528. [Online]. Available: http://arxiv.org/abs/1602.01528.

[63] K. M. A. Hasan and S. Chakraborty, "GPU Accelerated Tensor Computation of Hadamard Product for Machine Learning Applications," in *2021 International Conference on Information and Communication Technology for Sustainable Development (ICICT4SD)*, Feb. 2021, pp. 1–5. DOI: 10.1109/ICICT4SD50815.2021.9396980. [Online]. Available: https://ieeexplore.ieee.org/document/9396980.

[64] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008, ISSN: 1558-2256. DOI: 10.1109/JPROC.2008.917757. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4490127.

[65] E. Saleh and C. Shastry, "General Purpose Computing on Graphics Processing Units: From Fixed-Function Pipelines to Programmable Cores," in *2022 4th International Conference on Advances in Computing, Communication Control and Networking*, Dec. 2022, pp. 2064–2075. DOI: 10.1109/ICAC3N56670.2022.10074048. [Online]. Available: https://ieeexplore.ieee.org/document/10074048.

[66] S. Shariar and K. M. Azharul Hasan, "GPU Accelerated Indexing for High Order Tensors in Google Colab," in *2020 IEEE Region 10 Symposium (TENSYMP)*, Jun. 2020, pp. 686–689. DOI: 10.1109/TENSYMP50017.2020.9230789. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9230789.

[67] J. You, J.-W. Chung, and M. Chowdhury, "Zeus: Understanding and Optimizing {GPU} Energy Consumption of {DNN} Training," en, 2023, pp. 119–139, ISBN: 978-1-939133-33-5. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/you.

[68] R. Schoonhoven, B. Veenboer, B. van Werkhoven, and K. J. Batenburg, *Going green: Optimizing GPUs for energy efficiency through model-steered auto-tuning*, Nov. 2022. DOI: 10.48550/arXiv.2211.07260. [Online]. Available: http://arxiv.org/abs/2211.07260.

[69] D. I. Lyakh, T. Nguyen, D. Claudino, E. Dumitrescu, and A. J. McCaskey, "ExaTN: Scalable GPU-Accelerated High-Performance Processing of General Tensor Networks at Exascale," *Frontiers in Applied Mathematics and Statistics*, vol. 8, 2022, ISSN: 2297-4687. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fams.2022.838601.

[70] C. S. Hwang, S. K. Kim, and S. W. Lee, "Mass-Production Memories (DRAM and Flash)," en, in *Atomic Layer Deposition for Semiconductors*, C. S. Hwang, Ed., Boston, MA: Springer US, 2014, pp. 73–122, ISBN: 978-1-4614-8054-9. DOI: 10.1007/978-1-4614-8054-9_4. [Online]. Available: https://doi.org/10.1007/978-1-4614-8054-9_4.

[71] D. Schmidt and N. Wehn, "DRAM power management and energy consumption: A critical assessment," in *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, ser. SBCCI '09, New York, NY, USA: Association for Computing Machinery, Aug. 2009, pp. 1–5, ISBN: 978-1-60558-705-9. DOI: 10.1145/1601896.1601937. [Online]. Available: https://dl.acm.org/doi/10.1145/1601896.1601937.

[72] A. Nekooei and S. Safari, "Compression of Deep Neural Networks based on quantized tensor decomposition to implement on reconfigurable hardware platforms," *Neural Networks*, vol. 150, pp. 350–363, Jun. 2022, ISSN: 0893-6080. DOI: 10.1016/j.neunet.2022.02.024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S089360802200065X.

[73] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105–118, Jan. 2011, ISSN: 1558-2183. DOI: 10.1109/TPDS.2010.107. [Online]. Available: https://ieeexplore.ieee.org/document/5473222.

[74] T. Allen and R. Ge, "Characterizing Power and Performance of GPU Memory Access," in *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, Nov. 2016, pp. 46–53. DOI: 10.1109/E2SC.2016.012. [Online]. Available: https://ieeexplore.ieee.org/document/7830508.

[75] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the Memory Hierarchy of Modern GPUs," en, in *Network and Parallel Computing*, C.-H. Hsu, X. Shi, and V. Salapura, Eds., Berlin, Heidelberg: Springer, 2014, pp. 144–156, ISBN: 978-3-662-44917-2. DOI: 10.1007/978-3-662-44917-2_13.

[76] T. Y. Phuong, D.-Y. Lee, and J.-G. Lee, "Impacts of optimization strategies on performance, power/energy consumption of a GPU based parallel reduction," en, *Journal of Central South University*, vol. 24, no. 11, pp. 2624–2637, Nov. 2017, ISSN: 2227-5223. DOI: 10.1007/s11771-017-3676-5. [Online]. Available: https://doi.org/10.1007/s11771-017-3676-5.

[77] J. Lucas and B. Juurlink, "MEMPower: Data-Aware GPU Memory Power Model," en, in *Architecture of Computing Systems – ARCS 2019*, M. Schoeberl, C. Hochberger, S. Uhrig, J. Brehm, and T. Pionteck, Eds., Cham: Springer International Publishing, 2019, pp. 195–207, ISBN: 978-3-030-18656-2. DOI: 10.1007/978-3-030-18656-2_15.

[78] D. Patterson, J. Gonzalez, Q. Le, *et al.*, *Carbon Emissions and Large Neural Network Training*, Apr. 2021. DOI: 10.48550/arXiv.2104.10350. [Online]. Available: http://arxiv.org/abs/2104.10350.

[79] A. Babuta, B. Gupta, A. Kumar, and S. Ganguli, "Power and energy measurement devices: A review, comparison, discussion, and the future of research," *Measurement*, vol. 172, p. 108 961, Feb. 2021, ISSN: 0263-2241. DOI: 10.1016/j.measurement.2020.108961. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0263224120314354.

[80] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "RAPL in Action: Experiences in Using RAPL for Power Measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no. 2, 9:1–9:26, 2018, ISSN: 2376-3639. DOI: 10.1145/3177754. [Online]. Available: https://dl.acm.org/doi/10.1145/3177754.

[81] M. Jay, V. Ostapenco, L. Lefevre, D. Trystram, A.-C. Orgerie, and B. Fichel, "An experimental comparison of software-based power meters: Focus on CPU and GPU," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2023, pp. 106–118. DOI: 10.1109/CCGrid57682.2023.00020. [Online]. Available: https://ieeexplore.ieee.org/document/10171575.

[82] R. A. Bridges, N. Imam, and T. M. Mintz, "Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods," *ACM Computing Surveys*, vol. 49, no. 3, 41:1–41:27, Sep. 2016, ISSN: 0360-0300. DOI: 10.1145/2962131. [Online]. Available: https://dl.acm.org/doi/10.1145/2962131.

[83] *NVIDIA Management Library (NVML)*. [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml.

[84] L. Lannelongue, J. Grealey, and M. Inouye, *Green Algorithms: Quantifying the carbon footprint of computation*, Dec. 2020. DOI: 10.48550/arXiv.2007.07610. [Online]. Available: http://arxiv.org/abs/2007.07610.

[85] Mlco, *GitHub - mlco2/impact: ML has an impact on the climate. But not all models are born equal. Compute your model's emissions with our calculator and add the results to your paper with our generated latex template.* [Online]. Available: https://github.com/mlco2/impact.

[86] Mlco, *GitHub - mlco2/codecarbon: Track emissions from Compute and recommend ways to reduce their impact on the environment.* [Online]. Available: https://github.com/mlco2/codecarbon.

[87] A. Song, D. Chen, and Z. Zong, "Unveiling the Truth: An Analysis of the Energy and Carbon Footprint of Training an OPT Model using DeepSpeed on the H100 GPU," in *Proceedings of the 14th International Green and Sustainable Computing Conference*, ser. IGSC '23, New York, NY, USA: Association for Computing Machinery, 2024, pp. 36–38. DOI: 10.1145/3634769.3634806. [Online]. Available: https://dl.acm.org/doi/10.1145/3634769.3634806.

[88] A. S. Luccioni, S. Viguier, and A.-L. Ligozat, *Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model*, Nov. 2022. DOI: 10.48550/arXiv.2211.02001. [Online]. Available: http://arxiv.org/abs/2211.02001.

[89] F. S. Martínez, R. Parada, and J. Casas-Roma, "CO2 impact on convolutional network model training for autonomous driving through behavioral cloning," *Advanced Engineering Informatics*, vol. 56, p. 101 968, Apr. 2023, ISSN: 1474-0346. DOI: 10.1016/j.aei.2023.101968. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474034623000964.

[90] V. Liu and Y. Yin, *Green AI: Exploring Carbon Footprints, Mitigation Strategies, and Trade Offs in Large Language Model Training*, Apr. 2024. DOI: 10.48550/arXiv.2404.01157. [Online]. Available: http://arxiv.org/abs/2404.01157.

[91] K. Lottick, S. Susai, S. A. Friedler, and J. P. Wilson, *Energy Usage Reports: Environmental awareness as part of algorithmic accountability*, Dec. 2019. DOI: 10.48550/arXiv.1911.08354. [Online]. Available: http://arxiv.org/abs/1911.08354.

[92] L. F. W. Anthony, B. Kanding, and R. Selvan, *Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models*, Jul. 2020. DOI: 10.48550/arXiv.2007.03051. [Online]. Available: http://arxiv.org/abs/2007.03051.

[93] R. Selvan, N. Bhagwat, L. F. Wolff Anthony, B. Kanding, and E. B. Dam, "Carbon Footprint of Selecting and Training Deep Learning Models for Medical Image Analysis," en, in *Medical Image Computing and Computer Assisted Intervention – MICCAI 2022*, L. Wang, Q. Dou, P. T. Fletcher, S. Speidel, and S. Li, Eds., Cham: Springer Nature Switzerland, 2022, pp. 506–516, ISBN: 978-3-031-16443-9. DOI: 10.1007/978-3-031-16443-9_49.

[94] B. Meulemeester and D. Martens, "How sustainable is "common" data science in terms of power consumption?" *Sustainable Computing: Informatics and Systems*, vol. 38, p. 100 864, Apr. 2023, ISSN: 2210-5379. DOI: 10.1016/j.suscom.2023.100864. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537923000197.

[95] D. Geißler, B. Zhou, M. Liu, S. Suh, and P. Lukowicz, *The Power of Training: How Different Neural Network Setups Influence the Energy Demand*, May 2024. DOI: 10.48550/arXiv.2401.01851. [Online]. Available: http://arxiv.org/abs/2401.01851.

[96] F. Valeye, "Tracarbon — Track your device's carbon footprint - Florian Valeye - Medium," *Blogpost*, Jan. 2023. [Online]. Available: https://medium.com/@florian.valeye/tracarbon-track-your-devices-carbon-footprint-fb051fcc9009.

[97] S. A. Budennyy, V. D. Lazarev, N. N. Zakharenko, *et al.*, "eco2AI: Carbon Emissions Tracking of Machine Learning Models as the First Step Towards Sustainable AI," en, *Doklady Mathematics*, vol. 106, no. 1, S118–S128, Dec. 2022, ISSN: 1531-8362. DOI: 10.1134/S1064562422060230. [Online]. Available: https://doi.org/10.1134/S1064562422060230.

[98] D. A. Maevsky, E. J. Maevskaya, and E. D. Stetsuyk, "Evaluating the RAM Energy Consumption at the Stage of Software Development," en, in *Green IT Engineering: Concepts, Models, Complex Systems Architectures*, ser. Studies in Systems, Decision and Control, V. Kharchenko, Y. Kondratenko, and J. Kacprzyk, Eds., Cham: Springer International Publishing, 2017, pp. 101–121, ISBN: 978-3-319-44162-7. DOI: 10.1007/978-3-319-44162-7_6. [Online]. Available: https://doi.org/10.1007/978-3-319-44162-7_6.

[99] S. Pletenev, V. Chekalina, D. Moskovskiy, M. Seleznev, S. Zagoruyko, and A. Panchenko, "A Computational Study of Matrix Decomposition Methods for Compression of Pre-trained Transformers," in *Proceedings of the 37th Pacific Asia Conference on Language, Information and Computation*, C.-R. Huang, Y. Harada, J.-B. Kim, *et al.*, Eds., Hong Kong, China: Association for Computational Linguistics, Dec. 2023, pp. 723–742. [Online]. Available: https://aclanthology.org/2023.paclic-1.73.

[100] O. Shaikh, J. Saad-Falcon, A. P. Wright, *et al.*, "EnergyVis: Interactively Tracking and Exploring Energy Consumption for ML Models," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '21, New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–7, ISBN: 978-1-4503-8095-9. DOI: 10.1145/3411763.3451780. [Online]. Available: https://dl.acm.org/doi/10.1145/3411763.3451780.

[101] T. G. Kolda and B. W. Bader, "Tensor Decompositions and Applications," en, *SIAM Review*, vol. 51, no. 3, pp. 455–500, Aug. 2009, ISSN: 0036-1445, 1095-7200. DOI: 10.1137/07070111X. [Online]. Available: http://epubs.siam.org/doi/10.1137/07070111X.

[102] A. Joyal and R. Street, "The geometry of tensor calculus, I," en, *Advances in Mathematics*, vol. 88, no. 1, pp. 55–112, Jul. 1991, ISSN: 00018708. DOI: 10.1016/0001-8708(91)90003-P. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/000187089190003P.

[103] X. Chen, "Intuitive understanding of tensors in machine learning," *Blogpost*, Feb. 2023. [Online]. Available: https://medium.com/@xinyu.chen/intuitive-understanding-of-tensors-in-machine-learning-33635c64b596.

[104] R. Penrose, "A generalized inverse for matrices," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 51, no. 3, pp. 406–413, 1955. DOI: 10.1017/S0305004100030401.

[105] G. Tomasi and R. Bro, "A comparison of algorithms for fitting the parafac model," *Computational Statistics Data Analysis*, vol. 50, no. 7, pp. 1700–1734, 2006, ISSN: 0167-9473. DOI: https://doi.org/10.1016/j.csda.2004.11.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167947304003895.

[106] N. (M. Faber, R. Bro, and P. K. Hopke, "Recent developments in candecomp/-parafac algorithms: A critical review," *Chemometrics and Intelligent Laboratory Systems*, vol. 65, no. 1, pp. 119–137, 2003, ISSN: 0169-7439. DOI: https://doi.org/10.1016/S0169-7439(02)00089-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169743902000898.

[107] R. Pajarola, S. K. Suter, R. Ballester-Ripoll, and H. Yang, "Tensor Approximation for Multidimensional and Multivariate Data," en, in *Anisotropy Across Fields and Scales*, E. Özarslan, T. Schultz, E. Zhang, and A. Fuster, Eds., ser. Mathematics and Visualization, Cham: Springer International Publishing, 2021, pp. 73–98, ISBN: 978-3-030-56215-1. DOI: 10.1007/978-3-030-56215-1_4.

[108] G. Golub, A. Hoffman, and G. Stewart, "A generalization of the Eckart-Young-Mirsky matrix approximation theorem," en, *Linear Algebra and its Applications*, vol. 88-89, pp. 317–327, Apr. 1987, ISSN: 00243795. DOI: 10.1016/0024-3795(87)90114-5. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0024379587901145.

[109] Z. Li, Q. Fang, and G. Ballard, "Parallel Tucker Decomposition with Numerically Accurate SVD," in *Proceedings of the 50th International Conference on Parallel Processing*, ser. ICPP '21, New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–11, ISBN: 978-1-4503-9068-2. DOI: 10.1145/3472456.3472472. [Online]. Available: https://dl.acm.org/doi/10.1145/3472456.3472472.

[110] R. Minster, Z. Li, and G. Ballard, *Parallel randomized tucker decomposition algorithms*, 2023.

[111] M. Che, Y. Wei, and H. Yan, *Efficient algorithms for tucker decomposition via approximate matrix multiplication*, 2023.

[112] A. H. Phan and A. Cichocki, "Fast and efficient algorithms for nonnegative tucker decomposition," in *Advances in Neural Networks - ISNN 2008*, F. Sun, J. Zhang, Y. Tan, J. Cao, and W. Yu, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 772–782, ISBN: 978-3-540-87734-9.

[113] L. T. Thanh, K. Abed-Meraim, N. L. Trung, and R. Boyer, "Adaptive algorithms for tracking tensor-train decomposition of streaming tensors," in *2020 28th European Signal Processing Conference (EUSIPCO)*, 2021, pp. 995–999. DOI: 10.23919/Eusipco47968.2020.9287780.

[114] D. Aksoy, D. J. Gorsich, S. Veerapaneni, and A. A. Gorodetsky, *An incremental tensor train decomposition algorithm*, 2023.

[115] T. Shi, M. Ruth, and A. Townsend, *Parallel algorithms for computing the tensor-train decomposition*, 2021.

[116] S. Miron, Y. Zniyed, R. Boyer, *et al.*, "Tensor methods for multisensor signal processing," en, *IET Signal Processing*, vol. 14, no. 10, pp. 693–709, 2020, ISSN: 1751-9683. DOI: 10.1049/iet-spr.2020.0373. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1049/iet-spr.2020.0373.

[117] I. V. Oseledets, "Tensor-Train Decomposition," en, *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, Jan. 2011, ISSN: 1064-8275, 1095-7197. DOI: 10.1137/090752286. [Online]. Available: http://epubs.siam.org/doi/10.1137/090752286.

[118] D. Ghimire, D. Kil, and S.-h. Kim, "A Survey on Efficient Convolutional Neural Networks and Hardware Acceleration," en, *Electronics*, vol. 11, no. 6, p. 945, Jan. 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11060945. [Online]. Available: https://www.mdpi.com/2079-9292/11/6/945.

[119] P. Dhilleswararao, S. Boppu, M. S. Manikandan, and L. R. Cenkeramaddi, "Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey," *IEEE Access*, vol. 10, pp. 131 788–131 828, 2022, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3229767. [Online]. Available: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/9988986.

[120] B. Jacob, S. Kligys, B. Chen, *et al.*, *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*, Dec. 2017. DOI: 10.48550/arXiv.1712.05877. [Online]. Available: http://arxiv.org/abs/1712.05877.

[121] C.-C. Tsai, "A Hardware-friendly Quantization and Model Fine Tuning with STEBC for Object Detection," in *2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2023, pp. 1040–1044. DOI: 10.1109/MWSCAS57524.2023.10405997. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10405997.

[122] H. Zhang, L. Liu, Y. Huang, X. Lei, L. Tong, and B. Wen, "InstKD: Towards Lightweight 3D Object Detection With Instance-Aware Knowledge Distillation," *IEEE Transactions on Intelligent Vehicles*, pp. 1–13, 2024, ISSN: 2379-8904. DOI: 10.1109/TIV.2024.3401461. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10531046.

[123] C. Deng, Z. Deng, Y. Han, D. Jing, and H. Zhang, "GradQuant: Low-Loss Quantization for Remote-Sensing Object Detection," *IEEE Geoscience and Remote Sensing Letters*, vol. 20, pp. 1–5, 2023, ISSN: 1558-0571. DOI: 10.1109/LGRS.2023.3308582. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10230297.

[124] C. Yang, Z. Lin, Z. Lan, R. Chen, L. Wei, and Y. Liu, "Evolutionary channel pruning for real-time object detection," *Knowledge-Based Systems*, vol. 287, p. 111 432, Mar. 2024, ISSN: 0950-7051. DOI: 10.1016/j.knosys.2024.111432. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705124000674.

[125] D. Ham, J. Jeong, J.-K. Park, *et al.*, *Scalable Object Detection on Embedded Devices Using Weight Pruning and Singular Value Decomposition*, Mar. 2023. DOI: 10.48550/arXiv.2303.02735. [Online]. Available: http://arxiv.org/abs/2303.02735.

[126] Y. Zou and C. Liu, "A light-weight object detection method based on knowledge distillation and model pruning for seam tracking system," *Measurement*, vol. 220, p. 113 438, Oct. 2023, ISSN: 0263-2241. DOI: 10.1016/j.measurement.2023.113438. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0263224123010023.

[127] M. Shen, L. Mao, J. Chen, *et al.*, "Hardware-Aware Latency Pruning for Real-Time 3D Object Detection," in *2023 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2023, pp. 1–6. DOI: 10.1109/IV55152.2023.10186732. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10186732.

[128] L. Liebenwein, C. Baykal, B. Carter, D. Gifford, and D. Rus, *Lost in Pruning: The Effects of Pruning Neural Networks beyond Test Accuracy*, Mar. 2021. DOI: 10.48550/arXiv.2103.03014. [Online]. Available: http://arxiv.org/abs/2103.03014.

[129] G. Yang, Y. Tang, Z. Wu, J. Li, J. Xu, and X. Wan, "DMKD: Improving Feature-Based Knowledge Distillation for Object Detection Via Dual Masking Augmentation," in *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 2024, pp. 3330–3334. DOI: 10.1109/ICASSP48485.2024.10446978. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10446978.

[130] J. Rao, X. Meng, L. Ding, *et al.*, "Parameter-Efficient and Student-Friendly Knowledge Distillation," *IEEE Transactions on Multimedia*, vol. 26, pp. 4230–4241, 2024, ISSN: 1941-0077. DOI: 10.1109/TMM.2023.3321480. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10272648.

[131] U. Ojha, Y. Li, A. S. Rajan, Y. Liang, and Y. J. Lee, *What Knowledge Gets Distilled in Knowledge Distillation?* Nov. 2023. DOI: 10.48550/arXiv.2205.16004. [Online]. Available: http://arxiv.org/abs/2205.16004.

[132] X. Ou, Z. Chen, C. Zhu, and Y. Liu, *Low Rank Optimization for Efficient Deep Learning: Making A Balance between Compact Architecture and Fast Training*, Mar. 2023. DOI: 10.48550/arXiv.2303.13635. [Online]. Available: http://arxiv.org/abs/2303.13635.

[133] Z. Chen, M. Gong, L. Ge, and B. Du, "Compressed Self-Attention for Deep Metric Learning with Low-Rank Approximation," en, in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, Yokohama, Japan: International Joint Conferences on Artificial Intelligence Organization, Jul. 2020, pp. 2058–2064, ISBN: 978-0-9992411-6-5. DOI: 10.24963/ijcai.2020/285. [Online]. Available: https://www.ijcai.org/proceedings/2020/285.

[134] Q. Xie, Q. Zhao, Z. Xu, and D. Meng, "Color and direction-invariant nonlocal self-similarity prior and its application to color image denoising," en, *Science China Information Sciences*, vol. 63, no. 12, p. 222 101, Nov. 2020, ISSN: 1869-1919. DOI: 10.1007/s11432-020-2880-3. [Online]. Available: https://doi.org/10.1007/s11432-020-2880-3.

[135] X. Li, Y. Ye, and X. Xu, "Low-Rank Tensor Completion with Total Variation for Visual Data Inpainting," en, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, Feb. 2017, ISSN: 2374-3468. DOI: 10.1609/aaai.v31i1.10776. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10776.

[136] H. Xue, S. Zhang, and D. Cai, "Depth Image Inpainting: Improving Low Rank Matrix Completion With Low Gradient Regularization," eng, *IEEE transactions on image processing*, vol. 26, no. 9, pp. 4311–4320, Sep. 2017, ISSN: 1941-0042. DOI: 10.1109/tip.2017.2718183. [Online]. Available: https://doi.org/10.1109/TIP.2017.2718183.

[137] Y. Liu, Z. Long, and C. Zhu, "Image Completion Using Low Tensor Tree Rank and Total Variation Minimization," *IEEE Transactions on Multimedia*, vol. PP, pp. 1–1, Jul. 2018. DOI: 10.1109/TMM.2018.2859026.

[138] Y. Panagakis, J. Kossaifi, G. G. Chrysos, *et al.*, "Tensor Methods in Computer Vision and Deep Learning," en, *Proceedings of the IEEE*, vol. 109, no. 5, pp. 863–890, May 2021, ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2021.3074329. [Online]. Available: https://ieeexplore.ieee.org/document/9420085/.

[139] M. A. O. Vasilescu and D. Terzopoulos, "Multilinear Analysis of Image Ensembles: TensorFaces," en, in *Computer Vision — ECCV 2002*, A. Heyden, G. Sparr, M. Nielsen, and P. Johansen, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 447–460, ISBN: 978-3-540-47969-7. DOI: 10.1007/3-540-47969-4_30.

[140] I. Macedo, E. Vital Brazil, and L. Velho, "Expression Transfer between Photographs through Multilinear AAM's," Oct. 2006, pp. 239–246. DOI: 10.1109/SIBGRAPI.2006.18.

[141] S. Ge, Z. Luo, S. Zhao, X. Jin, and X.-Y. Zhang, "Compressing deep neural networks for efficient visual inference," en, in *2017 IEEE International Conference on Multimedia and Expo (ICME)*, Hong Kong, Hong Kong: IEEE, Jul. 2017, pp. 667–672, ISBN: 978-1-5090-6067-2. DOI: 10.1109/ICME.2017.8019465. [Online]. Available: http://ieeexplore.ieee.org/document/8019465/.

[142] J. Kossaifi, A. Bulat, G. Tzimiropoulos, and M. Pantic, "T-Net: Parametrizing Fully Convolutional Nets With a Single High-Order Tensor," English, IEEE Computer Society, Jun. 2019, pp. 7814–7823, ISBN: 978-1-72813-293-8. DOI: 10.1109/CVPR.2019.00801. [Online]. Available: https://www.computer.org/csdl/proceedings-article/cvpr/2019/329300h814/1gyrhm6sn4I.

[143] S. Indolia, A. K. Goswami, S. P. Mishra, and P. Asopa, "Conceptual Understanding of Convolutional Neural Network- A Deep Learning Approach," *Procedia Computer Science*, International Conference on Computational Intelligence and Data Science, vol. 132, pp. 679–688, Jan. 2018, ISSN: 1877-0509. DOI: 10.1016/j.procs.2018.05.069. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918308019.

[144] T.-J. Yang, Y.-H. Chen, J. Emer, and V. Sze, "A method to estimate the energy consumption of deep neural networks," en, in *2017 51st Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA: IEEE, Oct. 2017, pp. 1916–1920, ISBN: 978-1-5386-1823-3. DOI: 10.1109/ACSSC.2017.8335698. [Online]. Available: http://ieeexplore.ieee.org/document/8335698/.

[145] R. Lv, D. Wang, J. Zheng, Y. Xie, and Z.-X. Yang, "Realistic acceleration of neural networks with fine-grained tensor decomposition," *Neurocomputing*, vol. 512, pp. 52–68, Nov. 2022, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2022.09.057. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231222011377.

[146] A. Murua, R. Ramakrishnan, X. Li, R. H. Yang, and V. P. Nia, *Tensor train decompositions on recurrent networks*, Jun. 2020. [Online]. Available: http://arxiv.org/abs/2006.05442.

[147] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "TIE: Energy-efficient tensor train-based inference engine for deep neural network," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 264–278, ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322258. [Online]. Available: https://dl.acm.org/doi/10.1145/3307650.3322258.

[148] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, "Tensorizing neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15, Cambridge, MA, USA: MIT Press, Dec. 2015, pp. 442–450.

[149] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation," in *Advances in Neural Information Processing Systems*, vol. 27, Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2014/hash/2afe4567e1bf64d32a5527244d104cea-Abstract.html.

[150] M. Astrid and S.-I. Lee, "Cp-decomposition with tensor power method for convolutional neural networks compression," in *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2017, pp. 115–118. DOI: 10.1109/BIGCOMP.2017.7881725.

[151] A.-H. Phan, P. Tichavský, and A. Cichocki, "Error Preserving Correction: A Method for CP Decomposition at a Target Error Bound," *IEEE Transactions on Signal Processing*, vol. 67, no. 5, pp. 1175–1190, Mar. 2019, ISSN: 1941-0476. DOI: 10.1109/TSP.2018.2887192. [Online]. Available: https://ieeexplore.ieee.org/document/8579207.

[152] L. Veeramacheneni, M. Wolter, R. Klein, and J. Garcke, "Canonical convolutional neural networks," in *2022 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2022, pp. 1–8. DOI: 10.1109/IJCNN55064.2022.9892607. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9892607.

[153] Y. Liu and M. K. Ng, "Deep neural network compression by Tucker decomposition with nonlinear response," *Knowledge-Based Systems*, vol. 241, p. 108 171, Apr. 2022, ISSN: 0950-7051. DOI: 10.1016/j.knosys.2022.108171. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705122000326.

[154] P. Hao, X. Li, and F. Wu, "Learning Tucker Compression for Deep CNN," in *2022 Data Compression Conference (DCC)*, Mar. 2022, pp. 332–341. DOI: 10.1109/DCC52660.2022.00041. [Online]. Available: https://ieeexplore.ieee.org/document/9810720.

[155] M. Gabor and R. Zdunek, "Compressing convolutional neural networks with hierarchical Tucker-2 decomposition," *Applied Soft Computing*, vol. 132, p. 109 856, Jan. 2023, ISSN: 1568-4946. DOI: 10.1016/j.asoc.2022.109856. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S156849462200905X.

[156] Z. Zhong, F. Wei, Z. Lin, and C. Zhang, "ADA-Tucker: Compressing deep neural networks via adaptive dimension adjustment tucker decomposition," *Neural Networks*, vol. 110, pp. 104–115, Feb. 2019, ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.10.016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608018303010.

[157] L. Huyan, Y. Li, D. Jiang, *et al.*, "Remote Sensing Imagery Object Detection Model Compression via Tucker Decomposition," en, *Mathematics*, vol. 11, no. 4, p. 856, Jan. 2023, ISSN: 2227-7390. DOI: 10.3390/math11040856. [Online]. Available: https://www.mdpi.com/2227-7390/11/4/856.

[158] T. Garipov, D. Podoprikhin, A. Novikov, and D. Vetrov, *Ultimate tensorization: Compressing convolutional and FC layers alike*, Nov. 2016. DOI: 10.48550/arXiv.1611.03214. [Online]. Available: http://arxiv.org/abs/1611.03214.

[159] A. Taskynov, V. Korviakov, I. Mazurenko, and Y. Xiong, *Tensor Yard: One-Shot Algorithm of Hardware-Friendly Tensor-Train Decomposition for Convolutional Neural Networks*, Aug. 2021. DOI: 10.48550/arXiv.2108.04029. [Online]. Available: http://arxiv.org/abs/2108.04029.

[160] J. Qi, C.-H. H. Yang, P.-Y. Chen, and J. Tejedor, "Exploiting Low-Rank Tensor-Train Deep Neural Networks Based on Riemannian Gradient Descent With Illustrations of Speech Processing," en, *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 31, pp. 633–642, 2023, ISSN: 2329-9290, 2329-9304. DOI: 10.1109/TASLP.2022.3231714. [Online]. Available: https://ieeexplore.ieee.org/document/10006412/.

[161] X. Ou, P. Yan, Y. Zhang, *et al.*, "Moving Object Detection Method via ResNet-18 With Encoder–Decoder Structure in Complex Scenes," *IEEE Access*, vol. 7, pp. 108 152–108 160, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2931922. [Online]. Available: https://ieeexplore.ieee.org/document/8781779/figures#figures.

[162] D. Anderson, "Algorithms for minimization without derivatives," *IEEE Transactions on Automatic Control*, vol. 19, no. 5, pp. 632–633, Oct. 1974, ISSN: 0018-9286, 1558-2523, 2334-3303. DOI: 10.1109/TAC.1974.1100629. [Online]. Available: https://ieeexplore.ieee.org/document/1100629/.

[163] J. T. Schuurmans, K. Batselier, and J. F. P. Kooij, *How Informative is the Approximation Error from Tensor Decomposition for Neural Network Compression?* Aug. 2023. [Online]. Available: http://arxiv.org/abs/2305.05318.

[164] M. A. Mansournia, M. Nazemipour, A. I. Naimi, G. S. Collins, and M. J. Campbell, "Reflection on modern methods: demystifying robust standard errors for epidemiologists," *International Journal of Epidemiology*, vol. 50, no. 1, pp. 346–351, Dec. 2020, ISSN: 0300-5771. DOI: 10.1093/ije/dyaa260. [Online]. Available: https://doi.org/10.1093/ije/dyaa260.

[165] Y. Kim, J. Lee, J.-S. Kim, H. Jei, and H. Roh, "Comprehensive techniques of multi-GPU memory optimization for deep learning acceleration," en, *Cluster Computing*, vol. 23, no. 3, pp. 2193–2204, Sep. 2020, ISSN: 1573-7543. DOI: 10.1007/s10586-019-02974-6. [Online]. Available: https://doi.org/10.1007/s10586-019-02974-6.

[166] L. De Lathauwer and J. Castaing, "Tensor-based techniques for the blind separation of DS–CDMA signals," *Signal Processing*, Tensor Signal Processing, vol. 87, no. 2, pp. 322–336, Feb. 2007, ISSN: 0165-1684. DOI: 10.1016/j.sigpro.2005.12.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0165168406001745.

[167] J. B. Kruskal, "Three-way arrays: Rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics," *Linear Algebra and its Applications*, vol. 18, no. 2, pp. 95–138, Jan. 1977, ISSN: 0024-3795. DOI: 10.1016/0024-3795(77)90069-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0024379577900696.

Glossary

List of Acronyms

GPU	Graphics Processing Unit
CPU	Central Processing Unit
TPU	Tensor Processing Unit
ALU	Arithmetic Logic Unit
RAM	Random-Access Memory
DRAM	Dynamic Random-Access Memory
SRAM	Static Random-Access Memory
CU	Control Unit
GPGPU	General-Purpose Graphics Processing Unit
IPMI	Intelligent Platform Management Interfaces
ML	Machine learning
DL	Deep learning
DNN	Deep Neural Networks
API	Application Programming Interface
TDP	Thermal Design Power
NVML	Nvidia Management Library
RAPL	Running Average Power Limit
PUE	Power Usage Effectiveness
PARAFAC	Parallel Factors
CANDECOMP	Canonical Decomposition
CP	Parallel Factors (PARAFAC) Canonical Decomposition (CANDECOMP)
ALS	Alternating Least Squares
3MFA	Three-mode factor analysis

SVD Singular Value Decomposition

3MPCA Three-mode PCA

HOSVD Higher-order Singular Value Decomposition (SVD)

PCA Principle component analysis

MLSVD Multilinear SVD

HOOI Higher-order orthogonal iteration

TT Tensor Train

CNN Convolutional Neural Network

PAPI Performance API

GB GigaBytes

FPO floating point operations

AI Artificial Intelligence

AV autonomous vehicle

NLS nonlinear least squares

EPC Error Preserving Correction

VBMF Variational Bayesian Matrix Factorization

ADA-Tucker Adaptive Dimension Adjustment Tucker decomposition

SCADA-Tucker Shared Core Adaptive Dimension Adjustment Tucker decomposition (ADA-Tucker)

RNN Recurrent neural network

AAM's Active Appearance Models

MAC Multiply-Accumulate operation

MAD Multiply-Add operation

KD Knowledge distillation

FPGA Field-Programmable Gate Array

RMSE Root-mean-square error

VIF Variance Inflation Factor

OLS Ordinary Least Squares

SE Standard Error