UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

BRENO FANCHIOTTI ZANCHETTA

Deep Learning in Edge: Evaluation of Models and Frameworks in ARM Architecture

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Claudio R. Geyer

Porto Alegre
February 2022

CIP — CATALOGING-IN-PUBLICATION

Zanchetta, Breno Fanchiotti

Deep Learning in Edge: Evaluation of Models and Frameworks in ARM Architecture / Breno Fanchiotti Zanchetta. – Porto Alegre: PPGC da UFRGS, 2022.

122 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Claudio R. Geyer.

I. Geyer, Claudio R.. II. Título.

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

AGRADECIMENTOS

Agradeço a Deus por mais um dia de vida.

Agradeço à minha família, pelo amparo à todas as minhas necessidades materiais, principalmente minha Mãe Jucilene, que nunca desistiu e nem duvidou de mim.

Agradeço aos colegas do Grupo de Processamento Paralelo e Distribuído (GPPD): Prof. Dr. Claudio R. Geyer por sua orientação. Dr. Julio dos Anjos Santos, por sua co-orientação, paciência e votos de confiança nos momentos de turbulência. MsC. Kassiano Matteussi, pelas inúmeras lições profissionais, técnicas e acadêmicas. Também por sua dedicação, paciência e amizade, que ajudaram muito a persistir nessa pós-graduação.

Extendo meus agradecimentos à Universidade Federal do Rio Grande do Sul (UFRGS), pela hegemônica conduta em pesquisas científicas, que é exemplo primordial da importância da educação no Rio Grande do Sul, no Brasil e no mundo.

Saúdo o Intituto de Informática da UFRGS (INF) pela organização, determinação, e primosia com que tem auxiliado os estudantes de pós-graduação. Sem vocês o caminho da pós-graduação se tornaria muito maior.

Agradeço aos funcionários de segurança, limpeza, recepção e a todos que eu não conseguir lembrar agora, mas que foram fundamentais para o exercício de minha pesquisa no dia-a-dia. Agradeço também aos professores, pesquisadores, técnicos, servidores, alunos, médicos e psicológos que participaram de minha caminhada, ajudando-me desde assuntos pequenos até às resoluções de situações complexas.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) por me agraciar com uma bolsa de mestrado com dedicação exclusiva pelo período de 24 meses. Compreendo que, em momentos de tensão política e regressão de valores, estudar tem se tornado um previlégio -infelizmente-. Por isso, extendo meus agradecimentos aos milhões de cidadãos que permitiram esta bolsa por meio de suas contribuições.

Agradeço também ao programa da NVIDIA Grant pela doação de duas unidades de NVIDIA Jetson TX2, permitindo assim a existência de meus experimentos.

Agradeço à minha amiga de graduação Stéphanie Ames por me auxiliar com a escrita final deste trabalho.

Agradeço à minha grande amiga Mariana de Nunes Flores e Silva pelo companherismo, por ter passado as dificuldades comigo sempre acreditando na minha vitória.

Agradeço à União do Vegetal, por me devolver à vontade e à fé que me permitiram estar vivo até hoje.

ABSTRACT

The boom and popularization of edge devices have molded its market due to stiff competition that provides better functionalities at low energy costs. The ARM architecture has been unanimously unopposed in the huge market segment of smartphones and still makes a presence beyond that: in drones, surveillance systems, cars, and robots. Also, it has been used successfully for the development of solutions for chains that supply food, fuel, and other services. Up until recently, ARM did not show much promise for high-level computation, i.e., thanks to its limited RISC instruction set, it was considered power efficient but weak in performance compared to x86 architecture. However, most recent advancements in ARM architecture pivoted that inflection point up thanks to the introduction of embedded GPUs with DMA into LPDDR memory boards. Since this development in boards such as NVIDIA TK1, NVIDIA Jetson TX1, and NVIDIA TX2, perhaps it finally became feasible to study and perform more challenging parallel and distributed workloads directly on a RISC-based architecture. On the other hand, the novelty of this technology poses a fundamental question of whether these boards are gaining a meaningful ratio between processing power and power consumption over conventional architectures or if they are bound to have reached their limitations. This work explores the Parallel Processing of Deep Learning on embedded GPUs of NVIDIA Jetson TX2 to evaluate the question above comprehensively. Thus, it uses 4 ARM boards, with 2 Deep Learning frameworks, 7 CNN models, and one medium-sized dataset combined into six board settings to conduct experiments. The experiments were conducted under similar environments, all built from the source. Altogether, the experiments ran for a total of 4,804 hours and revealed a slight advantage for MxNet on GPU-reliant training and a PyTorch overall advantage in total execution time and power, but especially for CPU-only executions. The experiments also showed that the NVIDIA Jetson TX2 already makes feasible some complex workloads directly on its SoC.

Keywords: Deep Learning. Embedded-GPU Devices. ARM Evaluation.

LIST OF ABBREVIATIONS AND ACRONYMS

AE	Auto Encoders
AI	Artificial Intelligence
AM	Application Metric
ARM	Advanced RISC Machine
CPU	Central Processing Unit
CNN	Convolutional Neural Networks
DMA	Direct Memory Access
DS	Data set
DL	Deep Learning
EIE	Efficient Inference Engine
FFT	Fast Fourier Transform
FGPA	Field-Programmable Gate Array
FNN	Feed-forward Neural Networks
FP	Footprint
GAN	Generative Adversarial Networks
GPU	Graphical Processing Unit
HM	Hardware Metric
ILSVRC	ImageNet Large-Scale Visual Recognition Challenge
LSTM	Long-Short Term Memory
ML	Machine Learning
MLP	Multilayer Perceptron
MPI	Message Passing Interface
RNN	Recurrent Neural Networks
SoC	Service-on-Chip

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The rise in worldwide internet access has led to high web access and content production rates. According to (WEBSITE,), it is predicted that 3.5 billion people (nearly half the world's population) accessed the internet in 2016. Also, according to the company Internet of Things Analytics [1], as of this 2019, we already had about 8 billion gadgets connected to the internet, which also reveal the vast potential sources of data creation.

The reasons for this significant number may be linked to the popularization of smartphones and tablet devices. This increase in technology consumption implies market sales competition, which dramatically reduces the cost of purchasing sensor-based technology. These appliances and their sensors may generate spatial position, velocity, acceleration, pictures, videos, and audio. These data are generated in huge volumes at unpredictable rates, which must be dealt with accordingly.

This vast number of sensor-based technologies differ from the traditional x86 architecture in such a way that a new paradigm called Edge Computing emerged. Thus, this paradigm is based on in-situ processing to provide analytics at the data source (FRANKLIN, 2017). Exploring the Locality Principle, this paradigm allows gadgets to relieve some or all of the workload from the cloud, thus alleviating the performance fluctuations that it suffers on high demands.

Also, the study of Edge Computing has been highlighting the increase of gadgets' overall performance and autonomy. Aligned with that, recent strands of cost-effective Graphical Processing Unit (GPU) embedded edge devices have elaborated fertile ground for many complex applications, such as in the field of Artificial Intelligence (AI). It is noteworthy to mention that Deep Learning (DL), a sub-field of AI (GOODFELLOW et al., 2016), adapted well to these Edge devices in tasks such as facial recognition, speech detection, and image processing.

These System-on-Chip (SoC) have been the focus of many recent works, which attest to their advantages in terms of low purchase cost, portability, and high energy efficiency with good performance. However, these SoC evolve rapidly with noteworthy gains but lack experimentation regarding the new boundaries of its evolution.

For instance, the NVIDIA Jetson TX1 and TX2 were released in November 2015 and March 2017 and counted with 4 GB and 8 GB of RAM, respectively. For instance, NVIDIA Jetson Xavier was released in December 2018 with 16 GB of RAM.

[1] https://iot-analytics.com/

However, even though these boards evolve at such high speed, there still is a lack of studies that attest to the full potential of these boards on Deep Learning, especially regarding the training of DL models directly on the board.

1.1 Problem statement

Regarding deep learning execution in embedded systems, it is customary to perform training steps of a deep neural network offline and only deploy it on an embedded architecture. However, there has been an increase in frameworks and applications production in the past few years. This work intends to tackle the following question:

Is it possible to train and deploy deep learning applications at the edge?

There are a few questions that need to be answered in order to respond to this question, such as:

1. What are the limits while training DL on embedded platforms?
2. What frameworks and models perform the best training and deployment on Advanced RISC Machine (ARM) architecture?

1.2 Motivation

Since many DL frameworks and models have been created for regular x86 architectures, they face compatibility issues with arm64 and face differences in implementations and optimizations. Thus, it is essential to measure and compare the performance of these frameworks and models under a proper methodological experimental setup.

This work's motivation is to perform a performance evaluation of Deep Learning Frameworks on an ARM-based architecture environment to compare performances and determine the best frameworks, models, and datasets.

1.3 Objectives

In order to attest to the goals of this work, it seeks to conduct a study of Deep Learning on ARM-based edge devices. In order to do so, specific goals have been assigned:

- To compare practical performances among two famous DL frameworks under a similar and fair environment configuration and upon the same models written similarly from the source.
- To understand the behavior of DL applications on SoC under various performance scenarios.
- To conduct and analyze a performance evaluation with a few famous Convolutional Neural Networks (CNN) models on ARM NVIDIA Jetson TX2.
- To detect limitations and spot a progress curve for CNNs training at GPU embedded ARM devices.
- To determine the best framework and models for embedded ARM devices in complex DL training workloads.

1.4 Contributions

This work is experimental research that aims to test a combination of DL frameworks, models, and datasets under a parameterized and fair training environment. This experimentation scenario focuses on performing training loads on SoC boards to compare two major frameworks' performance by checking training discrepancies and efficiency limitations.

This work evaluates the feasibility of training and deploying Deep Neural Networks on embedded GPU devices with and without GPU support. These networks are implemented under two different Deep Learning frameworks and focus on training CNNs for computer vision problems.

Also, this work evaluates and compares the performance of these two deep learning frameworks upon similar implementations of the same seven models, using reliable data set on top of the NVIDIA Jetson TX2 embedded platform.

Therefore, this work aims to conduct a performance evaluation study on current embedded GPU devices' most famous CNN models and datasets. Thus this was achieved in an NVIDIA Jetson TX2 cluster with four units using PyTorch and MxNet frameworks, each running of 7 CNN models and one medium-sized dataset into three configurations of CPU and two for GPU, thus totaling six possible hardware configurations.

The experiments revealed a characteristic better behavior for MxNet and another better scenario for PyTorch, both under specific hardware configurations that will be further revealed and discussed in this work's experiments.

These experiments measured the impact of intense computations such as CNN models training and validation when processed with PyTorch framework regarding hardware metrics such as RAM memory, CPU and GPU utilization, but also investigating software metrics such as total execution time, average execution time per epoch, average power consumption, accuracy, loss, and top5 accuracy.

Besides that, this work has multiple applicable scenarios, such as mitigation systems for general network pane, especially in traffic monitoring systems and robotics (GAYA et al., 2016), and perhaps, also included in spatial technology.

Finally, this work contributed by elaborating a well-defined step-by-step tutorial [2] to aid future Jetson TX2 users to install and configure the board's environments, and also to help anyone who wishes to obtain the codes and organization for experimental research with CNNs.

1.5 Work Organization

To approach all the previously mentioned topics, this work was divided as follows:

1. Chapter 2 approaches deep learning background, which includes the entire explanation of techniques, key concepts, models, and frameworks. It is followed by state-of-the-art, which briefs the reader about important contributions to deep learning research in the recent past. After that, the related work subsection unveils the collection of works that attempted on doing similar topics.
2. Chapter 3 contains a detailed study regarding the related works to this research topic.
3. Chapter 4 contains a theoretical formalization that led to this work's solution and also highlights a few key contributions for this study.
4. Chapter 5 describes the test scenarios, specifications, results, and analysis.
5. Chapter 6 holds the closing remarks, future works, and acknowledgments.

[2]https://github.com/Bfzanchetta/DLARM

2 BACKGROUND

This chapter introduces the essential concepts of Deep Learning. This conceptual section highlights important techniques that evolved into Deep Learning and the current frameworks, such as MxNet and PyTorch, and shows some differences. This section also unravels different network types, models, and data sets for further understanding.

2.1 General View

Over the years, many concepts have been developed on AI, most of which achieved great success on practical issues, such as Image Recognition, Facial Detection, Object Detection, Speech Recognition, Textual Prediction, and much more. Because of the success of this broad spectrum of applications, Deep Learning became a notorious learning option for algorithms.

Computational applications are often developed in logical steps to evolve from static into dynamic solutions, i.e., cancel the need for constant manual adjustments and automatize the so wished process. The desire to guide decision-making and introduce knowledge learning on computational applications guided the development of AI.

Before AI started to be recognized by what is known in current literature, many researchers attempted to introduce knowledge on computational systems. Two distinct approaches tackled different perspectives of acquiring and generating knowledge using logic but differing on specific approaches. Initial methods aimed to represent data as numbers and approximate functions to learn valuable information, while others aimed to store knowledge in databases and create rules of knowledge to guide the algorithm.

The attempt to introduce knowledge manually to the machine was named Knowledge Base Systems. These systems may be described as manual questions and answers tuples, stored in data structures and fed into logic programmable structures, thus generating a static organized base of knowledge. This approach depended on the manual entry of information, so it relied on specialists who could provide correct questions and answers precisely.

The literature cites (LENAT; GUHA, 1989) as one famous case, which presented two significant components: One inference engine and one database of statements in a language called CycL. Human staff entered statements into the database and struggled to develop formal rules of inference so that the algorithm could generate questions that the staff would answer. The staff entered a story about Fred shaving in the morning during the development. The system knew beforehand that people did not have electrical parts, but since Fred was holding an electric razor, the system inferred that "FredWhileShaving" possessed electrical parts. The system then asked whether Fred continued being a person while shaving.

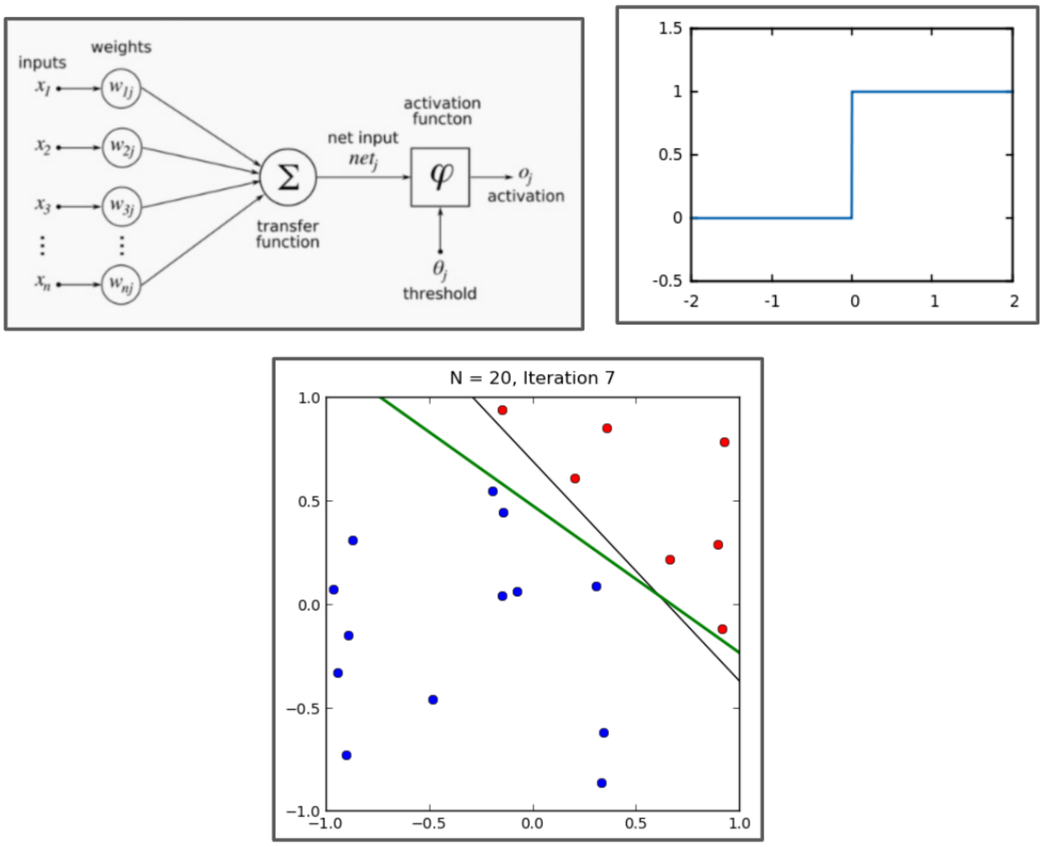
Even though such systems appeared to gain knowledge, the algorithm was not learning because it did not apply self-made or nonlinear computations. Also, since the machine did not learn new information, it was inherent that the complete development of a particular instance of this model would make the specialist who fulfilled its database ultimately obsolete. These facts led this first attempt to fateful disuse. Still, these challenges of hard-coding knowledge demonstrated that it was essential that machines needed a technique to gain their knowledge with pattern extraction from raw data. Thus, the scientific community focused on initiatives that bestowed knowledge acquisition almost entirely to the machine, creating techniques that allowed the creation of techniques known today as Machine Learning (ML).

Almost at the same period as the earlier knowledge experiments, in 1939, with the publication of the Hodgkin-Huxley Model (HODGKIN; HUXLEY, 1939), science learned first-hand about neuron shape and the direction of nervous electrical response. The novelty of neuron cells and their functioning inspired McCulloch and Pitts in 1943 to publish a logical calculus basis for programmable neurons (MCCULLOCH; PITTS, 1943).

By doing so, this publication set formal guidelines that determined how to define and combine neural units in computational terms while also postulating that the neuron would only transmit the signal forward if it surpassed a threshold of inhibition into an excitation value. It was only with the introduction of this work that many researchers could implement computational neurons and thus, start experimenting with AI. Indeed, the bio-inspired neuron is still the primary line of thinking and understanding neural networks.

In 1958, Frank Rosenblatt introduced Perceptron (ROSENBLATT, 1958), the first neural model that used a linear function to perform the learning. To do so, the model (seen on Figure 2.1 to the left) multiplies data entries by the Perceptron weight's array and adds this result to a bias columnar array in order to generate an output. This output is passed on an activation function (see Figure 2.1 to the right) on some occasions and then subtracted by the expected output value of the previously annotated entry data to generate a prediction.

Figure 2.1: Perceptron Model Architecture (left), Activation Function (right) and Output (bottom)



The model stops training when the loss achieves a minimum value provided by the programmer, but until then, the training performs alterations on the weights to find great values for them and converge into a solution. It is vital to notice that the entry data is fed into the model, and the training generates an output, fixating a new and unique combination of weight values on the neuron itself. These weights represent the model's contribution that granted real learning to this network against no authentic learning from Knowledge Base Systems.

Finally, Perceptron's initial predictions (see Figure 2.1 at the bottom) on top of two groups of training data started agglutinating into positions that could be split into two distinct groups by at least one line. Thus, the Perceptron proved that it could learn to classify a binary dataset with success, thus gaining the fame of Linear Classifier.

In 1960, Widrow e Hoff published Adaline (WIDROW; HOFF, 1960), a Linear Unit neuron that may be considered as a similar alternative to the Perceptron. Both Perceptron and Adaline performed calculations on data characteristics known as features. They discovered how to linearly separate features into classes into an approach called Logistic Regression. By doing so, they learned how to approximate the best function that separates the data groups. The more complex these learned functions are, the more classes of data it can classify, but it also increases the training complexity and computational costs.

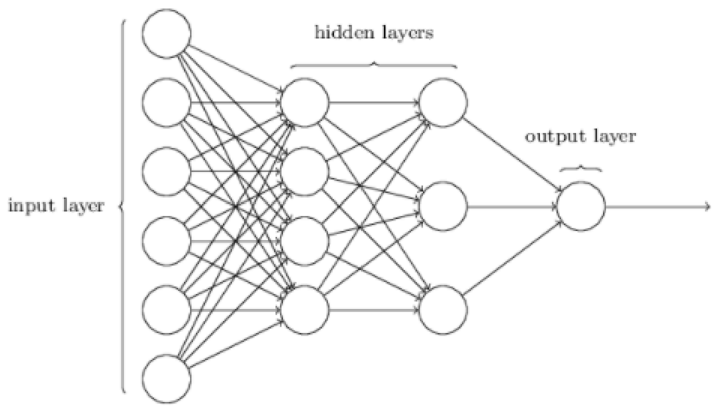
In a nutshell, Perceptron defined the training concept on neural networks that are still being used in the twenty-first century, which symbolizes the action of performing many calculation steps on a structure or object (model) with a training conjunct (data set or dataset) in hopes of regulating this model's parameters to perform a task with such data better. With this new concept, all network successors must contain a training method or function on which the model performs many proceedings to adjust its internal parameters for a very particular task.

Training a network is a pretty straightforward process that culminates on a set of model's internal variables (weights) specialized towards a problem representation and may be exported for later use once it converges. In this way, training mitigated the need to repeat itself every time the application needs to be used. In other words, a programmer who finds the best combination of weights that solves a particular problem may save the configuration for sharing or not. Also, due to the high number of operations over data, training consumes more computational resources and takes more time to finish than any other network proceedings. More details regarding training costs will be discussed further in this work.

The data entry conjunct is called dataset (DS) and is formed by many instances of the data used as input on the training step of the model. Since Perceptron was being developed, there was a great need for generous datasets, for small datasets would provide too little information for the model's adaptation. Datasets may contain images, text, or sounds and may or not present the respective labels which identify to what class a particular instance of the dataset belongs. More details regarding datasets will be discussed further in this work.

Some complex datasets revealed weak points on Perceptron's ability to converge its model into a complex solution. This problem was mitigated by attaching multiple Perceptron units into a computation graph, also named Multilayer Perceptron (MLP). This combination, seen in Figure 2.2, increased the entire network's time but also enabled the network to learn to classify more classes by dataset and to increase the prediction rates for the same early datasets of a single Perceptron.

Figure 2.2: Multi-layer Perceptron (MLP) Architecture

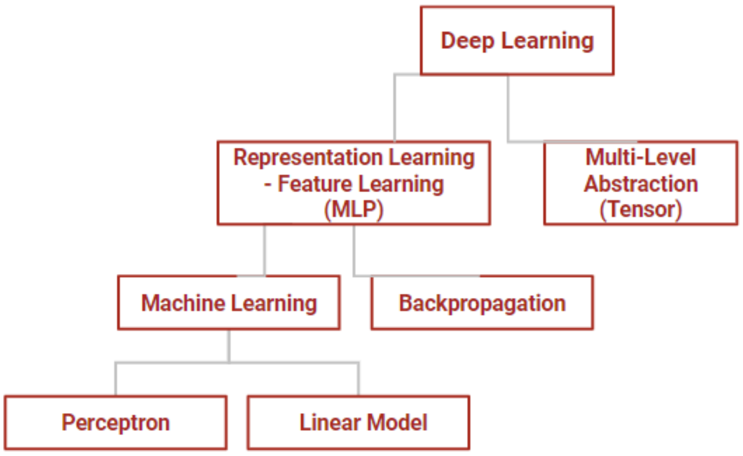


Also, regarding MLP, the first most remarkable difference from a single Perceptron resides in its architecture. Table 2.2 shows that the first Perceptron units in the MLP layer are responsible for being the first units to receive the input and transform it, thus gaining the name of the input layer. The intermediate Perceptrons receive the result of the input layer and the result of each other until it arrives in the last layer. This middle layer is called the hidden layer, and it is the most responsible for the procedures that engage in the gaining of learning. Finally, the output layer consists of neurons that specialize in classifying the process of the hidden layers into a probabilistic output, which is called the prediction.

This multilayer design fits well in the DL approach because this depth enables sequential learning as minor parts of a unique computer program. Each layer represents a memory footprint of the computer's state prior to the parallel execution of other sets of instructions. In this way, more depth leads to a higher number of instructions being executed in parallel. There are advantages in processing instructions sequentially, for current state instructions may access previous step values to aid in its processing.

The second and last key difference between MLP and early Perceptron resides in how these networks adjust their training. Early Perceptron networks performed a forward training and then conducted changes only in the following training step to reduce the model's prediction loss. MLP, on the other hand, allied Perceptron's logic above with a backward loss adjustment that took place right after a complete forwards pass. This new technique is called backpropagation, and it contributed towards a more significant gain of performance of the neural networks.

Figure 2.3: AI Hierarchical Vision


Deep Learning
Representation Learning - Feature Learning (MLP)
Multi-Level Abstraction (Tensor)
Machine Learning
Backpropagation
Perceptron
Linear Model

To summarize the evolution of AI, Figure 2.3 shows that Perceptron is the crucial element of Deep Learning, primarily due to its story that started thanks to its ability to separate classes linearly. Then, the concept of Machine Learning started by exploring the Perceptron and its strengths and checking its weakness. By uniting Perceptron with the backpropagation algorithm, the MLP started a field of ML, also called Representation Learning, or Feature Learning. Thus, networks strive to obtain the best weights to learn how to characterize features of the data.

As a solution to solve problems that Representation Learning could not solve, Deep Learning (DL) introduced representations that a combination of simpler ones may express. In this way, DL allows the model to create its complex concepts out of simple concepts. A clear concept of DL cannot be defined only with the definition of the Multilayer Perceptron (MLP), i.e., the MLP consists of a model that combines n-layers of simple Perceptrons, but simply connecting many MLPs only generates another bigger MLP. On the other hand, by allowing the algorithms to create their fundamental matrix of knowledge with the MLP, one must provide the data and let the algorithm develop its own more complex matrices of knowledge, thus increasing the model's representation scope.

For instance, a fundamental self-created matrix of an individual MLP in the hidden layer might learn to identify a point. Then, still without human interaction, the neurons can combine many instances of point with learning a higher concept such as line, or curve, which can then be used to develop even higher concepts such as surfaces, or eyes, or legs, and so on.

Summarizing DL, it may be described as teaching the machine to perform tasks that are intuitive to human beings but very complex to define in algorithms due to their variations. In this way, DL enables machines to learn their concepts and evolve these into more complex concepts, mitigating anomalies and granting more possibilities to the models' scope.

2.2 Frameworks

Until Deep Learning, most models were hard-coded, but with the massive leap of complexity that DL provided, the development of DL frameworks became essential. New open-source frameworks were released to facilitate the implementation, training, evaluation, tests, and predictions on deep neural models.

Several of these frameworks presented special optimizations that could only be found in them. Since all frameworks are open source, developers soon realized how significant specific optimizations were and started doing it themselves. This scenario provided a race for better framework optimizations and tools for visualization. In this section, some of the most noteworthy of these frameworks will be briefed further in Table 2.1.

PyTorch (COLLOBERT; BENGIO; MARIÉTHOZ, 2002) introduces a deep learning tensor library with a series of optimizations, with GPU and Central Processing Unit (CPU). PyTorch has been created over regular Torch. So, it was developed to support Python targeting two main niches: a substitute of NumPy for GPU usage optimization and a deep learning platform with the promise of faster and flexible computations.

MxNet (CHEN et al., 2015) is another widely used DL framework. It is very famous for supporting great flexibility, primarily through viable options in image processing -part of it thanks to its tool Gluon-. This framework is among the pioneers in multi-GPU processing with Deep Learning and keeps good performance with state-of-the-art Convolutional and Long-Short Term Memory (LSTM) models.

Theano was dis-considered for this work because this project has been discontinued. The framework deeplearning4j was opted out of this work because it proved difficult to configure and install, with very confusing, incomplete, and void weblink tutorials.

Caffe and TensorFlow have opted out because they contain too many optimizers and stable releases that turned a simple installation and configuration into a difficult task. Finally, MxNet and PyTorch have very similar scripts, installation steps, sub-packages and match many versions of the sub-packages with little effort, thus reducing the time to install and configure the fair test environment.

Table 2.1: Details of the most used DL frameworks

Deep Learning Frameworks	Details		
	OS Support	PL Implementation	PL Support
PyTorch (COLLOBERT; BENGIO; MARIÉTHOZ, 2002)	Linux, Android, Mac OS X, iOS	Lua, LuaJIT, C, CUDA, C++	C, Lua
MxNet (CHEN et al., 2015)	Windows, Linux, Mac OS	C++, Python, R, Julia, JavaScript, Scala, Go, Perl	C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl, Wolfram Language
Theano (BASTIEN et al., 2012)	Windows, Linux, Mac OS, CentOS 6	Custom C, Python	C/C++, CUDA, OpenCL, PTX, CAL, AVX, ...
Caffe (JIA et al., 2014)	Windows, Linux, Mac OS	C++	Python, C and C++
deeplearning4j (GIBSON et al., 2016)	Windows, Linux, Mac OS	Java, C, C++, CUDA	Java, Scala, Clojure, Kotlin
TensorFlow (ABADI et al., 2016)	Windows, Linux, Mac OS	C, C++, Python	Python, Java, C++, Go

2.3 Network Types

As researches in network design started benefiting from more complex learning functions, advanced optimization algorithms, and model design, scientists felt a necessity to create a taxonomy on deep neural networks' scope. The convention is that each deep model must be separated depending on its primordial operation.

An Artificial Neural Network (ANN), or just Neural Network (NN), may be defined by its purpose. Literature highlights three critical neural networks, which vary in their techniques and computational methods.

The first type of NN is called Feed Forward Neural Networks (FNN), and its history comes from plural attachments of single neurons like MLP into a connected graph. The critical point in this network is the training and optimizer functions that are called back in the training and fine-tuning stages. A simple linear model may be considered FF-type if it does not contain any other networks in its calculations.

This type of network is primitive in the Deep Learning field but still widely used. Since this network is known for its excellent connectivity, the results of co-adjacent units of the same network rely on the processing time of the previous unit, which respectively also depends on all previous connections and their times. For more straightforward tasks and simple networks, training might take less time and cost less -computationally speaking-, whereas higher networks with more connections and higher datasets increase the costs dearly.

Thus, most state-of-the-art networks introduced a technique called dropout, which consists of a rate between zero and one-hundred percent that will be the rate on which the network connections will be disabled during an entire forward network interaction.

If the dropout is 0.5 (50%), half the network connections will be randomly turned off during each complete interaction between the input and output layers. This process reduces training time significantly because it forces the network to reinforce the connection between some neurons instead of training the influence of all-to-all paths. The state-of-the-art results show that deep models, especially those with over 1000 neurons in the Fully Connected layer, require some dropout to enable real training time.

CNN is considered one of the essential neural networks, especially in the image processing domain. The most outstanding contribution of this network is the introduction of convolution operations between tensors, which leverages over many matrix operations that would be costly in other techniques. Also, the image processing performed with the aid of the kernel technique boosts the convolution results and improves feature extraction.

For instance, current state-of-the-art convolutional network applications may easily contain over one million units. In this way, most deep networks benefit from parallel computing to perform convolutional and other costly operations.

In practical terms, convolution corresponds to the conversion of the input and kernel to the frequency domain using Fourier transform, performing point-wise multiplication of these two signals and converting back to the time domain by inverse Fourier transform (GOODFELLOW et al., 2016).

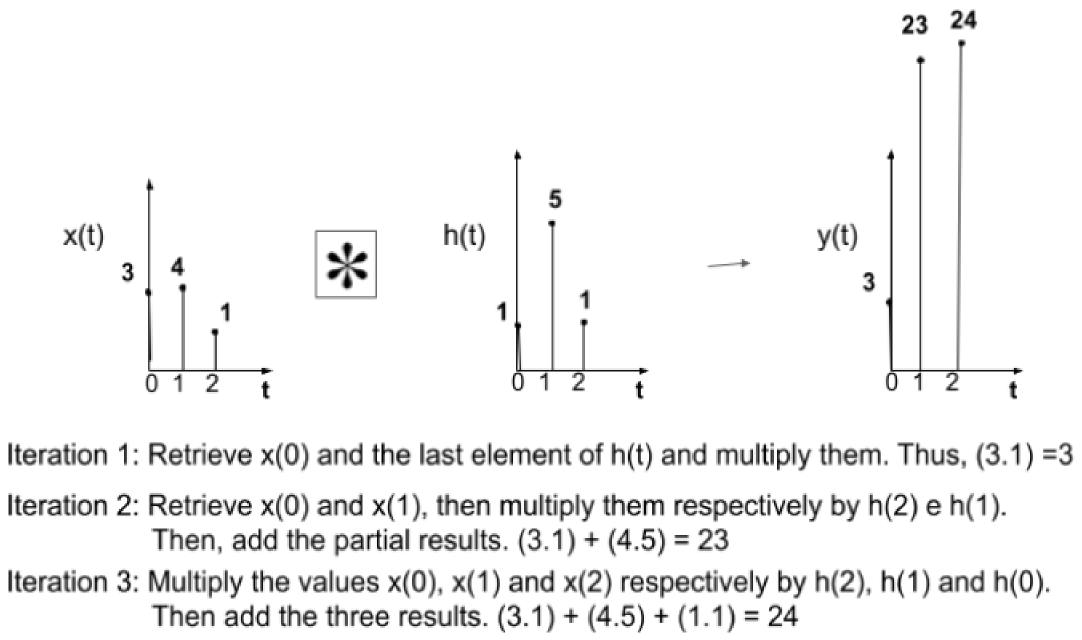
Figure 2.4 shows step-by-step on how the convolution takes place. The picture sample is stored into a data structure which is represented by the discrete pixel values in the *x(t)* function (to the left of 2.4). It is important to highlight that all discrete *t* values from *x(t)*, *h(t)* and *y(t)* functions do not symbolize time, but simply represent a particular data spot for data representation.

The second element represented by *h(t)* of the convolution may represent another input data in other cases. However, in this case, it represents the convolutional kernel that will assist the transformation of this data during a specific convolutional layer. More information regarding the used kernels will be described in the next section and Table 4.2. Finally, both the input data (*x(t)*) and the kernel (*h(t)*) are processed by an iterative logic described in the captions of Figure 2.4 in order to obtain that particular convolutional result *(y(t))*.

Furthermore, the output observed in Figure 2.4 reveals the exaggerating effect of the convolutional filters in the input data, i.e., strictly positive kernels enhance all data values. In contrast, kernels with mixed positive and negative values might isolate only specific values. In a nutshell, these impacts of convolution improved the ability of Fully

Connected neurons to acquire more critical weights during training and thus, improved the overall prediction accuracy of neural networks in tasks such as Facial Recognition and Object Detection (GOODFELLOW et al., 2016).

Figure 2.4: Illustrated Example of Convolution



Iteration 1: Retrieve x(0) and the last element of h(t) and multiply them. Thus, (3.1) =3

Iteration 2: Retrieve x(0) and x(1), then multiply them respectively by h(2) e h(1). Then, add the partial results. (3.1) + (4.5) = 23

Iteration 3: Multiply the values x(0), x(1) and x(2) respectively by h(2), h(1) and h(0). Then add the three results. (3.1) + (4.5) + (1.1) = 24

Even though Convolution gives name to complex image recognition AI models, this operations is not new, it is a well-known mathematical operation. However, it still offers great advantages result-wise.

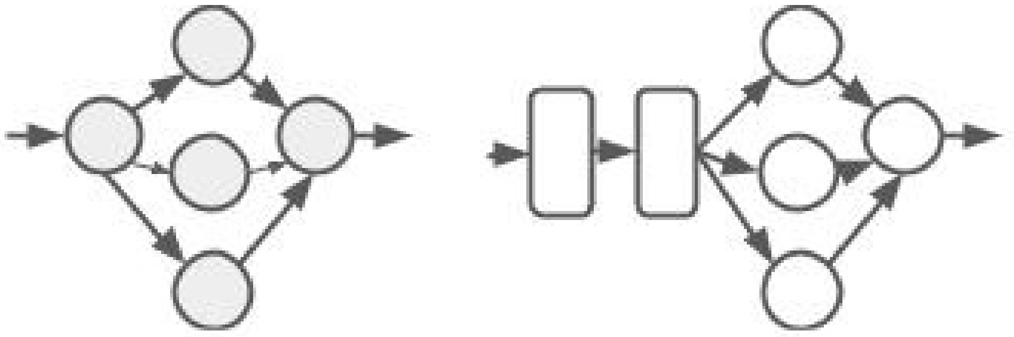
Regarding CNNs, this work unveils three of the basic operations in most Convolutional Neural Networks. A CNN comprises Convolutional Layer, Pooling Layer, or Dense Layer. Convolutional Layers usually take a sub-region of the image to apply a series of filters, kernels, and activation functions to produce a single output - also named a feature map.

Pooling layers serve as a tool to reduce the dimensionality of the data feature map by many computer vision techniques that reduce its size in the case of images. This reduction is made to shorten the processing time of the training steps, mainly because the images of several CNNs are too heavy, and their processing at total capacity takes a lot of computational energy and time, turning the project unfeasible.

The Dense Layer is usually applied in the final stage of the CNN, which contains the aforementioned MLP-based connections. Thus it performs the classification of the features that the convolutional layer enhanced. Theoretically, the more units in the Dense layer, the higher capacity of learning the model has. However, two factors must be accounted for: increasing few dense units causes significant complexity increases in training algorithms, and also, it is known that the excessive addition of neurons in MLP has a roof limit for performance gains.

Several typical applications use CNN, but due to this convolutional kernel, most applications focus on image problems and video -which is a series of images-.

Figure 2.5: Graphical Representation of the Models



FNN (in the left) unites multiple units that perform calculations (training), which modifies weights until their impact on data representation becomes feasible, i.e. it learns to extract features from the data; CNN (in the right) introduces early Computer Vision transformations to facilitate FNNs feature extraction

2.4 Models

The development of Deep Learning was fueled by many real-world problems that did not have accurate solutions until then, such as pattern detection, image recognition, image extraction, object detection, speech recognition, and word prediction.

These problems had to be solved by algorithms specializing in one of the solutions mentioned earlier by intensive training. The formal specification of data ingestion and transformation to solve these problems into a programming language is described as a DL model. The subsection below will further explain some of the CNN models used in the further tests.

2.4.1 CNN

Many CNN models were proposed in the literature. Among them, this work chose to describe five instances of models in order to attest to a better model choice for experimentation. These chosen models were: LeNet, AlexNet, VGG-16, ResNet, and Inception-V3.

LeNet is the first fruitful convolutional neural network developed in 1990 by Yan LeCunn and served as a precursor of more profound and more complex convolutional networks. Its primary purpose was to identify handwritten digits on zip codes, personal checks, and others. This task is described as classification on literature.

It was updated up to a LeNet5 version, which received 32x32 images and passed them through a convolutional layer that produces six feature maps reduced down to 28x28 bits. Then, these maps are subsampled, producing six feature images measuring 14x14 units in length. Then, it is passed to a further convolutional layer that generates 16 feature maps of size 10x10, followed by second downsampling, which returns 16 feature map units of 5x5 size. Then, the network feeds these 16 feature maps to a first fully connected layer with 120 units, followed by a second 84-units fully connected layer. The output layer contains ten units, which is vital for classifying ten-digit classes.

The differential of LeNet is that it was the first successful convolutional neural network in deep learning, and it introduced a 5x5 convolution filter kernel.

AlexNet was developed in 2012 by a group of researchers from the University of Toronto to classify 1.2 million images into one thousand classes, winning the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) that year. It takes ImageNet data set images and passes them through 5 convolutional layers, followed by three fully connected layers. The first and second convolutional layers are interleaved with Max Pooling layers only after the fifth convolutional layer.

This model takes 224x224x3 input images then passes them to a first convolutional layer that generates 96 feature maps, followed by a second convolutional layer generating 256 feature maps. Then, these maps are fed to Max Pooling layers, then fed to the third convolutional layer. This third layer is unique because its kernel maps are connected to both kernels from the two groups of feature maps of the second layer, whereas the second, fourth, and fifth layers only consider the previous layer, but not the parallel feature maps. This third convolutional layer contains 384 kernels sizing 3x3x256 connected to the normalized and pooled output from the second layer. The fourth layer possesses 384 kernels of 3x3x192, followed by the fifth convolutional layer with 256 kernels of the same size as the fourth. Then, the 256 feature maps pass through a Max Pooling, only to be fed to three fully connected layers, with 4096 neurons each and a dropout rate of 50%.

It is noteworthy that this model beat the other models on ILSVRC'12 by a considerable margin that year. Its five best predictions revealed a top error of 15.4%, while the second place obtained 26.2% for the same parameters. AlexNet contains more layer groups that may be computed in parallel, such is the fact that it results in a high number of layers, and this consequently demands higher memory than LeNet for intermediate data storage.

The VGG team entry in the ILSVRC-2014 image classification competition designed a 16 layer convolutional network, trained, validated, and tested with the ILSVRC-2012 dataset introducing new properties to convolutional networks.

The VGG-16 model receives RGB images and passes them through a series of convolutional layers with reduced kernel sizes. The authors argue that composing two 3x3 convolution layers, for instance, equals one 5x5 convolution, but with fewer training parameters and increasing nonlinearity due to twice as many ReLU operations. The model comprises two convolutions, max pool, two convolutions, max pool, three convolutions, max pool, three convolutions, max pool, three convolutions, max pool, and three fully connected layers with two dropouts between them. With a total of 13 convolutional layers and three fully connected, coming to a total of 16 great layers, which justifies the model's name.

Once the training starts, the model receives 224x224x3 input images, then it passes by two convolutional layers of size 3x3, a stride of 1 and pad equals to 1 that produce 64 feature maps each. Then, ReLU activation and max-pooling with a receptive field of 2x2 and stride 2, which stays on the exact configuration for all max pooling operations of the network. The third and fourth layers convolute to generate 128 feature maps each, then applied max-pooling again. This convolutional and max-pooling configuration happens until the last convolution. Then, the fifth, sixth and seventh convolutional layers generate 256 feature maps, followed by max-pooling, followed by another three convolutional layers: the eighth, ninth, and tenth layers, all of which generate 512 feature maps. After these three layers, max-pooling takes place, and then the last three convolutional layers generate other 512 feature maps, namely: the eleventh, twelfth, and thirteenth convolutional layers. A final max pool follows these layers, only directed to three sequential fully connected layers of 4096 neurons with two 0.5 dropouts between them. The model then outputs the SoftMax layer with one thousand classes, thus finishing the training.

This network is significant because the input's volume has reduced spatial dimensions due to both max pooling and convolution. However, the number of feature maps generated on the training increases because kernel filters also increase as the model gets deeper layerwise. This option presents a viable option for DL scientists who focus more on higher datasets with a bit of trade-off to the model's graph training.

Built by Microsoft, ResNet is known for its 152-layers-deep convolutional network for object detection and image classification tasks. This model proved its success in ILSVRC 2015 competition by grating victory with a top error rate of 3.6%, and such an impressive rate represents lower values than the human error rate of 5-10%.

Previous networks were good function approximators. Thus, they learned an identity function f(x)=x that theoretically could be added to a good network without changing the output or compromising the network quality. However, in practical terms, the model needs to train all the network weights simultaneously, so this approach significantly reduced deeper network accuracy rates when combining identity networks. Still, the idea that a small network could learn identity function motivated this work to make this integration plausible. In order to tackle this issue, researchers presaged that by feedbacking the input directly to the output of this small network (f(x)+x), thus making identity mapping nearly optimal and simply allowing solvers to guide the weights of the many nonlinear layers toward zero in order to approach the small network identity mappings. With this crucial concept of a residual block, this work has a greater probability of learning the smaller network's identity function when f(x) equals zero than learning a new deep model's identity function from scratch. It proved a handy technique to train deeper models without degrading accuracy. In ResNet specifically, for every triad of convolution, ReLUs, and convolutional operations, the first convolutional operation's input is passed to the operation's output in the hope of learning the optimal identity mapping. In other networks, the attempt is to fit subjacent mapping to the original input to solve the classification problem.

- ResNet-152 receives 224x224x3 input image, and then the first convolution layer generates 64 feature maps of size 7×7 and stride 2, followed by a max pool of 3x3 and stride 2. Then, the authors described sets of building blocks that correspond to each residual block previously mentioned.
- The first set of building blocks is set with three layers' repetitions: 1x1 convolution generating 64 feature maps, 3x3 convolution generating 64 feature maps, and 1x1

convolution generating 256 feature maps. These three repetitions generate nine convolution layers.

- The second set of building blocks are set with eight repetitions of layers: 1x1 convolution generating 128 feature maps, 3x3 convolution generating 128 feature maps, and 1x1 convolution generating 512 feature maps. These three repetitions generate 24 convolution layers.
- The third set of building blocks are set with thirty-six repetitions of layers: 1x1 convolution generating 256 feature maps, 3x3 convolution generating 256 feature maps, and 1x1 convolution generating 1024 feature maps. These three repetitions generate 108 convolution layers.
- The fourth set of building blocks has three repetitions of layers: 1x1 convolution generating 512 feature maps, 3x3 convolution generating 512 feature maps, and 1x1 convolution generating 2048 feature maps. These three repetitions generate nine convolution layers.

Summing all convolutional layers so far, it comes to 151 convolutional layers. Finally, the model passes by an average pooling, fully connected, and SoftMax layer, 1000 classes.

VGG highly influenced this model because it preserved the decision to use smaller-sized kernel filters to the detriment of higher dimension kernels. Also, it is essential to mention that ResNet focused more on the residual blocks than on the numbers of neurons in the fully connected layer.

InceptionV3 is a Google convolutional neural network that is one of the state-of-the-art models of this model's niche. It was designed as a more complex GoogLeNet, winning the ImageNetILSVRC-2014 competition in tasks such as Object detection with additional training data, classification, and localization with provided training data.

The model trained a more profound and broader convolutional neural network on top of the ILSVRC2012 dataset to perform image classification, localization, and object detection with high accuracy. This work introduced an unprecedented contribution: The Inception module. In a nutshell, the Inception module consists of a series of convolutions and max-pooling in a well-defined, parallel, but restricted order. Then, the results of these feature maps are merged later.

After performing a convolution, traditional methods perform either another convolution or a max-pooling in series during intermediate layers. When it comes to InceptionV3, the Inception module generalizes one start point into four parallel flows of convolution and max-pooling of predefined dimensions. These Inception modules may be seen as a process that generates four threads to explore different features from each flow during the filter concatenation phase. Anyone can write a custom Inception module and put it to test however this work introduced the three best modules tested and approved.

1. The first Inception module consists of four flows: One flow containing 1x1 convolution, followed by two 3x3 convolutional layers, a second flow with 1x1 and 3x3 convolution, then the third flow with a pool followed by 1x1 convolution, and finally the fourth flow with one 1x1 convolution. It is important to highlight that these flows execute in parallel and have their final feature maps merged in the Filter Concatenation block, where the particular Inception module ends.
2. The second Inception module is similar to the first, but with different units of kernels and different sizes, which consists of four flows: The first flow has five sequential convolutional layers, with kernels: 1x1, 1x7, 7x1, 1x7, 7x1. The second flow has three sequential convolutional layers, with kernels: 1x1, 1x7, 7x1. The third and fourth flow remains the same.
3. The third Inception module repeats the same logic moreover as the previous two, but adds a sub-flow on flows, the description of their flows is as follows: The first flow has two serial convolutions sized 1x1 and 3x3, and it branches into two parallel 1x3 and 3x1 convolutions. The second flow has a first 1x1 convolution, branching into two parallel 1x3 and 3x1 convolutions. The third and fourth flow remains the same.

It is essential to highlight that these flows execute in parallel and have their final feature maps merged in the Filter Concatenation block, where the particular Inception module ends. This new contribution works because different kernel sizes extract unique granular levels of feature knowledge.

The work's model is presented in a well-defined set of layers: Starts with 3x3 convolution with stride 2, followed by 3x3 convolution with stride one and 3x3 padded convolution with stride 1. Then a 3x3 convolution with stride 2, followed by another 3x3 convolution with stride 1, one 3x3 convolution with stride two, and one more 3x3 convolution with stride 1. The feature maps are passed through 3 units of the first Inception module, followed by five units of the second Inception module and two units of the third Inception module. Concluding the network sequence, the following steps consist of passes on pool, linear, and softmax layers.

This model allows deeper models to be designed without much accuracy reduction while keeping kernel size relatively small and fewer fully connected neurons. The parallel convolutional layer model that Inception explored allowed the development of better networks.

2.4.2 Datasets

In order to obtain knowledge towards problem-solving, NNs require samples of data that may be annotated or not called datasets. These data sets may contain many domains and sizes that serve specific purposes, such as images for facial recognition and robotics, sounds for music processing, and texts for natural processing. Some of the most famous image datasets used in CNN models are listed below.

- CIFAR-10 (KRIZHEVSKY; NAIR; HINTON, 2014): it has 60,000 RGB images of 32x32x3 pixels representing 10 groups of animals and vehicles.
- MNIST (LECUN; CORTES; BURGES, 2010): contains 70,000 examples of monochromatic handwritten digits from 0 to 9 in 28x28x1 shape.
- ILSVRC'12 (Imagenet) (RUSSAKOVSKY et al., 2015): divides 1,281,167 RGB pictures (256x256x3) into 1000 classes of objects and animals, containing nearly 144 GB.

The datasets that were used in all related works and the present work, and also the reasons for using them in the present work, will be further discussed in Chapter 4.

2.5 Background Final Remarks

This chapter described a DL overview, followed by its most noteworthy frameworks and network types. The network types section described many of the quintessential concepts of deep neural networks and their operations. Then, the model sections further explained the most famous networks in the state-of-the-art that have been used in this work, followed by an explanation about the datasets. Finally, the next chapter presents the fine-grained study regarding this work's research problem of the related studies.

3 RELATED WORK

Training and evaluating Deep Learning models can be challenging due to hardware and time constraints. In order to address these issues, many works have proposed an evaluation on testing DL models on GPU-based SoC. However, most of these works perform training on x86-based machines with GPU and only validate their models on SoCs, or at the very least perform fine-tuning on the SoC and return to testing. However, some related work studied these DL on SoC by profiling some of the board's critical elements with several exciting strategies. These points were tabled and formed a good strategy for defining some of this work's experimental choices. These works are at this moment summarized.

3.1 Deep Learning on Desktops and GPU-based SoC

Overall, GPUs are more expensive than CPUs; however, they show considerable gains in mathematical computations due to an increase in floating-point arithmetic and fast data processing parallelism. Also, the fast development of complex computer games and simulations engines boosted GPU evolution and availability thanks to market competition and the quick rate of evolution of these components.

Due to these factors, the DL model's processing on GPU dramatically reduced implementation and test times, thus allowing researchers to focus on frameworks and models advancements. Several works decided to take advantage of GPU processing on the Desktop environment to implement their DL solutions (HAN et al., 2016; QI; SPARKS; TALWALKAR, 2016) while recurring to SoC hardware for testing and deployment.

The **Efficient Inference Engine (EIE)** (HAN et al., 2016) [7] was proposed as an inference engine with an accelerator that performs custom multiplication of sparse matrix vectors and optimized weight handling on top of compressed Deep Neural Networks. These deep compressed nets were proposed previously in the literature under the coined term 'Deep Compression,' which essentially allows fitting large DNNs (such as AlexNet and VGGNet) entirely in on-chip SRAM. This compression is achieved by detecting and trimming the redundant connections of the network and by forcing many connections to share the same weight (HAN et al., 2015).

The work EIE evaluated AlexNet, VGGNet, and NeuralTalkLSTM models on Intel Core i-7 5930k desktop for CPU, NVIDIA GeForce GTX Titan X for desktop GPU, and NVIDIA Tegra K1 for mobile GPU. The work tested the models in compressed and uncompressed states on the Caffe framework. The experiments ran on nine benchmarks, and each experiment ran on batch sizes of 1 and 64. Then, they compared these experiments with custom EIE implementation. They were evaluated in function of performance -Computation Time (S), Gigaoperations per second (GOP/s) and speedup -, energy (economy generated by redundant operation removal) and design space exploration (which leverages Queue Depth, SRAM Width, and Arithmetic Precision).

This work is significant for SoC research because it belongs to a group of a select few that provide training test results. In contrast, other works focus primarily on fine-tuning models trained on the cloud or GPU desktops on the SoC due to its relatively shorter time to prepare a model that was already trained. However, that was not the case because EIE aimed to validate the way models responded to their engine even on SoC, thus contributing to its research.

Paleo (QI; SPARKS; TALWALKAR, 2016) [9] introduces an analytically performance model of deep learning frameworks. Its authors argue that each model's architecture carries declarative constraints related to its particular programming techniques on a training and evaluation basis. They also argue that these constraints may be extracted and mapped on hardware, software, and network guidelines to better scalability and performance evaluation. In order to do this, it uses techniques such as Hardware Acceleration, Software Acceleration, Parallelism, and Communication Schemes.

In this way, Paleo executes CNN experiments with AlexNet and VGG-16 while comparing TensorFlow's native metrics with their own regarding backward and forward pass-time (ms) in a 4GB NVIDIA TITAN X GPU desktop environment.

They also executed tests with NiN model, Inception V3, and AlexNet in Caffe, TensorFlow, and Cuda-convnet2 frameworks, respectively, and executed on NVIDIA K20X, NVIDIA K20, and NVIDIA K20 desktops, respectively. These tests aimed they describe in the results several graphics of comparison between Paleo and these frameworks, projecting future collaborations to readers.

Further experimentation tested NiN with FireCaffe instead of Paleo's Train Time and Speedup measurements, permutated on four scenarios with different workers and batch sizes on a Titan single-GPU hardware.

Also, they executed a prediction on Hybrid parallelism and Data parallelism (Train time in hours and speedup) instead of other work's AlexNet to validate Paleo on a desktop with 8 GPUs.

Finally, they conducted Generative Adversarial Networks (GAN), AlexNet, and Inception model evaluations on multiple server clusters with NVIDIA K80 GPUs linked through a 20 Gbps network. They oscillated the batch size number of workers and measured its effects on accuracy and speedup. This single experiment may classify this work as Cloud-based. However, it was a minority on the total experiments. Therefore it was classified as desktop-based.

The Paleo Project is significant due to its fine-grained performance evaluations and many frameworks and models. However, their more notable shortcomings relate to how vaguely they studied this vast amount of models and frameworks against one another. In other words, their work approaches many parts of deep learning, which gives readers perception of excellent workload volume. However, they only permute its variables on a select few states, which subsequently contradicts the reader's earlier notion.

3.2 Deep Learning on Cloud

Cloud computing allows users to purchase on-demand resources online and run applications with real-time monitoring of Memory, CPU power, GPU power, and storage capacity. It leverages the scalability, allowing the user's infrastructure to grow -horizontal and vertical growth- or reduce based on the user's demand or pre-configured preemptive settings.

The most significant upside of hiring Cloud computing revolves around the cost of such infrastructure. It costs less to instantiate a cloud virtual machine on a high-performance infrastructure than to purchase, install and maintain the physical cluster itself. The main bottleneck on Cloud computing refers to the network -mainly on the user's end since the traffic becomes essential to manipulate, transfer and monitor data and applications processed in the Cloud's infrastructure, as opposed to *in situ* which usually incurs in fewer network contentions.

Also, due to GPU's high purchasing costs, cloud providers implemented solutions that leveraged multiple powerful GPU cards allied with memory boosts, thus generating the GP-GPU clusters. These GP-GPUs can arbitrarily execute code as opposed to just executing rendering sub-routines, as it was performed until then (GOODFELLOW et al., 2016). NVIDIA's new programming language, CUDA, introduced this advancement, which allowed high-scale parallelism with higher memory bandwidth. Not long after its creation, this platform started being used on DL's research (CIREŞAN et al., 2012;

RAINA; MADHAVAN; NG, 2009).

The article (ICHINOSE et al., 2017) [6] regards cloud technologies privacy, with a specific experiment on sensor pieces of equipment towards pipeline processing in the Cloud. It proposes utilizing the Caffe framework, with a client symbolized by Raspberry Pi and the cloud server represented by GPGPU NVIDIA GeForce GTX 980. The analysis was conducted on a single feed-forward neural network: CIFAR-10 on the CIFAR-10 dataset.

The article confirmed that it was still more advantageous to perform the entire deep learning training workload on the Cloud than running it on the device itself at the time of this study. The results also showed how the limitations of Raspberry Pi in terms of graphical processing and non-volatile memory have greatly affected the experimental design on SoC.

3.3 Deep Learning Performance Evaluation Towards SoC

The work (ZHANG; WANG; SHI, 2018) [1] was conducted with the intent of testing multiple Deep Learning frameworks, alongside a few models on top of mobiles such as MacBook Pro, Intel FogNode, NVIDIA Jetson TX2, Raspberry Pi, and Nexus 6P. Amongst these Deep Learning frameworks, one may find TensorFlow, Caffe2, MXNet, PyTorch, and TensorFlow Lite. The latency, energy consumption, and memory footprint were measured in two experiment cases. The models executed on CNN cases were AlexNet and SqueezeNet but in limited combinations of the previously mentioned frameworks and devices.

The work was very generalist regarding the choice of frameworks and devices, but it failed to mention the data sets. The training and validation methodologies were also not mentioned, just as the explanations regarding the metrics extraction. The article moves so quickly on the methodology section that it fails to explain if the training routines were executed in batches, their sizes, and these choices motivations. However, it is a very broad-sighted work that must be recognized for its contributions to many-to-many experiments with frameworks and devices.

The work (TAYLOR et al., 2018) [2] evaluates the performance of Inception, ResNet, and MobileNet CNN models with the ILSVRC 2012 dataset. The experiments ran on TensorFlow and NVIDIA Jetson TX2 hardware, proposing an evaluation of these models with a self-implemented machine learning algorithm based on KNN. The work used inference time, energy consumption, accuracy, precision, recall, and F1-score as validation metrics to compare their implementation and the two CNN models.

Also, (ESHRATIFAR; PEDRAM, 2018) [3] revealed how DNN has gained more practical utilities in the last few years. Amongst those networks, the work cites iPhone's Siri algorithm that executes DNN models on the Cloud. They argue that data transfer from mobile phones to the Cloud at all times is costly in terms of battery, 3G, and 4G wireless transfers.

Also, it is not always feasible to transfer these applications' data with success due to server overload. That is why the authors proposed a framework that distributes the DNN workload and opts to send only part of the computation to the Cloud, instead of its totality, with the promise of increasing execution gains and reducing energy consumption. The experiments used TensorFlow with six models divided into three subcategories each. The experimental environment was NVIDIA Jetson TX2, the mobile representative of mobiles, and NVIDIA Tesla K40C, representing the server.

The results revealed that AlexNet upholds better results when executing on mobile standalone than on Cloud. Cloud, however, returned better results on VGG16 and NiN, reducing the energy consumption by a factor of 2.11 and increasing speedup between the range of 1.42 and 3.07 when optimized for performance.

Then, work (HUYNH; LEE; BALAN, 2017) [4] proposed an experimental study to evaluate DNN models and the so-called "Shallow Models" with direct training targeted to Mobile Devices. The chosen models were VGG-VeryDeep-16, VGG-Face, YOLO, AlexNet, VGG-F, VGG-M, and LRCN. Experimental conjunct was trained on Samsung S7, Samsung Note 4, and Sony Z5 mobiles using Pascal VOC 2007 dataset, collecting latency, precision, and energy consumption as main evaluation metrics.

Monsoon Power Monitor measured the energy consumption. This choice follows the tendency of most related works, which opt by using expensive energy measuring machinery towards energy expenditure validation.

Their prototype DeepMon executes the experiments on top of the DeepX framework and validates an execution on Cloud using the Caffe framework.

Beyond a fair comparison of object detection and classification deep neural algorithms, this work introduces critical concepts that help classify experimental workloads by volume and work locality. They were divided into three majority classes of experiment on this niche: edge-strong, remote-strong, and remote-weak. According to the authors, most related works decide on edge-strong and remote-weak combinations for their tests.

The work (CANZIANI; PASZKE; CULURCIELLO, 2016) [5] conducts experiments amongst many Deep Learning models, evaluating accuracy, memory footprint, parameters, operations count, inference time, and power consumption metrics in order to choose the best models.

Amongst these models were a total of 14 instances, being: AlexNet, BN-AlexNet, BN-NIN, ENet, GoogLeNet, ResNet-18, VGG-16, VGG-19, ResNet-34, ResNet-50, ResNet-101, ResNet-152, Inception-v3 and Inception-v4. The experiments were executed on the Torch7 framework, using an NVIDIA Jetson TX1 running a JetPack-2.3 installation.

Even though their research contains many models and metrics applied towards ARM experimentation, it contains a few shortcomings. For instance, it fails to mention what dataset was used in the experiments, and it also does not justify the choice of Torch7 to the detriment of other broadly used frameworks.

This article (LIN et al., 2018) [8] proposes adaptations on some DNNs in order to reduce these original models' checkpoint storage size and training volume without losing much efficiency.

This tradeoff is performed using Fast Fourier Transform (FFT), implemented in Java and C++ with OpenCV. Also, the experiments were performed in LG Nexus 5, Odroid XU3, and Huawei Honor 6X mobiles built on ARMv8-A architecture (aarch64), using MNIST and CIFAR-10 datasets. The results claim that this OpenCV prototype requires less storage for the storage of execution parameters.

Euphrates (ZHU et al., 2018) [10] enters the related works section due to some reasons: First, it approaches the topic of object detection with CNNs, adding one more case of study on deep learning functional appliances. Also, it conducted tests in Jetson TX1 and TX2. The prototype implements and tests some optimizations from other related works, such as a tradeoff between the image processing engines to save energy without losing much system energy.

The work (ZHU et al., 2016) [11] approaches the practical utility of performing the design of neural networks and their implementations with the parallel processing in FPGAs aligned with a GPU.

By doing so, the system allows a tradeoff between performance and energy, which is why their framework's architecture was based on the calculation of convolutional neural network accelerators. This technique represents a logic of pre-processing, which aims to decipher the best configuration of workload distribution. CNN Lab distributes the workload amongst either Field-programmable Gate Array (FPGA) accelerators or GPU.

The hardware for the experiments was composed of Intel Corei7-4770 for CPU, Intel-Altera DE5 for FPGA, and Nvidia K40 for GPU. This work is worthy of being in the related works section due to its convolutional neural network experiments on an embedded platform and GPU experimentation. Also, this work projected future works to perform new experiments with Apache Spark and TensorFlow.

The article (WANG et al., 2016) [12] introduces a qualitative study of deep learning in distributed environments and also raises particular points towards the datasets. It selects six mainstream frameworks of Deep Learning and labels them by the presence or absence of special optimizations: operation scheduling, memory management, parallelism, and consistency.

Besides, the article mentions some DL models that may be adapted for database applications to attempt processing speed optimization. For instance, it mentions that RNNs could reduce NLP queries and translate these operations to SQL, which are intrinsically faster. Also, the work continues tackling database-related problems such as Query Planning, which also may leverage from an RNN model that would learn the database's elements and metadata.

The actual evolution of Deep Learning has been allowing new researches with several insights into large datasets. This assertion introduces the work (SHAMS et al., 2017) [13], which also stated the importance of evaluating resource management and performance because as these Deep Learning applications evolve, such resources become increasingly scarce.

For this, the article proposes an experiment to analyze the performance of three Deep Learning frameworks: Caffe, TensorFlow, and Apache. The tests were performed on hardware Intel E5-2680v2 Xeon Processor, IBM Power8 Processor, IBM Power8+ Processor with NVLink interface, NVIDIA Tesla P100 with NVLink interface, Intel Phi 7230 processor (KNL). These frameworks were tested in five Deep Learning models: AlexNet, VGG-19, GoogLeNet, LeNet, MNIST, ConvNet. The datasets used were: ILSVRC'12, MNIST, and CIFAR-10. The article provided valuable insights in comparing frameworks in computing time and results in scalability.

The work (SHI et al., 2016) [14] is a comparison between different Deep Learning frameworks with FCN, CNN, and RNN models, with synthetic and natural datasets. The experiments used the frameworks: Caffe, CNTK, MXNet, TensorFlow, and Torch. The models chosen to work on synthetic datasets were: FCN-S for FCN networks, AlexNet-S, and ResNet-50 for CNN networks. For real datasets, the following were chosen: FCN-

R model for FCN network, AlexNet-R, and ResNet-56 models for CNN networks, and LSTM model for RNN network.

The hardware used was composed of Intel CPU i7-3820, Intel CPU E5-2630 (x2), GTX 980, GTX 1080 2560, Tesla K80 GK210, and K80(x2). The experimental workload had many frameworks, machines, and models to test. However, it does not justify the choice of batch size in the training phase. Furthermore, the execution is specific to HPC, whereas it could benefit from GPU-to-edge processing architectures. The article ends with the realization that no framework can consistently outperform others in execution. It ends up leaving the question of performance optimization in these frameworks open for future work.

The work (SHI; CHU, 2017) [15] uses four Deep Learning frameworks: Caffe-MPI, CNTK, MXNet, TensorFlow. Four GPU Cluster nodes are used, each with four NVIDIA Tesla P40s, for 16 units. Several parameters of the stochastic gradient descent (SGD) algorithm of the above frameworks and hardware combined with CNN AlexNet, GooGleNet, and ResNet-50 models were evaluated. This work raises the excellent point that the models were well chosen in the methodology so that all combinations. Although the work uses several frameworks, it only uses high-performance graphics cards, while the present work aims to investigate the execution of edge devices. Furthermore, the decision to test just a few CNN models could have been reversed by inserting more CNN models or even FFN and RNN models.

The work (BAHRAMPOUR et al., 2015) [16] compares Caffe, Neon, TensorFlow, Theano, and Torch for AE and CNN's with MNIST and ImageNet datasets and LSTM networks using the IMDB dataset. The LeNet model was trained with the MNIST dataset for 11 CPU configurations and 8 GPU configurations with six different batch sizes. The AlexNet model was trained with the ImageNet dataset in 5 different CPU framework configurations and 12 GPU framework configurations with five batch sizes.

Autoencoders were trained with 11 CPU framework configurations and 5 GPU framework configurations. To test LSTM's, it was not possible to test Caffe as it did not have support for this functionality at the time of the study. Also, Neon and TensorFlow did not support variable-length input data within the same LSTM batch. Therefore, the comparison was made between Torch and Theano: Two configurations for CPU and two for GPU.

All experiments looked for Forward Time and Gradient Computation Time as metrics. The experiments were run on Intel Xeon CPU E5-1650 v2 @3.50GHz; Nvidia

GeForce GTX Titan X/PCIe/SSE2; 32 GiB DDR3 memory; SSD hard drive. The article ends by bringing the comparison results and providing valuable information about the frameworks.

The work (RUNGSUPTAWEEKOON; VISOOTTIVISETH; TAKANO, 2017) [17] tested the YOLO real-time object detection system. The training was run on Lenovo ThinkPad X230t with Ubuntu OS 16.04 and tested on NVIDIA Jetson TX1 and TX2. The results were validated with an NVIDIA Tesla P40. It is presented as an example of a purely experimental bias that uses TX2. However, very limited in terms of the scarcity of models and frameworks.

The work (ATTARAN et al., 2018) [18] experiments with KNN and SVM implementations in FPGAs, NVIDIA Tegra TX1, NVIDIA Tegra TX2, and Raspberry Pi 3 on a stress assessment system that takes into account body sensors to remove signals. Enter as yet another work that uses AI technique on TX2 Boards.

Table 3.1: Works with DL on SoCs and their Metrics

#	**FEATURES**													
	Type	**Models**	**Metrics**							**Frameworks**				
			Energy	*Inference Time*	*Accuracy*	*Latency*	*Memory Footprint*	*Training Time*	*Others*	*TF*	*Caffe*	*Theano*	*Torch*	*Others*
[1]	CNN	-AlexNet -SqueezeNet	Yes			Yes	Yes			Yes	Yes		Yes	Yes
[2]	CNN	-Inception -ResNet -MobileNet	Yes	Yes	Yes				Yes	Yes				
[3]	CNN AE GAN	-AlexNet -OverFeat -VGG16 -NiN -Chair -Pix2Pix	Yes			Yes			Yes	Yes				
[4]	CNN RNN (CNN+LSTM)	-AlexNet -VGG-F -VGG-VeryDeep-16 -YOLO -Fast R-CNN	Yes		Yes	Yes					Yes			Yes
[5]	CNN	-AlexNet, -customAlexNet -NiN -ENet(for ImageNet) -GoogLeNet -VGG-16 and VGG-19 -ResNet-18,-34,-50,-101,-152 -Inception-v3 -Inception-v4	Yes	Yes	Yes		Yes		Yes				Yes	
[6]	FNN	-CIFAR-10						Yes			Yes			
[7]	CNN RNN(LSTM)	-AlexNet -VGG16 -NeuralTalk	Yes			Yes			Yes		Yes			
[8]	CNN CNN(FFT)	******			Yes			Yes	Yes					Yes
[9]	CNN GAN	-AlexNet -VGG16 -NiN -Inception-v3						Yes	Yes	Yes	Yes			Yes
[10]	CNN	-YOLOv2Tiny -YOLO.MDNet	Yes		Yes									Yes
[11]	CNN	******	Yes					Yes	Yes					Yes
[12]	******	******	******							******				
[13]	CNN	-AlexNet -VGG-19 -GoogLeNet -LeNet						Yes	Yes	Yes	Yes			Yes
[14]	CNN FFN RNN	-AlexNet -ResNet-50 -FCN -LSTM		Yes	Yes					Yes	Yes		Yes	Yes
[15]	CNN	-AlexNet -GoogLeNet -ResNet-50							Yes	Yes	Yes			Yes
[16]	AE CNN RNN(LSTM)	-LeNet -AlexNet -AE -LSTM		Yes				Yes	Yes	Yes	Yes	Yes	Yes	Yes
[17]	DNN(CNN)	-YOLO -Tiny YOLO	Yes						Yes					Yes
[18]	ML	-SVM -KNN	Yes			Yes			Yes					Yes

After the careful reading of the articles of the state-of-the-art, they were registered on Table 3.1. This table allows the present work to define some methodological aspects of research in the next chapter. Also, regarding Table 3.1, the cells that contain asterisks mean that the work failed to specify that particular feature, even though it is considered to be in the state-of-the-art thanks to its contributions on the present work subject.

It is a well-established fact that CNN is the more famous type of network in Deep Learning literature. Both Tables 3.1 and 3.2 (see below), we verify that about 53,33% of the total of network types are CNNs. Also, 14 out of 18 works used it.

Therefore, it makes sense to include such a hegemonic network type in this work's scope. On the other hand, a small portion of readers might argue that this choice has been exhausted and does not deserve space in this work.

This work argues that the SoC devices are under speedy development in the face of these thoughts. This quick rise in a newer, more power-efficient architecture expands the SoC and DL test possibilities. Tests on these SoCs are feasible and necessary because different configurations of the same problem might give insights for better solutions while giving users a safe rock in the choice of their experimental combinations.

The second most used type was Recurrent Neural Network (RNN), with 16,67% occurrences on the total network types. These RNNs provide higher insights in time series and audio processing. Therefore they are not crucial to this work since the focus is on the majoritarian CNNs.

Finally, FNN presents a 10% occurrence rate in the network types. These networks can be simpler to program than CNNs and RNNs, but still, solve many problems with relatively low complexity and cost.

On the other hand, some network types were less used in these related works. For instance, Auto Encoders (AE) consist of nonsupervised learning methods, which raises the training complexity of the system. Because of this fact, this work chose not to engage in AE experimentation.

Under the circumstances of this work, few metrics are worthy of attention. As Table 3.1 shows, the most common metrics between the related works are Training Time, Precision, Accuracy, and Energy.

Training time consists of the time that the model's algorithm takes to perform all steps and epochs of training. It can be measured by evoking two-time variables during the extreme moments of the execution. Both variables receive the machine's time in String type, one at the beginning of the training stage and one at the end of the script. Then, the script casts the String to double and subtracts the end time with the starting time.

Since the training phase is still the most costly stage of most deep learning models and since SoCs have limited resources, it makes sense to focus on this specific set of frameworks, experiments, models, and datasets.

For instance, a complete experimental work under only one framework takes much time and does not give much insight into their benefits or challenges on ARM. On the other hand, a few well-chosen models co-scripted on proper installations of two different frameworks provide a better picture of DL on ARM. That is why this work focused on just two DL frameworks while emphasizing the number of models.

3.4 Very Similar Related Works

Since the NVIDIA Jetson board family is very recent, too few works have come close to this work's proposal to train the DL models in the ARM boards. Therefore, only a single work has made an effort to overview those of higher similarity.

The work of (LIU et al., 2019) performs an experimental study with up to ten CNN models on the NVIDIA Jetson TX2 and the TensorFlow framework. The work is very detailed-oriented regarding the metrics collection and seeks to determine if neural networks can be trained in mobile devices.

One short-come of this work is that it justifies the use of TensorFlow because there was no training framework, especially targeting training DL models on mobile devices. This statement is just not true since most frameworks have been supporting aarch64 builds from source since prior to the time of this work. Perhaps, the mentioned work only cared to use pre-built engines of the framework, which also does not give much insight when lacking results at another framework to serve as a frame of reference. On the other hand, the present work used two frameworks built from the source, which give plenty of comparison cases of real insight on how DL works on mobile devices.

Also, all of the experiments of (LIU et al., 2019) are focused on a single hardware configuration with all cores available plus GPU, which does not give a complete view of the hardware's shortcomings as they announce in their contributions.

Three-quarters of their experiments were executed on the CIFAR-10 dataset, which is ten times smaller than the mImgNet10 used in this present work.

However, the work does differ in contributions from the present work since it attempts to investigate more models and types of networks with more datasets and batch sizes, which brings some depth to its analysis. As far as the present work is concerned, the previous related work is the only one to perform similar experimentation closely.

Table 3.2: Hardware and Datasets of the State-of-the-art

	HARDWARE	DATASET
[1]	-MacBook Pro -Intel FogNode -NVIDIA Jetson TX2 -Raspberry Pi -Nexus 6P	******
[2]	-NVIDIA Jetson TX2 -NVIDIA P40 GPU	-ILVRSC 2012
[3]	-NVIDIA Jetson TX2 -NVIDIA Tesla K40C	******
[4]	-Samsung S7 -Samsung Note 4 -Sony Z5	-ILSVRC2012 -Pascal VOC 2007 -UCF101 -LENA
[5]	-NVIDIA Jetson TX1	-ILSVRC15
[6]	-Intel(R) Xeon(R) CPU W5590 @3.33 GHz -GPGPU NVIDIA GeForce GTX 980	-CIFAR-10 -ILSVRC2012
[7]	-CPU Core i7-5930k -GPU Titan X -mGPU (Tegra TK1)	-ImageNet
[8]	-LG Nexus 5 -Odroid XU3 -Huawei Honor 6X -IBM TrueNorth	-MNIST-CIFAR-10
[9]	-NVIDIA TITAN X GPU -NVIDIA K20X -NVIDIA K20	-LSUN dataset
[10]	-NVIDIA Jetson TX2	-VOT -Euprathes
[11]	-Intel Corei7-4770-Intel-Altera DE5 FPGA -NVIDIA K40	******
[12]	******	******
[13]	-Intel E5-2680v2 Xeon Processor -IBM Power8 Processor -IBM Power8+ Processor with NVLink interface -NVIDIA Tesla P100 with NVLink interface -Intel Phi 7230 processor(KNL)	-ILSVRC2012 -MNIST-CIFAR-10
[14]	-Intel CPU i7-3820 -Intel CPU E5-2630x2 -GTX 980 -GTX 1080 -Telsa K80 GK210	-ImageNet -MNIST -CIFAR-10
[15]	-NVIDIA Tesla P40	-ILSVRC2012
[16]	-Intel Xeon CPU E5-1650 v2 -Nvidia GeForce GTX Titan X	-ImageNet -MNIST -IMDB
[17]	-NVIDIA Jetson TX1 -NVIDIA Jetson TX2 -NVIDIA Tesla P40 -Lenovo ThinkPad X230t	-ILSVRC2012 -Pascal VOC -COCO
[18]	-ARM A53 (baseline) -TX2 GPU -TX1 GPU -Artix-7 100T FPGA -ASIC -Raspberry Pi 3B	-LifeShirt

4 METHODOLOGY

This chapter aims to explain the methodology that has been chosen for the proposed work. The following few sections explain the methodology, emphasizing the classification and its approach. Then, it describes the metrics used in the experiments, going through their types, conceptual differences, and practical usage. Finally, it ends with a validation section and then summarizing the essential items.

This work is experimental research that aims to test a combination of DL frameworks, models, and datasets under a parameterized and fair training environment. This experimentation scenario focuses on performing training loads on SoC boards to compare two powerful frameworks' performance by checking training discrepancies and efficiency limitations.

The state-of-the-art attested to the energy efficiency of the ARM architecture against x86 for many applications. The Jetson boards contain an embedded GPU and special memory, which make costly computer vision applications more feasible at the edge. Among these applications, Deep Learning gained much notoriety due to its considerable computational resource constraints proved a viable solution on many state-of-the-art articles.

Even though Deep Learning algorithms execution has been exhaustively proven on the ARM architecture, all these works lack a study regarding the training itself on the architecture mentioned above. The usual rule of thumb is to train the deep model in the cloud (offline training), then converge the model to a set of parameters that may be saved and loaded. This trained model can be loaded and adjusted into a different device from which it was trained through fine-tuning. Therefore, the traditional approach outsources the training process to more powerful x86-based architecture machines on the cloud.

Thus, the humble Jetson boards have not been investigated under Deep Learning training, which may be due to the higher effectiveness of the offline training. However, it is essential to mention that these boards have been evolving rather quickly. For instance, NVIDIA Jetson TX1 was released in 2016 with 4 Gigabytes of RAM, NVIDIA Jetson TX2 was released in late 2017 with 8 Gigabytes of RAM (as seen on Table 4.1), and the newest release of 2018 Jetson Xavier has 16 GB of RAM. This means that approximately every year, NVIDIA has introduced a new low-power board with twice the memory for more complex models and its data. Besides that, this memory called LPDDR5 may be scheduled to process for the Cortex A57 and Denver2 CPU cores, but it also may be

Table 4.1: ARM in Comparison with x86

Hardware	NVIDIA Jetson TX2	Azure NC24
Processor	ARM Cortex-A57 @ 2GHz / NVIDIA Denver2 @ 2GHz	Intel Xeon® E5-2690 v3 @ 2.60GHz
Cores	4 / 2	24
Architecture	ARM v8-A (64 bits)	x86_64
Memory	8 GB	224 GB
Network	1000 Mbps	1000 Mbps
GPU	256-core Pascal @ 1300MHz	4 x K80 GPU (2 Physical Cards)
GPU Memory	8 GB*	32 GB
Disk	1 TB HHD	1.44 TB SSD
Operating System	Ubuntu 16.04	Ubuntu 16.04

*Both CPU and GPU share the same board's memory.

reserved towards the CUDA-cores (GPU) through Direct Memory Access (DMA).

This interesting architectural approach raises the question: What are the limitations of these Jetson boards on Deep Learning training? From an objective perspective, many Deep Learning algorithms were not considered to be trained until the boom of GPUs and the introduction of the CUDA library. Perhaps the quick evolution of these boards deserves a closer inspection, thus allowing more insights on the future behavior and financial feasibility of the GPU-embedded products implemented with ARM architecture.

Furthermore, the quick evolution of smartphones that contain the same hardware architecture is noteworthy, mainly thanks to the rise in processing capabilities of the arm CPU, the embedded GPUs, and the battery life cycles. All these advancements emphasize the need for a study that unveils the limitations of ARM64 under current workloads.

Besides that, some cloud providers are investing in changing x86-based clusters into arm-based clusters. The main reason is because of the small energy consumption of arm computers, but also it is so because of the excessive heat that x86 machines produce when operating under high capacity, which disturbs the cluster's performance if not controlled and also increases energy costs, even more, thanks to the need of proper a cooling system.

This author lists a few scenarios where this study could aid: 1) edge topology with multiple devices with a poor or unreliable network that could take the training stages under its expenditures; 2) edge devices that suffer repeatedly unpredictable changes on its scenarios. Thus devices that must adapt quickly may profit more from an in-situ training; 3) a repetition of the second scenario with network unavailability; 4) students and professionals entering the AI studies, thanks to the low cost of the Jetson development kit and low power consumption.

As far as this author is concerned, this is the first work to attempt a thorough comparative study of DL training on GPU-embedded ARM devices with two frameworks. This work intends to attest to the success or incapacity of training state-of-the-art models and datasets on ARM64 boards. Thus, this work evaluates the hardware and application metrics within a well-controlled training scenario. Also, this work focuses on isolating the Cortex A57 CPU, Denver2 CPU, and CUDA Cores on scenarios that may provide more knowledge regarding these boards.

4.1 Methodological Approach

Regarding the state-of-the-art, this work prioritized the state-of-the-art research on respected academic research engines such as Google Scholar, IEEEXplore, Research Gate, arXiv, ACM Digital Library, and others. Besides journals and conferences, this work also performed a thorough search in the UFRGS Lumi database to determine if some related work had been conducted.

In order to ensure that this research's related work is up-to-date, this work opted to set a minimum time limit of three years prior to this its publication, with a margin of tolerance due to how little this topic has been explored, but also how it keeps rising as a trending topic in Computer Science.

When it comes to the DL models used in this work's experiments, this work studied the most used models from the related works to implement them manually on the frameworks. Also, this work initially compared the five main models that were described in most DL books, and surveys (GOODFELLOW et al., 2016; PATTANAYAK; PATTANAYAK; JOHN, 2017), i.e., LeNet, AlexNet, VGG, ResNet, and Inception, and compared them by three specific features -e.g., Number of Convolutional Layers, number of FC neurons, and the presence or absence of dropout layer- as seen on table 4.2.

The number of neurons in the Fully Connected layers raises the training time considerably. However, for profound models, the convolution parameters add quite an overload. Therefore, the convolution is the main factor for choosing a model's relevance for this work. At the same time, the FC layer presents a secondary factor, and the dropout is mainly used for unties.

Five models seemed too few to contemplate a deep analysis during the initial experiments. Thus, two more CNN models were added to the experiments: DenseNet and SqueezeNet. Besides that, all seven models were hard-coded according to MxNet and

Table 4.2: Comparison of CNN models - The table presents the most famous CNN models in literature sorted by appearance.

CNN MODELS	**Conv. Layers**	**Conv. Filter Kernels Size**	**Feature Maps**	Neurons in FC	**Dropout**
LeNet(LECUN et al., 1989)	2	C1, C2:5x5	22	204	No
AlexNet(KRIZHEVSKY; SUTSKEVER; HINTON, 2012)	5	C1:11x11 C2:5x5 C3,C4,C5:3x3	1376	8192	0.5
VGG(SIMONYAN; ZISSERMAN, 2014)	13	C1-C13:3x3	4224	9192	0.5
Inception(HE et al., 2016)	95	C1-C6:3x3 C7:(1x1,3x3,3x3); (1x1,3x3); (1x1); (1x1) x3 C8:[(1x1,1x7,7x1,1x7,7x1), (1x1,1x7,7x1), (1x1), (1x1)]x5 C9:[(1x1,3x3,(1x3),(3x1)), (1x1,(1x3),(3x1)), (1x1), (1x1)]x2	13552	1024	0.4
ResNet(SZEGEDY et al., 2016)	151	C1:7x7 C2:(1x1,3x3,1x1)x3 C3:(1x1,3x3,1x1)x8 C4:(1x1,3x3,1x1)x36 C5:(1x1,3x3,1x1)x3	71872	1000	No
DenseNet(HUANG et al., 2017)	117	C1:7x7 C2:(1x1,3x3)x6 C3:(1x1,3x3)x12 C4:(1x1,3x3)x24 C5:(1x1,3x3)x16	Over 100k	1000	0.5
SqueezeNet(IANDOLA et al., 2016)	2882	*Not similar to the others	*Does not apply	None	0.5

Pytorch API. The codes are available on the author's Github repository[1].

Naturally, the small MNIST dataset is easily stored for model training and does not give much memory or computational trouble in ARM64 boards. Models like LeNet and AlexNet usually present values higher than 99% regarding accuracy when training in this simple dataset. Thus, this dataset has been unconsidered for this work's experiments.

CIFAR-10 was initially considered for this work's experimental setup; however, its small size (as can be seen further on Table 5.3) did not provoke a significant difference in the experiments between frameworks, thus making the Comparisons of this work's experiments harder to be verified. Also, many CIFAR-10 results got lost during the initial research stages thanks to a primary partition malfunction on the centralized disk used for this work. After that, all results of this work started to be stored under triple redundancy with backups on the cloud. Still, thanks to the reasons mentioned earlier, the CIFAR-10 dataset was not used in this work's experiments.

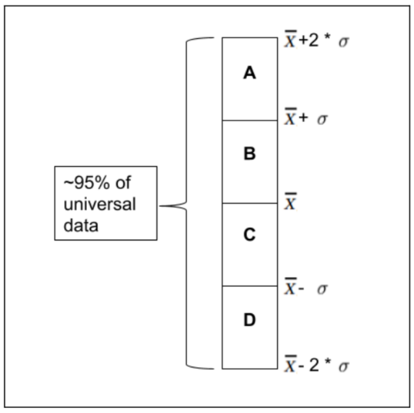
Furthermore, it is impossible to train a model with an ImageNet dataset on SoCs such as used in this work, since these ARM boards have only up to 8GB of RAM (see Table 4.1) and the ImageNet (RUSSAKOVSKY et al., 2015) dataset has over 140GB -i.e., only 5,44% of the total needed only to load the dataset-. Besides loading the Operating System and all programs into memory, the boards would allocate memory for the

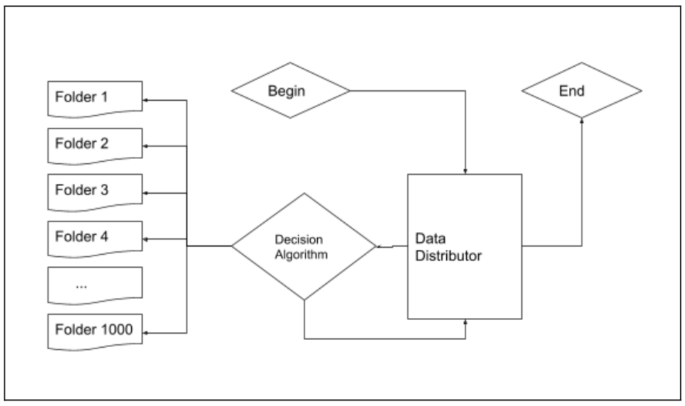
[1]https://github.com/Bfzanchetta/DLARM

giant dataset and store all the training parameters (weights), which would at the very least double the requirements of the entire ImageNet dataset. In order to address this inherent memory limitation, this work developed a modest but fair dataset selection and distribution algorithm.

First, the algorithm pre-calculated the average file size of ImageNet's 1.282.167 pictures, followed by its standard deviation and quarters. The results gave an average file size of 116.672,43639588 Bytes (116,7KB), a standard deviation of 338.589 Bytes. Also, since the full dataset comprises one thousand different directories with natural pictures, it is noteworthy to mention that it also contains an average of 1282,167 picture files per class folder.

Figure 4.1: Normal Data Distribution (left) and Dataset Algorithm's Model (right)





Then, the subject that uses this dataset selection algorithm must decide on a trade-off: either have many folders -and thus many classes- with very little information on each folder, or few folders with much data. Naturally, this work decided on the second choice because it is less likely to cause underfitting.

Finally, the dataset mentioned above's size elements passed into a Shell Script that would select the new reduced and balanced dataset baptized as mImgNet10 according to a pre-defined logic, see the algorithm's pseudocode in Figures 4.2 and 4.3.

Also, it is noteworthy that a random draw in zone A specifies a picture that contains the average value of Bytes in size plus up to twice the standard deviation also in Bytes. Then, according to the algorithm defined in 4.3, the next draw for the subsequent folders (e.g., from two to ten) must be in the same data size zone. Then, a balance is updated by the end of all draws from the same data zone. Random draws from zones A and B are added to the balance, and values from zone C and D are subtracted. This balance restricts the zone of the next random draw in order to equalize the dataset.

Figure 4.2: ImageNet Subset Selection Algorithm for Specific Purposes: Pseudocode

```
ImageNet Subset Selection Algorithm for Specific Purposes: Pseudocode

Input: Dataset Folder, Dataset Labels File
Output: Chosen Array

Begin
1.  Define an integer N as the number of classes in the desired subset.     #e.g.: N=12
2.  Subfolders[ ] ←read_input(Data Labels File)
3.  Define a 1-D array.      #e.g.:Chosen[ ]
4.  For i=0 to N do
5.      sortedNumber←random(1000)
6.      If Chosen[ ].length() == 0 then
7.          Chosen[ i ]←Subfolders[sortedNumber]
8.      Else
9.          For j=Chosen[ ].length() downto i do
10.             If sortedNumber == Chosen[ j ] then
11.                 sortedNumber←random(1000)
12.                 j=Chosen[ ].length()-1
13.             End-if
14.             Chosen[ i ]←Subfolders[sortedNumber]
15.     End-If
16. End-for
End
Return Chosen[N]
```

Figure 4.3: ImageNet Subset Statistical Analysis: Pseudocode

```
ImageNet Subset Statistical Analysis: Pseudocode

Input: $\overline{X}$, $\sigma$, Chosen Array, N
Output: Subset S validated for use

Begin
1. Balance := 0
2. For i=0 to 1000 do
3.     For j=0 to N do
4.         aux := PickRandomFile( )
5.         If sizeof(aux) >= (X-(2*σ)) &&  sizeof(aux) < (X-σ) then          #Zone D
6.                 ChosenArray.add(aux)
7.                 Balance += X- sizeof(aux)
8.         Else If sizeof(aux) >= (X-σ) &&  sizeof(aux) < X then          #Zone C
9.                 ChosenArray.add(aux)
10.                Balance += X- sizeof(aux)
11.        Else If sizeof(aux) >= X &&  sizeof(aux) < (X+σ) then          #Zone B
12.                ChosenArray.add(aux)
                   VerifyTheBalance()
13.                Balance += sizeof(aux) - X
14.        Else If sizeof(aux) >= X+(σ) &&  sizeof(aux) < (X+2*σ) then          #Zone A
15.                ChosenArray.add(aux)
16.                Balance += sizeof(aux) - X
17.        Else
18.                DrawAgain();                          # Rare cases.
19.        End If
20.    End For
21. End For
End
Return Chosen[ ]
```

The methodology that has been applied in this work is inspired on (JAIN, 1990), which suggests 30 workloads repetitions by experimental scenario. However, thanks to these board's hardware constraints in processing power and the empirical observation of low standard deviation between the results, many steps had to be adjusted to obtain a factual conclusion time for the entire experimental set.

With that in mind, this work reduced the total required repetitions to up to 5 times for each model. The only exception for this value was detected on CPU-only cases, mainly because many experiments took too much time could not be conducted as many times as the GPU-aided. Some of these long-timed experiments also presented difficulty to log the results thanks to runtime problems, operating system malfunctions, and energy peaks, which happened more noticeably on these cases thanks to their extended execution time.

All training sets were passed into the model with shuffle operation, which randomizes each execution into a unique experiment, thus increasing experimental fairness between every experimental instance. Allied to that, since each model training initiates the weight matrices randomly, it is worth mentioning that there is no reason that any training obtained in this work's board would surpass an x86-based machine training based on the board's properties or model implementation. Since each execution is initiated randomly and suffers more random transformations along with the training procedure, the only factor that would contribute to a better or worse training result is eventuality or luck, i.e., since x86 machines are inherently faster than ARM, more random executions of the same algorithm incurs in more trials to achieve a more significant result.

Many learning algorithms rely on gradient descent learning to perform its error back-propagation. These algorithms can be trained either by batches or online training (mini-batches or data stream). The main difference between these two techniques lies in the approach they use for gradient computing: batches calculate the weight changes by each epoch, and online training computes weights chances by each entry, even within the same epoch (WILSON; MARTINEZ, 2003).

A single epoch value does not suffice the training algorithms. Thus, the author of the models must increase or decrease the number of epochs to obtain a well-tuned model. Few epochs incur ill-trained models that usually do not present good accuracy results. Too many epochs usually surpass optimal algorithm convergence and start harming the network's accuracy rates thanks to a process named overfitting. Based on these facts, this work empirically tested good values to generate an epoch number that would demonstrate good time and accuracy convergence without damaging the total experimental execution time. The final value of 10 training epochs proved to suffice this work's purposes.

On the other hand, larger batch values usually entail more accurate gradient approximations with a trade-off regarding linearity reduction, thus reducing the model's capacity (GOODFELLOW et al., 2016). Moreover, multicore architectures suffer resource underutilization on such workloads. However, since the proposed experiments are developed mainly for SoC boards with limited processors, this effect will be mitigated. Even though the authors (WILSON; MARTINEZ, 2003) concluded that online training outperforms batch training -which contradicts the literature's common sense-, it also stipulates that small-sized batches can cause a regularizing effect on (WILSON; MARTINEZ, 2003). For the previously mentioned findings, this work defined the batch size of 4 as the standard value for all applications in this work's experiments.

Two basic types of metrics monitored the experiments: the first type is the Application Metric (AM), which runs together with the application runtime in the Python engine; the second type is the Hardware Metric (HM) which is collected directly by the board's proprietary firmware tegrastats, and it was set to be logged at a 1-second interval.

As seen below in Table 4.3, the time metric was measured through Python due to its high precision. Both frameworks collected Loss (i), Top1 (ii) and (iii) Top5 Accuracy, which is defined as: (i) The distance between the model's gradient and the given Data; (ii) the class which gets more correct predictions on the test phase based on an annotated dataset, but without previous knowledge of the class name beforehand; (iii) the five classes with most incredible prediction accuracy. These Application Metrics were gathered at Python's runtime and logged into text files formatted such as this:

.

```
Epoch 0. Loss: 2.1653266438665644, Train acc 0.211615384615, Test acc 0.216
Epoch 0 training: err-top1=0.863154 err-top5=0.437385
Epoch 0 time cost: 445.511252
Epoch 1. Loss: 2.0503304842100443, Train acc 0.269153846154, Test acc 0.318
Epoch 1 training: err-top1=0.816077 err-top5=0.370077
Epoch 1 time cost: 446.088975
...
Epoch 9. Loss: 1.399470929865164, Train acc 0.545076923077, Test acc 0.606
Epoch 9 training: err-top1=0.633992 err-top5=0.187554
Epoch 9 time cost: 446.179313
```

The Hardware Metrics were collected by tegrastats monitor, an NVIDIA proprietary monitoring firmware with good precision for software analysis. The tool is configured for collection on every pre-determined second, whereas the output is logged into a disk text or comma-separated file. Tegrastats was configured to collect metrics every 1 second to register hardware execution metric collection because smaller period values increase the log's length significantly. Every second of the firmware collection presented this form:

.

RAM 573/7860MB (lfb 1067x4MB) **SWAP** 0/2048MB (cached 0MB) **CPU** 48%@2035, off,off,34%@2035,30%@2035,38%@2035 **EMCFREQ** 0% **GR3DFREQ** 0% **PLL**@36.5C **MCPU**@36.5C **PMIC**@100C **Tboard**@32C **GPU**@35C **BCPU**@36.5C **thermal**@36.5C **Tdiode**@33.75C **VDDSYSGPU** 229/229 **VDDSYSSOC** 611/611 **VDD4V0WIFI** 19/19 **VDDIN** 3824/3824 **VDDSYSCPU** 1146/1146 **VDDSYSDDR** 713/713

The HMs were parsed into smaller CSV files, which were then plotted into hardware consumption graphs that can be seen on the Figures of Annex 2 (or Chapter8). The energy use collected by tegrastats is present in the log variables VDDSYSGPU, VDDSYSSOC, VDD4V0WIFI, VDDIN, VDDSYSCPU, and VDDSYSDDR. These values attest to each board component's instantaneous power consumption in Watts. These values were confirmed by power measuring smart-plugs WeMo Insight that attested whether the amount of energy registered by tegrastats was factual or not with the plug's instantaneous power consumption and throughout the experiments. The sum of the six-part values obtains the final power consumption. Also, it is worth mentioning that the CPU-only experiments never registered any values other than zero for GPU power consumption.

Table 4.3: Experiment Metrics, Units, Origins and Types

ID	Metric Name	Unit	Tool	Type
M1	Time	Seconds	Python and tegrastats	AM
M2	Loss	Likelihood	PyTorch and MxNet	AM
M3	Top1 Accuracy	Percentage	PyTorch and MxNet	AM
M4	Top5 Accuracy	Percentage	PyTorch and MxNet	AM
M5	CPU	Percentage	tegrastats	HM
M6	GPU	Percentage	tegrastats	HM
M7	RAM Memory	Utilization	tegrastats	HM
M8	SWAP Memory	Utilization	tegrastats	HM
M9	Power	Watts	tegrastats	HM

It is essential to mention that the tegrastats registered no power consumption on the GPUs during the entire CPU-only experiments. On the other hand, GPU-aided experiments correctly logged power consumption on all components, such as CPU and GPU.

4.2 Research Classification

This section mentions the classification that fits this particular research, just as it explains the criteria used to select the study's variables and prepare the experimental environment.

As the study grows, problem formulation, hypothesis construction, and variable relationship identification constitute critical steps of the research's conceptual system. According to (GIL, 2008), the research needs to confront the problem's theoretical overview in an attempt to classify it regarding its objectives and theoretical proceedings. Regarding the objectives, this research may be considered exploratory and descriptive.

It may be declared exploratory due to the inclusion of the identification and familiarization with the problem. This way, the problem becomes explicit and allows the construction of a hypothesis, allowing bibliographic research and case study shape. Also, it can be classified as descriptive because it uses standardized techniques to establish variable relations and describe phenomenon characteristics.

In parallel with the objectives, this research can be classified by technical proceedings, considered bibliographic and experimental. It may be considered bibliographic because it is developed on top of previously elaborated scientific material -books and articles- to achieve a specific goal. It may be considered experimental because it determines a study object and selects the variables that influence or may influence it. Only then, defining controlling and observation techniques towards the variable's effect on the object of study.

5 EXPERIMENTS AND RESULTS

This chapter presents this work's hardware and software setup, the experimental configurations and workloads, and the interpretation of the results.

5.1 Hardware Setup

The NVIDIA Jetson family contains energy-efficient GPU-embedded platforms, allowing AI to achieve better energy efficiency than x86 architectures, with good performance over the edge.

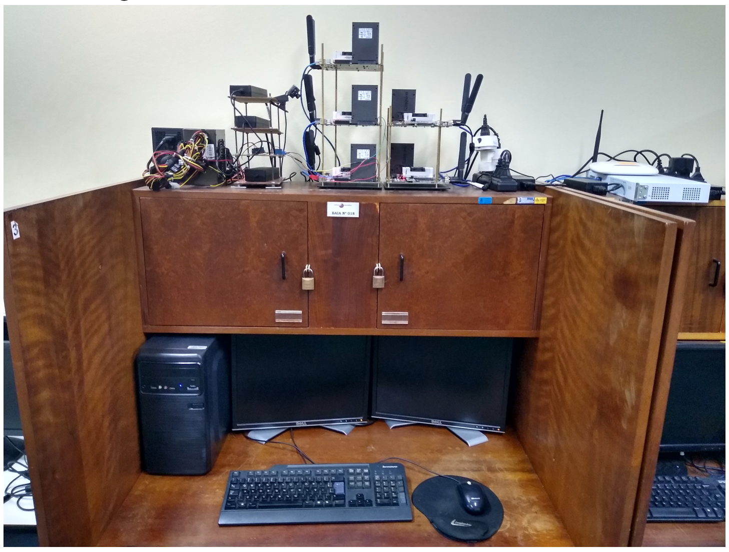
This work opted to use the board NVIDIA Jetson TX2 as the System on Chip of choice. The TX2 is equipped with Denver 2 and 4 Cortex-A57 processors, with 64-bits architecture ARMv8. The ARMv8 boards' technical specifications are detailed in Table 5.1.

Table 5.1: Hardware Specifications

Device	**NVIDIA Jetson TX1**	**NVIDIA Jetson TX2**
Architecture	ARMv8-A (64 bits)	ARMv8-A (64 bits)
Processor	Cortex-A57/ Cortex-A53	Denver2/ Cortex-A57
Cores	4/4	2/4
Max Clock	1,9 GHz/ 1,3 GHz	2 GHz/ 2GHz
Memory	4 GB LPDDR4 @1600MHz	8 GB LPDDR4 @1866MHz
L1 Cache	48KB(I)/48KB(D)/ 32KB(I)/32KB(D)	128KB(I)/64KB(D)/ 32KB(I)/32KB(D)
L2 Cache	2 MB shared/ 512 KB shared	2 MB shared/ 512 KB shared
GPU	256 Maxwell Cores @1000Mhz	256 Pascal Cores @1300MHz

Also, this work's test environment is comprised of four NVIDIA Jetson TX2 boards, with 240 GB Kingston SSD each, connected to a Gigabit Ethernet Switch with a link speed of 1000 Mbps.

Figure 5.1: ARM CLuster at Research Lab

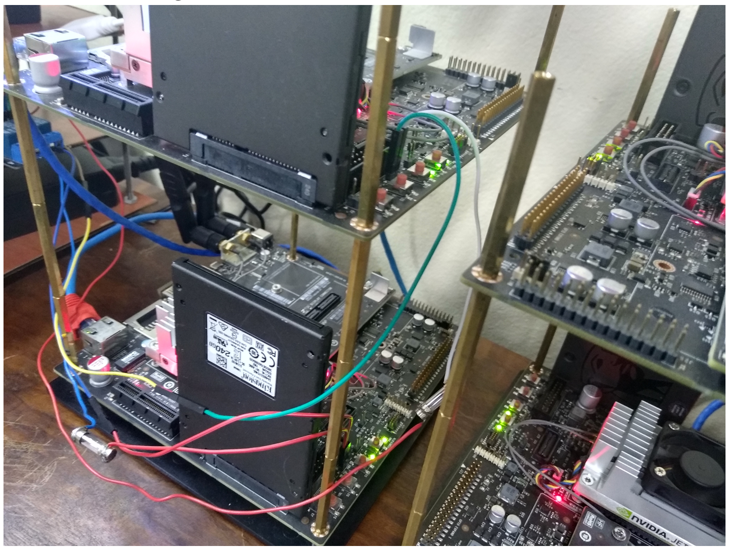


The Figures 5.1 and 5.2 demonstrate two different stacks: a smaller stack with two TX2 and a more significant stack with two TX2 and a TX1. However, TX1 was not used in this works' experiments and was just used for software setup, model implementation, and specific debugs.

The boards were all stacked by twenty purchased M3 brass standoff spacers. A base desktop (seen below the boards on Figure 5.1) was placed in the desktop in order to assist boards' flash and software update.

Also, all boards were configured following the same precise configuration steps to avoid different packages and environment mismatches. Furthermore, each boards' fan was set automatically to always work by default.

Figure 5.2: Detailed ARM Cluster



5.2 Software Stack

The boards are flashed with NVIDIA JetPack v4.2.1, which comprises an altered version of Ubuntu 18.04 LTS specific for Tegra boards. This work uses the frameworks PyTorch at version 1.3.0a0+5ec1c29 and MxNet at version 1.5.0, both built from the source.

The most noteworthy packages used in these boards configuration are listed with their versions and usage below:

- CUDA 10.0.2: NVIDIA's tool implements many C optimizations for matrix and Floating-Point operations that accelerate graphics.
- cuDNN 8.0.0: NVIDIA's deep neural network optimizer performs calculations on the early stage of training to reduce additional stage time for convergence.
- OpenCV 3.4.8: library utilized for image or video acquisition, manipulation, and rendering, great for mobile and image applications.
- Protobuf libprotoc3.8.0: is a package developed by Google to serialize structured data as JSON or XML, promising to be more optimized. Largely used for many deep learning framework build-ups.
- NCCL 2.7.5-ga: NVIDIA's library serves as a back-end for multi-GPU distributed process communication. It is not supported by TX2 but is required for some builds.
- oMPI 3.0.0: is an adapted version of the Message Passing Interface (MPI), serves to provide the back-end for distributed training but is also required for most builds.
- Java 8u281: Java Development Kit 8 for aarch64. It can be used for model implementation but mostly for special installation procedures during framework setup.
- Python 3.6.9: the main programming language in which the models are written. This work started with Python 2.7, but it got recently discontinued.
- Gluon: Computer vision toolkit that aids with object loading transformations and many tasks of MxNet.
- Torchvision: Respective computer vision library for PyTorch, with many data and model transforms, and other DL useful features.

5.3 Workloads

This work created the experimental workload by attempting the permutation of the classic five CNN models described in the Background chapter plus the two late-entries CNN (i.e., DenseNet and SqueezeNet) with a reduced ImageNet dataset -here called mImgNet10-.

The models were hard-coded relying on the frameworks API documentation[1][2] and Github[3],[4]. This was done in order to increase the degree of fairness in the experiments, i.e., to ensure that the models' transformations written for MxNet are as close as the PyTorch API allows, and vice-versa. The set of coded models, experimental setup, and tutorials are available at the author's Github repository[5].

The seven CNN models that can be seen in table 5.2 were mapped as in their MxNet and PyTorch implementation. So, model 1 stands for LeNet, and the codes 1M and 1P stands for LeNet implemented on MxNet and PyTorch, respectively.

Table 5.2: Model Mapping for Experiments

Model Name	**Shortcut**	**MxNet's ID**	**PyTorch's ID**	**Color Tag**
LeNet(LECUN et al., 1989)	LE	1M	1P	Red
AlexNet(KRIZHEVSKY; SUTSKEVER; HINTON, 2012)	ALX	2M	2P	Orange
VGG(SIMONYAN; ZISSERMAN, 2014)	VGG	3M	3P	Yellow
Inception(HE et al., 2016)	INC	4M	4P	Green
ResNet(SZEGEDY et al., 2016)	RES	5M	5P	Blue
DenseNet(HUANG et al., 2017)	DNS	6M	6P	Indigo
SqueezeNet(IANDOLA et al., 2016)	SQZ	7M	7P	Violet

These models were combined with reduced mILSVRC (mImgNet10) dataset accordingly. For instance, MNIST and CIFAR were not used in this work, but they are specified in the table below for comparison purposes. Just as the models, the datasets were also branded with specific IDs (see Table 5.3) to facilitate the reading of the results.

[1]https://pytorch.org/docs/stable/index.html
[2]https://mxnet.apache.org/versions/1.5.0/
[3]https://github.com/pytorch/pytorch
[4]https://github.com/apache/incubator-mxnet/tree/1.5.0
[5]https://github.com/Bfzanchetta/DLARM

Table 5.3: Dataset Size Specification

ID	Dataset	Application	Volume	Size
D1	MNIST	Image Classification	11.59 MB	Tiny
D2	CIFAR-10	Image Classification	163 MB	Small
D3	mImgNet10	Image Classification	1.6 GB	Medium

It is essential to mention that all models had their configuration parameters matched amongst frameworks due to experimental fairness. All models were trained for ten epochs, and the datasets were loaded in batches of size 4 with shuffle enabled.

Each model-dataset combination has been executed only with CPU and with GPU to test this board's chip performance. Also, NVIDIA Jetson TX2 allows manual enabling/disabling of specific CPU cores, allowing different experimental settings. Thus, this work isolated the four Cortex A57 and the Denver2 into two separate cases, plus the third entire 6-cores case (4xA57 + 2xDenver2).

On top of these three CPU configurations, the experiments were designed to test them with and without GPU. Based on that, some scenarios were proposed and labeled accordingly (see Table 5.4).

Table 5.4: Possible Hardware Configurations

TX2 Environment	Hardware Settings
CPU-CFG1	4x Cortex A57
GPU-CFG1	4x Cortex A57 + GPU
CPU-CFG2	2x Denver2
GPU-CFG2	2x Denver2 + GPU
CPU-CFG3	4x Cortex A57 + 2x Denver2
GPU-CFG3	4x Cortex A57 + 2x Denver2 + GPU

It is essential to mention that other settings were initially proposed, such as i) a single Denver2 core; ii) a single Cortex57 core; However, these cases increased dearly the already expressive training times displayed in this work. So, these fine-grained cases had to be withdrawn.

5.4 Experimentation

In order to conduct this experimental study, comparisons had to be proposed. These allow model and framework fair comparisons of the training and validation. Table 5.5 shows the test scenarios that concluded training with success, and thus will be discussed in the sections that follow.

Table 5.5: Table of Complete List of Experiments in mImageNet10 dataset with Results

		LE	ALX	VGG	INC	RES	DNS	SQZ
CPU-CFG1	MxNet	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	PyTorch	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CPU-CFG2	MxNet	Yes	Yes	Yes	No	No	No	No
	PyTorch	No	No	No	No	No	No	No
CPU-CFG3	MxNet	Yes	Yes	Yes	No	No	No	No
	PyTorch	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GPU-CFG1	MxNet	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	PyTorch	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GPU-CFG2	MxNet	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	PyTorch	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GPU-CFG3	MxNet	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	PyTorch	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Upon a quick look at the previous table, it is noticeable that not all configurations have a concrete case for comparisons. That is due to multiple factors, such as model script errors, automation script errors, results lost/file corruption. Also, every completed experiment was thoroughly checked for correctness, and only the complete ones were included in this table. So, this work's comparisons are listed and described below:

Comparison 1: GPU-CFG1 trained and validated at D3 A collection of experiments attests to the extent of the collaboration between the CortexA57 and the GPU.

Comparison 2: GPU-CFG2 trained and validated in D3 Gathers the quantitative results of the Denver2 and GPU experiments.

Comparison 3: GPU-CFG3 trained and validated in D3 This Comparison gathers the joint case of both CortexA57, Denver2, and GPU.

Comparison 4: CPU-CFG1 trained and validated at D3 Follows the same pattern as Comparison 1 but disabling the GPU.

Comparison 5: CPU-CFG2 trained and validated at D3 Executes tests solely with the Denver2, thus without the GPU.

Comparison 6: CPU-CFG3 trained and validated at D3 Comprises similar experiments but with all six available CPU cores and no GPU.

5.4.1 Comparison 1: GPU-CFG1 trained and validated at D3

This experiment was conducted on all seven models (see Table 5.2) on top of the mImgNet10 dataset (D3) over a hardware configuration that is comprised solely of four available Cortex A57 CPU-cores and all of the GPU's 256 Pascal cores. All of the fourteen test cases finished their 10-epochs training and validations with success in both frameworks, as listed in Table 5.5.

Table 5.6: CNN Models on mImgNet10 - Results at GPU-CFG1

ID	Total Time	Mean Time/Epoch	Power	Loss	Average Val. Acc	Top5
1M	3094	303.9±0.8	17.96±0.8	1.576±0.15	0.507±0.07	0.832
1P	2910	285.6±11.4	15.84±0.6	1.659±0.19	0.466±0.08	0.897
2M	4462	439.1±0.2	24.31±0.9	1.728±0.24	0.456±0.12	0.724
2P	8231	817.1±4.2	17.05±0.4	1.358±0.37	0.592±0.14	0.935
3M	14387	1413.5±5.6	29.22±0.9	1.948±0.21	0.342±0.10	0.649
3P	34181	3377.1±5.5	17.67±0.3	1.741±0.27	0.466±0.11	0.898
4M	8582	844.1±2.3	17.96±0.5	2.133±0.18	0.236±0.06	0.581
4P	15077	1486.7±1.4	17.89±0.4	1.521±0.28	0.571±0.11	0.931
5M	29741	2928.8±2.0	20.75±0.3	1.573±0.19	0.513±0.06	0.811
5P	50721	5012.1±3.5	18.51±0.2	1.557±0.34	0.476±0.13	0.897
6M	64648	6355.6±159.4	17.42±0.8	2.034±0.08	0.304±0.04	0.707
6P	29028	2875.8±0.9	17.92±0.4	1.184±0.27	0.702±0.09	0.960
7M	6665	656.5±0.3	17.74±0.5	2.235±0.09	0.195±0.03	0.519
7P	10440	1026.0±1.0	16.22±0.3	1.475±0.29	0.527±0.10	0.914
Units:	**s**	**s**	**W**	**loss***	**%**	**%**

The results in Table 5.6 reveal that most models converged acceptably with average validation accuracy of a little over 50%. It is essential to mention that the Accuracy's average and standard deviation values are not a product of the ARM platform but derive from the experimental choices applied in this work, such as dataset selection methodology, number of epochs, random weight initialization, and random dataset shuffle and distribution. Besides that, this work does not aim to find ideal or optimal training weights for these models. Thus, even though the accuracy results might be considered low in many cases, such as in 7M and 4M, they are still relevant to check if a model suffered a huge convergence problem (such as values below 5 to 10%) or not.

The MxNet models perform relatively worse than PyTorch, except for the two simpler CNNs: LeNet and AlexNet. Still, MxNet's average time per epoch is smaller than PyTorch in 5 out of 7 test cases. Furthermore, MxNet registers a shallow standard deviation time per epoch, unlike PyTorch, which registers great values proportional to the scale of each experiment. These facts point out to a tendency that PyTorch is much more unpredictable than MxNet, and thus more prone to oscillations during the training process.

It is also important to mention that the significant difference of time values throughout all Comparisons, such as 2910 seconds in 1P and 64648 in 6M, is a reflection of the deepness increase of the models, i.e., the increase of convolutional layers and kernels, and the increase of neurons in the fully connected layer. And since the models were categorized based on deepness, it is expected that lower ID models present shorter training times than high ID models.

Also, every case but LeNet and DenseNet obtained a faster training and validation performance on MxNet rather than PyTorch. Table 5.7 shows that these five cases obtained from 36.16% to 57.91% shorter execution times in MxNet as opposed to PyTorch.

Table 5.7: Ranking and Evaluation for CNN Models on mImgNet10 at GPU-CFG1

FEATURE	MODELS	LE	ALEX	VGG	INC	RES	DNS	SQZ
Total Execution Time	Faster Framework	PT	MX	MX	MX	MX	PT	MX
	Difference(s)	184.0	3769.0	19794.0	6495.0	20980.0	35620.0	3775.0
	Difference(%)	5.95	45.79	57.91	43.08	41.36	55.10	36.16
Avg.Time/ Epoch	Faster Framework	PT	MX	MX	MX	MX	PT	MX
	Difference(s)	18.3	378.0	1963.6	642.6	2083.3	3479.8	369.5
	Difference(%)	6.02	46.26	58.14	43.22	41.57	54.75	36.01
Avg. Power	Most Efficient Framework	PT	PT	PT	PT	PT	MX	PT
	Difference(W)	2.120	7.260	11.550	0.070	2.240	0.500	1.520
	Difference(%)	11.80	29.86	39.53	0.39	10.80	2.79	8.57
Avg. Loss	Best Converging Framework	MX	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.083	0.370	0.207	0.612	0.016	0.850	0.760
	Difference(%)	5.00	21.41	10.63	28.69	1.02	41.79	34.00
Avg. Val. Accuracy	Most Precise Framework	MX	PT	PT	PT	MX	PT	PT
	Difference(abs)	0.041	0.136	0.124	0.335	0.037	0.398	0.332
	Difference(%)	8.80	29.82	36.26	141.95	7.77	130.92	170.26
Avg. Top 5	Best Prediction Framework	PT	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.065	0.211	0.249	0.350	0.086	0.253	0.395
	Difference(%)	7.81	29.14	38.37	60.24	10.60	35.79	76.11

Six of seven model comparisons presented minor power consumption on PyTorch than on the MxNet framework regarding the energy consumption. This energy behavior, added with the shorter execution times on MxNet, suggests that the MxNet framework is more optimized than PyTorch for GPU, with slightly higher power consumption as a trade-off. The energy results of this and all Comparisons will be further analyzed in the section Summary of Comparison Best Results.

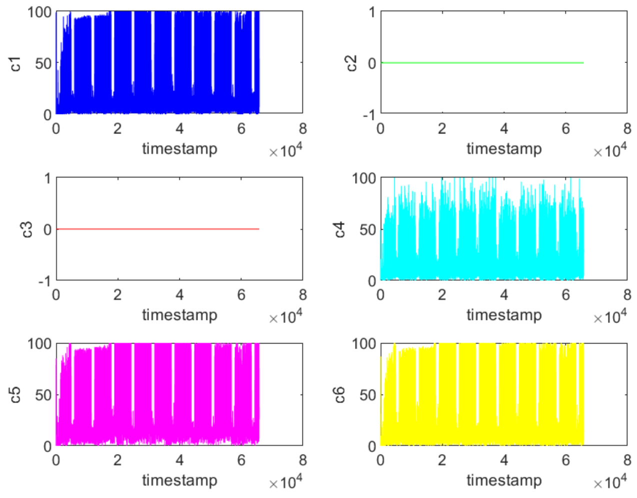
Furthermore, the difference in time and energy between models from both frameworks becomes more noticeable starting from the second model (AlexNet), where MxNet beats PyTorch's execution time twofold but with almost the same increase in energy consumption. Then, for deeper layered models, time keeps a similar 2:1 ratio, although energy consumption piratically levels between both frameworks. This may point to some GPU optimizations on MxNet that might perform a better workload scheduling for very small or medium model training.

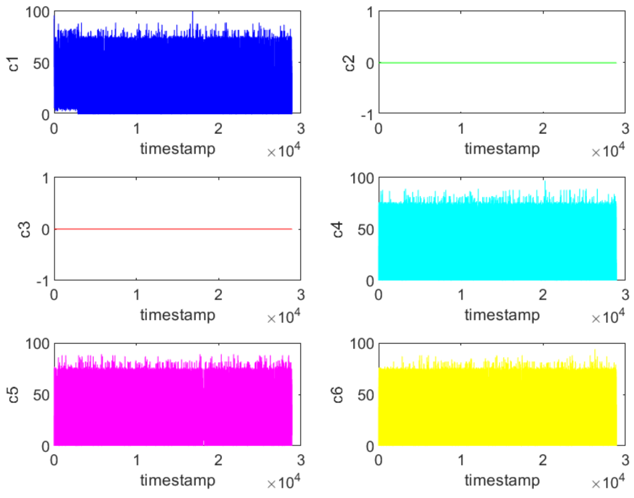
On the other hand, model 6 (DenseNet) presented a divergent behavior from its peers, whereas they behaved in a 2:1 execution time from PyTorch to MxNet, this ratio flipped by more than two in favor of PyTorch's execution time. The figures presented in Table 5.8 show the hardware monitor metrics for DenseNet in both frameworks. For this particular model, the framework MxNet shows more difficulty in resuming sequential epochs of training, with ten apparent gaps of hardware not being used at all.

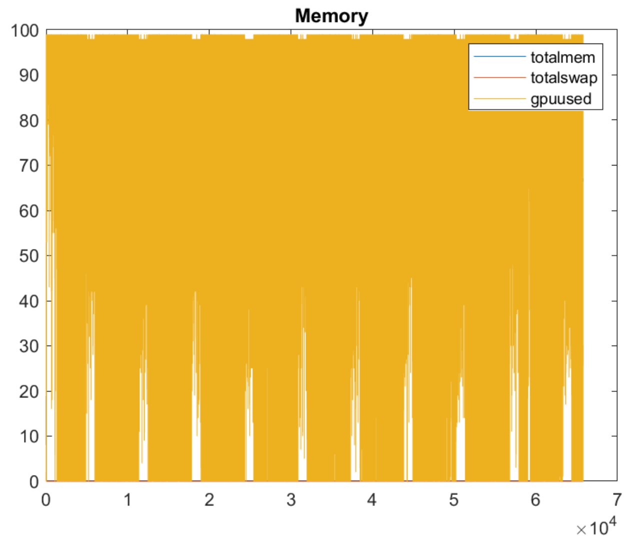
Also, by comparing MxNet's graph on the left to PyTorche's on the right, the memory graph reveals that PyTorch's GPU has not performed under 10% of its capacity. So, even though PyTorch's CPU works below 90% as opposed to MxNet's high CPU frequency, this accounts for better overall performance for a deep neutral training. In other words, MxNet struggled to synchronize CPU and GPU frequency, which created a bottleneck on the board's CPU and highly affected the execution time.

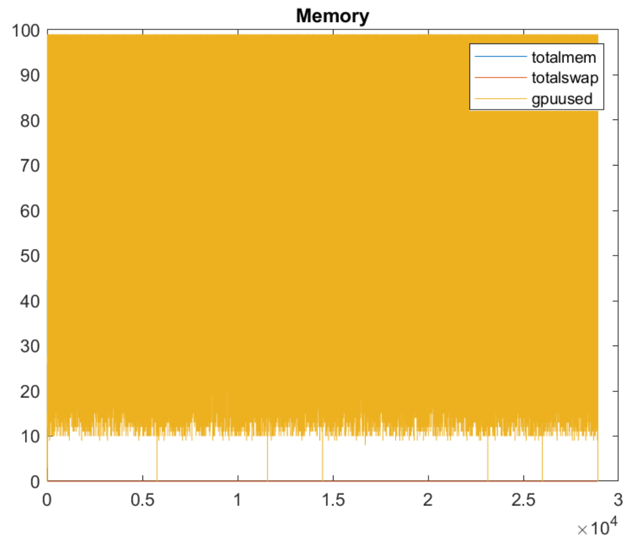
MxNet did better in 5 of the cases against two of PyTorch regarding time. This behavior repeated itself on average time. When it comes to average power, MxNet did better in 1 case against 6 cases of PyTorch. The same 6:1 ratio was repeated regarding loss. MxNet was outperformed in average Accuracy by 2:5 against PyTorch. Finally, PyTorch dominated the top5 results in this test case.

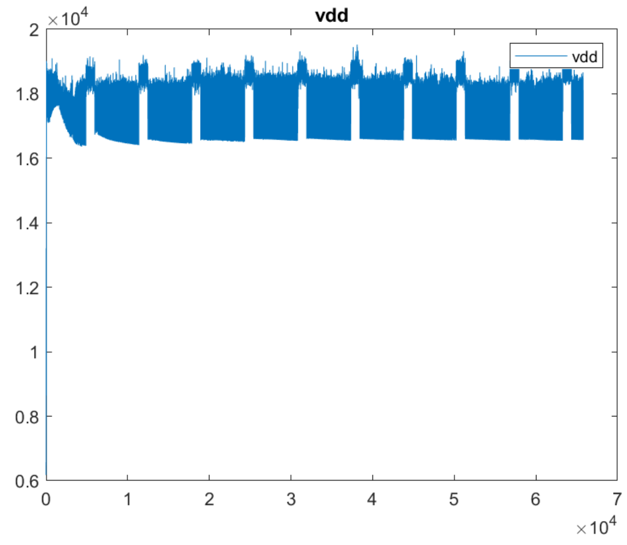
Table 5.8: Hardware Metrics Comparison of CPU (top), Memory (middle) and Energy (bottom) for DenseNet in GPU-CFG1 for mImgNet dataset in MxNet (left) and PyTorch (right), respectively

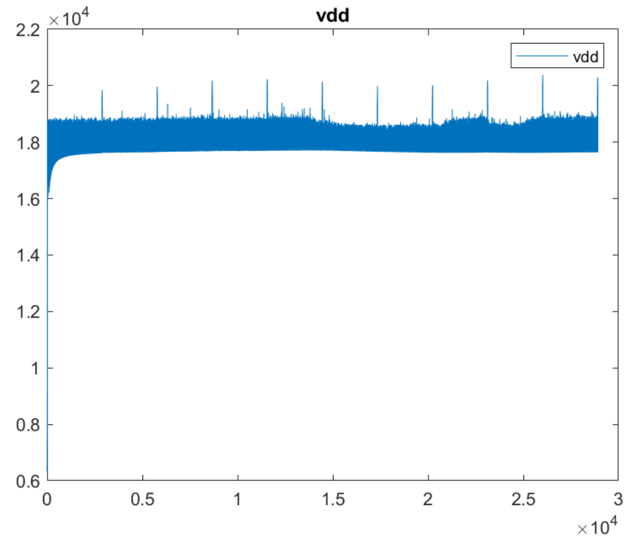












5.4.2 Comparison 2: GPU-CFG2 trained and validated in D3

This second experiment was performed on all available models (see Table 5.2) with the mImgNet10 dataset (D3) over a hardware set that consists of two functional Denver2 cores and all of the GPU's 256 Pascal cores (notice that all Cortex A57 are disabled in this case). The fourteen models concluded the 10-epochs training and validations successfully, as can be seen in Table 5.5.

Table 5.9: CNN Models on mImgNet10 - Results at GPU-CFG2

ID	**Total Time**	**Mean Time/Epoch**	**Power**	**Loss**	**Average Val. Acc**	**Top5**
1M	3873	380.5±1.4	18.35±0.8	1.565±0.15	0.498±0.06	0.832
1P	3661	352.3±32.2	14.22±0.8	1.624±0.20	0.494±0.07	0.906
2M	5639	555.0±1.6	19.09±0.7	1.752±0.22	0.441±0.12	0.719
2P	8716	858.6±3.5	17.56±0.5	1.291±0.34	0.626±0.11	0.941
3M	17848	1753.5±2.0	21.08±0.5	1.998±0.18	0.342±0.10	0.647
3P	34743	3431.3±6.5	18.39±0.3	1.786±0.26	0.453±0.11	0.887
4M	10274	1010.5±3.8	17.61±0.4	2.122±0.19	0.231±0.07	0.578
4P	17750	1748.3±8.9	17.25±0.4	1.492±0.29	0.599±0.12	0.942
5M	33227	3272.2±1.4	19.11±0.3	1.578±0.22	0.506±0.06	0.806
5P	56305	5567.7±94.5	18.93±0.5	1.560±0.33	0.471±0.10	0.885
6M	64922	6382.6±150.9	18.05±0.8	2.041±0.07	0.295±0.04	0.704
6P	34167	3366.5±10.5	18.47±0.6	1.180±0.26	0.689±0.09	0.962
7M	7690	757.4±0.4	18.27±0.4	2.235±0.09	0.183±0.03	0.516
7P	11670	1142.8±29.5	17.32±0.5	1.437±0.28	0.553±0.10	0.937
Units:	**s**	**s**	**W**	**loss***	**%**	**%**

By isolating the two Denver2 cores with the GPU, the results in Table 5.9 show very similar behavior to the results observed in the previous experiment. However, almost every model in both frameworks has increased their total training time. Denver2 cores are designed to enhance the SoC's performance with Footprint (FP) calculations, but the 4 Cortex A57 has surpassed them.

In this scenario, the behavior in time difference stays similar to scenario 1, with minor variations that reduced the gap between the two frameworks by little. The most noteworthy reduction has been related to DenseNet, which presented a 55.1% time difference in scenario 1, but now came closer to 47.37%, i.e., a 7,73% reduction from the previous scenario.

However, what is curious is that with only two working CPU cores, MxNet's time did not change radically from the first scenario's 64648s to this scenario's 64922s execution time. On the other hand, PyTorch's execution time for DenseNet increased from 29028s to 34167s, a 17% increase of execution time.

This might show that PyTorch performs better with multi-CPU scenarios but suffers a drastic performance loss for cases with few available CPUs. So, if this case consistently repeats itself, it might point to a better CPU optimization of PyTorch as opposed to MxNet, which may be more dependent on GPU.

Table 5.10: Ranking and Evaluation for CNN Models on mImgNet10 at GPU-CFG2

FEATURE	MODELS	LE	ALEX	VGG	INC	RES	DNS	SQZ
Total Execution Time	Faster Framework	PT	MX	MX	MX	MX	PT	MX
	Difference(s)	212.0	3077.0	16895.0	7476.0	23078.0	30755.0	3980.0
	Difference(%)	5.47	35.30	48.63	42.12	40.99	47.37	34.10
Avg.Time/ Epoch	Faster Framework	PT	MX	MX	MX	MX	PT	MX
	Difference(s)	28.2	303.6	1677.8	737.8	2295.5	3016.1	385.4
	Difference(%)	7.41	35.36	48.90	42.20	41.23	47.26	33.72
Avg. Power	Most Efficient Framework	PT	PT	PT	PT	PT	MX	PT
	Difference(W)	4.130	1.530	2.690	0.360	0.180	0.420	0.950
	Difference(%)	22.51	8.01	12.76	2.04	0.94	2.27	5.20
Avg. Loss	Best Converging Framework	MX	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.059	0.461	0.212	0.630	0.018	0.861	0.798
	Difference(%)	3.63	26.31	10.61	29.69	1.14	42.19	35.70
Avg. Val. Accuracy	Most Precise Framework	MX	PT	PT	PT	MX	PT	PT
	Difference(abs)	0.004	0.185	0.111	0.368	0.035	0.394	0.370
	Difference(%)	0.81	41.95	32.46	159.31	7.43	133.56	202.19
Avg. Top 5	Best Prediction Framework	PT	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.074	0.222	0.240	0.364	0.079	0.258	0.421
	Difference(%)	8.89	30.88	37.09	62.98	9.80	36.65	81.59

Regarding power consumption, the experiments registered at Table 5.9 show a similar behavior from Scenario 1, with higher average power consumption on MxNet in all cases but one: DenseNet.

As mentioned in the last scenario, this slightly higher power consumption in PyTorch for DenseNet is attributed to the higher GPU work frequency in this particular framework case, where the GPU almost does not stand idle.

Finally, it is worth mentioning that the DenseNet's execution time behavior from Scenarios 1 and 2 still occurs in this scenario. However, the presence of more available CPU cores shows a significant time reduction for MxNet's execution, reducing from 64922s to 54983s (a 15.3% reduction).

Regarding the total execution time and average time per epoch shown in Tables 5.9 and 5.10, MxNet did better than PyTorch in five of the seven scenarios. When it came to average power consumption and loss, PyTorch performed better six times against one better performance of MxNet. MxNet was better in two cases against five of PyTorch regarding average Accuracy. Also, regarding top5, all seven best cases were registered in PyTorch.

These results, united with the rest of the tests, show that MxNet is more optimized towards the GPU than CPU and show that this framework needs more available CPU cores to run these optimizations for deeper workloads.

Opposed to that, PyTorch did not gain any performance by adding the extra CPU cores, mainly because its optimizations focus more on the CPU workloads and do not implement real gains for embedded GPU yet.

5.4.3 Comparison 3: GPU-CFG3 trained and validated in D3

Comparison number 3 also ran training and validation on all seven models (see Table 5.2) with the same mImgNet10 dataset (D3). Also, the hardware setting for this set of experiments contains all CPU-cores, i.e., two functional Denver2 CPU-cores, all four Cortex A57 cores, and all of the GPU's 256 Pascal cores. All the fourteen models concluded training and validations successfully, as registered in Table 5.5.

Table 5.11 shows a noticeable gain on total training time of models 1M and 2M compared to the results obtained in Comparison 5.4.1. LeNet (1M) and AlexNet (2M) registered respectively 42,98% and 24,33% reduced conclusion times on all-cores training opposed to four CortexA57-configuration experiments.

Table 5.11: CNN Models on mImgNet10 - Results at GPU-CFG3

ID	Total Time	Mean Time/Epoch	Power	Loss	Average Val. Acc	Top5
1M	2840	278.9±0.7	23.81±1.1	1.789±0.26	0.491±0.06	0.822
1P	2634	258.4±1.2	15.33±0.7	1.646±0.19	0.470±0.10	0.907
2M	4594	452.2±0.5	24.45±1.1	1.701±0.19	0.437±0.12	0.724
2P	8187	812.7±4.1	17.18±0.5	1.271±0.32	0.627±0.09	0.951
3M	14845	1458.5±3.9	28.59±0.8	1.972±0.18	0.350±0.11	0.647
3P	31737	3147.7±5.5	25.86±0.6	1.606±0.32	0.539±0.12	0.924
4M	8155	802.1±1.6	23.57±0.6	2.124±0.20	0.248±0.06	0.579
4P	15076	1490.6±0.9	22.36±0.9	1.468±0.28	0.603±0.11	0.944
5M	27037	2662.6±1.5	26.96±0.5	1.893±0.31	0.52±0.06	0.806
5P	48122	4777.1±3.6	25.88±0.6	1.573±0.32	0.488±0.09	0.894
6M	54983	5405.5±145.5	23.57±0.8	1.796±0.25	0.294±0.04	0.703
6P	35909	3556.9±0.9	21.85±1.0	1.170±0.26	0.699±0.09	0.965
7M	6340	624.4±0.5	23.17±0.7	2.234±0.09	0.195±0.03	0.517
7P	6249	614.9±5.3	25.46±1.3	1.444±0.30	0.550±0.11	0.927
Units:	**s**	**s**	**W**	**loss***	**%**	**%**

Just as in Comparison 5.4.1, results in the present comparison show normal behavior on total execution times of MxNet's models, which can be detected thanks to the almost constant increasing time between the current and next model by a factor close to two.

Table 5.12: Ranking and Evaluation for CNN Models on mImgNet10 at GPU-CFG3

FEATURE	MODELS	LE	ALEX	VGG	INC	RES	DNS	SQZ
Total Execution Time	**Faster Framework**	PT	MX	MX	MX	MX	PT	PT
	Difference(s)	206.0	3593.0	16892.0	6921.0	21085.0	19074.0	91.0
	Difference(%)	7.25	43.89	53.22	45.91	43.82	34.69	1.44
Avg.Time/ Epoch	**Faster Framework**	PT	MX	MX	MX	MX	PT	PT
	Difference(s)	20.5	360.5	1689.2	688.5	2114.5	1848.6	9.5
	Difference(%)	7.35	44.36	53.66	46.19	44.26	34.20	1.52
Avg. Power	**Most Efficient Framework**	PT	PT	PT	PT	PT	PT	MX
	Difference(W)	8.480	7.270	2.730	1.210	1.080	1.720	2.290
	Difference(%)	35.62	29.73	9.55	5.13	4.01	7.30	8.99
Avg. Loss	**Best Converging Framework**	PT	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.143	0.430	0.366	0.656	0.320	0.626	0.790
	Difference(%)	7.99	25.28	18.56	30.89	16.90	34.86	35.36
Avg. Val. Accuracy	**Most Precise Framework**	MX	PT	PT	PT	MX	PT	PT
	Difference(abs)	0.021	0.190	0.189	0.355	0.032	0.405	0.355
	Difference(%)	4.47	43.48	54.00	143.15	6.56	137.76	182.05
Avg. Top 5	**Best Prediction Framework**	PT	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.085	0.227	0.277	0.365	0.088	0.262	0.410
	Difference(%)	10.34	31.35	42.81	63.04	10.92	37.27	79.30

Also, Tables 5.11 and 5.12 show that most MxNet models kept an average validation accuracy close to 50%, which is a decent middle-ground margin between the application performance on low-processing devices and the assessment of the hardware consumption of this training.

In other words, even though the validation accuracy is not ideal, it symbolizes model executions that were reasonably close to good convergence and most clearly avoided the underfitting. Perhaps Inception, DenseNet, and SqueezeNet may have underfitted in this scenario on the MxNet framework, with 24.8%, 29.4%, and 19.5% average validation accuracy.

It is more probable that these models' performance was affected by a very similar subset of mImgNet10, or perhaps the model lacked a few better training parameters and a few more trials to achieve higher rates. The detection of similar small values for the same three cases as mentioned above in test Scenarios 1 and 2 also implies that these values were no accident.

Perhaps these models needed more training epochs to escape this local validation slope towards more good values. However, since most models presented a stable average rate, these results should be further analyzed based on the hardware metrics.

Curiously, Tables 5.11 and 5.12 prove that 6M and 7M models showed similar smallest validation's average and standard deviation on both Comparisons 5.4.1 and 5.4.3, which could mean that these particular models require more epochs of training or perhaps more data for better specialization. Alternatively, perhaps this behavior could be explained by some discrepancy between the default values for specific transformations or routines from MxNet's gluon API and PyTorch's torchvision API.

To summarize these Comparisons' results, MxNet was faster than PyTorch in Total Execution and Average Time in four out of seven test cases. Regarding the average power, PyTorch nearly beat MxNet in all cases except one. When it came to average loss and average Top5 metrics, PyTorch was unanimously superior to MxNet. Finally, the reported average validation accuracy indicates that PyTorch predicts better (in this scenario) than MxNet in five out of seven test cases.

5.4.4 Comparison 4: CPU-CFG1 trained and validated at D3

Following the same model pattern, but with a different approach, comparison 4 experimented with all seven models (see Table 5.2) with the same mImgNet10 dataset (D3). However, in contrast with comparisons 1, 2, and 3, the comparison's four hardware setting is comprised solely of CPU cores without GPU. In other words, this test enables only the four Cortex A57 cores. The fourteen models finished training and validations correctly (see Table 5.5).

Table 5.13: CNN Models on mImgNet10 - Results at CPU-CFG1

ID	**Total Time**	**Mean Time/Epoch**	**Power**	**Loss**	**Average Val. Acc**	**Top5**
1M	95570	9386.8±36.4	7.14±0.2	1.593±0.16	0.512±0.06	0.828
1P	14075	1379.3±67.0	10.85±0.2	1.669±0.18	0.465±0.07	0.893
2M	245375	24148.9±2.5	9.13±0.4	1.760±0.24	0.444±0.13	0.715
2P	134027	13190.9±116.1	10.33±0.6	1.321±0.33	0.605±0.10	0.943
3M	1806216	177453±122.8	6.84±0.2	1.998±0.15	0.354±0.09	0.654
3P	521840	51288.1±541.6	11.0±0.7	1.778±0.27	0.442±0.11	0.879
4M	430022	42295.5±13.3	6.93±0.1	2.133±0.18	0.230±0.08	0.578
4P	283719	27916.6±498.6	10.3±0.6	1.516±0.28	0.573±0.11	0.941
5M	2834601	279147±327.9	6.95±0.1	1.669±0.20	0.435±0.04	0.727
5P	1612913	158696.3±5706.6	9.64±0.6	1.576±0.33	0.472±0.12	0.891
6M	471315	46335.4±1.3	7.31±0.3	1.583±0.15	0.518±0.05	0.825
6P	733727	72215.9±1509.2	10.3±0.6	1.181±0.26	0.715±0.09	0.963
7M	224034	22066.1±2.8	9.06±0.1	2.229±0.09	0.206±0.03	0.519
7P	165347	16288.1±456.9	9.59±0.4	1.625±0.26	0.489±0.10	0.829
Units:	**s**	**s**	**W**	**loss***	**%**	**%**

Table 5.13 gathers the results for this test scenario. In it, PyTorch unveils its vast superiority against MxNet on all-CPU executions. The executions register a variation between 26.2% and 85.27% time difference in the frameworks. The only model that behaves strangely is DenseNet, which flips the execution time ratio between frameworks. Overall, it seems that MxNet is not designed for proper use of the ARM CPUs, that is so because nearly all models take much more time to finish and the average power consumption in this framework is always smaller than PyTorch (varying from 5.53% to

37.82%).

Interestingly, this scenario shows how the first three not-so-deep models registered the most excellent rates in a time difference of this experimental set. Also, two of these three cases were amongst the most significant average power differences. This behavior could mean either that one framework is vastly more well-equipped for any workloads while the other is only cut for some, or perhaps that this behavior is unilateral -i.e., only one of these two suffers dearly with very low workloads on CPU standalone executions-.

Table 5.14: Ranking and Evaluation for CNN Models on mImgNet10 at CPU-CFG1

FEATURE	MODELS	LE	ALEX	VGG	INC	RES	DNS	SQZ
Total Execution Time	Faster Framework	PT	PT	PT	PT	PT	MX	PT
	Difference(s)	81495.0	111348.0	1284376.0	146303.0	1221688.0	262412.0	58687.0
	Difference(%)	85.27	45.38	71.11	34.02	43.10	35.76	26.20
Avg.Time/ Epoch	Faster Framework	PT	PT	PT	PT	PT	MX	PT
	Difference(s)	8007.5	10958.0	126164.9	14378.9	120450.7	25880.5	5778.0
	Difference(%)	85.31	45.38	71.10	34.00	43.15	35.84	26.18
Avg. Power	Best Framework	MX	MX	MX	MX	MX	MX	MX
	Difference(W)	3.710	1.200	4.160	3.370	2.690	2.990	0.530
	Difference(%)	34.19	11.62	37.82	32.72	27.90	29.03	5.53
Avg. Loss	Best Framework	MX	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.076	0.439	0.220	0.617	0.093	0.402	0.604
	Difference(%)	4.55	24.94	11.01	28.93	5.57	25.39	27.10
Avg. Val. Accuracy	Best Framework	MX	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.047	0.161	0.088	0.343	0.037	0.197	0.283
	Difference(%)	10.11	36.26	24.86	149.13	8.51	38.03	137.38
Avg. Top 5	Best Framework	PT	PT	PT	PT	PT	PT	PT
	Difference(abs)	0.065	0.228	0.225	0.363	0.164	0.138	0.310
	Difference(%)	7.85	31.89	34.40	62.80	22.56	16.73	59.73

Going back to Table 5.13 and comparing each pair of values individually, it becomes evident that MxNet does not seem to perform well for small models at all. For instance, the discrepancies between the two frameworks' values become smaller and more stable starting from the fourth model. However, the time-related differences from the first-to-second, second-to-third, and fourth-to-third test cases do not benefit MxNet's choice for CPU.

For instance, PyTorch has its time almost ten times (9.52X) greater from LeNet to AlexNet, while MxNet's difference between the same two models is only 2.56 times greater. Regarding the difference between AlexNet and VGG, PyTorch registered total times 3.89 times greater than one another, while MxNet moved up to a 7.36 ratio difference. Finally, the difference from VGG to Inception represents a 0,54 difference ratio on PyTorch and 0.23 ratio for MxNet.

The union of these ratios with the percent time difference of these model cases in Table 5.14 show how poorly MxNet performs for CPU standalone cases. For the LeNet-AlexNet comparison, the time gap between frameworks was reduced almost by half, while the time as mentioned earlier ratio was reduced fourfold. When it came to the AlexNet-VGG case, the time difference between frameworks went in a different direction than LeNet and increased by about half (56.69%). But, the previously calculated time ratios between these models increased only twofold. This behavior is curious, especially since VGG has nearly 2.5 times more convolutional layers than AlexNet, just as AlexNet has almost 2.5 times more convolutional layers than LeNet. Perhaps the previously mentioned 56.69% increase may be linked to MxNet's inability to handle any changes in convolutional layers on models deeper than eight-thousand neurons in the Fully-Connected layer (see Table 4.2).

It could be that MxNet was developed to rely more heavily on the GPU to alleviate the stress of the FC layer's intense operations and thus tends to suffer a performance loss for CPU-only executions.

Finally, the VGG-Inception case shows a similar strange behavior, as the time difference between frameworks again shifts towards a reduction of 52.15% as the time ratios are reduced twofold. This behavior reiterates the previous conclusions about MxNet since a considerable increase in convolutional deepness does not cause the same effects as it did in the previous case, mostly thanks also to the significant reduction of neurons in the fully-connected layer. This compensation resumes the stability in the results since the consecutive models differ less in convolutional depth while maintaining the same number of neurons in the FC.

Furthermore, the results in Table 5.14 show that all models except DenseNet favor quicker PyTorch executions, while all cases favor MxNet as shorter and more stable average power throughout the experiments. And overall, regarding total execution time, average time per epoch, average loss, and average Accuracy, PyTorch outperformed MxNet six times out of seven. When it came to power, MxNet performed better in all seven cases.

However, regarding the top5 metric, PyTorch did better in all cases.

5.4.5 Comparison 5: CPU-CFG2 trained and validated at D3

Similar to comparison 4, comparison 5 ran all seven models (see Table 5.2) with the same mImgNet10 dataset (D3) only with two available Denver2 cores and no assistance of the GPU. In this case, only three models from MxNet's framework finished their training and validations correctly (see Table 5.5).

Since even a single epoch within this configuration takes a long time to execute, it is possible that the training procedures were affected by memory/data corruption or perhaps occurred too many faults within the Operating System.

Also, by the time that the current work has been written, it was detected that VGG's reported time was indeed inconsistent with the hardware metrics, which pointed to an early stop of the ten training epochs at the third-to-fourth going. This might have been due to an Operating System deadlock or system freeze thanks to the significant and prolonged computational workload. And since this work is based on the actual output of complete training metrics, it would not be ethical nor fair to simply extrapolate its results by a simple proportion.

Table 5.15: CNN Models on mImgNet10 - Results at CPU-CFG2

ID	Total Time	Mean Time/Epoch	Power	Loss	Average Val. Acc	Top5
1M	126212	12396.5±64.6	8.64±0.2	1.573±0.16	0.503±0.06	0.829
2M	264483	26029.6±67.0	8.86±0.2	1.768±0.25	0.439±0.12	0.715
3M	284429	27943.9±49.5	17.81±0.9	1.978±0.18	0.338±0.10	0.654
Units:	**s**	**s**	**W**	**loss***	**%**	**%**

Therefore, the data in Table 5.15 are only presented in this work in order to serve as a baseline for comparison with the other scenarios and also to aid decisions for future parallel and distributed tests scenarios.

5.4.6 Comparison 6: CPU-CFG3 trained and validated at D3

Finally, comparison 6 performed training and validation of all seven models (see Table 5.2) with the same mImgNet10 dataset (D3) with all four Cortex A57 and two Denver2 cores but no assistance of the GPU. From all the fourteen models, only three finished their training and validations correctly in both frameworks (see Table 5.5). As to the other four models, they were only successful in the PyTorch framework and did not finish for MxNet.

Table 5.16: CNN Models on mImgNet10 - Results at CPU-CFG3

ID	**Total Time**	**Mean Time/Epoch**	**Power**	**Loss**	**Average Val. Acc**	**Top5**
1M	111879	10988.7±45.3	10.07±0.3	1.562±0.15	0.510±0.06	0.834
1P	13711	1343.9±53.5	10.90±0.1	1.692±0.17	0.446±0.06	0.891
2M	257964	25388±418.7	10.01±0.4	1.752±0.26	0.437±0.12	0.714
2P	139222	13700.1±54.0	10.62±0.7	1.303±0.34	0.624±0.10	0.941
3M	2127183	208986±6085.0	9.25±0.3	1.957±0.19	0.357±0.10	0.654
3P	516868	50789.9±191.2	11.58±0.8	1.845±0.23	0.441±0.10	0.879
4P	346662	34150.6±634.6	10.05±0.6	1.466±0.27	0.597±0.11	0.952
5P	1562707	153931.0±4222.5	9.39±0.5	1.545±0.33	0.501±0.11	0.904
6P	831122	81716.3±3563.9	9.74±0.6	1.193±0.26	0.690±0.09	0.961
7P	188384	18533.1±307.7	9.75±0.3	1.421±0.29	0.553±0.12	0.922
Units:	**s**	**s**	**W**	**loss***	**%**	**%**

In this scenario, the comparison Table 5.16 shows that MxNet's execution suffered a time increase in LeNet from 95570s to 111879s, in AlexNet from 245375s to 257964s, and in VGG from 1806216s to 2127183s, thus registering respective increases of 17.06%, 5.13%, and 17.17% between scenario 4 and 6. This once more proves that MxNet has trouble with multi-CPU workloads on ARM64, whereas PyTorch obtained a few better results and some worse results, but all with little time variation. This once more proves that PyTorch is better suited for execution that relies more on CPU than MxNet.

Going further than the previous findings, Table 5.17 detail that PyTorch really had set an experimental trend for the deep models except when it came to Average Power. This behavior will be further discussed further on the power consumption analysis.

Table 5.17: Ranking and Evaluation for CNN Models on mImgNet10 at CPU-CFG3

MODELS	FEATURE																	
	Total Execution Time			Avg. Time/ Epoch			Avg. Power			Avg. Loss			Avg. Val. Accuracy			Avg. Top 5		
	Best	Diff (s)	Diff (%)	Best	Diff (s)	Diff (%)	Best	Diff (s)	Diff (%)	Best	Diff (s)	Diff (%)	Best	Diff (s)	Diff (%)	Best	Diff (s)	Diff (%)
LE	PT	98168	87.74	PT	9644.8	87.77	MX	0.830	7.61	MX	0.13	7.68	MX	0.064	14.35	PT	0.057	6.83
ALEX	PT	118742	46.03	PT	11687.9	46.04	MX	0.610	5.74	PT	0.449	25.63	PT	0.187	42.79	PT	0.227	31.79
VGG	PT	1610315	75.7	PT	58196.1	75.7	MX	2.33	20.12	PT	0.112	5.72	PT	0.084	23.53	PT	0.225	34.40

In summary, for Scenario 6, the experiments show that PyTorch was faster than MxNet three out of three times. Also, it outperformed MxNet by the same ratio regarding Average Top5. When it came to average power, MxNet dominated PyTorch in all test cases. However, regarding loss and average validation accuracy, the score was one to MxNet against two to PyTorch.

5.5 Summary of Comparison Best Results

This section focuses on comparing all the prior scenarios that contain results in both frameworks. The table below (Table 5.18) describes, for each model, on what framework it obtained most desirable (smaller in this case) total time and average power, followed by the percentage difference of the best framework. For instance, the model LE is presented with Total Time equals to PT 5,95% and Avg. Power equals PT 11,8%. This means that, overall, the model LeNet has outperformed PyTorch in both time and power by the two previous smaller rates, respectively.

Upon an aggregate evaluation, PyTorch outperformed MxNet by 16 to 15 regarding total execution times, and it also outperformed MxNet in 18 out of 13 cases regarding average energy consumption, which puts PyTorch slightly in favor of any framework choice in the long run. Regarding CPU-only executions, MxNet did better one time as opposed to nine better execution times from PyTorch. However, MxNet demonstrated to have a better average power consumption on all the 10 CPU-only comparisons. So, this proves that PyTorch is faster for executions that rely more on CPU than MxNet, but overall, MxNet is more energy-friendly.

Finally, on scenarios with GPU, MxNet outperformed PyTorch by 14 to 7 regarding total execution time, but this came with a cost, where PyTorch has been more power-efficient in 18 cases as opposed to 3 of MxNet.

To summarize, if a deep model-related process on ARM64 relies more on multi-core CPU execution, PyTorch presents the faster but more power-consuming options. In contrast, the scenarios with ARM64 embedded-GPU workloads count with the quicker but more power costly MxNet framework.

Table 5.18: Compilation of Best Total Time and Average Power Results

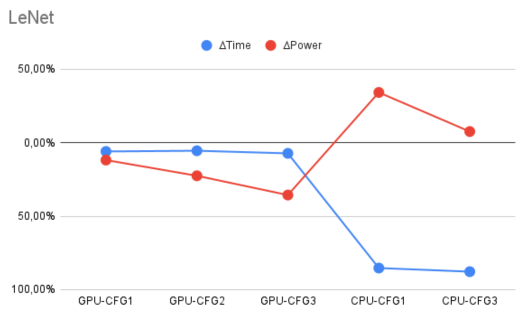
MODEL	METRIC	GPU-CFG1	GPU-CFG2	GPU-CFG3	CPU-CFG1	CPU-CFG2	CPU-CFG3
LE	Total Time	PT 5,95%	PT 5,47%	PT 7,25%	PT 85,27%	N/A	PT 87,74%
	Avg. Power	PT 11,8%	PT 22,5%	PT 35,6%	MX 34,2%	N/A	MX 7,6%
ALX	Total Time	MX 45.79%	MX 35,3%	MX 43,89%	PT 45.38%	N/A	PT 46.03%
	Avg. Power	PT 29.86%	PT 8,01%	PT 29,73%	MX 11.62%	N/A	MX 5.74%
VGG	Total Time	MX 57.91%	MX 48,63%	MX 53,22%	PT 71.11%	N/A	PT 75.70%
	Avg. Power	PT 39.53%	PT 12,76%	PT 9,55%	MX 37.82%	N/A	MX 20.12%
INC	Total Time	MX 43.08%	MX 42,12%	MX 45.91%	PT 34.02%	N/A	N/A
	Avg. Power	PT 0.39%	PT 2,04%	PT 5,13%	MX 32.72%	N/A	N/A
RES	Total Time	MX 41.36%	MX 40,99%	MX 43,82%	PT 43.10%	N/A	N/A
	Avg. Power	PT 10.8%	PT 0,94%	PT 4,01%	MX 27.90%	N/A	N/A
DNS	Total Time	PT 55,1%	PT 47,37%	PT 34,69%	MX 35.76%	N/A	N/A
	Avg. Power	MX 2,79%	MX 2,27%	PT 7,3%	MX 29.03%	N/A	N/A
SQZ	Total Time	MX 36,16%	MX 34,1%	PT 1,44%	PT 26.2%	N/A	N/A
	Avg. Power	PT 8,57%	PT 5,2%	MX 8,99%	MX 5.53%	N/A	N/A

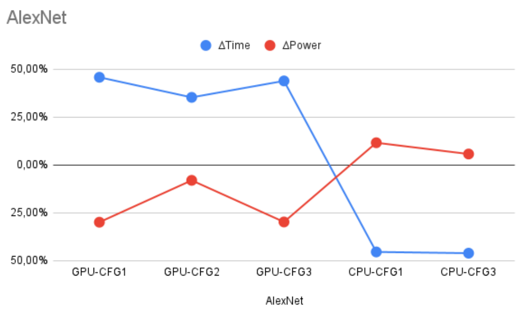
Overall, the images located at 5.19 show a discrete representation of the results from Table 5.18, which are not sorted by any particular order other than the order established on this works results sections. In this picture, the commutation of co-adjacent points does not factor into a polynomial behavior, nor does it improve/weaken any results. The only inference in this graph is this: values above the x-axis mean that the result is better for MxNet, and below is better for PyTorch. Blue points are the time-related results, and red points regard the power results.

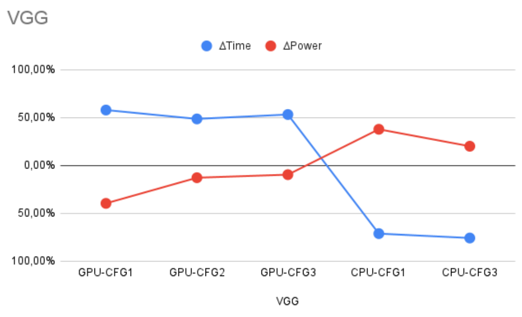
These graphs point out a similar tendency for time (blue) and energy (red) for all cases except LeNet and DenseNet. For instance, AlexNet, VGG, Inception, ResNet, and SqueezeNet obtain a nearly 50% better result in MxNet according to the time metric. However, the power metric does not follow the same close pattern for all these cases.

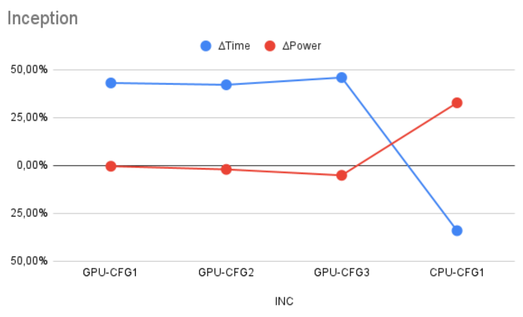
By analyzing the Figures located in Table 5.19 regarding time, it is possible to notice that LeNet, AlexNet, VGG, Inception, and ResNet presented very similar behav-

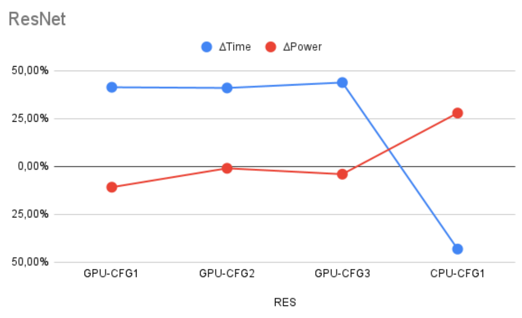
Table 5.19: Best Results Comparison for all Configurations by Model: Above Horizontal Line Means Favorable to MxNet and Below Means Better for PyTorch

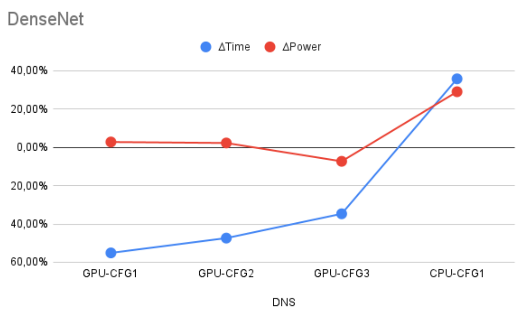


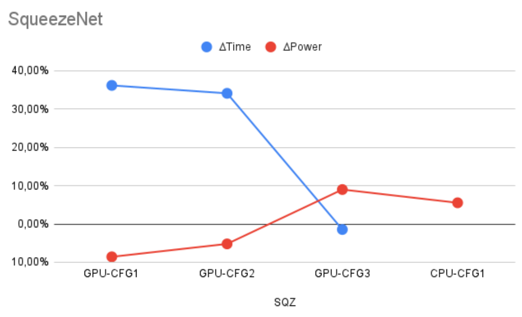












iors. The time difference in these models for GPU experiments has kept approximately constant and favorable towards MxNet, with a noticeable skew of this behavior towards PyTorch in the CPU configurations.

ResNet registered the most constant time difference percentages throughout the entire GPU configurations but not for CPU, which may be explained due to the fact that both frameworks contain several GPU optimizations in their API developed mainly for x86 architecture, but also sustained towards ARMv8. On the CPU side, this constant ResNet behavior can be explained by the findings discussed in 5.4.4, which pointed out that MxNet tends to decay in performance for higher values of convolutional layers and neurons in the FC layer. Thus, even though deeper models starting from Inception (4M and 4P) do not suffer as much as LeNet, AlexNet, and VGG on MxNet, their performances still suffer considerably compared to PyTorch in CPU.

The only peculiar behavior within the same GPU execution was registered by SqueezeNet, which logged the best execution on MxNet for the first and second configurations, but then the third result is better on PyTorch. First thoughts could credit this approximation between frameworks results thanks to the addition of two or four extra CPU cores to this configuration. However, not only did the remainder of the experiments not suffer such effects, but they also accentuated the stats even more towards PyTorch. A second inspection identifies that it was not MxNet that improved or declined along with the configurations, but PyTorch that improved its performance greatly during this particular transition by reducing over 40% of time.

By analyzing the hardware metric graphs, it becomes evident that whatever reason induced this behavior, it also caused a considerable and steady increase in average power consumption. The graphs of SqueezeNet's experiments on GPU-CFG1 and GPU-CFG2 show the power oscillating near the 18-16kW band. On the other hand, the graphs of the same model on GPU-CFG3 point out to power between the 23-25kW band.

The Memory and GPU utilization graphs of the two implemented models in these three GPU configurations do not inform a significant difference between their executions. On the other hand, the CPU utilization graphs show that even though the four Cortex A57 of GPU-CFG1 worked heterogeneously, they still were over-tasked and thus did not deliver a good CPU-to-GPU task scheduling. Also, on GPU-CFG2, the pattern of the CPU graphs tells that one of the two Denver2 cores was very stressed handling most of the computations, while the second core under-performed regularly below 10%.

This strange behavior that only prioritizes the already scarce available CPU resources does not repeat on GPU-CFG3. The graphs of the CPU utilization, in this case, show how c1, c4, c5, and c6 (which stands for the four Cortex A57) differ from c2 and c3 (Denver2 cores). This graph shows that thanks to the availability of four cores of a similar type, the two Denver2 were able to successfully co-work in higher and more frequent bands.

For instance, the two Denver2 cores in the previously mentioned figure show a vertical dislocation of workload moments after the beginning until a little bit after the second 2000 of the experiment. This could indicate that the algorithm of PyTorch's optimizers attempted to redistribute the workload between the cores until reaching an optimal utilization rate between them. Perhaps this behavior did not repeat on GPU-CFG1 because it only affects Denver2 cores, while the GPU-CFG2 could not reach such optimization thanks to factors such as the handling of OS routines and interruptions, which might have prevented the algorithm from finding the optimal distribution between the two cores.

When it comes to the total execution time metric as pairs of results, the desired value for an experiment should be lower than its peer at all costs. oThe average power per experiment also behaves according to this logic -even more so because the experiments did not detect any abnormal standard deviations throughout the experiments-. However, results differ significantly in execution time within the same configuration and between different ones. So, this gap may add to considerable power consumption for the experiments.

Table 5.20: Total Power Consumption Results

MODEL	**CONFIGURATION**						**VAL**
NAME	**GPU-CFG1**	**GPU-CFG2**	**GPU-CFG3**	**CPU-CFG1**	**CPU-CFG2**	**CPU-CFG3**	**SUM**
LE-MX	15.43	19.74	18.78	189.55	302.91	312.95	859.36
LE-PT	12.80	14.46	11.22	42.42	-	41.51	122.41
ALX-MX	30.13	29.90	31.20	622.30	1308.46	717.28	2739.27
ALX-PT	38.98	42.51	39.07	384.58	-	410.70	915.84
VGG-MX	116.77	104.51	117.89	3431.81	1407.13	5465.68	10643.79
VGG-PT	167.77	177.48	227.98	1594.51	-	1662.59	3830.33
INC-MX	42.81	50.26	53.39	827.79	-	-	974.25
INC-MX	74.92	85.05	93.64	811.75	-	967.76	2033.12
RES-MX	171.42	176.38	202.48	5472.35	-	-	6022.63
RES-PT	260.79	296.07	345.94	4319.02	-	4076.06	9297.88
DNS-MX	312.82	325.51	359.99	957.03	-	-	1955.35
DNS-PT	144.49	175.30	217.95	2099.27	-	2248.65	4885.66
SQZ-MX	32.84	39.03	40.80	619.83	-	-	732.50
SQZ-PT	47.04	56.15	44.19	440.47	-	510.21	1098.06
VAL SUM	1469.01	1592.35	1804.52	21812.68	3018.50	16413.39	46110.45
Units	wH	wH	wH	wH	wH	wH	wH

Thus, every able case had its power consumption calculated by the product between the average power with the total execution time, and the result is in *wH*, which is shown in Table 5.20.

The results obtained for every model in Table 5.20 and Figure 5.21 show that every model, with two exceptions, had similar behavior in the comparison between the frameworks. In other words, every graph of each individual model with its performance under different configurations approximates a very similar set of points or even a polynomial function. Perhaps some of these results may differ slightly between max and min values, but under a qualitative comparison, these functions only seem a little more transposed regarding the y-axis than the other. These findings help to ensure that, overall, this works' experiments were stable and also set trends that could be analyzed deeper in future works.

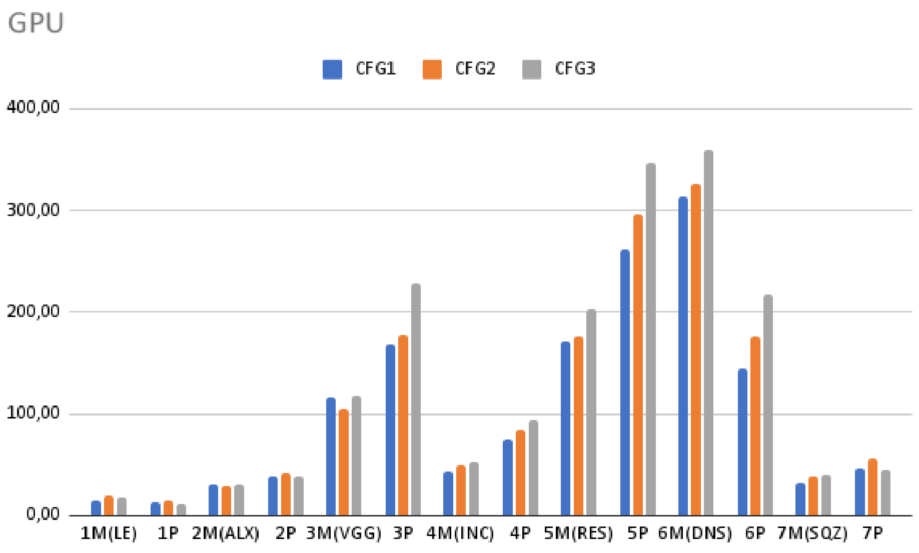
Overall, the models LeNet, AlexNet, and SqueezeNet implemented in MxNet and PyTorch demonstrated the smallest values regarding energy consumption. Regarding the models ResNet on PyTorch and DenseNet on MxNet, those models introduced the highest values, being the last one also the highest consumption case of the third configuration, with a power consumption nearly thirty-three times higher than the lowest value (LeNet in Pytorch). The previously mentioned values may be high, but they are also expected since they either introduce higher convolutional depths or a high number of neurons in the FC layer.

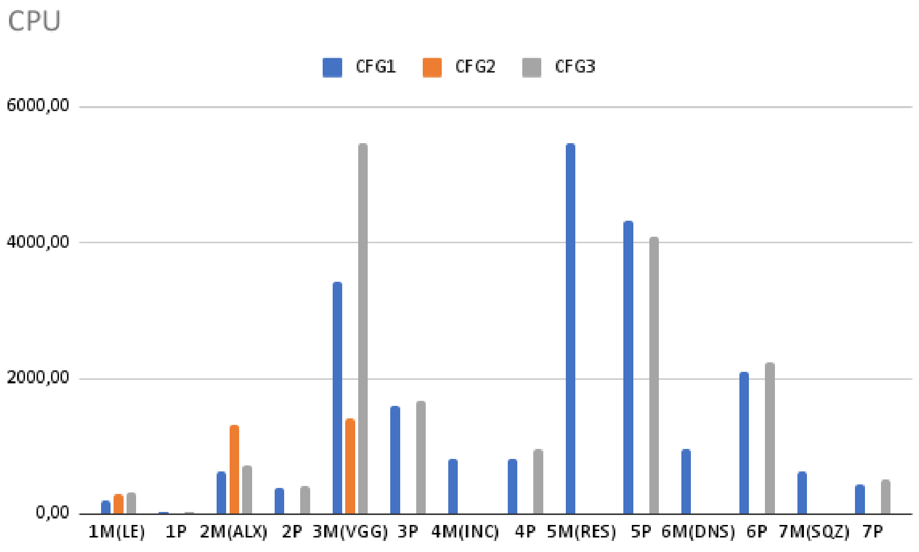
On the other hand, when it comes down to CPU configurations, the values are different. For instance, in the CPU configuration 1, ResNet in MxNet and PyTorch and VGG on MxNet obtained the highest values of power consumption, while LeNet, AlexNet, Inception, and SqueezeNet obtained the lowest consumption values.

Besides that, CPU configuration 3 followed a similar pattern to the previously mentioned configuration 1, and configuration two could not be mentioned due to its insufficient amount of comparable cases.

Inception figures show that both frameworks have almost the same average power for all GPU-related experiments. ResNet on configurations for GPU-CFG2 and GPU-CFG3 also show almost equal power consumption averages.

Table 5.21: Summary of Power Consumption Table





6 CLOSING REMARKS AND FUTURE WORKS

This chapter presents this research's conclusions and the future works that may be conducted to continue expanding this work's experiments.

6.1 Closing Remarks

This work introduced an experimental evaluation of Deep Learning convolutional models and frameworks on GPU-embedded arm boards to attest to the feasibility of performing in-depth learning training on these boards and provide a fair comparison between DL frameworks on ARM64 boards.

Initially, this work aimed to conduct a performance evaluation study on current embedded GPU devices' most famous CNN models and datasets. Thus this was achieved in an NVIDIA Jetson TX2 cluster with four units using PyTorch and MxNet frameworks, each running of 7 CNN models and one medium-sized dataset into three configurations of CPU and two for GPU, thus totaling six possible hardware configurations.

Experiments measured the impact of intense computations such as CNN models training and validation when processed with PyTorch framework regarding hardware metrics such as RAM, CPU, and GPU utilization, but also investigating software metrics such as total execution time, average execution time per epoch, average power consumption, accuracy, loss, and top5 accuracy.

The experiments showed that most model training performed with the MxNet framework tends to be faster than PyTorch for scenarios with embedded-GPU support. However, the experiments also showed that this comes with a power consumption trade-off, whereas PyTorch consumes less average energy than MxNet.

Also, further experiments showed that PyTorch outperforms MxNet in CPU-only scenarios, thus presenting a significantly shorter execution time. However, just as in GPU-only executions, this comes with a price that makes PyTorch less energy-friendly in CPU-only scenarios than MxNet.

Overall, both in CPU-only and GPU-aided cases, PyTorch demonstrates a slight advantage both in total execution time and average power consumption, beating MxNet by 16 to 15 and 18 to 13, respectively.

Finally, it is safe to say that MxNet presents itself as a better choice for GPU workloads, and PyTorch presents itself as a more viable option for CPU-only scenarios, i.e., workloads on less powerful ARM64 boards may gain more performance on top of PyTorch.

6.2 Future Works

Regarding some paths for future works, it is worth mentioning:

- The correction in some parameters of a few study cases revealed poor convergence at most scenarios, especially models SqueezeNet and DenseNet on the MxNet framework.
- The introduction of at least one RNN or LSTM unit model in the scope of the future experiments.
- Addition of more datasets and models of current experiment's scope.
- Install and configure the frameworks with distributed processing support for arm, and also develop and test the deep learning models for distributed multi-CPU and multi-GPU training.

6.3 Tips of Choice

Since some readers might have doubts regarding which hardware to invest in, this section brings a few tips that might help to guide their decision. These tips are based on a few key factors, such as i) the field of study that the boards will be used towards; ii) the hardware consumption of the application; iii) the need for an embedded GPU in the application; iv)

For instance, if the subject of study is Deep Learning, the user must verify how greater or smaller the network is in this work. If the networks are more profound, the user may benefit more by purchasing the boards higher than the NVIDIA Jetson TX2, like the NVIDIA Jetson Xavier or NVIDIA Jetson AGX Orin.

Also, users must regard if the application can be executed by embedded GPU or not. If not, perhaps an NVIDIA Jetson TX2, NVIDIA Jetson Nano, a Cubbietruck, or a Raspberry Pi 3 could suffice.

6.4 Acknowledgments

This work acknowledges UFRGS - Federal University of Rio Grande do Sul, Brazil- and CNPq - Brazilian National Council for Scientific Development for funding this research. This work's authors gratefully acknowledge the support of NVIDIA Corporation with the donation of two NVIDIA Jetson TX2 used for this research.

REFERENCES

ABADI, M. et al. Tensorflow: a system for large-scale machine learning. In: **OSDI**. [S.l.: s.n.], 2016. v. 16, p. 265–283.

ATTARAN, N. et al. Embedded low-power processor for personalized stress detection. **IEEE Transactions on Circuits and Systems II: Express Briefs**, IEEE, 2018.

BAHRAMPOUR, S. et al. Comparative study of deep learning software frameworks. **arXiv preprint arXiv:1511.06435**, 2015.

BASTIEN, F. et al. Theano: new features and speed improvements. **arXiv preprint arXiv:1211.5590**, 2012.

CANZIANI, A.; PASZKE, A.; CULURCIELLO, E. An analysis of deep neural network models for practical applications. **arXiv preprint arXiv:1605.07678**, 2016.

CHEN, T. et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. **arXiv preprint arXiv:1512.01274**, 2015.

CIREŞAN, D. et al. Multi-column deep neural network for traffic sign classification. **Neural networks**, Elsevier, v. 32, p. 333–338, 2012.

COLLOBERT, R.; BENGIO, S.; MARIÉTHOZ, J. **Torch: a modular machine learning software library**. [S.l.], 2002.

ESHRATIFAR, A. E.; PEDRAM, M. Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment. In: ACM. **Proceedings of the 2018 on Great Lakes Symposium on VLSI**. [S.l.], 2018. p. 111–116.

FRANKLIN, D. Nvidia jetson tx2 delivers twice the intelligence to the edge. **NVIDIA Accelerated Computing| Parallel Forall**, 2017.

GAYA, J. O. et al. Vision-based obstacle avoidance using deep learning. In: **2016 XIII Latin American Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR)**. [S.l.: s.n.], 2016. p. 7–12.

GIBSON, A. et al. Deeplearning4j: Distributed, open-source deep learning for java and scala on hadoop and spark. May, 2016.

GIL, A. C. **Métodos e técnicas de pesquisa social**. [S.l.]: 6. ed. Ediitora Atlas SA, 2008.

GOODFELLOW, I. et al. **Deep learning**. [S.l.]: MIT press Cambridge, 2016.

HAN, S. et al. Eie: efficient inference engine on compressed deep neural network. In: IEEE. **Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on**. [S.l.], 2016. p. 243–254.

HAN, S. et al. Learning both weights and connections for efficient neural networks. **arXiv preprint arXiv:1506.02626**, 2015.

HE, K. et al. Identity mappings in deep residual networks. In: SPRINGER. **European conference on computer vision**. [S.l.], 2016. p. 630–645.

HODGKIN, A. L.; HUXLEY, A. F. Action potentials recorded from inside a nerve fibre. **Nature**, Nature Publishing Group, v. 144, n. 3651, p. 710, 1939.

HUANG, G. et al. Densely connected convolutional networks. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2017.

HUYNH, L. N.; LEE, Y.; BALAN, R. K. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In: ACM. **Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services**. [S.l.], 2017. p. 82–95.

IANDOLA, F. N. et al. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. **arXiv preprint arXiv:1602.07360**, 2016.

ICHINOSE, A. et al. Pipeline-based processing of the deep learning framework caffe. In: ACM. **Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication**. [S.l.], 2017. p. 97.

JAIN, R. **The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling**. [S.l.]: John Wiley & Sons, 1990.

JIA, Y. et al. Caffe: Convolutional architecture for fast feature embedding. In: ACM. **Proceedings of the 22nd ACM international conference on Multimedia**. [S.l.], 2014. p. 675–678.

KRIZHEVSKY, A.; NAIR, V.; HINTON, G. The cifar-10 dataset. **online: http://www. cs. toronto. edu/kriz/cifar. html**, 2014.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2012. p. 1097–1105.

LECUN, Y. et al. Backpropagation applied to handwritten zip code recognition. **Neural computation**, MIT Press, v. 1, n. 4, p. 541–551, 1989.

LECUN, Y.; CORTES, C.; BURGES, C. Mnist handwritten digit database. **AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist**, v. 2, 2010.

LENAT, D. B.; GUHA, R. V. Building large knowledge-based systems; representation and inference in the cyc project. Addison-Wesley Longman Publishing Co., Inc., 1989.

LIN, S. et al. Fft-based deep learning deployment in embedded systems. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018**. [S.l.], 2018. p. 1045–1050.

LIU, J. et al. Performance analysis and characterization of training deep learning models on mobile device. In: IEEE. **2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)**. [S.l.], 2019. p. 506–515.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, Springer, v. 5, n. 4, p. 115–133, 1943.

PATTANAYAK, S.; PATTANAYAK; JOHN, S. **Pro Deep Learning with TensorFlow**. [S.l.]: Springer, 2017.

QI, H.; SPARKS, E. R.; TALWALKAR, A. Paleo: A performance model for deep neural networks. 2016.

RAINA, R.; MADHAVAN, A.; NG, A. Y. Large-scale deep unsupervised learning using graphics processors. In: ACM. **Proceedings of the 26th annual international conference on machine learning**. [S.l.], 2009. p. 873–880.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, American Psychological Association, v. 65, n. 6, p. 386, 1958.

RUNGSUPTAWEEKOON, K.; VISOOTTIVISETH, V.; TAKANO, R. Evaluating the power efficiency of deep learning inference on embedded gpu systems. In: IEEE. **Information Technology (INCIT), 2017 2nd International Conference on**. [S.l.], 2017. p. 1–5.

RUSSAKOVSKY, O. et al. ImageNet Large Scale Visual Recognition Challenge. **International Journal of Computer Vision (IJCV)**, v. 115, n. 3, p. 211–252, 2015.

SHAMS, S. et al. Evaluation of deep learning frameworks over different hpc architectures. In: IEEE. **Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on**. [S.l.], 2017. p. 1389–1396.

SHI, S.; CHU, X. Performance modeling and evaluation of distributed deep learning frameworks on gpus. **arXiv preprint arXiv:1711.05979**, 2017.

SHI, S. et al. Benchmarking state-of-the-art deep learning software tools. In: IEEE. **Cloud Computing and Big Data (CCBD), 2016 7th International Conference on**. [S.l.], 2016. p. 99–104.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. **arXiv preprint arXiv:1409.1556**, 2014.

SZEGEDY, C. et al. Rethinking the inception architecture for computer vision. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2016. p. 2818–2826.

TAYLOR, B. et al. Adaptive selection of deep learning models on embedded systems. **arXiv preprint arXiv:1805.04252**, 2018.

WANG, W. et al. Database meets deep learning: Challenges and opportunities. **ACM SIGMOD Record**, ACM, v. 45, n. 2, p. 17–22, 2016.

WEBSITE, S. **Internet usage worldwide - Statistics Facts**. Available from Internet: <https://www.statista.com/topics/1145/internet-usage-worldwide/>.

WIDROW, B.; HOFF, M. E. **Adaptive switching circuits**. [S.l.], 1960.

WILSON, D. R.; MARTINEZ, T. R. The general inefficiency of batch training for gradient descent learning. **Neural Networks**, Elsevier, v. 16, n. 10, p. 1429–1451, 2003.

ZHANG, X.; WANG, Y.; SHI, W. pcamp: Performance comparison of machine learning packages on the edges. 2018.

ZHU, M. et al. Cnnlab: a novel parallel framework for neural networks using gpu and fpga-a practical study with trade-off analysis. **arXiv preprint arXiv:1606.06234**, 2016.

ZHU, Y. et al. Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. **arXiv preprint arXiv:1803.11232**, 2018.

7 ANNEX 1: RESUMO EM PORTUGUÊS

A explosão e a popularização de componentes de Computação de borda (Edge) têm moldado este mercado graças à competição árdua desses componentes e de suas cada vez melhores funcionalidades a um valor de eficiência energética cada vez mais baixo.

A arquitetura ARM tem sido unanimamente presente no grande segmento de mercado de smartphones, e ainda assim, possui grande presença em tecnologias como drones, sistemas de monitoramento, carros e robótica.

Além disso, os componentes ARM64 de borda vêm sendo utilizados com sucesso para o desenvolvimento de soluções para cadeias de produtividade e logística no setor alimentício e energético. Grande parte dessa presença se dá por conta de sua arquitetura baseada no modelo reduzido no conjunto de instruções RISC, o que comparado ao computador tradicional de arquitetura x86, possui poder computacional reduzido em consumo energético.

Além disso, muitos avanços atuais na arquitetura ARM possibilitaram ganhos maiores de performance graças à introdução de GPUs embarcadas com acesso direto à memória (DMA) e memórias RAM de baixo consumo energético (LPDDR).

Desde o desenvolvimento de placas com GPU-embarcadas como NVIDIA TK1, NVIDIA Jetson TX1 e NVIDIA TX2, finalmente se tornou possível aplicar cargas de trabalho mais onerosas e algoritmos paralelos e distribuídos nesta arquitetura antes limitada.

Por outro lado, a novidade dessas placas revela a baixa quantidade de trabalhos investigativos acerca de suas capacidades e limitações. Além disso, a introdução de cargas de trabalho mais poderosas nestas placas impõem uma dúvida sobre o ganho real entre o poder de processamento e o consumo energético que tais GPUs embarcadas acarretam sobre as mesmas, e também se há limitações para tais usos.

Por essas razões, este trabalho explora o processamento paralelo de Deep Learning em GPUs embarcadas da NVIDIA Jetson TX1 e TX2 para realizar uma avaliação do poder e das limitações dessas placas sobre estas cargas específicas.

Por isso, este trabalho realizou a instalação e preparação de ambiente de testes de dois frameworks PyTorch e MxNet em cinco placas ARM, com o desenvolvimento igual de sete modelos de Redes Neurais Convolucionais do estado-da-arte. Estes sete modelos foram combinados com um dataset adaptado do ImageNet, com tamanho médio, e permutados em seis configurações de hardware. Estas configurações são basicamente a presença ou ausência de GPU nos treinamentos permutada com: i) configuração 1 que contém os quatro cores Cortex A57 ligados e os dois cores Denver2 desligados; ii) configuração 2 que contém os quatro cores Cortex A57 desligados e os dois cores Denver2 ligados; iii) configuração 3 que contém tanto os quatro cores Cortex A57 ligados quanto os dois cores Denver2 ligados.

Os experimentos de treinamento e de validação destas cargas de trabalho foram executadas por um total de 4804 horas, com treinamento bem sucedido para todas as configurações, com exceção de alguns casos do PyTorch e todos casos do MxNet nas configurações 2 e 3 para treinamento apenas com CPU.

Estes testes revelaram um leve favoritismo para MxNet em questão de tempo para cargas de trabalho mais dependentes da GPU-embarcada. Por outro lado, os experimentos também revelaram um favoritismo em geral para o framework PyTorch tanto em tempo de execução quanto em consumo energético, especialmente em casos de execução baseados apenas em CPU sem o auxílio de GPU.

Para trabalhos futuros, planeja-se a continuidade dos experimentos, expandindo os mesmos para execução de forma completamente distribuída, a fim de validar os dados encontrados nestes experimentos locais.

8 ANNEX 2: EXPERIMENTAL HARDWARE PLOTS

Figure 8.1: CPU, MEM and VDD uses for LeNet and AlexNet on MxNet at GPU-CFG1

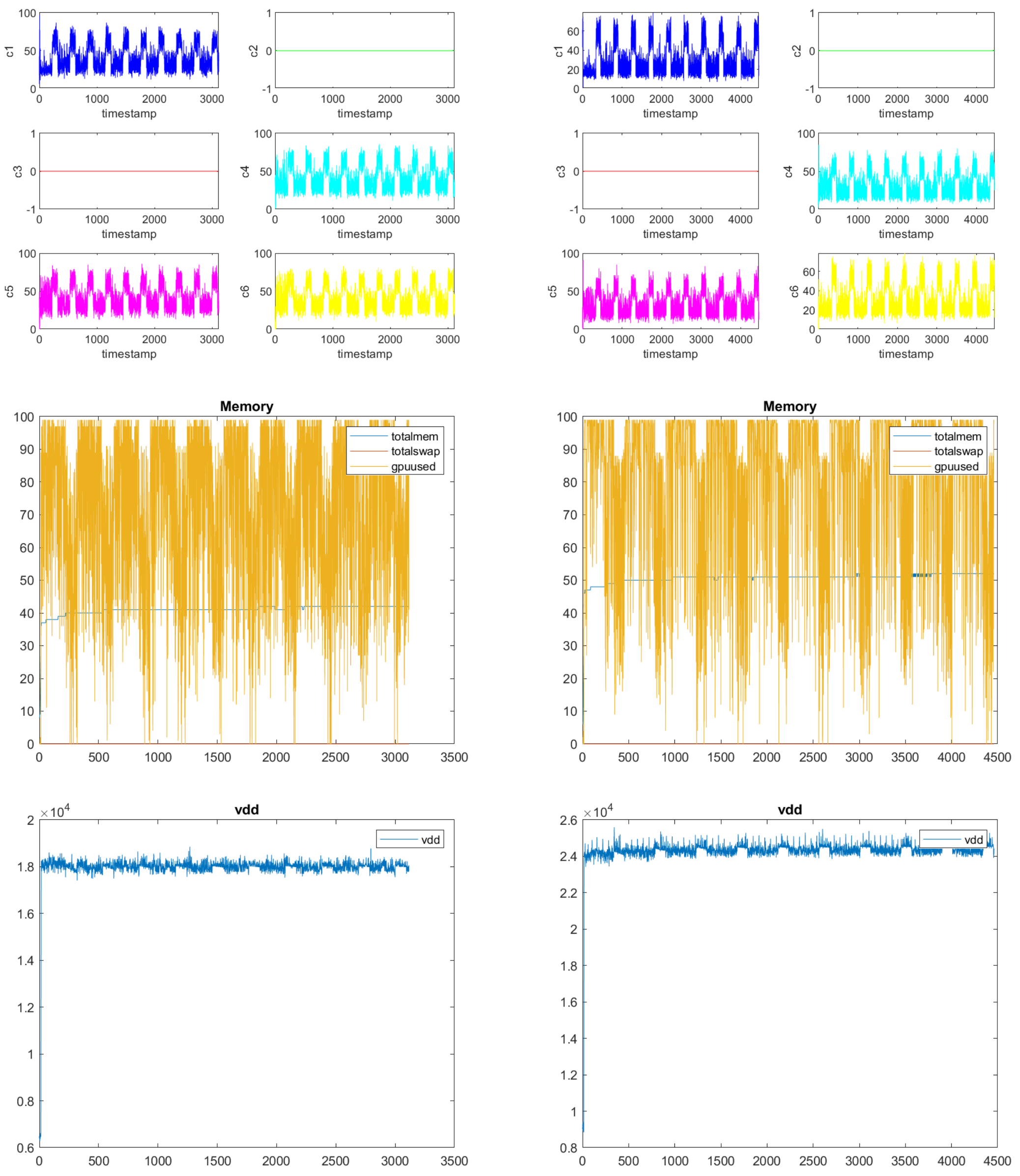


Figure 8.2: CPU, MEM and VDD uses for VGG and Inception on MxNet at GPU-CFG1

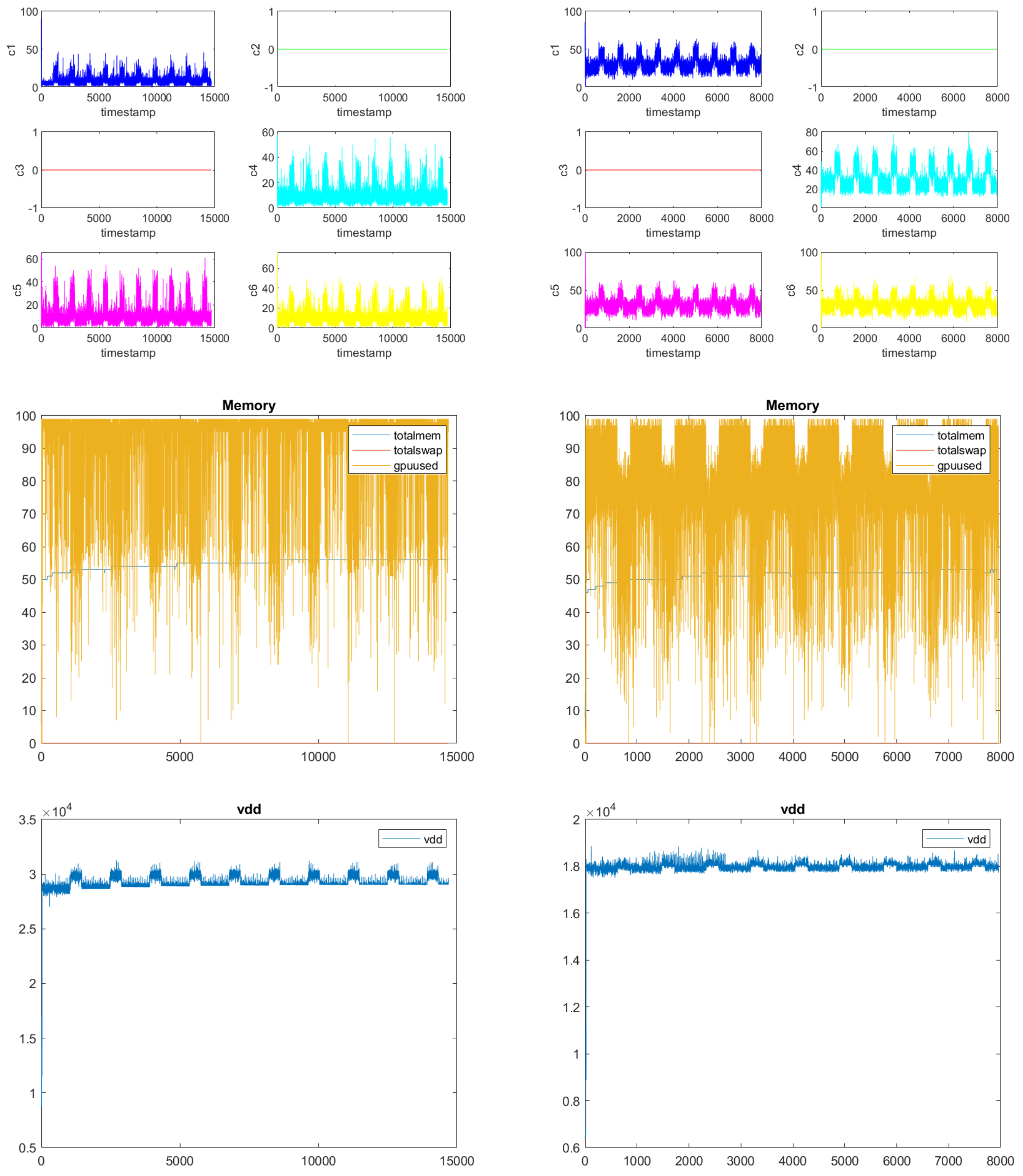


Figure 8.3: CPU, MEM and VDD uses for ResNet and DenseNet on MxNet at GPU-CFG1

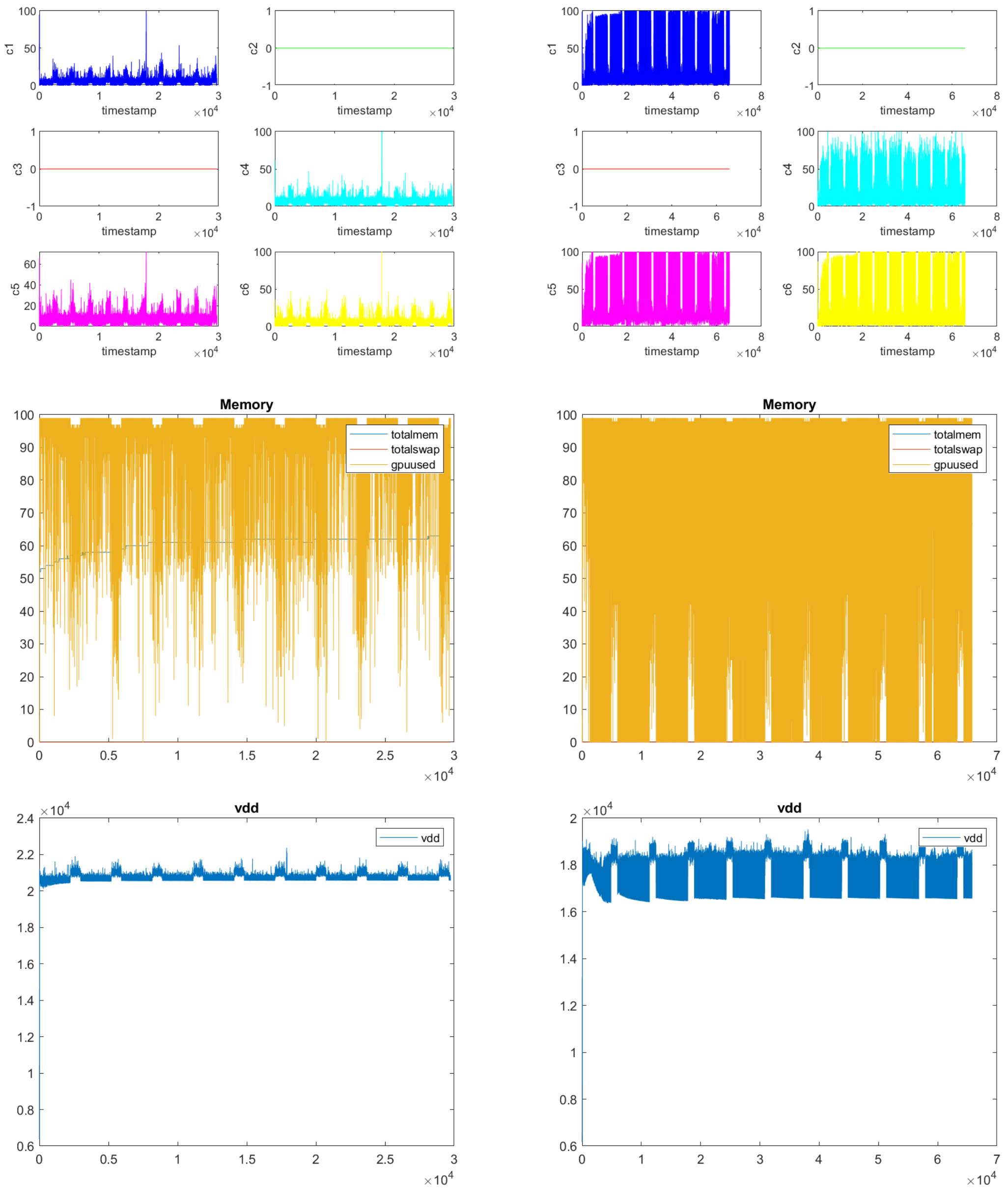
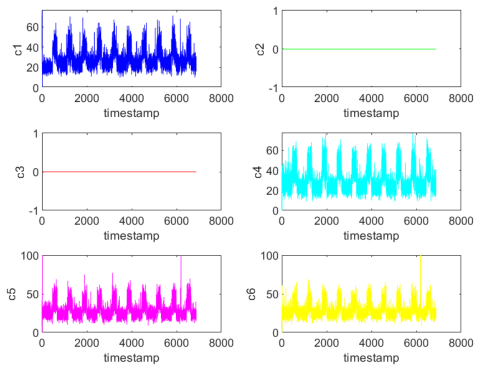
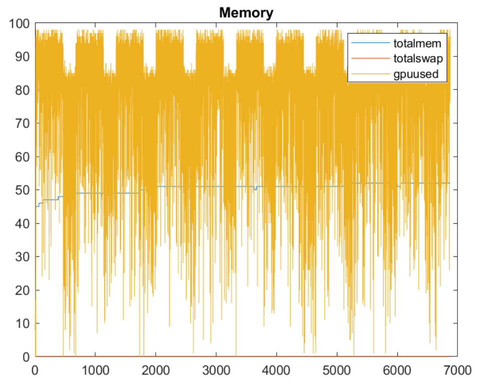


Figure 8.4: CPU, MEM and VDD uses for SqueezeNet on MxNet at GPU-CFG1





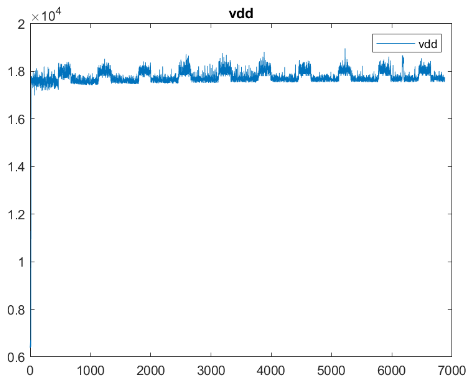


Figure 8.5: CPU, MEM and VDD uses for LeNet and AlexNet on PyTorch at GPU-CFG1

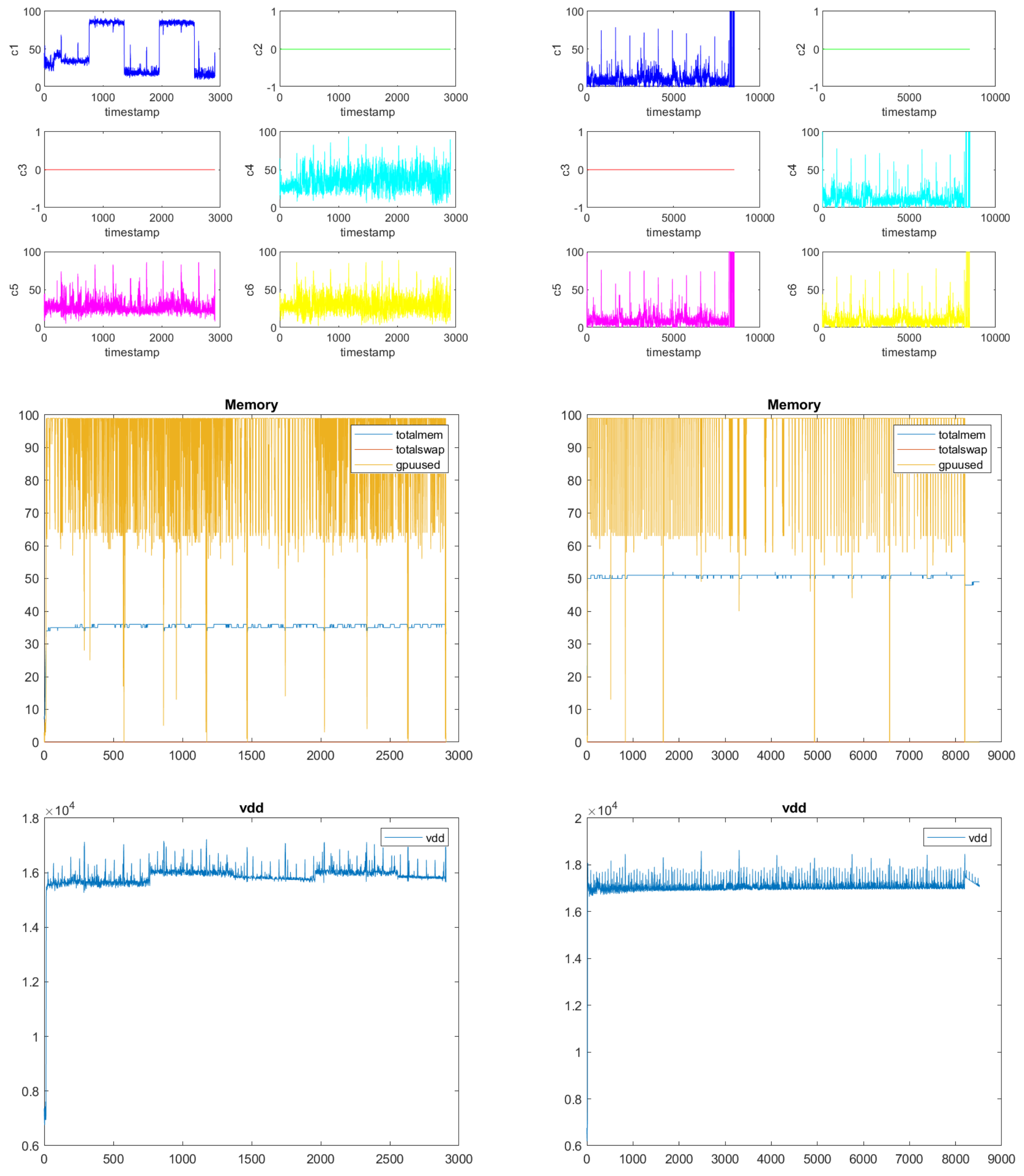


Figure 8.6: CPU, MEM and VDD uses for VGG and Inception on PyTorch at GPU-CFG1

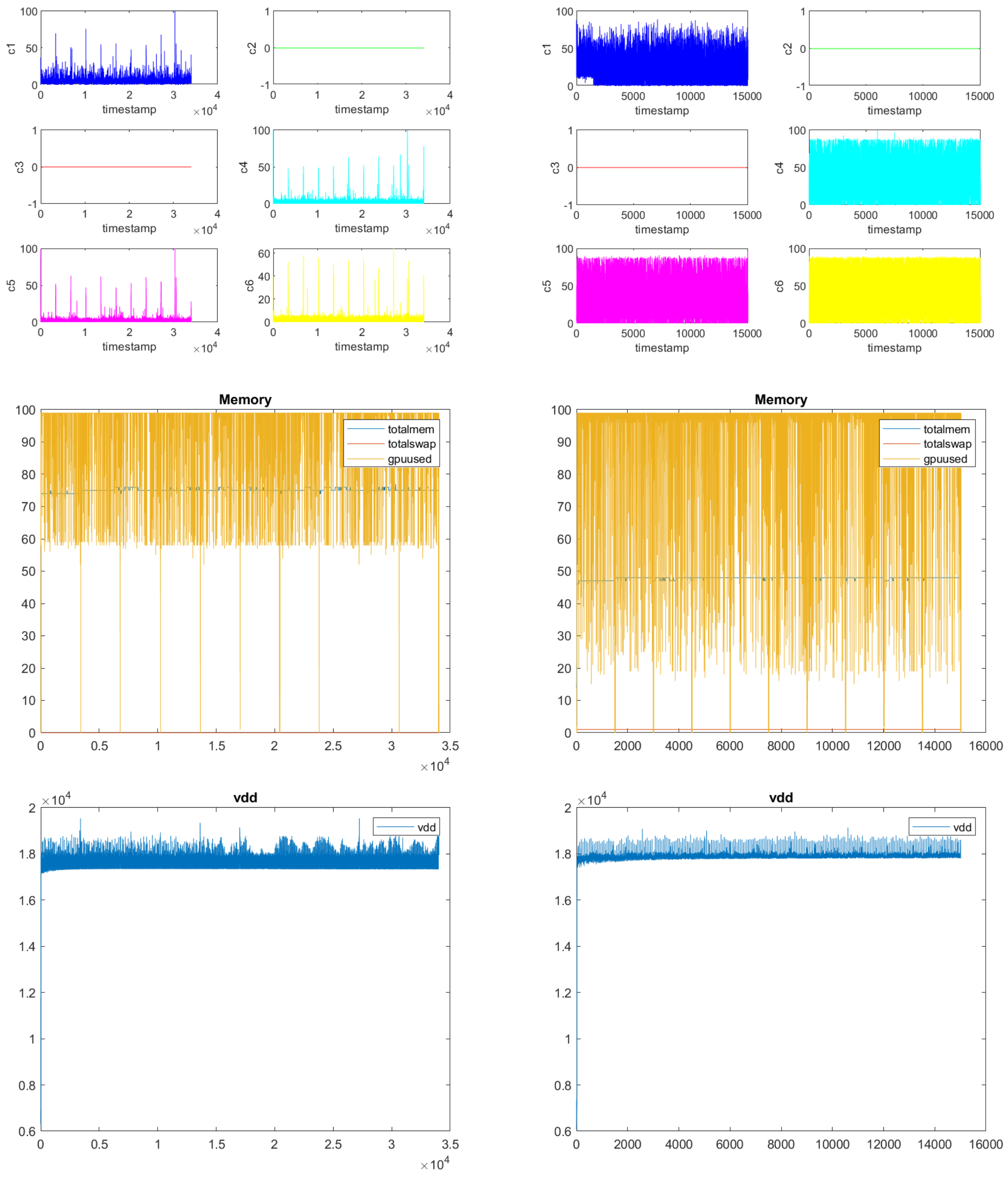


Figure 8.7: CPU, MEM and VDD uses for ResNet and DenseNet on PyTorch at GPU-CFG1

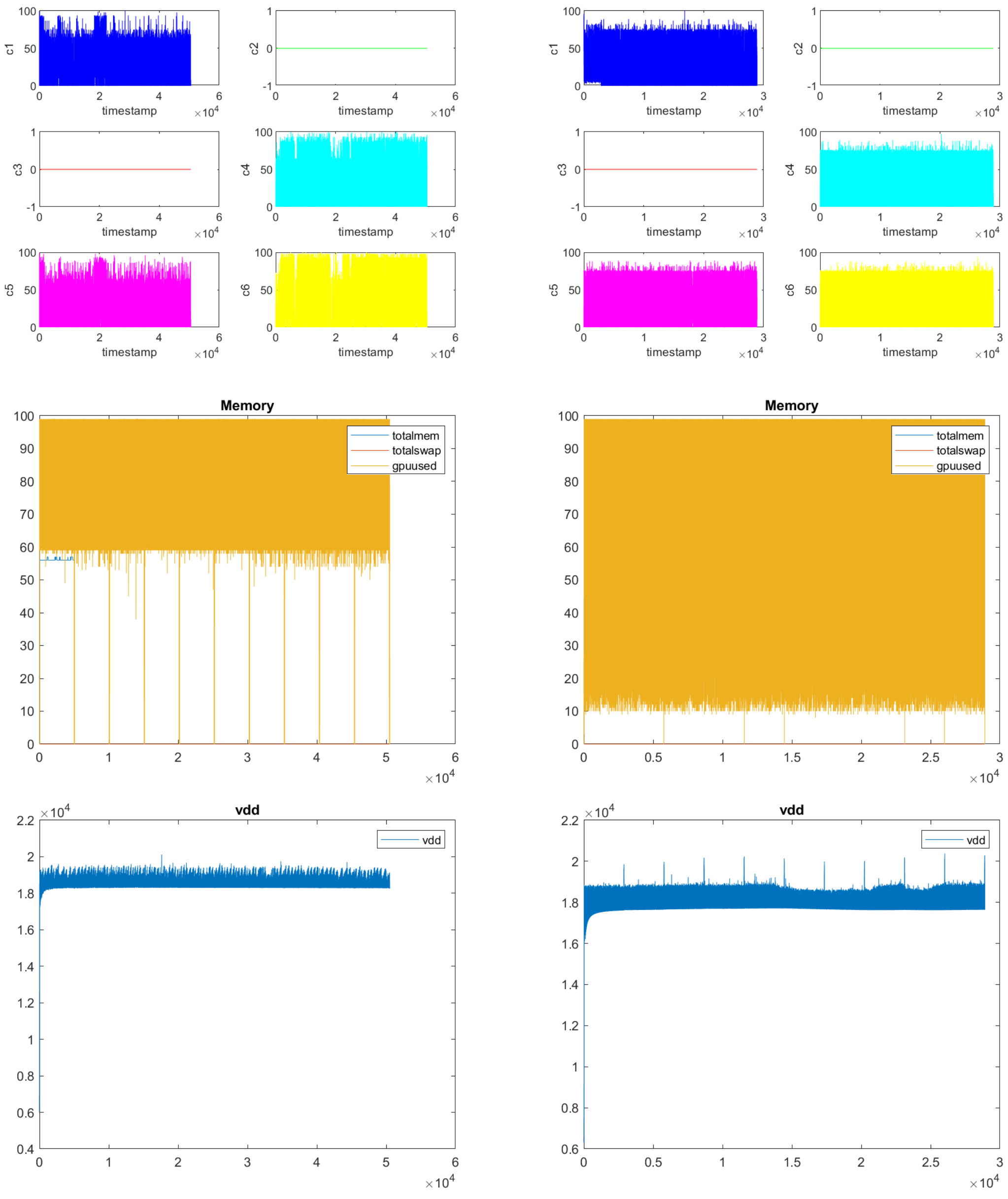


Figure 8.8: CPU, MEM and VDD uses for Squeeze on PyTorch at GPU-CFG1

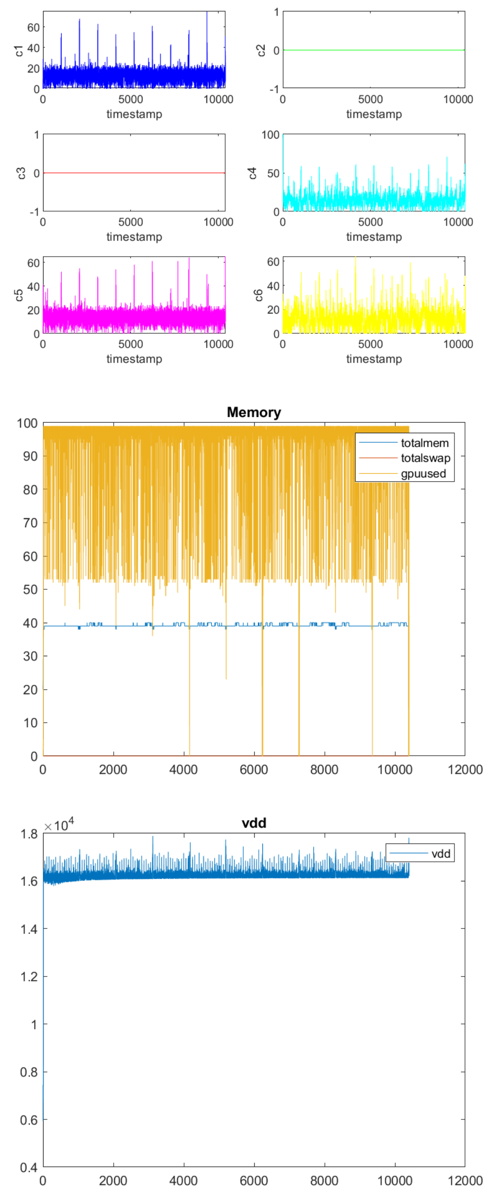


Figure 8.9: CPU, MEM and VDD uses for LeNet and AlexNet on MxNet at GPU-CFG2

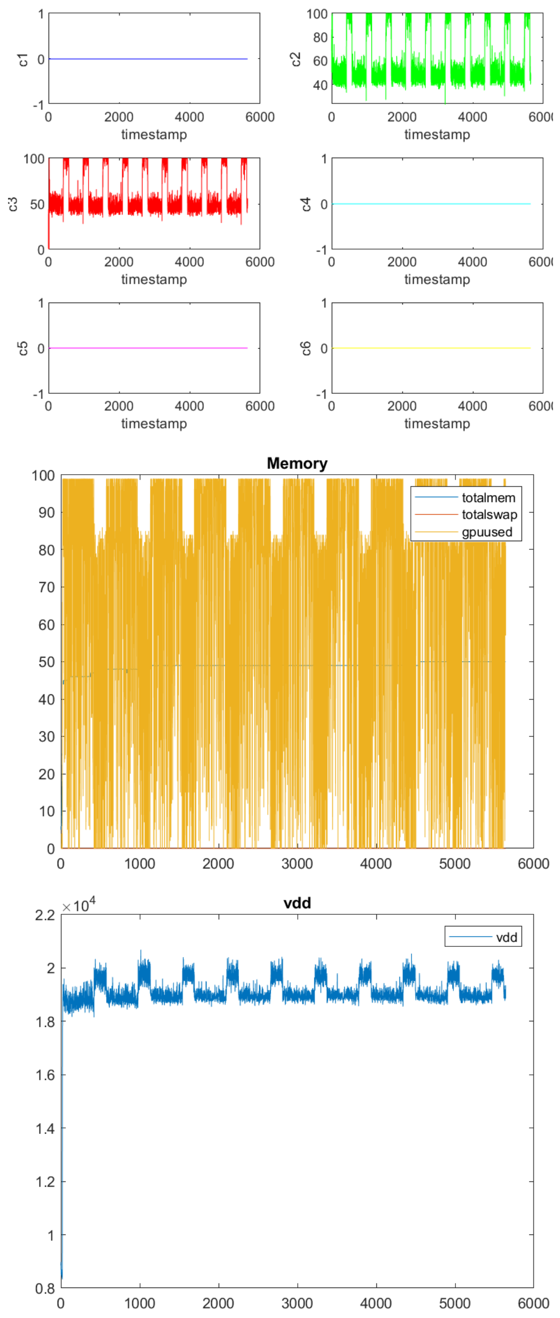


Figure 8.10: CPU, MEM and VDD uses for VGG and Inception on MxNet at GPU-CFG2

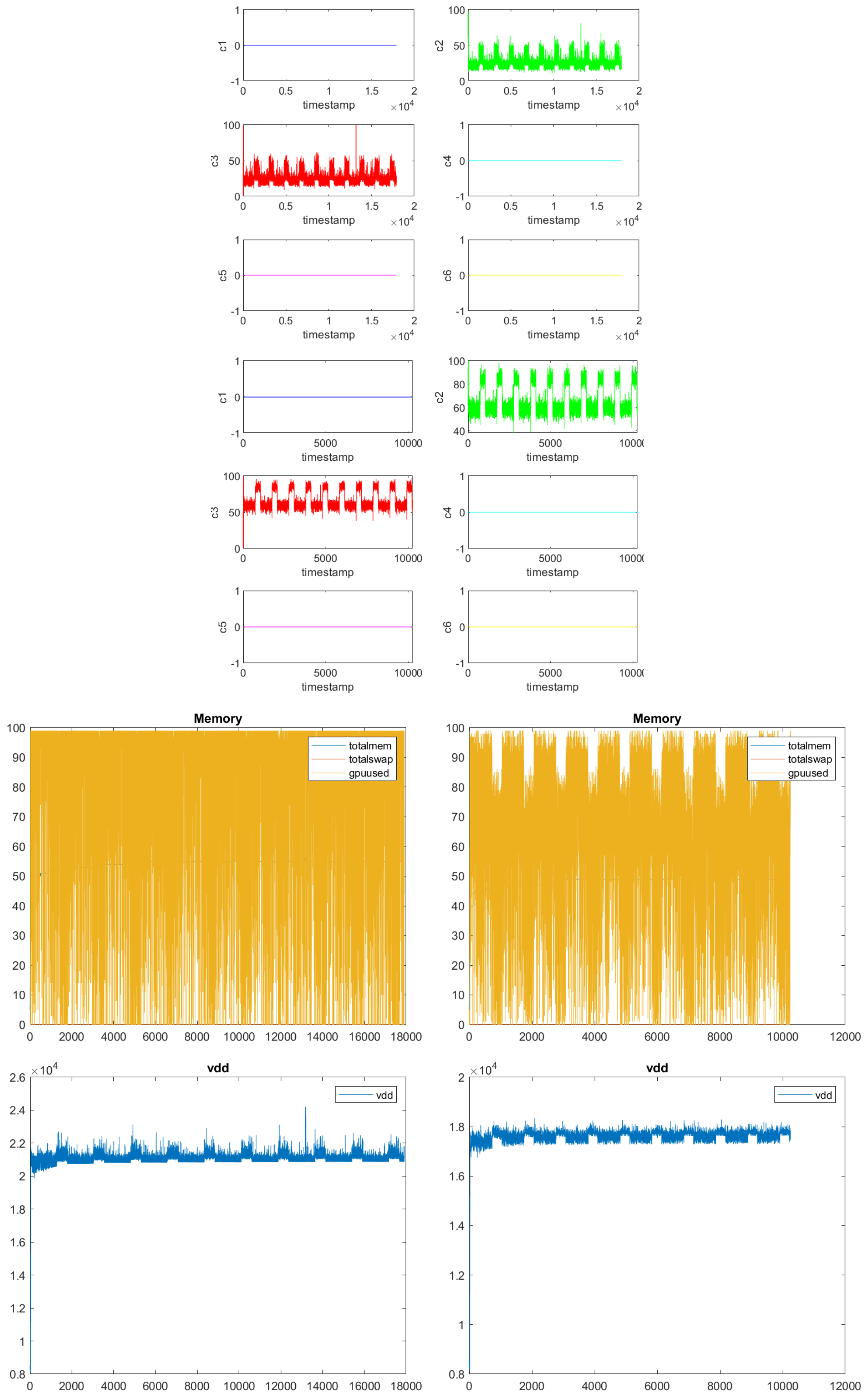


Figure 8.11: CPU, MEM and VDD uses for ResNet and DenseNet on MxNet at GPU-CFG2

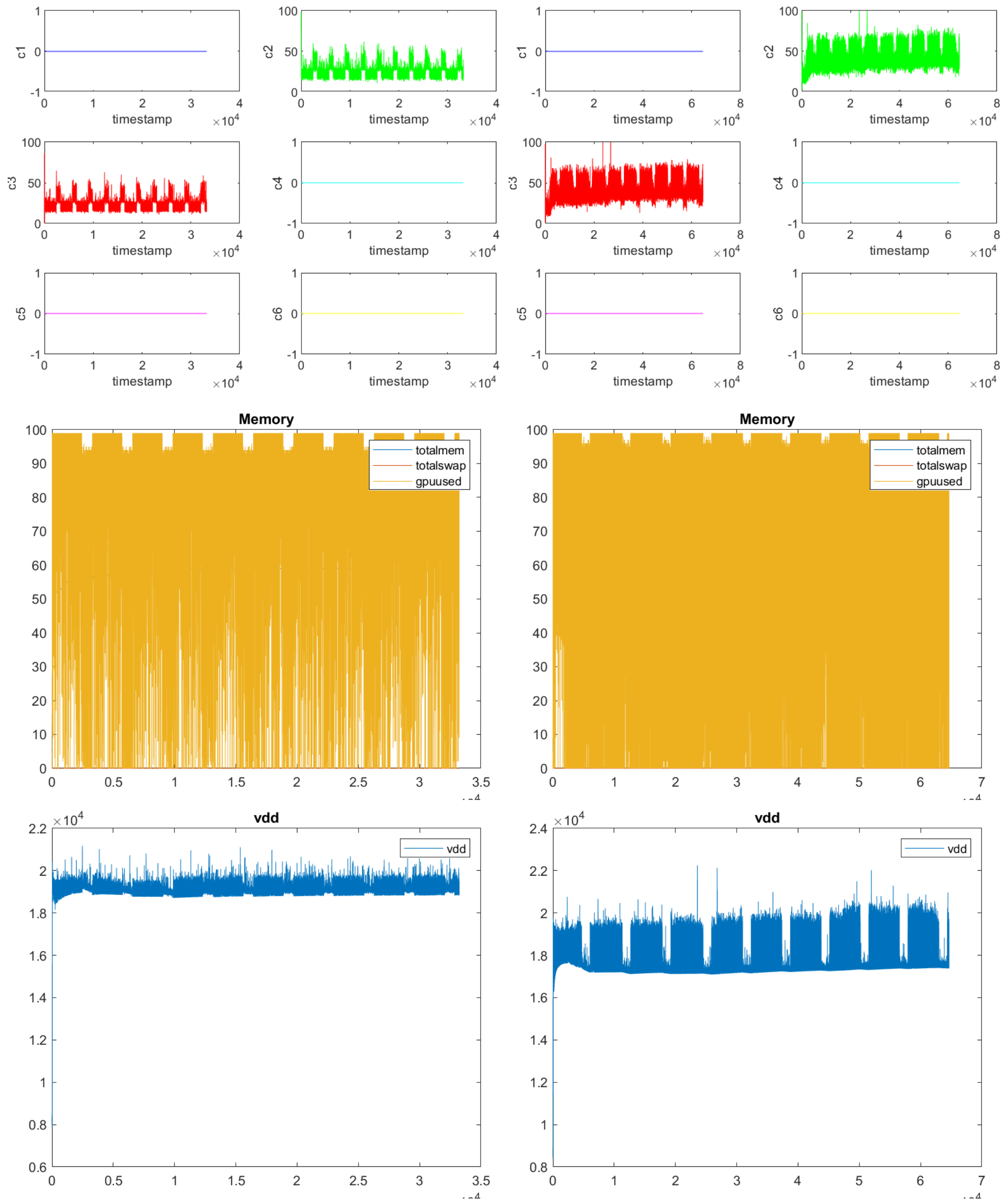


Figure 8.12: CPU, MEM and VDD uses for Sqeeze on MxNet at GPU-CFG2

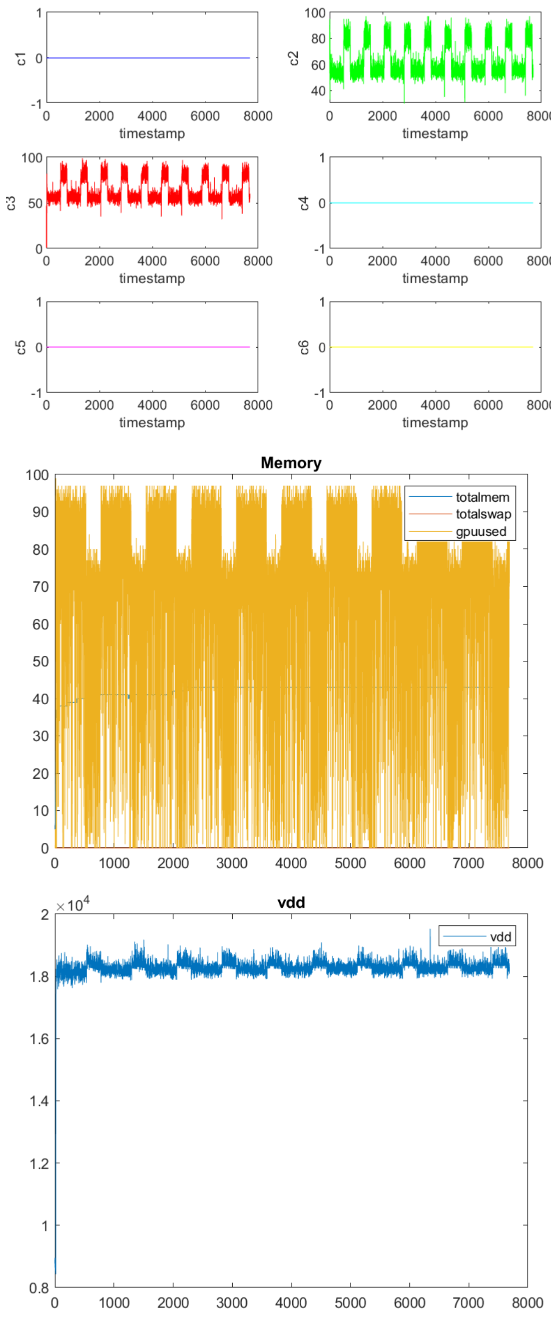


Figure 8.13: CPU, MEM and VDD uses for LeNet and AlexNet on PyTorch at GPU-CFG2

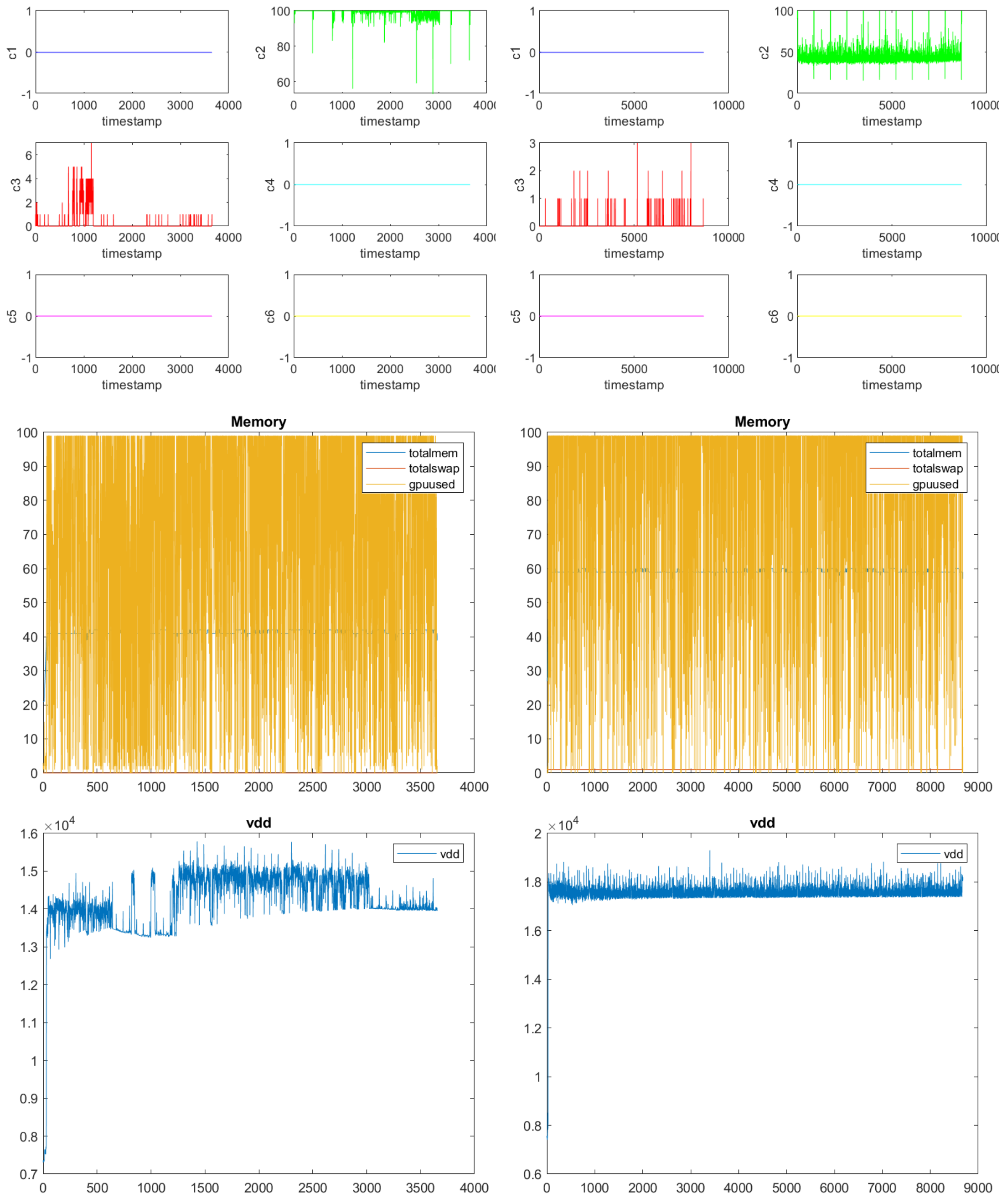


Figure 8.14: CPU, MEM and VDD uses for VGG and Inception on PyTorch at GPU-CFG2

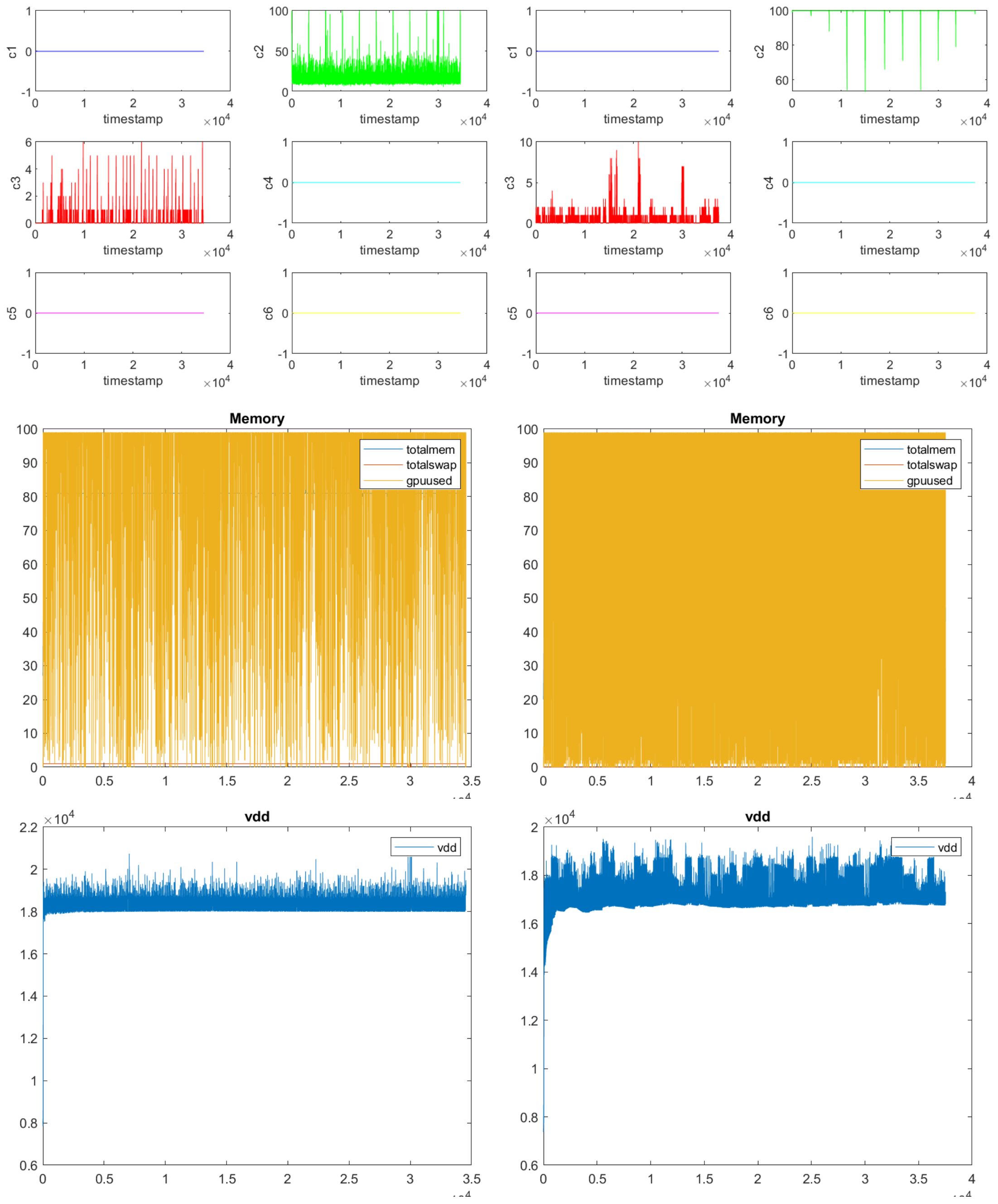


Figure 8.15: CPU, MEM and VDD uses for ResNet and DenseNet on PyTorch at GPU-CFG2

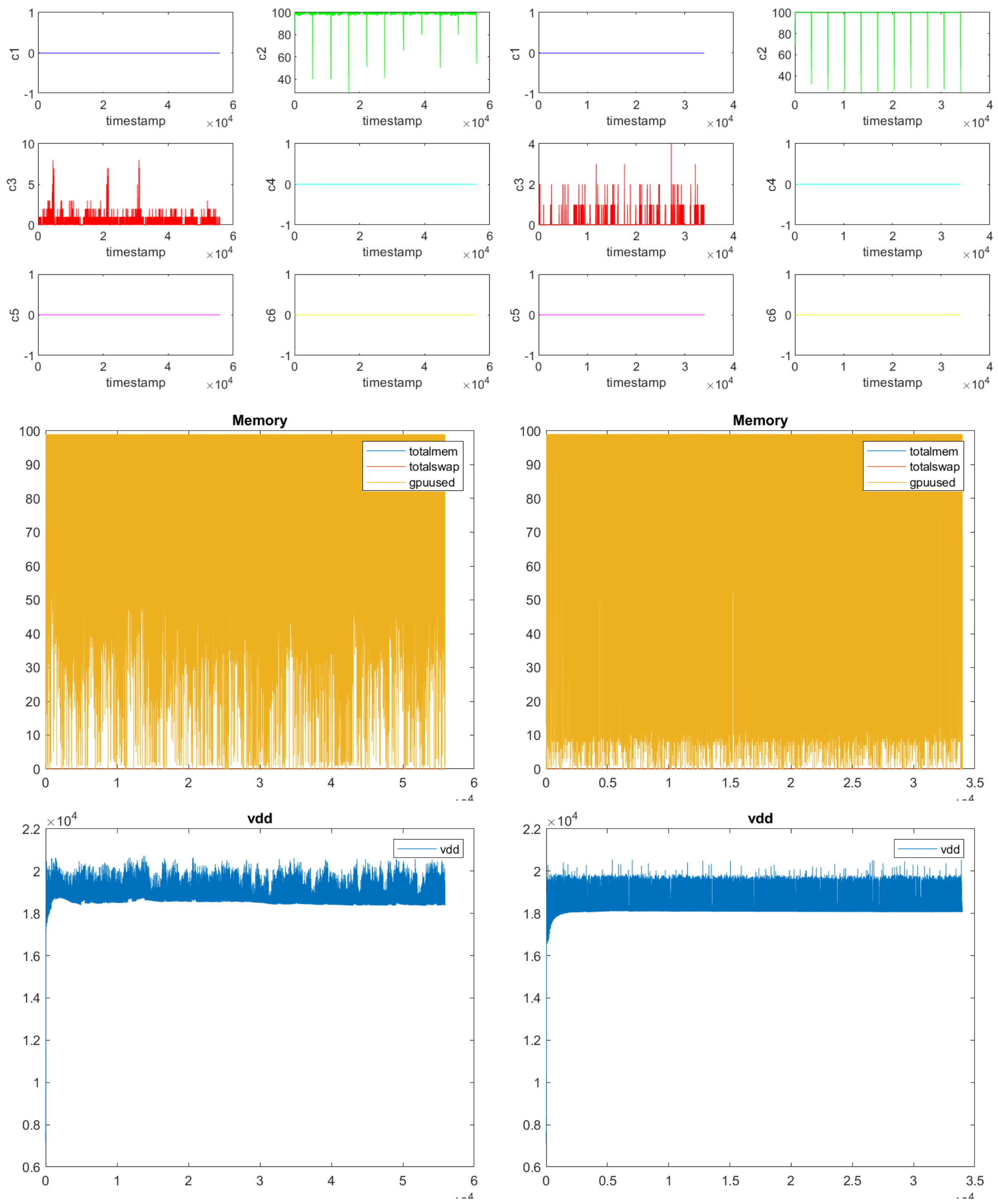


Figure 8.16: CPU, MEM and VDD uses for SqeezeNet on PyTorch at GPU-CFG2

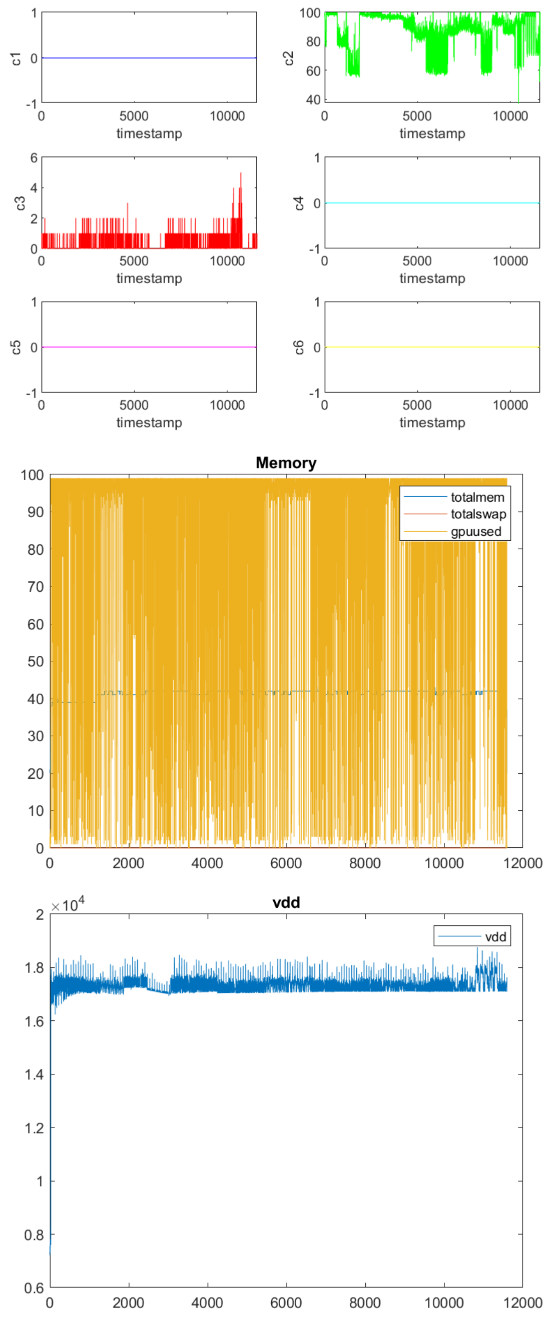


Figure 8.17: CPU, MEM and VDD uses for LeNet and AlexNet on MxNet at GPU-CFG3

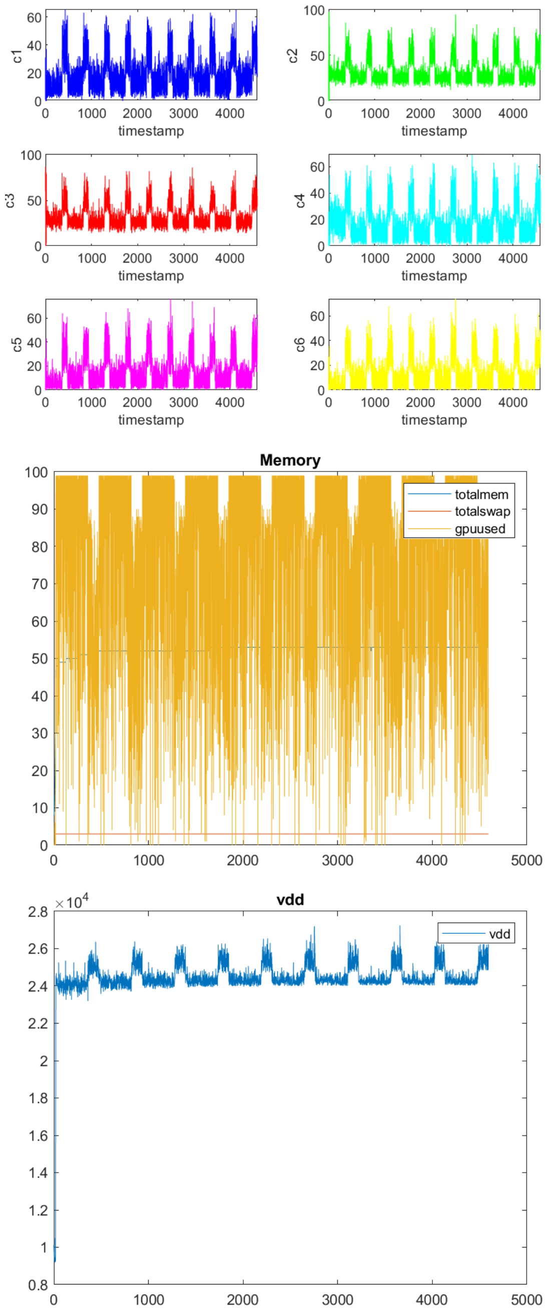
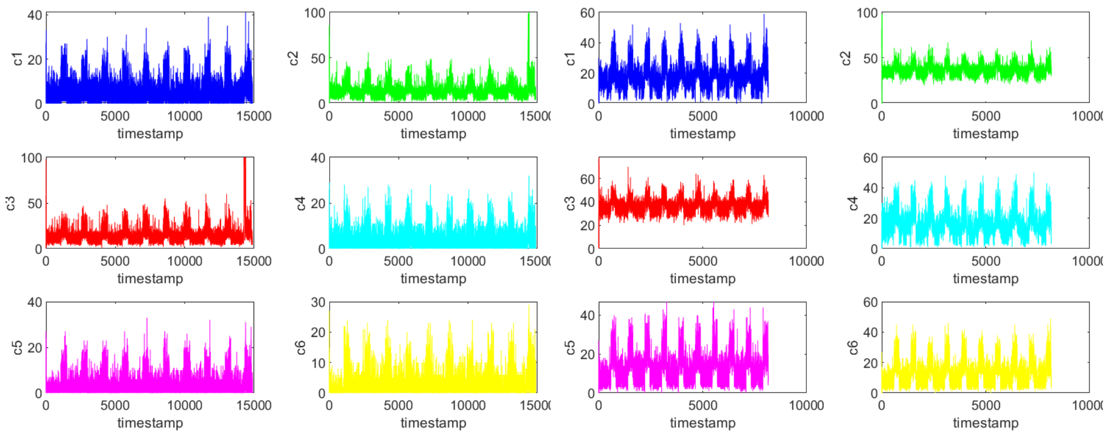
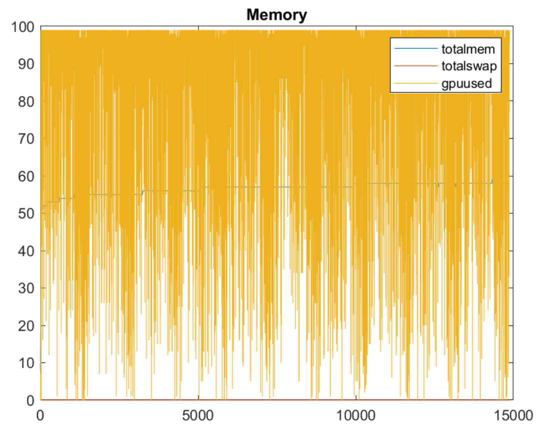
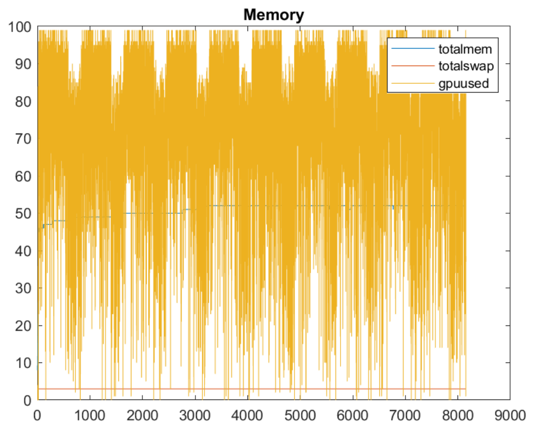
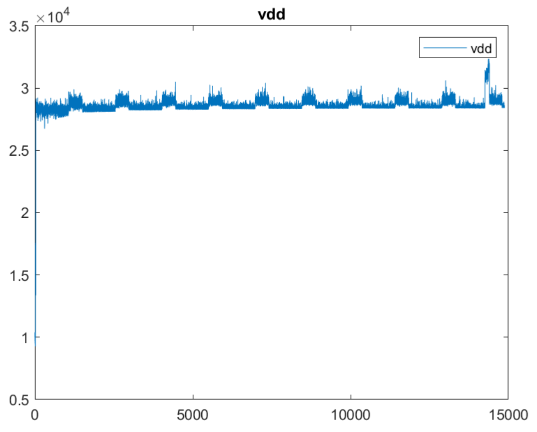


Figure 8.18: CPU, MEM and VDD uses for VGG and Inception on MxNet at GPU-CFG3









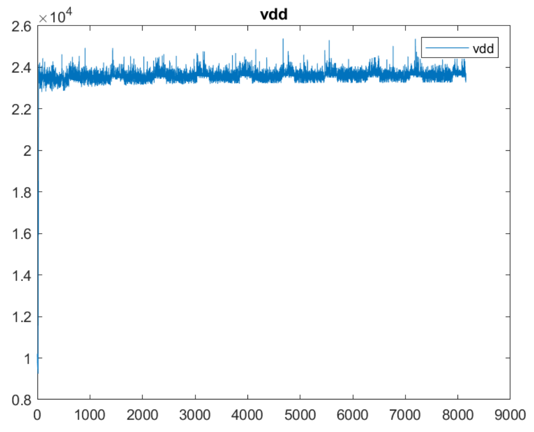


Figure 8.19: CPU, MEM and VDD uses for ResNet and DenseNet on MxNet at GPU-CFG3

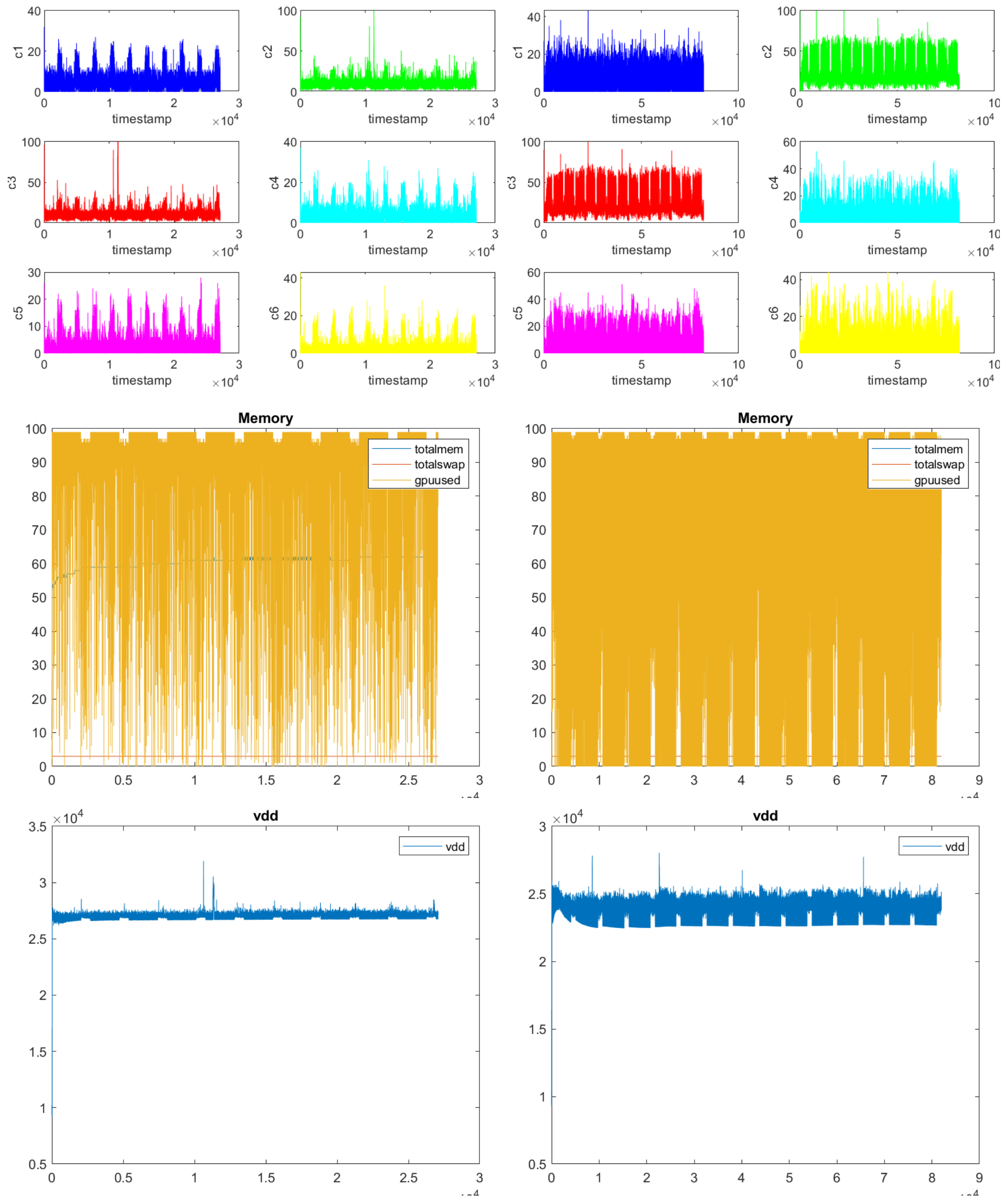


Figure 8.20: CPU, MEM and VDD uses for Sqeeze on MxNet at GPU-CFG3

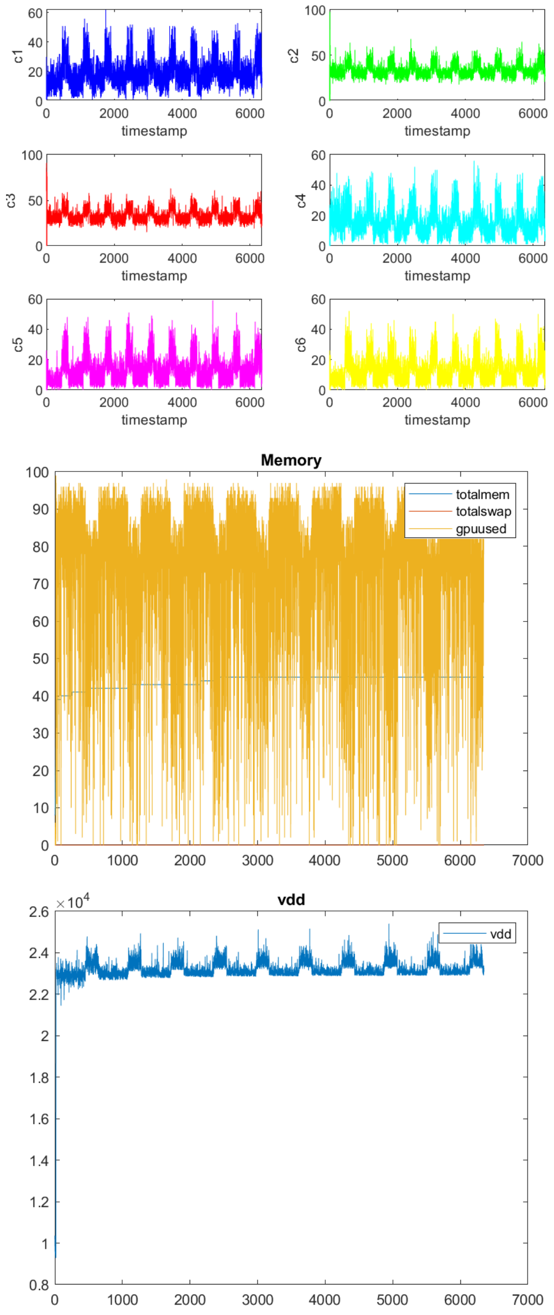
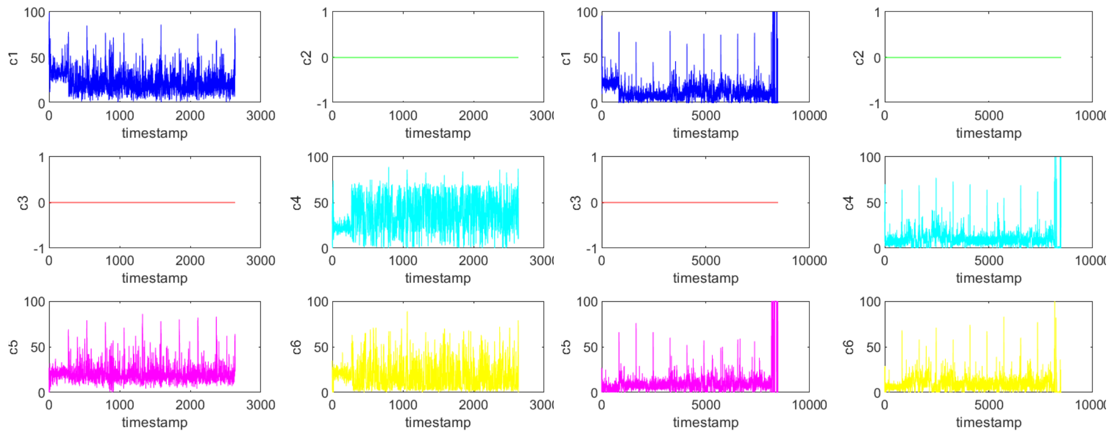
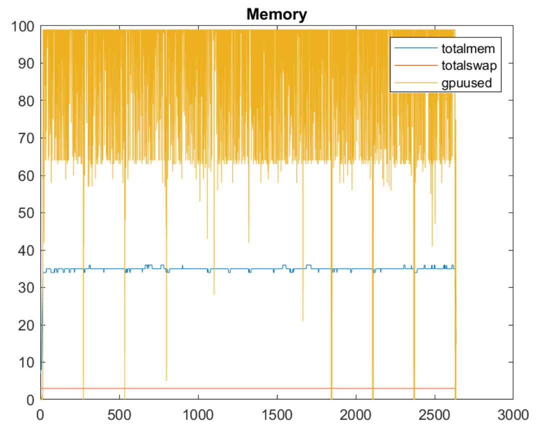
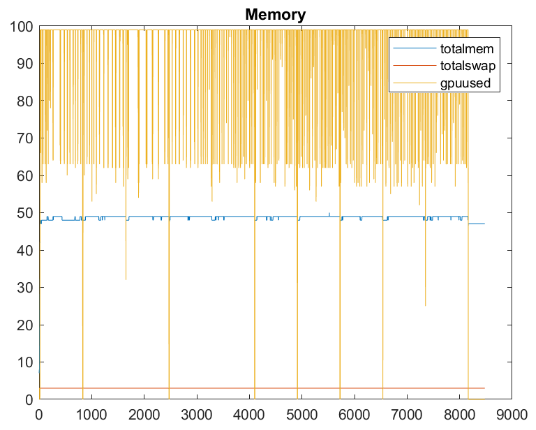
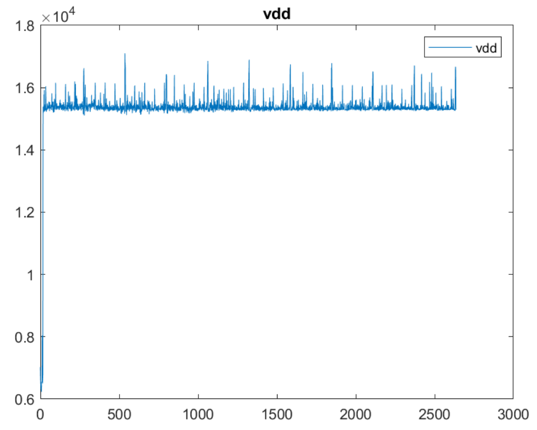


Figure 8.21: CPU, MEM and VDD uses for LeNet and AlexNet on PyTorch at GPU-CFG3









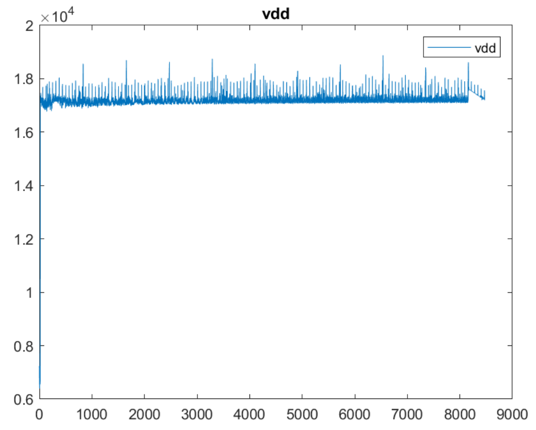


Figure 8.22: CPU, MEM and VDD uses for VGG and Inception on PyTorch at GPU-CFG3

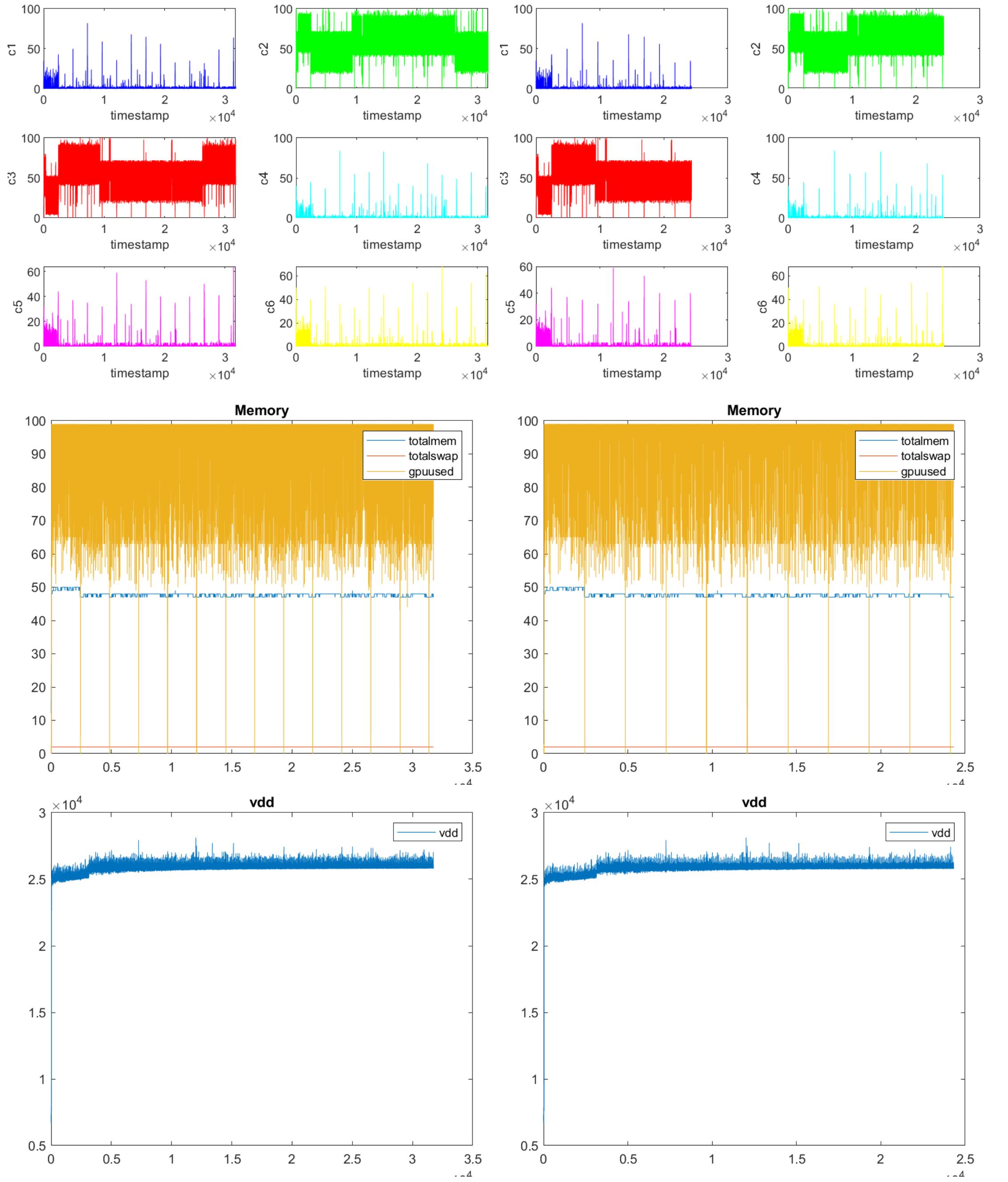


Figure 8.23: CPU, MEM and VDD uses for ResNet and DenseNet on PyTorch at GPU-CFG3

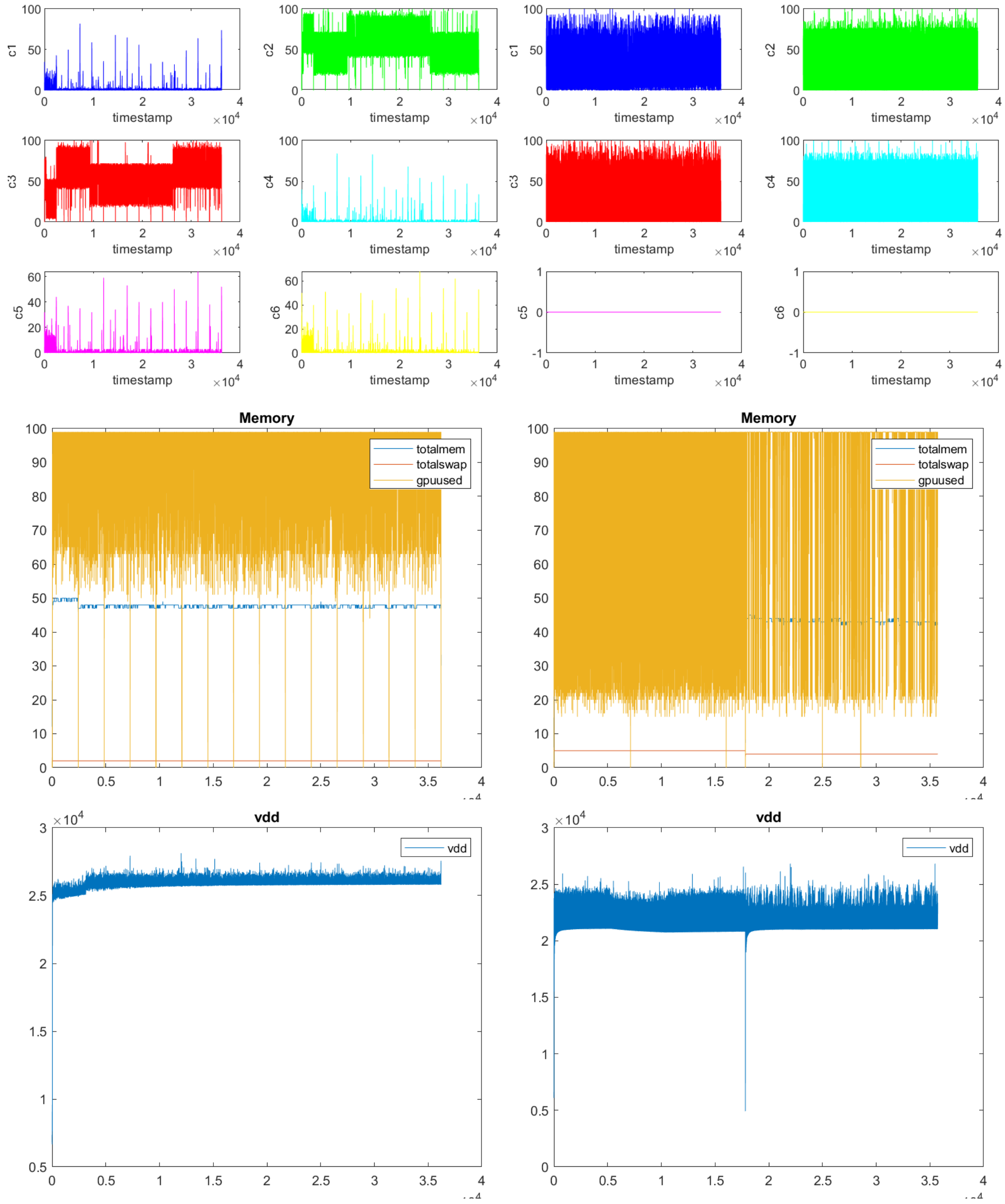
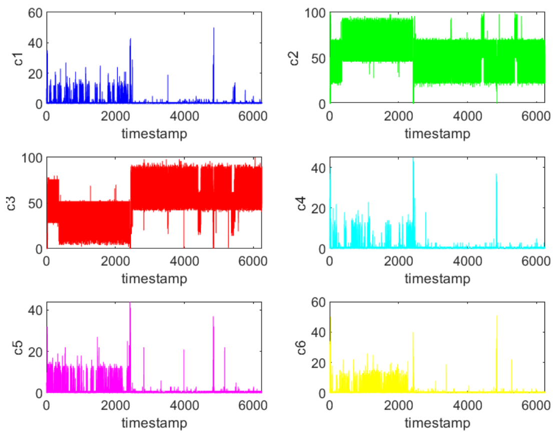
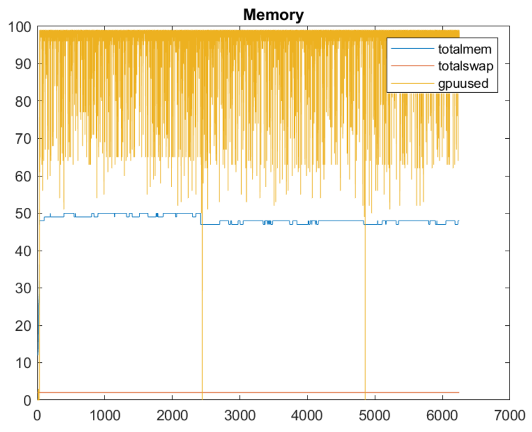


Figure 8.24: CPU, MEM and VDD uses for SqeezeNet on PyTorch at GPU-CFG3





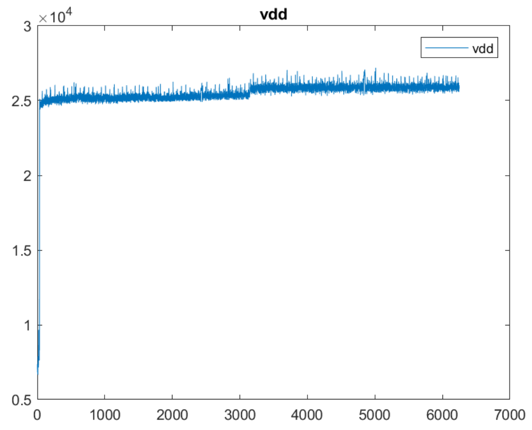


Figure 8.25: CPU, MEM and VDD uses for LeNet, AlexNet, VGG and Inception, ResNet, DenseNet and SqeezeNet on MxNet at CPU-CFG1

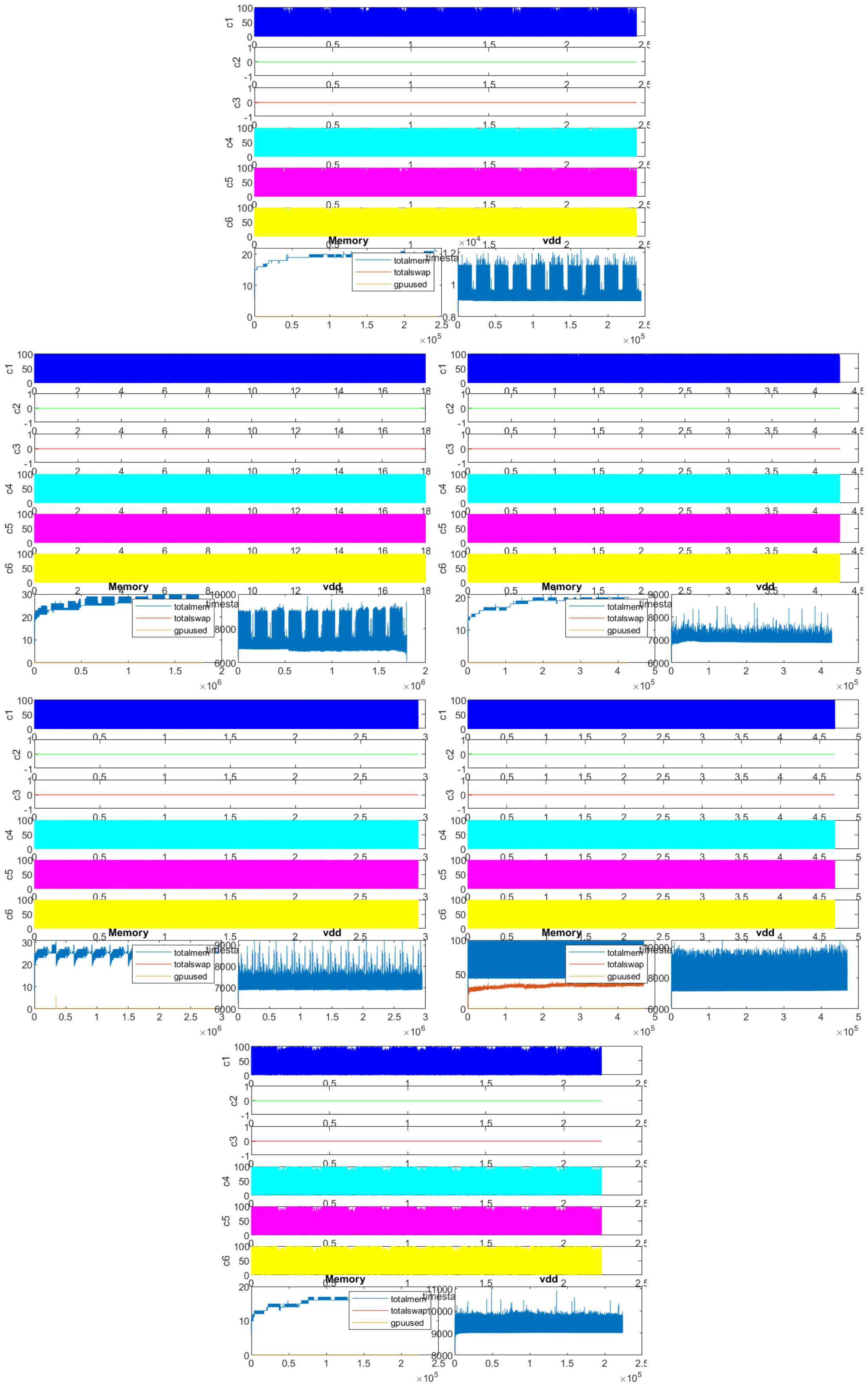


Figure 8.26: CPU, MEM and VDD uses for LeNet and AlexNet, VGG and Inception, ResNet and DenseNet and SqeezeNet on PyTorch at CPU-CFG1

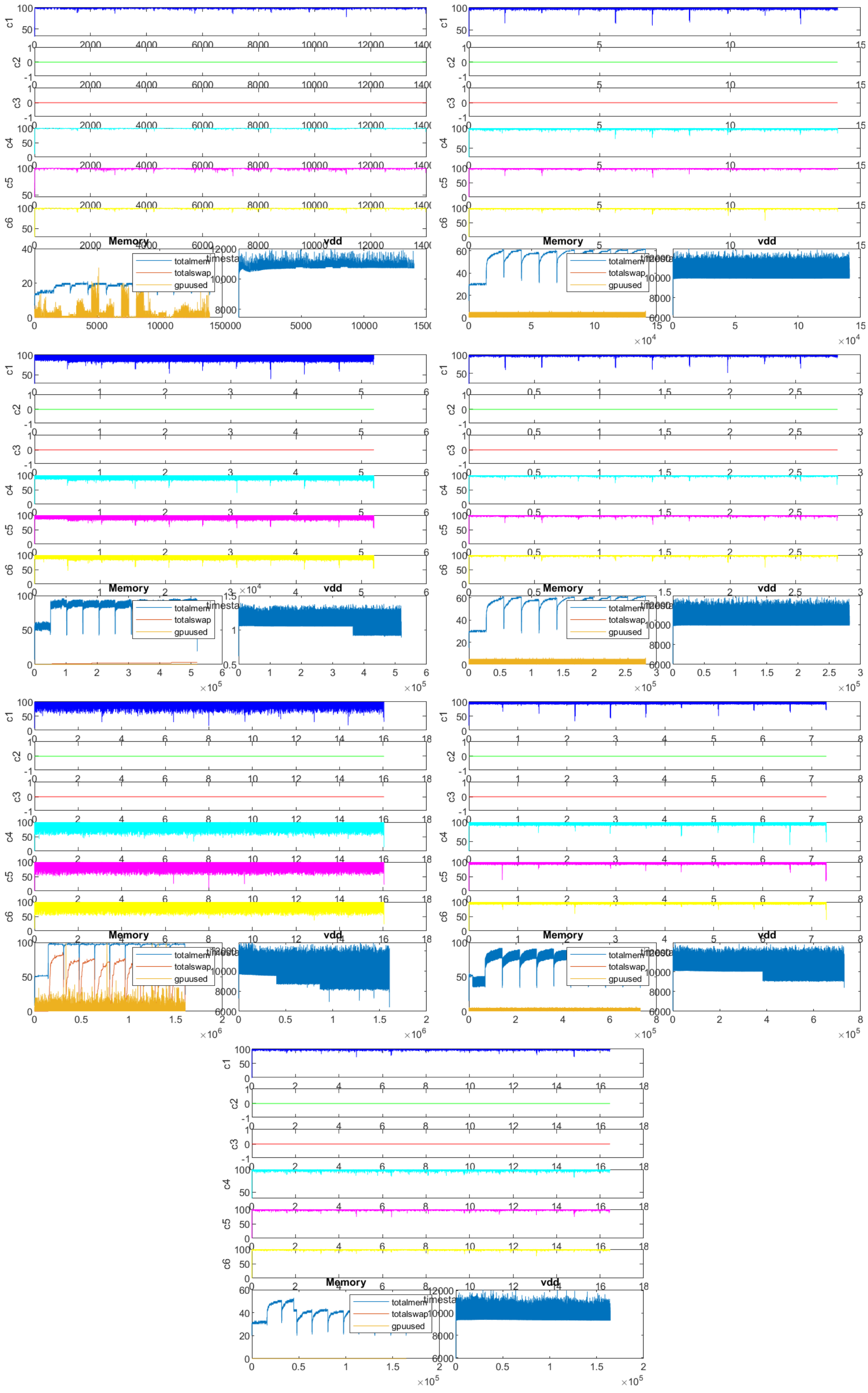


Figure 8.27: CPU, MEM and VDD uses for LeNet and AlexNet, CNN e VGG on MxNet at CPU-CFG1

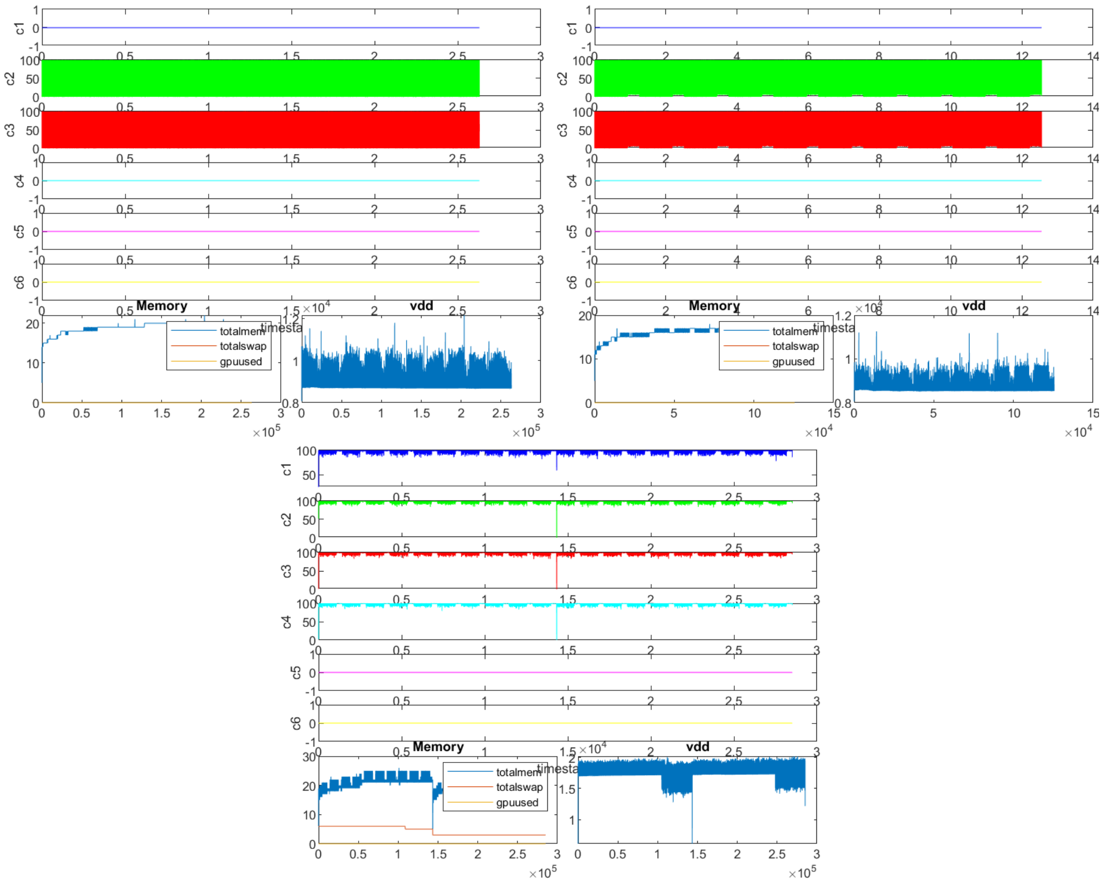


Figure 8.28: CPU, MEM and VDD uses for LeNet and AlexNet, CNN e VGG on MxNet at CPU-CFG1

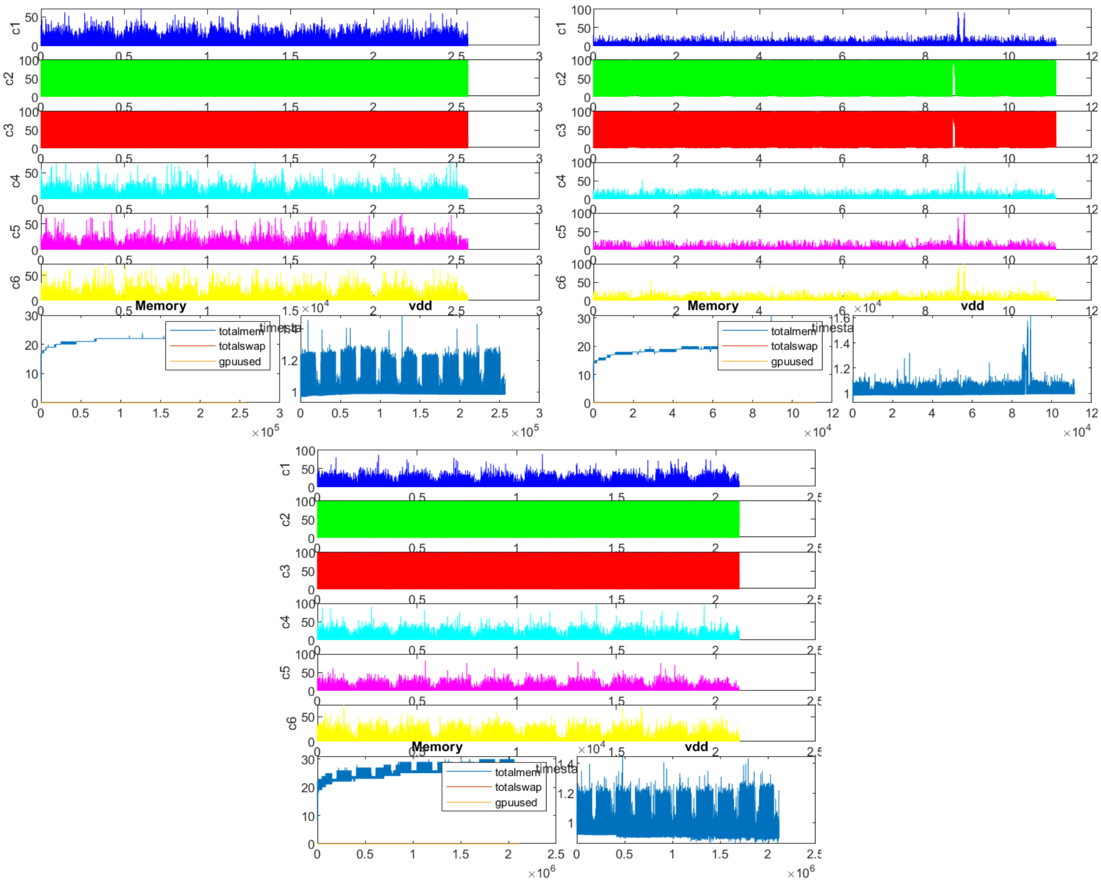
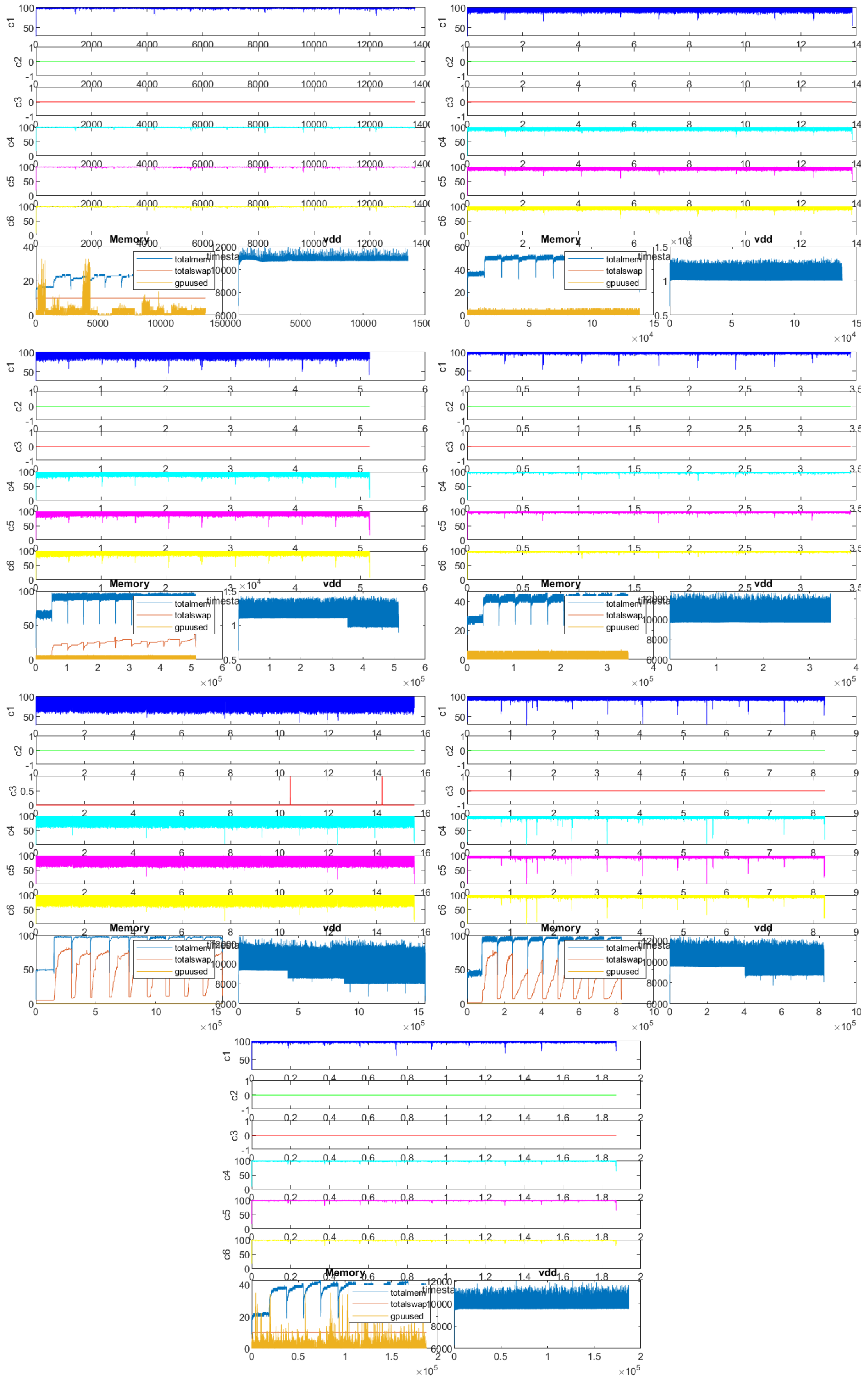


Figure 8.29: CPU, MEM and VDD uses for LeNet, AlexNet, VGG, Inception, ResNet,DenseNet and SqeezeNet on MxNet at CPU-CFG1


c1
c2
c3
c4
c5
c6
Memory
vdd
totalmem
totalswap
gpuused
timestamp