Thesis for The Degree of Licentiate of Engineering

Multi-LSTM Acceleration and CNN Fault Tolerance

Stefano Ribes

Division of Computer Engineering
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2021

Multi-LSTM Acceleration and CNN Fault Tolerance

Stefano Ribes

Advisor: Ioannis Sourdis, Prof. at Chalmers University of Technology
Co-Advisor: Pedro Trancoso, Prof. at Chalmers University of Technology
Co-Advisor: Vassilios Papaefstathiou, Post Doc. researcher at FORTH-ICS
Examiner: Ulf Assarsson, Prof. at Chalmers University of Technology
Discussion Leader: Theocharis Theocharides, Prof. at University of Cyprus

Technical Report No 197L
ISSN 1652-876X
Department of Computer Science & Engineering
Division of Computer Engineering
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2021.

Abstract

This thesis addresses the following two problems related to the field of Machine Learning: the acceleration of multiple Long Short Term Memory (LSTM) models on FPGAs and the fault tolerance of compressed Convolutional Neural Networks (CNN). LSTMs represent an effective solution to capture long-term dependencies in sequential data, like sentences in Natural Language Processing applications, video frames in Scene Labeling tasks or temporal series in Time Series Forecasting. In order to further boost their efficacy, especially in presence of long sequences, multiple LSTM models are utilized in a Hierarchical and Stacked fashion. However, because of their memory-bounded nature, efficient mapping of multiple LSTMs on a computing device becomes even more challenging. The first part of this thesis addresses the problem of mapping multiple LSTM models to a FPGA device by introducing a framework that modifies their memory requirements according to the target architecture. For the similar accuracy loss, the proposed framework maps multiple LSTMs with a performance improvement of 3× to 5× over state-of-the-art approaches. In the second part of this thesis, we investigate the fault tolerance of CNNs, another effective deep learning architecture. CNNs represent a dominating solution in image classification tasks, but suffer from a high performance cost, due to their computational structure. In fact, due to their large parameter space, fetching their data from main memory typically becomes a performance bottleneck. In order to tackle the problem, various techniques for their parameters compression have been developed, such as weight pruning, weight clustering and weight quantization. However, reducing the memory footprint of an application can lead to its data becoming more sensitive to faults. For this thesis work, we have conducted an analysis to verify the conditions for applying OddECC, a mechanism that supports variable strength and size ECCs for different memory regions. Our experiments reveal that compressed CNNs, which have their memory footprint reduced up to 86.3× by utilizing the aforementioned compression schemes, exhibit accuracy drops up to 13.56% in presence of random single bit faults.

Keywords

FPGA, Machine Learning, LSTMs, SVD, HLS, Roofline Model, CNNs,

Fault Tolerance, Compression, Caffe

Acknowledgment

To all people I've met during my years at Chalmers, thank you. I learned a lot from each and everyone of you.

Спасибо, бабушка, царство тебе небесное.

This work is supported by the European Commission under the Horizon 2020 Program through the ECOSCALE (grant agreement 671632) and SHARCS (grant agreement 644571) projects as well as by the European Research Council (ERC) under the MECCA project (Contract No. 340328).

List of Publications

Appended publications

This thesis is based on the following publications:

[I] S. Ribes, P. Trancoso, I. Sourdis, C.-S. Bouganis,
"Mapping Multiple LSTM models on FPGAs",
Int'l Conf. on Field-Programmable Technology (FPT), December, 2020.

[II] S. Ribes, A. Malek, P. Trancoso, I. Sourdis,
"Reliability Analysis of Compressed CNNs",
Technical Report.

Contents

Chapter 1

Introduction

Artificial intelligence and in particular Machine Learning (ML) have gain high popularity in the recent years. This popularity is due to its outstanding results in tasks such as image classification, language translation, sentiment analysis and so on. ML applications rely on Deep Neural Networks (DNNs), special algorithms which process given inputs through a series of non-linear functions.

The way DNNs are built follows a process called training. In fact, DNNs can be seen as non-linear function estimators which combine a series of differentiable functions, which may, or may not, include a set of tunable parameters. Because of that, the network output will depend on the series of functions and on the values of such parameters. By defining a proper loss function which associates the network outputs with the expected outputs from a given probability distribution, one can tune, *i.e.* train, the network parameters to model such distribution. In image classification tasks for example, by running a network through an image set, called training set, and by comparing network outputs with the labels associated to each image in the set, a DNN can be trained to classify new unseen images, *i.e.* outside the training set.

The process of utilizing trained DNNs on novel input data, *i.e.* never seen at training time, is defined as inference. Training is typically performed offline, usually with full precision parameters, and can require days or even weeks to complete for certain network architectures. Inference on the other hand requires rapid response times since the network only processes a single input data (or a small batch), compared to a commonly large training set.

In this thesis we focus on the DNNs inference process. Some of the most popular networks include Long Short Term Memory (LSTMs) and Convolutional Neural Networks (CNNs). In particular, running inference of LSTMs, CNNs and ML workloads is in general compute and data intensive. One way of achieving high performance in inference is through acceleration, typically utilizing GPUs, ASICs or FPGAs. FPGAs in particular can achieve performance comparable to GPUs, with the advantage of flexibility and high energy efficiency. Their flexibility often allows to exploit compression techniques applied to DNNs. In fact, the most common data that DNNs require is in form of matrices or tensors, which can be compressed, *e.g.* made sparse. The process of sparsifying DNNs data without significantly impacting DNNs output is referred as pruning and consists of an active area of research. Other techniques for compressing

networks can consist of dimensionality reduction, weight clustering, parameter quantization, *et cetera*.

Even when compressed, the large amounts of data that ML applications require are usually stored in DRAM as they cannot fit on chip. Such data can be sensitive to attacks or faults, which can lead to Silent Data Corruption (SDC) faults, *i.e.* to significant drops in the network output accuracy. In ML applications running one or multiple DNNs inference processes, faults can happen in different regions of a network, potentially affecting the output and so the accuracy of the network. The problem can become even more severe when the DNN memory footprint is compressed, eventually leading to even higher accuracy drops.

1.1 Problem Statements

This thesis focuses on two main topics: (*i*) accelerating and mapping multiple LSTM models on an FPGA and (*ii*) the fault tolerance of CNNs applications.

Since LSTM models execution is dominated by vector-matrix multiplications involving their parameters, *i.e.* weight and bias values, LSTMs are inherently memory bound. The problem is further amplified when multiple LSTM networks are utilized in parallel and accelerated by the same device.

When accelerating CNNs instead, compressing their parameters can lead to better performance by simply trading-off negligible losses in accuracy. However, some compression mechanisms can make the networks more sensitive to single and multi bit flips faults.

1.1.1 Acceleration of SVD-based multi-LSTMs

When accelerating deep neural networks on FPGAs, the limited off-chip memory bandwidth of a given device may become a significant bottleneck due of the large amount of data that such applications require. LSTM networks in particular, not only result in a memory-bound application, but also exhibit data dependencies across their execution. In fact, compared to other types of DNNs, LSTMs are best suitable to capture the relations between different elements in sequences of data, even if "distant" from each. One example of this are the words in a sentence to be translated: if one attempts to translate "*The cat, which is on the chair, is red*" to Italian, by calling the LSTM on each word, its internal memory state would easily keep track of the relation between 'cat' (a masculine noun in Italian) and the second distant 'is' in order to properly conjugate the adjective 'red' to masculine, matching the gender of 'cat' (and not to feminine, which would match the word 'chair' in Italian).

One way to tackle the problem of the large memory footprint, and thereby the data dependency, is approximating the parameters of the given LSTM network, *i.e.* reducing the amount of data that need to be transferred from and to main memory. Such approximation can be achieved with numerous techniques, one of which consists in applying a zero mask to the weight parameters followed by a retraining step. However, retraining a network is often an expensive task, both in terms of execution time and compute power. A possible solution that

avoids retraining is applying Singular Value Decomposition (SVD) [29, 5, 28] to the weight matrices of the network.

The parameters of the matrices of LSTMs are typically full rank, meaning that the full amount of information is encoded in the matrices values. Applying SVD can reduce the rank of such matrices, thereby filtering excessive information. Because of that, SVD approximation can significantly compress the parameter space of an LSTM and has the advantage of being performed offline. SVD, however, tends to considerably reduce the original accuracy of the network [10, 3]. On top of that, when machine learning applications utilize multiple LSTM networks in parallel, *e.g.* multiple layers on a single device, time-multiplexing resources, the memory may become a more severe bottleneck. In such scenarios, SVD alone may not be sufficient to achieve high performance, thus requiring more advanced algorithms and mechanisms to approximate the networks parameters.

1.1.2 Reliability Analysis of CNNs Workloads

Safety critical software applications can be found in various domains, like aerospace, automotive or biomedical, and all demand a high level of fault tolerance. Example of such applications are satellite launch, spacecrafts navigation systems, self-driving cars, medical images devices, *et cetera*. Because of deep learning potential and achievements, in recent years DNNs have become an attractive instrument for the aforementioned applications.

However, it's hard to efficiently measure the safety of ML applications running inference, due to their nature of function estimators. On top of that, ML inference requires large datasets of parameters that are usually stored in DRAM. DRAM memory cells can be particularly susceptible to faults or attacks, leading a given ML application to possible crashes or wrong outputs, *e.g.* high drops in accuracy, thereby making them even less suitable for safety critical tasks.

In order to mitigate the problem and design more robust deep learning applications, ML workloads can be deployed on machines with fixed Error Correcting Codes (ECCs) DRAM protection. However, such solution suffers from significant DRAM capacity, latency and energy overheads [19].

With this respect, approximating and/or compressing the network parameters can reduce the application memory footprint and in turn improve the network performance and diminish the ECC size. However, a smaller application data region can make the application itself more sensitive to faults. Because of that, DNNs compression can eventually lead to a less accurate network, thus requiring costly fault tolerance techniques, *i.e.* DRAM ECC, effectively resulting in a vicious cycle.

1.2 Thesis Objectives and Contributions

This thesis is divided in two parts, each of which giving a different set of contributions, as explained below.

1.2.1 Acceleration of SVD-based multi-LSTMs

The objective of the first part of the thesis is to address the problem of approximating multiple LSTM cells on a single FPGA device so to reduce their data volume, reduce memory bandwidth pressure and thereby improve their performance.

Related Work: LSTMs and in general Recurrent Neural Networks (RNNs) are inherently memory-bounded applications. As such, research has focused on reducing the amount of data to be either transferred, between the main memory and the accelerator, or stored, in main memory or directly on the device.

Early work on compressing LSTMs include ESE [11], which introduces a load-balancing aware compression methodology based on parameter pruning and quantization. Such methodology is then exploited by the proposed FPGA accelerator, able to efficiently process sparse, *i.e.* pruned, matrices and vectors.

Research in the direction of storing all the parameters on-chip has shown significant performance gains [4, 15, 25, 24]. In fact, the proposed designs do not need to access the off-chip memory, since they can accommodate all the compressed LSTM weights on the device memory. However, this solution is not viable for large networks.

Finally, a work closer to our approach, which we use to build upon our contributions, is a SVD-based compression proposed by Kouris *et. al.* [14] and Rizakis *et. al.* [23]. Their solution is applied to LSTMs which are handled separately by the accelerator, without exploiting possible redundancies in their weights and thus wasting memory bandwidth.

Thesis Approach: our work takes advantage of the similarities across multiple LSTMs' in order to approximate their parameters and reduce the memory footprint of the entire application. The proposed approximation is based on a SVD-based algorithm and is included in a framework able to identify the best design points by trading off performance and network accuracy. After selecting the best designs, the LSTMs are finally executed on our proposed FPGA accelerator, written in High Level Synthesis (HLS) language.

Contributions: the thesis contributions reported in the first part can be summarized as follows:

- an SVD-based algorithm that approximates multiple LSTMs by exploiting the redundancy of their weight parameters and that does not require any retraining,
- an FPGA accelerator for executing multiple LSTM models that operate on a set of synchronized inputs,
- a systematic framework for exploring the large design space that identifies the best trade-offs in terms of accuracy of the approximated LSTMs and performance of the FPGA accelerator.

1.2.2 Reliability Analysis of CNNs Workloads

The objective of the second part of the thesis is to analyze and investigate the response to faults, and in turn the tolerance to faults, of highly compressed DNNs, in particular CNNs. Our final goal is to verify if the conditions for utilizing an advanced and more scalable ECC protection scheme like Odd-ECC [19] are met. In fact, if different CNN parameters data have different sensitivity to faults, one can then take advantage of that and protect the data in DRAM by using Odd-ECC, which supports variable strength and size ECCs for different memory regions.

Related Work: Most works on DNN fault tolerance are divided in conducting a sensitivity analysis of faults targeting accelerators, like GPUs, ASICs and FPGA, happening in either their datapath [16, 26, 31] or their buffers [22, 16], or attacks against DNNs storing their parameters in main memory [21, 1], in particular row hammer, laser beam, gradient descend [17], backdoor [6, 7] or trojan attacks [30, 32, 2, 18].

Many works analyze not only full precision networks, but also networks utilizing quantized parameters [22, 16]. In [13], faults are injected in an FPGA accelerator implementing Binarized Neural Networks (BNN). BNNs represent an extreme case of network compression where the weights are reduced to either binary or ternary values (-1, 0 or 1).

Other works also address the general problem of detecting and correcting faults happening in DNNs accelerators. In Li et al. [16], selective latch hardening is utilized to detect and correct faults happening in the DNNs accelerators datapath. Qin *et al.* [20] instead propose an algorithm for detecting faults and correct them by setting the affected parameter to zero. They also introduced a binary representation for real numbers that limits the distortion caused by bit flips errors. Focusing on ECC mechanisms, the idea of Guan et al. [8] is to exploit the unused MSB of quantized CNN parameters to store error check bits. To do so, they developed a new training technique to regularize the spatial distribution of large weights and thereby control the available unused space for allocating the ECC bits. A more general approach, *i.e.* not tailored to DNNs, for protecting DRAM applications is the one of Malek et al. [19], which dynamically selects the memory fault tolerance of allocated pages according to the criticality of the respective data.

The closest work to our research is by Segee et al.[27], in which the authors test the fault tolerance of a pruned a single-input-single-output feed-forward neural network.

Thesis Approach: ML acceleration often involves the compression of the DNN parameters. Techniques such as pruning, weight-sharing and quantization reduce the amount of data required by the networks. In this work we are interested in exploring whether such techniques make ML applications and in particular CNNs more sensitive to faults in specific regions of the reduced parameter space.

In order to measure the sensitivity of compressed CNNs, we modified Caffe [12], a popular machine learning framework, to implement Caffe Macchiato, a framework that applies pruning and weight-sharing to the target network

parameters. Caffe Macchiato also exploits Ristretto [9], a forked version of Caffe, to quantize the network parameters to fixed point representation. Caffe Macchiato is also able to inject faults in different parts of the ML applications and collect statistics about the affected accuracy of the networks. We experimented with different fault types, in particular single and multi bit flips, injected in either the weight or bias parameters.

Contributions: in this thesis we propose a detailed methodology for compressing CNNs and testing their reliability to faults. In particular, the contribution of the second part of this thesis are as follows:

- Caffe Macchiato: a framework that provides different compression levels for deep neural networks and that represents a tool for injecting single and multi faults,
- a detailed set of fault injection experiments aiming to explore whether compressing techniques on CNNs, such as pruning, clustering and quantization, increase their sensitivity to faults, and if so, whether compressed CNNs have different sensitivity to faults on different parts of their compressed parameter data.

1.3 Thesis Outline

The remainder of the thesis is organized as follows. In Paper I we present our proposed framework for mapping multiple LSTMs on a FPGA device. Paper II presents a sensitivity analysis conducted on highly compressed CNNs.

References

[1] Wonseok Choi et al. "Sensitivity based error resilient techniques for energy efficient deep neural network accelerators". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6.

[2] Joseph Clements and Yingjie Lao. "Hardware trojan design on neural networks". In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2019, pp. 1–5.

[3] Emily L Denton et al. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in neural information processing systems*. 2014, pp. 1269–1277.

[4] J. C. Ferreira and J. Fonseca. "An FPGA implementation of a long short-term memory neural network". In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016, pp. 1–8.

[5] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448.

[6] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. "Badnets: Identifying vulnerabilities in the machine learning model supply chain". In: *arXiv preprint arXiv:1708.06733* (2017).

[7] Tianyu Gu et al. “Badnets: Evaluating backdooring attacks on deep neural networks”. In: *IEEE Access* 7 (2019), pp. 47230–47244.

[8] Hui Guan et al. “In-place zero-space memory protection for cnn”. In: *arXiv preprint arXiv:1910.14479* (2019).

[9] Philipp Gysel et al. “Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks”. In: *IEEE transactions on neural networks and learning systems* 29.11 (2018), pp. 5784–5789.

[10] Song Han, Huizi Mao, and William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. In: *arXiv preprint arXiv:1510.00149* (2015).

[11] Song Han et al. “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 75–84. ISBN: 9781450343541. DOI: 10.1145/3020078.3021745. URL: https://doi.org/10.1145/3020078.3021745.

[12] Yangqing Jia et al. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678.

[13] Navid Khoshavi, Connor Broyles, and Yu Bi. “Compression or Corruption? A Study on the Effects of Transient Faults on BNN Inference Accelerators”. In: *2020 21st International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2020, pp. 99–104.

[14] Alexandros Kouris et al. “Approximate LSTMs for Time-Constrained Inference: Enabling Fast Reaction in Self-Driving Cars”. In: *ArXiv* abs/1905.00689 (2019).

[15] M. Lee et al. “FPGA-Based Low-Power Speech Recognition with Recurrent Neural Networks”. In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2016, pp. 230–235.

[16] Guanpeng Li et al. “Understanding error propagation in deep learning neural network (DNN) accelerators and applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–12.

[17] Yannan Liu et al. “Fault injection attack on deep neural network”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 131–138.

[18] Yuntao Liu, Yang Xie, and Ankur Srivastava. “Neural trojans”. In: *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 45–48.

[19] Alirad Malek et al. “Odd-ECC: on-demand DRAM error correcting codes”. In: *Proceedings of the International Symposium on Memory Systems*. 2017, pp. 96–111.

[20] Minghai Qin, Chao Sun, and Dejan Vucinic. "Robustness of neural networks against storage media errors". In: *arXiv preprint arXiv:1709.06173* (2017).

[21] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. "Bit-flip attack: Crushing neural network with progressive bit search". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 1211–1220.

[22] Brandon Reagen et al. "Ares: A framework for quantifying the resilience of deep neural networks". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE. 2018, pp. 1–6.

[23] Michalis Rizakis et al. "Approximate FPGA-Based LSTMs Under Computation Time Constraints". In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications - 14th International Symposium, ARC 2018, Santorini, Greece, May 2-4, 2018, Proceedings*. 2018, pp. 3–15. DOI: 10.1007/978-3-319-78890-6_1. URL: https://doi.org/10.1007/978-3-319-78890-6_1.

[24] V. Rybalkin et al. "FINN-L: Library Extensions and Design Trade-Off Analysis for Variable Precision LSTM Networks on FPGAs". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 89–897.

[25] V. Rybalkin et al. "Hardware architecture of Bidirectional Long Short-Term Memory Neural Network for Optical Character Recognition". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 1390–1395.

[26] F. F. d. Santos et al. "Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs". In: *IEEE Transactions on Reliability* 68.2 (2019), pp. 663–677.

[27] Bruce E Segee and Michael J Carter. "Fault tolerance of pruned multilayer networks". In: *IJCNN-91-Seattle International Joint Conference on Neural Networks*. Vol. 2. IEEE. 1991, pp. 447–452.

[28] Yifan Sun et al. "Svdnet for pedestrian retrieval". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 3800–3808.

[29] Jian Xue, Jinyu Li, and Yifan Gong. "Restructuring of deep neural network acoustic models with singular value decomposition." In: *Interspeech*. 2013, pp. 2365–2369.

[30] Jing Ye, Yu Hu, and Xiaowei Li. "Hardware trojan in fpga cnn accelerator". In: *2018 IEEE 27th Asian Test Symposium (ATS)*. IEEE. 2018, pp. 68–73.

[31] Jeff Jun Zhang et al. "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator". In: *2018 IEEE 36th VLSI Test Symposium (VTS)*. IEEE. 2018, pp. 1–6.

[32] Yang Zhao et al. "Memory trojan attack on neural network accelerators". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1415–1420.

Chapter 2

Paper I

S. Ribes, P. Trancoso, I. Sourdis, C.-S. Bouganis,

Mapping Multiple LSTM models on FPGAs,

Int'l Conf. on Field-Programmable Technology (FPT), December, 2020.

Mapping Multiple LSTM models on FPGAs

Abstract

Recurrent Neural Networks (RNNs) and their more recent variant Long Short-Term Memory (LSTM) are utilised in a number of modern applications like Natural Language Processing and human action recognition, where capturing long-term dependencies on sequential and temporal data is required. However, their computational structure imposes a challenge when it comes to their efficient mapping on a computing device due to its memory-bounded nature. As recent approaches aim to capture longer dependencies through the utilisation of Hierarchical and Stacked RNN/LSTM models, *i.e.* models that utilise multiple LSTM models for prediction, meeting the desired application latency becomes even more challenging. This paper addresses the problem of mapping multiple LSTM models to a device by introducing a framework that alters their computational structure opening opportunities for co-optimising the memory requirements to the target architecture. Targeting an FPGA device, the proposed framework achieves 3× to 5× improved performance over state-of-the-art approaches for the same accuracy loss, opening the path for the deployment of high-performance systems for Hierarchical and Stacked LSTM models.

2.1 Introduction and Motivation

The recent advances in Machine Learning, especially Deep Neural Networks (DNN), have reignited the interest of researchers and practitioners on Neural Networks and their variations. With the abundant availability of data and computational capacity provided by modern GPUs, training of large DNNs with good generalisation properties became possible and led to unprecedented performance. Systems that rely on DNNs can efficiently perform a variety of tasks in applications from computer vision, to image understanding, scene analysis[3] and Natural Language Processing (NLP)[19].

In the case where long-term dependency capture is desired on sequential and temporal data, such as in the case of image captioning and NLP, Recurrent Neural Networks (RNN), a form of Neural Network with feedback connections, have demonstrated to be a suitable and efficient solution. However, standard RNNs suffer from vanishing and exploding gradients making their training a challenging task. An RNN variant, the Long-Short Term Memory (LSTM) network[9], addresses the above problem by introducing new structures, leading to their quick adoption in a large number of applications.

In case where the latency or throughput of the developed system is of concern, the mapping of LSTMs to a computing device is a challenging task, due to the low computation to communication ratio and the inherent dependencies in the LSTM operations. An LSTM network is based internally on structures (*i.e.* gates) that resemble networks with fully connected layers, that are manifested through matrix-vector multiplication operations, followed by non-linear functions. Any exploited parallelism is limited by the computational dependencies of the LSTM structure (*i.e.* recurrent connection). Moreover, the above problem is further amplified when multiple LSTMs need to be deployed as part of the application in the form of independent parallel executed LSTMs. Such case is where a number of independent outcomes are required based on the same input data, or in the utilisation of Stacked LSTMs [21], that extend the capabilities of LSTMs to longer time intervals. Example of applications can be found in [32] and [34], where Hierarchical Recurrent Neural Networks for skeleton-based action recognition are proposed, as well as in [29] where the authors propose a framework that utilises a two-stream Recurrent Neural Network pipeline for the task of action recognition. Finally, Li *et. al.* [18] propose a hierarchical LSTM model for building coherent long text for natural language generation and summarization.

Research effort in the efficient mapping of an LSTM to a device has focused only on the case where a single LSTM is required to be executed at any point of time [22, 13]. State-of-the-art approaches aim to increase the computation to communication ratio by reducing the memory accesses and computation cost through the investigation of parameter quantization and compression, as well as by pruning of connections (*i.e.* removing redundant network parameters [12]). Towards the above effort, the existing space can be divided into methods that require a re-training stage, allowing the methodologies to produce highly optimised designs [13], and approaches such as in [22] that assume no availability of data for retraining, focusing more on the generality of the approach.

This paper departs from the previously published approaches by focusing

on the problem of mapping multiple LSTMs in a device, and more specifically in the case where these LSTMs are independent of each other apart except that they are part of the same application. Also, focusing on the generality of the approach, no assumption on the availability of training data is made.

The main contributions of the paper are as follows:

- a methodology is proposed for approximating for the first time multiple LSTMs together, rather than each separately; the methodology allows iterative refinement of the LSTMs approximation leading to tunable and improved computation to communication ratios.
- an approach that exposes the computational and memory capabilities of the targeted device to the approximation algorithm, through structured pruning over the introduced refinement stages, leading to an architecture with improved device utilisation.

To the best of authors' knowledge this is the first work in the literature that addresses the important and timely problem of mapping multiple LSTMs on a device.

2.2 Background

2.2.1 LSTM Networks

An LSTM network processes an input $\mathbf{x}^t$ and produces an output $\mathbf{h}^t$ in every time-step t, where $\mathbf{x}$ and $\mathbf{h}$ denote vectors. Key to the operation of the LSTM is its recurrent connection of its output to its hidden units allowing the network to pass information over a number of time-steps, where regulation of the information flow is controlled through four modules called *gates*. Figure 2.1 illustrates the flow of an LSTM, where the details of the LSTM gates are given in Equation 2.1, where $\mathbf{b}$ and $\odot$ denote a bias vector and the element-wise multiplication operator. The *input* gate, $\mathbf{i}^t$, along with the *cell* gate $\mathbf{c}^t$ determine the amount of input information that propagates to the output of the network, whereas the *forget* gate, $\mathbf{f}^t$, controls the amount of previous information that will be maintained by the network. The *output* gate, $\mathbf{o}^t$, determines how much of the current state will be propagated to the network output.

The above gates are instantiated through non-linear functions, such as sigmoid $\sigma(\cdot)$ or hyperbolic tangent functions $tanh(\cdot)$, that operate on linear functions of the current input $\mathbf{x}^t$ and of the previous time-step output $\mathbf{h}^{t-1}$. Computationally, each gate is based on matrix-vector multiplications, and it is parameterised with a set of weight matrices, $\mathbf{W}_{cur}$ and $\mathbf{W}_{rec}$, responsible to modulate the current input and previous output.

$$
\begin{aligned}
\mathbf{i}^t &= \sigma\big(\mathbf{x}^t \cdot \mathbf{W}_{cur_i} + \mathbf{h}^{t-1} \cdot \mathbf{W}_{rec_i} + \mathbf{b}_i\big) \\
\mathbf{f}^t &= \sigma\big(\mathbf{x}^t \cdot \mathbf{W}_{cur_f} + \mathbf{h}^{t-1} \cdot \mathbf{W}_{rec_f} + \mathbf{b}_f\big) \\
\mathbf{c}^t &= \mathbf{f}^t \odot \mathbf{c}^{t-1} + \mathbf{i}^t \odot \tanh\big(\mathbf{x}^t \cdot \mathbf{W}_{cur_c} + \mathbf{h}^{t-1} \cdot \mathbf{W}_{rec_c} + \mathbf{b}_c\big) \\
\mathbf{o}^t &= \sigma\big(\mathbf{x}^t \cdot \mathbf{W}_{cur_o} + \mathbf{h}^{t-1} \cdot \mathbf{W}_{rec_o} + \mathbf{b}_o\big) \\
\mathbf{h}^t &= \mathbf{o}^t \odot \tanh(\mathbf{c}^t)
\end{aligned}
\tag{2.1}
$$

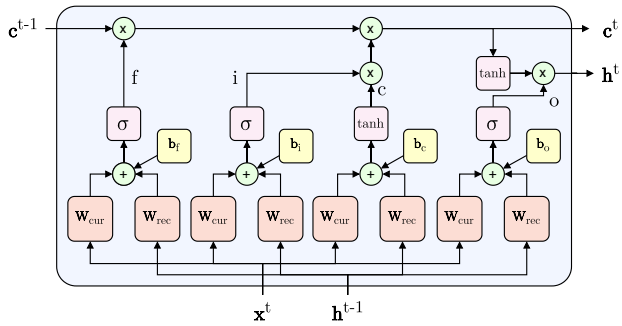

Figure 2.1: LSTM flow for processing output $\mathbf{h}^t$ and cell state $\mathbf{c}^t$ at timestep t.

2.2.2 SVD-Based Approximation

Typical DNNs, including LSTMs, utilise matrix-vector multiplication operations leading to designs whose performance is memory-bounded as a large number of parameters (*i.e.* weights matrix) needs to be accessed for the computation over a single input vector. Techniques to address this problem rely on batching multiple input vectors, sharing the weights access across multiple inputs, and/or pruning/approximating the weight matrices, reducing as such the data that need to be accessed per input. In the case of LSTMs, the former technique cannot be applied due to their recurrent connections, and effort is placed on the latter approach in order to improve the computation over communication ratio.

Possible techniques to prune/approximate the weights matrix include weights quantization, pruning of certain weights[13], as well as approximations of the weights matrix through rank-1 decomposition [16].

Decomposition of a matrix through rank-1 approximations expresses a matrix $\mathbf{W}$ as a linear combination of rank-1 matrices. The decomposition is achieved through the Singular Value Decomposition (SVD) algorithm that decomposes a given matrix $\mathbf{W}$ into 3 orthogonal matrices $\mathbf{U}$, $\mathbf{S}$, $\mathbf{V}$ as $\mathbf{W} = \mathbf{USV}^T$. The original matrix $\mathbf{W}$ can be approximated by selecting to utilise the first R rank-1 matrices of the decomposition (*i.e.* the ones that correspond to the largest eigenvalues), where the SVD algorithm guarantees the optimality of the approximation under the Mean Square Error (MSE) metric. As such, the matrix $\mathbf{W}$ can be approximated as:

$$\mathbf{W} \approx \sum_{i}^{R} s_i \ \mathbf{u}_i \ \mathbf{v}_i^T$$

where $\mathbf{u}_i$ and $\mathbf{v}_i$ correspond to the ith column and row of the $\mathbf{U}$ and $\mathbf{V}^T$ matrices respectively, while s_i is the ith element of the main diagonal of the diagonal matrix $\mathbf{S}$. The approximation leads to a reduction on the amount of data that need to be accessed as well as allows the matrix-vector multiplication computation to be performed through a series of dot-product calculations, as it will be shown later.

2.3 Problem Formulation

The work considers the general problem of accelerating the execution of multiple LSTM models that operate in parallel on synchronised inputs, and the device of choice is an FPGA. The parameters of the models are assumed to be stored in the off-chip memory, increasing the applicability of the approach to large problem sizes. The problem is formulated as follows: given a set of N LSTM models M_i with weight matrices $\mathbf{W}^i_{type}$, with $type \in \{cur_{gate}, rec_{gate}\}$ for $gate \in \{i, f, o, c\}$, and a target FPGA device D, derive an implementation that minimises the latency of their execution. More specifically, the work focuses on the case of a lossy mapping, where an error in the approximation on the final results of the computation is allowed but bounded by a user-specified threshold.

The proposed approach builds upon the work of Rizakis *et al.* [22], but it extends their problem formulation to address the case of multiple LSTMs. The key idea is to provide a decomposition of the weight matrices of the LSTMs in order to facilitate the necessary computations as a trade-off of latency and quality of the final result, along side with providing computational structures that would fully exploit the compute and memory capabilities of the targeted device.

Towards this, the proposed approach is based on the Singular Value Decomposition algorithm applied to a set of input matrices $\mathbf{W}_1, ..., \mathbf{W}_N$, producing a set of rank-1 matrices (*i.e.* matrices that can be expressed as the product of two vectors $\mathbf{u}^{(i)}$, $\mathbf{v}^{(i)}$) whose linear combination constructs the original input matrices. Such decomposition guarantees the least error in the approximation of the input matrices under the Mean Square Error (MSE) metric for a given number of rank-1 matrices used in the approximation [2].

As such, focusing on our problem formulation, the proposed approach aims to produce a *single set* of rank-1 matrices that approximates *all* the weight matrices of the same type $\mathbf{W}^i_{type}$ across the N LSTM targeted models. Thus, our approach allows us to *share* the $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ components across the N LSTM models M_j, and in doing so reduces the memory footprint of the models for a given targeted approximation error. Equation 2.2 indicates the approximation of a single matrix with R rank-1 matrices, where the *type* and *gate* indices have been dropped for clarity.

$$\mathbf{W}_{\mathrm{M}_j} \approx \sum_{i=1}^{R} {s_j}^{(i)} \odot \left(\mathbf{u}^{(i)} \cdot {\mathbf{v}^{(i)}}^T\right),\ j = 1, ..., N \tag{2.2}$$

Algorithm 1 lists the necessary steps for decomposing N given weight matrices $\mathbf{W}_1, ..., \mathbf{W}_N$ into the R components $\mathbf{u}^{(i)}$, ${s_j}^{(i)}$ and $\mathbf{v}^{(i)}$. The algorithm also sparsifies and quantizes such components to improve the mapping to the device.

The algorithm begins by initializing a set of error matrices $\mathbf{E}_1, ..., \mathbf{E}_N$ and one set of approximated weight matrixes $\widetilde{\mathbf{W}}_1, ..., \widetilde{\mathbf{W}}_N$. After initialization, for each of the refinement steps R, the algorithm first updates the error matrices by taking the difference between the original matrices and the partially reconstructed ones, *i.e.* approximated (line 5). Upon constructing the new error matrices, at line 6 we apply the decomposition described in [2] to obtain

ALGORITHM 1: Decomposition algorithm.

Data: $N \times \mathbf{W}$ weight matrices, R number of refinement steps, T_u and T_v number of tiles, ZT_u and ZT_v number of tiles to prune.

1 **begin**
2 $\quad \mathbf{E}_i \longleftarrow \mathbf{0}, \quad i = 1, ..., N$
3 $\quad \widetilde{\mathbf{W}}_i \longleftarrow \mathbf{0}, \quad i = 1, ..., N$
4 $\quad$ **for** i ***in*** R **do**
5 $\quad\quad \mathbf{E}_j \longleftarrow \mathbf{W}_j - \widetilde{\mathbf{W}}_j, \quad j = 1, ..., N$
6 $\quad\quad \mathbf{u}^{(i)}, s^{(i)}, \mathbf{v}^{(i)} \longleftarrow \texttt{decompose}(\mathbf{E})$
7 $\quad\quad$ **for** j **in** ZT_u **do**
8 $\quad\quad\quad zu^{(i)}[j] \longleftarrow \arg\min_k \{|\text{mean}(\mathbf{u}^{(i)}[k])|\}$
9 $\quad\quad\quad \mathbf{u}^{(i)}[zu^{(i)}[j]] \longleftarrow \mathbf{0}$ `// pruning`
10 $\quad\quad$ **end**
11 $\quad\quad$ **for** j **in** ZT_v **do**
12 $\quad\quad\quad zv^{(i)}[j] \longleftarrow \arg\min_k \{|\text{mean}(\mathbf{v}^{(i)}[k])|\}$
13 $\quad\quad\quad \mathbf{v}^{(i)}[zv^{(i)}[j]] \longleftarrow \mathbf{0}$ `// pruning`
14 $\quad\quad$ **end**
15 $\quad\quad \mathbf{A} \longleftarrow \mathbb{Q}(\mathbf{u}^{(i)}) \cdot \mathbb{Q}(\mathbf{v}^{(i)^T})$
16 $\quad\quad \widetilde{\mathbf{W}}_j \longleftarrow \mathbb{Q}(\widetilde{\mathbf{W}}_j) + \mathbb{Q}(s_j^{(i)}) \odot \mathbf{A}, \; j = 1, ..., N$
17 $\quad$ **end**
18 **end**

the $\mathbf{u}^{(i)}$, ${s_j}^{(i)}$ and $\mathbf{v}^{(i)}$ components. This decomposition aims to minimize the MSE of the approximated matrices reconstructed from the $\mathbf{u}^{(i)}$, ${s_j}^{(i)}$ and $\mathbf{v}^{(i)}$ elements.

It has been shown in the literature that neural networks are able to maintain their accuracy after the sparsification of their weight matrices, *i.e.* setting most of their weight values to zero, thanks to a process called pruning [12]. A standard de-facto way of pruning a network consists of an iterative process where a first step applies zero masks to the network matrices, followed by a fine-tuning step, *i.e.* retraining process. In this work, we propose a structured pruning of the $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ vectors that does *not* require retraining the network. Please note that ${s_j}^{(i)}$ are scalars and therefore are not pruned. In order to prune, we first divide the vectors $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ into T_u and T_v tiles respectively. Afterwards, we select the ZT_u tiles from vector $\mathbf{u}^{(i)}$ and the ZT_v tiles from $\mathbf{v}^{(i)}$ that contain the values with the minimum absolute magnitude, lines 8 and 12. Finally, all the elements of the selected tiles are assigned to zero. Pruning reduces both the amount of operations needed and the number of tiles, *i.e.* weight values, to be accessed.

Furthermore, a quantization operation of the pruned vectors $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ and the scalars ${s_j}^{(i)}$ (indicated with the $\mathbb{Q}(\cdot)$ operator) is performed, before the algorithm moves to the next refinement step. The quantized vectors are multiplied together to form a shared matrix $\mathbf{A}$ (line 15). The matrix $\mathbf{A}$ is then multiplied by the quantized scalars ${s_j}^{(i)}$ and added to the partial approximation matrix $\widetilde{\mathbf{W}}_j$.

At the next iteration, the approximation matrices $\widetilde{\mathbf{W}}_1, ..., \widetilde{\mathbf{W}}_N$ will include the errors introduced by both the pruning and quatization processes. The decomposition step will therefore generate components $\mathbf{u}^{(i)}$, $s^{(i)}$ and $\mathbf{v}^{(i)}$ that

account for such errors, minimizing the overall MSE.

2.4 Accelerator Design

In order to accelerate the execution of N LSTM layers, we applied our approximation algorithm to the weight matrices $\mathbf{W}_{type}$, with $type \in \{cur_{gate}, rec_{gate}\}$ and $gate \in \{i, f, c, o\}$. In particular, we approximated the weight matrices of the same type and gate together, because we empirically found them to have similar structures. For example, we made all the current forget gates ($\mathbf{W}_f^{cur}$) of the N LSTMs share the same $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ vectors (we will refer to this operation as *merging*). Merging same type of gate matrices overall yields lower MSE compared to stacking the gate weight matrices together and then approximating them. We believe that the reasons for this are twofold: first, the size of the approximated matrices is smaller, therefore the MSE can decrease quickly with fewer refinement steps R. Second, the gates across different LSTM models perform a similar function and so the information filtered out by our algorithm tends to be the same, thus improving the approximation MSE. Nevertheless, the above behaviour is application dependent and other constructions should be considered.

Our FPGA accelerator's key computation is the approximated vector-matrix multiplication of the N LSTM inputs with the gate weight matrices, as exemplified in Equation 2.3, which approximates the multiplication between the input vectors $\mathbf{x}_j^t$ with the current forget gates weight matrices $\mathbf{W}_f^{cur}$.

$$\mathbf{x}_j^t \cdot \mathbf{W}_{f_j}^{cur} \approx \sum_{i=1}^{R} \left(\mathbf{x}_j^t \cdot \mathbf{u}_f^{(i)}\right) \cdot s_{f_j}^{(i)}\right) \odot \mathbf{v}_f^{(i)},\ j = 1, ..., N \tag{2.3}$$

The matrix-vector multiplication is effectively decomposed in three parts: a dot product ($\mathbf{x}_j^t \cdot \mathbf{u}^{(i)}$), a scalar-scalar multiplication ($\cdot s_{f_j}^{(i)}$) and finally a scalar-vector multiplication ($\odot \mathbf{v}_f^{(i)}$). Equation 2.3 is applied to both the current and the recurrent gates of the LSTMs, just by using, for the recurrent gates, the previous output $\mathbf{h}_j^{t-1}$ and the properly sized vector components. Notice that all the elements of the equation are quantized and that the vectors $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ are also pruned. A visual example of the R dot products of Equation 2.3 is depicted in Figure 2.2. In this example all the $\mathbf{u}^{(i)}$ vectors contain exactly two non-pruned tiles and two pruned tiles. The pruned tiles of the $\mathbf{u}^{(i)}$ vectors are dashed. Only the non-pruned tiles participate to the final computation, thereby saving time and resources required to perform the product.

The accelerator is designed in a dataflow fashion, illustrated in Figure 2.3. Inputs and weights are stored in the DRAM external memory and fed to the FPGA accelerator through the four available high performance AXI ports which are directly connected to the memory controller. Each AXI port is connected to a Direct Memory Access (DMA) unit that feeds the processing kernels with the respective data. The accelerator is composed of the following main building blocks:

a) *SVD-Kernels*: they are responsible of the execution of the approximated matrix-vector operation of the LSTM gates, as reported in Equation 2.3.

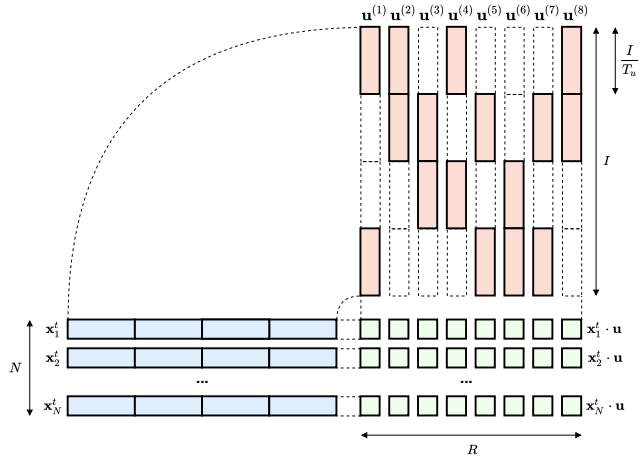

Figure 2.2: Reduce products $\mathbf{x}_j^t \cdot \mathbf{u}^{(i)}$ with $R = 8$, $T_u = 4$ and $ZT_u = 2$.

There are a total of 8 kernels, 4 for the current matrices of the LSTM gates and 4 for the recurrent ones.

b) *Input DMAs* and *tiles dispatchers*: they are in charge of transferring the inputs of the two LSTMs from main memory to the correct engines. In addition, they offer temporary on-chip buffers to store the N LSTMs' inputs $\mathbf{x}_j^t$ and $\mathbf{h}_j^{t-1}$ maximizing data reuse. Only the tiles corresponding to the non-pruned u-vector tiles are then read from the buffers and broadcasted into the MAC units of the SVD-kernels.

c) *u, s, v DMAs*: these DMA units fetch the non-zeroed tiles of the $\mathbf{u}^{(i)}$, $\mathbf{v}^{(i)}$, ${s_j}^{(i)}$ weight vectors to be streamed into the SVD-kernels.

d) *σ-Kernels*: their task is to apply the gate biases and the required non linear operations, listed in Equation 2.1, to the product of the inputs with the approximated weight matrices. There are N σ-kernels, one for each LSTM.

e) *σ DMAs*: these DMAs supply the data to the σ-kernels, *i.e.* the bias vectors and the previous LSTMs cell states. They also are used to write back the final computation to main memory.

The block diagram of the input DMA, tiles dispatchers and SVD-kernel is shown in Figure 2.4. The SVD-kernel computes Equation 2.3 and is composed of two types of units: U-unit and V-unit. Within the kernel, there are N U-units and N V-units. The U-units are responsible for computing the dot product reported in Equation 2.4.

$$\begin{aligned} {x_u}_j^{(i)} &= \mathbf{x}_j^t[nzu_k^{(i)}] \cdot \mathbf{u}^{(i)}[nzu_k^{(i)}], \\ j = 1, ..., N; \quad k &= 1, ..., T_u - ZT_u \end{aligned} \tag{2.4}$$

Each U-unit includes $T_u - ZT_u$ parallel multiply-accumulate blocks and an adder tree. In order for the U-units to perform their computation, the N input

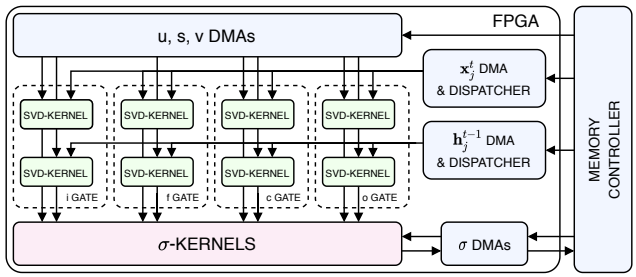

Figure 2.3: The proposed dataflow accelerator architecture. The approximated current and recurrent LSTMs gates are processed in parallel by eight SVD-kernels. The σ-kernels compute the final steps of the LSTMs algorithm. The DMAs stream the required inputs and weights from memory to the kernels.

tiles dispatcher supply the non-pruned input tiles, while the $\mathbf{u}^{(i)}$ tile dispatcher broadcasts the non-pruned tiles. Thanks to the list of indexes nzu the N input tiles dispatchers read the input tiles corresponding to the non-pruned tiles of $\mathbf{u}^{(i)}$ and then stream them from their on-chip buffers to the respective MACs within the corresponding U-unit (recall Figure 2.2).

The $N \times R$ scalars $x_{u_j}^{(i)}$ produced by the U-units are then multiplied by the $s_1^{(i)}, ..., s_j^{(i)}$ scalar components and forwarded to the kernel's V-units as $x_{s_j}^{(i)}$. The V-units perform the operations in Equation 2.5, *i.e.* the last step of the approximation process.

$$\begin{aligned} \mathbf{x}_j^t \cdot \widetilde{\mathbf{W}}_j &\approx \sum_{i=1}^{R} x_{s_j}^{(i)} \odot \mathbf{v}^{(i)}[nzv_k^{(i)}] \\ j = 1, ..., N; \quad k &= 1, ..., T_v - ZT_v \end{aligned} \tag{2.5}$$

Like for the U-units, there is a weight dispatcher which is in charge of supplying the V-unit's MACs with the non-pruned $\mathbf{v}^{(i)}$ vector tiles. In order to multiply and accumulate the $x_{s_j}^{(i)}$ scalars with the non-pruned $\mathbf{v}^{(i)}$ weight elements, each V-unit utilizes a partitioned accumulation buffer. The buffer is partitioned tile-wise to allow parallel access to it from the MACs. Once the refinement steps are completed, the V-units stream out the final approximated products $\mathbf{x}_j^t \cdot \tilde{\mathbf{W}}_j$ from their accumulation buffers.

Finally, the results of the SVD-kernels are streamed to the σ-kernels for applying the last non-linear functions required by the LSTMs.

2.5 Proposed Framework

In this section we describe a framework for identifying the combination of design parameters which best tradeoff the accuracy and execution time of accelerating N LSTM models. The initial part of the section describes our methodology, while the next and final one details the roofline model we use to estimate the performance of our accelerator during the design space exploration phase.

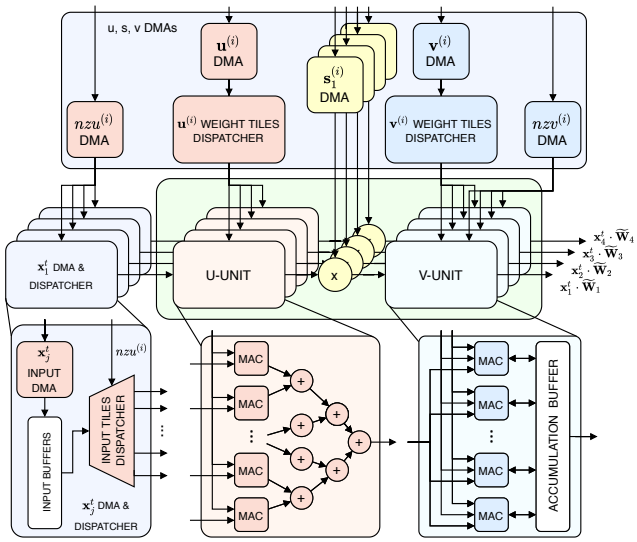

Figure 2.4: Dataflow architecture of one of the eight SVD-kernels for processing four LSTMs. The kernel is composed of two types of sub-blocks, the U-Unit and the V-Unit. There is one set of U-Unit and V-Unit per input. All U-Units (V-Units) are fed by the *same* $\mathbf{u}^{(i)}$ DMA and $\mathbf{u}^{(i)}$ weight tiles dispatcher ($\mathbf{v}^{(i)}$ DMA and $\mathbf{v}^{(i)}$ weight tiles dispatcher), since the $\mathbf{u}^{(i)}$ and $\mathbf{v}^{(i)}$ vectors are *shared* across the different LSTMs.

2.5.1 Methodology

There is a large number of design parameters, *i.e.* number of refinement steps, tile size, pruning percentage and quantization (detailed in Table 2.1), each having a large range of possible options, which make the design space huge and impractical to search exhaustively. We have defined the following methodology to select one, or at most a few, design points, which are promising for achieving a good performance-accuracy trade-off and fit in the target FPGA device.

First, we set particular performance and accuracy goals, as well as the resource constraints for our target design. Designs with accuracy below a certain threshold or excessive need for resources are not further considered. However, measuring actual accuracy requires heavy application-level simulation of the particular design point. Similarly, measuring actual resource requirements and performance requires a design implementation, which is time consuming. Our experiments indicate that the most critical and limited device resources are the DSP slices and BRAMs, for which analytical models have been developed. The proposed approach adopts analytical models that provide indications for accuracy, need for critical resources, as well as for performance for each design point. Based on these models we select the most promising design points for further evaluation and eventually implementation.

Second, we search the design space based on the accuracy criterion. We get an indication of the accuracy drop compared to the original application (before SVD approximation, pruning and quantization, etc.) using the average Mean Square Error (MSE) between the original stacked weight matrices $\mathbf{W}$

Table 2.1: List of design parameters of the framework.

Symbol	Description
R	Amount of refinement steps.
T_u	Number of tiles of the $\mathbf{u}^{(i)}$ vectors.
ZT_u	Number of pruned tiles of the $\mathbf{u}^{(i)}$ vectors.
T_v	Number of tiles of the $\mathbf{v}^{(i)}$ vectors.
ZT_v	Number of pruned tiles of the $\mathbf{v}^{(i)}$ vectors.
B	Byte size of the quantized LSTM's input and weight values.

and the SVD approximated ones $\widetilde{\mathbf{W}}$, defined in Equation 2.6. Subtraction and square operations are performed element-wise.

$$\mathrm{MSE}(\mathbf{W}, \widetilde{\mathbf{W}}) = \mathrm{mean}\big((\mathbf{W} - \widetilde{\mathbf{W}})^2\big) \tag{2.6}$$

Our conjecture, confirmed in the next section, is that a small MSE is a necessary condition for low accuracy drop. Consequently, design points with MSE below a MSE threshold (T_{MSE}) are selected for further evaluation. These design points are subsequently selected for simulation in order to measure their actual accuracy drop. Out of those, the design points with actual accuracy drop below our accuracy threshold (T_{acc}) are selected to continue in the next step of our design space exploration process.

Third, the design points that passed the accuracy check are then evaluated for their resource requirements and performance. In order to avoid generating a hardware implementation for all of them, the need for DSP slices and BRAMs is estimated. Designs that need more critical resources than available on the device are dropped. Subsequently, the attainable performance based on our roofline model (described in the following subsection) is used to estimate their execution time as in Equation 2.7.

$$t_{exe} = \frac{Nops}{Att_{perf}} \left[\frac{Ops}{Ops/s} \right] \tag{2.7}$$

The designs with the lowest attainable execution time are finally implemented in the FPGA board at hand and their actual performance is measured.

2.5.2 Roofline Model

For estimating the execution time of our FPGA accelerator we derive a roofline model for calculating the attainable performance of the possible designs [30, 33]. The attainable performance is defined in Equation 2.8 as the minimum value between the Computational Performance (CP) and the product between the maximum available bandwidth of the system B_w and the Communication To Computation ratio (CTC).

$$Att_{perf} = \min\big\{CP,\ CTC \cdot B_w\big\} \left[\frac{Ops}{s} \right] \tag{2.8}$$

The CP can be estimated as in Equation 2.9, where N_{ops} and N_{cycles} are the total number of performed fixed point operations and the estimated amount of execution cycles, respectively.

$$CP = \frac{Nops}{Ncycles \cdot \frac{1}{f_{clk}}} \left[\frac{Ops}{s}\right] \tag{2.9}$$

For our accelerator, the amount of required operations is reported in Equation 2.10. In an LSTM there are four gates, each including a pair of current and recurrent matrices, giving 8 matrices in total, four of which having dimension $I \times H$ and four $H \times H$. The U-units and V-units perform a series of MAC operations, so 1 MAC corresponds to two operations. The amount of non-linear operations on each hidden value is estimated to be equal to 24, leading to $Nops_\sigma$ amount of operations for the σ-kernel.

$$\begin{aligned} Nops &= N \cdot (Nops_u + Nops_s + Nops_v + Nops_\sigma) \\ &= N \cdot (Nops_u + R \cdot 8 + Nops_v + 24 \cdot H) \\ Nops_u &= R \cdot \left(4 \cdot (T_u - ZT_u) \cdot \left(\frac{I}{T_u} + \frac{H}{T_u}\right)\right) \cdot 2 \\ Nops_v &= R \cdot 8 \cdot (T_v - ZT_v) \cdot \frac{H}{T_v} \cdot 2 \end{aligned} \tag{2.10}$$

In order to finally compute the CP value, we need to estimate the required execution cycles, *i.e.* the accelerator's latency. The accelerator's latency is reported in Equation 2.11. Since the accelerator is designed in a dataflow fashion, we only consider the slowest accelerator's module and therefore the overall latency will be the maximum latency value among the hardware modules. Please notice that each LSTM weight matrix is mapped to a different SVD-kernel, so there are 8 SVD-kernels in total running in parallel. The N inputs, corresponding to N LSTM models, are also processed in parallel within each SVD-kernel.

$$\begin{aligned} Ncycles &= \max\left\{U_{latency}, S_{latency}, V_{latency}, \sigma_{latency}\right\} \\ &= \max\left\{U_{latency},\ R,\ R\ (T_v - ZT_v),\ 7\ \frac{H}{T_v}\right\} \\ U_{latency} &= R\ \max\left\{\frac{I}{T_u},\ \frac{H}{T_u},\ log_2(T_u - ZT_u)\right\} \end{aligned} \tag{2.11}$$

The last value we need for calculating the attainable performance is the CTC, which is reported in Equation 2.12. The CTC is defined as the ratio between the total number of operations $Nops$ in Equation 2.10 and the total amount of transferred data (in Bytes), reported in Equation 2.13.

$$CTC = \frac{Nops}{in + out + w + nz + bias} \left[\frac{Ops}{Byte}\right] \tag{2.12}$$

$$
\begin{aligned}
in + out &= N \cdot \Big((I + H) + 2\ H \Big) \cdot B \\
nz + bias &= R \cdot 8 \cdot (T_u + T_v)/8 + 4 \cdot H \cdot N \cdot B \\
w &= u_{size} + s_{size} + v_{size} \\
u_{size} &= R \cdot 4 \cdot (T_u - ZT_u) \cdot \Big(\frac{I}{T_u} + \frac{H}{T_u} \Big) \cdot B \\
s_{size} &= R \cdot 8 \cdot N \cdot B \\
v_{size} &= R \cdot 8 \cdot (T_v - ZT_v) \cdot H/T_v \cdot B
\end{aligned}
\tag{2.13}
$$

The values that need to be read and written are divided in several groups. The *in* and *out* values comprise the input and output vectors for the N LSTM models. The weights that the accelerator requires are the *bias* values, the non-zero indexes nz (which are bit vectors of size proportional to the amount of tiles T_u and T_v) and finally the approximated weight values w. The value of w includes the u, s and v components, which sizes are referred to as u_{size}, s_{size} and v_{size}.

2.6 Evaluation

In this section we describe the experimental setup and present a validation of the models described in the previous section followed by the evaluation of the proposed design in terms of performance and obtained accuracy.

2.6.1 Experimental setup

For the evaluation of the proposed framework, a multiple LSTM model that is trained for the Fashion MNIST dataset [31] is utilised. The Fashion MNIST dataset is a drop-in substitute for the MNIST dataset, but the classification task is considered more challenging [6, 1]. The targeted network model consists of two main branches, each containing an LSTM model [14]. For performance results, the software runs on the Processing System (PS) of the FPGA, which features four Cortex-A53 MPCore processors, ARMv8 architecture, running at 1.2GHz. The accuracy was tested by plugging in our HLS implementation to the Keras execution flow. The neural network was modeled in Keras 2.2.4 using Tensorflow 1.13.1 as a back-end.

For the evaluation of the proposed hardware architecture (denoted as *SVDn-HW*), we used a Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA. In order to generate the description of the hardware module from its high-level representation, we used Xilinx SDSoC 2018.3 tool.

We compared our proposed system against two software and two hardware implementations:

- *LSTM-SW*: Software implementation of baseline LSTM models using GEMV function from OpenBLAS library. Float32 values are used for both activations and weights.
- *LSTM-HW*: Hardware (FPGA) implementation of baseline LSTM models comprised of 8 parallel 1D systolic arrays for the dense matrix-vector computation (loosely inspired by [8]), followed by a non-linear unit.

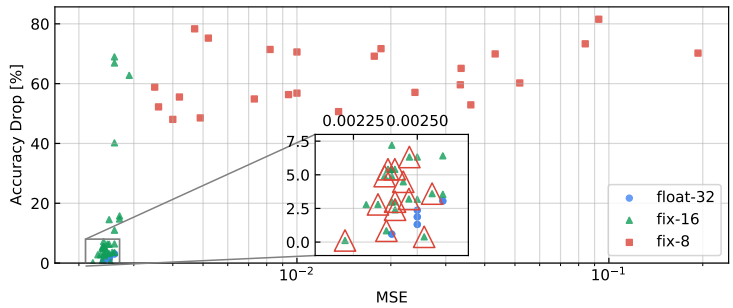

Figure 2.5: Correlation between accuracy drop and MSE of the approximation.

- *SVDn-SW*: Software implementation of the SVD optimization of the LSTM models that utilizes the same weight values of *SVDn-HW* before quantization. *SVDn-SW* performs computations on dense weight matrices, despite having many zero values since the OpenBLAS library does not support sparse computation.

- *SVD1-HW*: A hardware (FPGA) implementation following the design methodology described in [22], where the mapping of each LSTM model is optimised in isolation.

2.6.2 Validation of accuracy, performance and resource models

Next, we present a brief validation of the accuracy, performance and resource models presented in Section 2.5.

Regarding the validation of the accuracy model, in Figure 2.5 we show different design points for the proposed architecture characterized by the (average) MSE of its approximated LSTM weight matrices and the accuracy drop of the result, when compared to the correct output. Note that in this Figure we include design points for 32-bit floating-point as well as 8- and 16-bit fixed point implementations.

In general, it is possible to observe that the lower accuracy drop occurs for the lower values of MSE. Nevertheless, there are design points where a low MSE results in high accuracy drop. Consequently, choosing a design point with low MSE is a necessary but not sufficient condition for achieving a low accuracy drop of the result. An in-depth view is shown with the expansion of the bottom left corner, where it is possible to observe a correlation between the MSE and the accuracy drop. The values in that region are though very small, with very small differences between themselves. The red triangles in the expanded section show the design points that we have selected to explore in more detail.

Figure 2.6 shows the roofline model of our accelerator processing two LSTM layers. The design points in the roofline have different parametrizations of the elements reported in Table 2.1. We can notice that most of the design points hit the bandwidth limit, meaning that the computation is mainly memory-bound. The points highlighted by red triangles are the ones selected in Figure 2.5

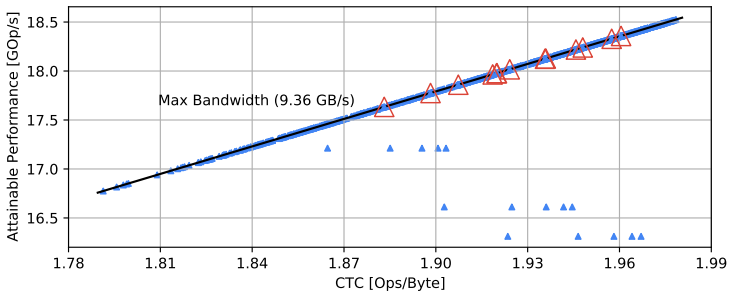

Figure 2.6: Roofline model for our FPGA accelerator processing two LSTM layers with $I = 1024$ and $H = 512$.

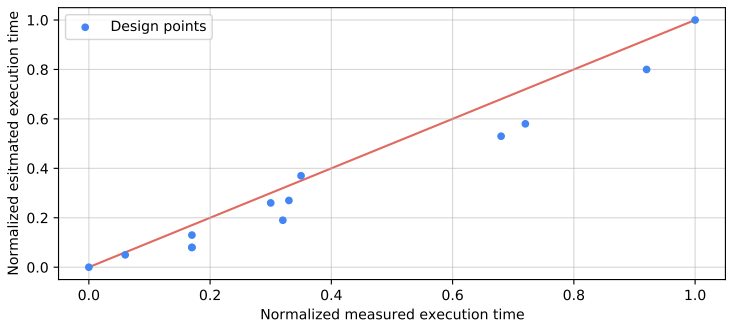

Figure 2.7: Normalized observed execution time versus estimated execution time.

based on accuracy. Based on the attainable performance of the roofline model, the validation of the execution time estimation model (in equation 2.7) is depicted in Figure 2.7. The estimated and measured values of execution time are normalized to each one's corresponding largest value. From this Figure it is possible to observe a high correlation between the estimated and the measured execution time, thus allowing us to use the model as a way to predict which designs achieve higher performance.

Lastly we validated the model for hardware resources. The DSP utilization estimate perfectly matches the count reported in place and route. The estimated BRAM usage shows a 1% relative error on average when compared with HLS reports and 18% versus post place and route results. We believe that the high error compared to post place and route BRAM utilization is because the Xilinx SDSoC 2018.3 tool introduces (when available) additional BRAMs for optimizations, which are hard to foresee and accurately estimate.

2.6.3 Evaluation of the proposed design

Next, the evaluation of the proposed design is presented in terms of execution time and accuracy drop of the output result and compare it to the alternative designs. The results are shown in Figure 2.8. The first observation is that, as expected, the baseline implementations without approximation (*LSTM-SW* and *LSTM-HW*) are the only ones achieving a 0% accuracy drop. Nevertheless,

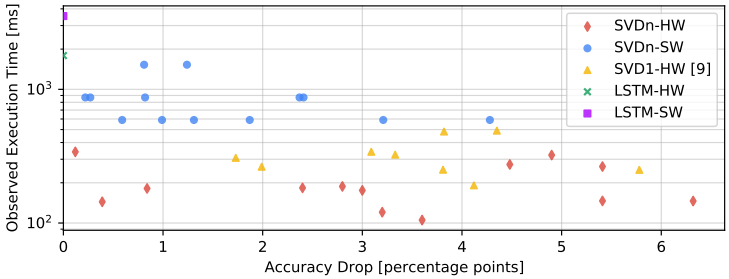

Figure 2.8: Actual exec. time vs. accuracy drop. Orig. network accuracy is 84.4%.

this is achieved at a high latency, higher than any other design presented. Another expected observation is the fact that all *SVDn-SW* points have a higher latency than the corresponding *SVDn-HW* points. The difference observed ranges between a factor of 3.1× and 5.6×. Another interesting comparison is between the proposed *SVDn-HW* and the previously proposed *SVD1-HW*. In particular, it can be observed that the fastest *SVDn-HW* design is 1.7× faster than the fastest *SVD1-HW*, considering all plotted points have acceptable accuracy. The most accurate *SVDn-HW* design has 14x lower accuracy drop than the most accurate *SVD1-HW*, considering all plotted points have acceptable performance. This is explained by the fact that *SVD1-HW* applies a similar SVD-based methodology as our approach but does not exploit possible redundancies between weight matrices across LSTM models. As there is a trade-off between accuracy drop and performance, the best *SVDn-HW* design in the pareto-front is 2× faster and 4.5× more accurate than the best *SVD1-HW*.

2.7 Related Work

Significant research effort has been focused on the efficient mapping of computationally heavy Convolutional Neural Networks on devices, leading to a number of automated toolchains [27][25, 26] and compression methods [28, 15, 12]. In contrast to the CNN mapping, mapping of Recurrent Neural Networks and their variants (LSTMs) pose different challenges as the systems are memory-bounded.

As such, previous research aiming to address the memory-bound limitation of accelerating the execution of given LSTM models have focused in the reduction of either the data volume transferred between the off-chip memory and accelerator, or the amount of data that needs to be stored, thus enabling their complete storage on device. Early representative works in this area are [5, 11]. Common investigated techniques include parameter pruning, parameter sharing, and compression using lossy and lossless schemes. In parallel, effort has been put on the design of accelerator architectures that support the sparsity of the data and the computational patterns introduced by those compression methods[12, 13].

In the case where the LSTM optimisation can be considered during the

training stage, research effort has focused on the extreme quantization of the parameters even to binary values[23]. However, the underlying assumption of availability of training data prohibits the application of those approaches in a large number of cases. Thus, effort has been placed on approaches that can be applied post-training. ESE [13] propose a load-balancing aware compression methodology, along-side an FPGA-specific architecture for speech processing. The compression scheme is based on parameter pruning and quantization, where their proposed architecture can operate directly with irregular patterns. To further address load balancing challenges stemmed from sparse parameter matrices, [20] and [4] propose novel sparse matrix formats, which allow improved load balancing capabilities across the processing elements. Nevertheless, even though the above methods do not require a training step, access to the training data is required for the pruning and the fine-tuning of the weights in order to achieve minimum penalty on the accuracy. Significant performance gains have been reported for custom hardware-based solutions in the case where the on-chip device memory can accommodate the parameters of the LSTM model, removing as such the requirement of accessing off-chip memory [7, 17, 24, 23]. However, such assumption severely restricts the application of these approaches and only few works [5, 11, 10, 13] address the general problem where the parameters of the compressed LSTM model do not fit in the on-chip memory, as is the case of the work presented in this paper.

Closer to this work are the works by Kouris *et. al.* [16] and Rizakis *et. al.* [22], that propose an SVD-based refining scheme for the approximation of the LSTM weight matrices.

The proposed work considers the more complex problem of mapping on a computing device multiple LSTM models that operate on synchronised inputs. The work focuses on exploiting any redundancies within and across the parameters of the models in order to produce a mapping that co-optimised the execution of all models. Previous pruning-based approaches can be used to further extend the impact of the proposed work through their application on each refinement stage, leading towards sparse computations, rather than aiming for a structured sparsity.

2.8 Conclusions

The paper presented a framework for the efficient mapping of multiple LSTMs on an FPGA device. By altering the structure of the computations it allows the co-optimisation of the scheduling of such computations and the underlying hardware parameters, while taking into account the resource constraints of the targeted device. The presented methodology offers the first compression scheme across multiple LSTM models. It offers better accuracy and performance compared to handling each LSTM separately and can be integrated with other existing lossy and lossless compression approaches. Even though a structured pruning approach is investigated in this work, the framework can be extended to allow a hybrid approach where each tile can be expressed through a sparse structure, allowing as such a finer design space exploration of the performance and computation to communication ratio.

References

[1] *Basic classification: Classify images of clothing*. `https://www.tensorflow.org/tutorials/keras/classification`. Accessed: 2020-11-13.

[2] Christos-S Bouganis et al. "Synthesis and optimization of 2D filter designs for heterogeneous FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 1.4 (2009), pp. 1–28.

[3] W. Byeon et al. "Scene labeling with LSTM recurrent neural networks". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 3547–3555.

[4] Shijie Cao et al. "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 63–72. ISBN: 9781450361378. DOI: 10.1145/3289602.3293898. URL: https://doi.org/10.1145/3289602.3293898.

[5] A. X. M. Chang and E. Culurciello. "Hardware accelerators for recurrent neural networks on FPGA". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2017, pp. 1–4.

[6] *Fashion-MNIST*. `https://github.com/zalandoresearch/fashion-mnist`. Accessed: 2020-11-13.

[7] J. C. Ferreira and J. Fonseca. "An FPGA implementation of a long short-term memory neural network". In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016, pp. 1–8.

[8] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. "Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis". In: *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 244–254.

[9] K. Greff et al. "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (2017), pp. 2222–2232.

[10] Y. Guan et al. "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017, pp. 152–159.

[11] Y. Guan et al. "FPGA-based accelerator for long short-term memory recurrent neural networks". In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, pp. 629–634.

[12] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[13] Song Han et al. "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 75–84. ISBN: 9781450343541. DOI: 10.1145/3020078.3021745. URL: https://doi.org/10.1145/3020078.3021745.

[14] *Hierarchical RNN (HRNN) to classify MNIST digits*. https://github.com/keras-team/keras/blob/master/examples/mnist_hierarchical_rnn.py. Accessed: 2020-06-08.

[15] A. Kouris, S. I. Venieris, and C. Bouganis. "CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 155–1557.

[16] Alexandros Kouris et al. "Approximate LSTMs for Time-Constrained Inference: Enabling Fast Reaction in Self-Driving Cars". In: *ArXiv* abs/1905.00689 (2019).

[17] M. Lee et al. "FPGA-Based Low-Power Speech Recognition with Recurrent Neural Networks". In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2016, pp. 230–235.

[18] Jiwei Li, Minh-Thang Luong, and Dan Jurafsky. "A Hierarchical Neural Autoencoder for Paragraphs and Documents". In: *arXiv preprint arXiv:1506.01057* (2015).

[19] Tomas Mikolov et al. "Recurrent neural network based language model". In: *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*. Ed. by Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura. ISCA, 2010, pp. 1045–1048. URL: http://www.isca-speech.org/archive/interspeech%5C_2010/i10%5C_1045.html.

[20] J. Park et al. "Balancing Computation Loads and Optimizing Input Vector Loading in LSTM Accelerators". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019), pp. 1–1.

[21] Razvan Pascanu et al. "How to construct deep recurrent neural networks". English (US). In: *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*. 2014.

[22] Michalis Rizakis et al. "Approximate FPGA-Based LSTMs Under Computation Time Constraints". In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications - 14th International Symposium, ARC 2018, Santorini, Greece, May 2-4, 2018, Proceedings*. 2018, pp. 3–15. DOI: 10.1007/978-3-319-78890-6_1. URL: https://doi.org/10.1007/978-3-319-78890-6_1.

[23] V. Rybalkin et al. "FINN-L: Library Extensions and Design Trade-Off Analysis for Variable Precision LSTM Networks on FPGAs". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 89–897.

[24] V. Rybalkin et al. “Hardware architecture of Bidirectional Long Short-Term Memory Neural Network for Optical Character Recognition”. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 1390–1395.

[25] S. I. Venieris and C. Bouganis. “fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs”. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016, pp. 40–47.

[26] S. I. Venieris and C. Bouganis. “fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs”. In: *IEEE Transactions on Neural Networks and Learning Systems* 30.2 (2019), pp. 326–342.

[27] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. “Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions”. In: *ACM Comput. Surv.* 51.3 (June 2018). ISSN: 0360-0300. DOI: 10.1145/3186332. URL: https://doi.org/10.1145/3186332.

[28] Erwei Wang et al. “Deep Neural Network Approximation for Custom Hardware: Where We’ve Been, Where We’re Going”. In: *ACM Comput. Surv.* 52.2 (May 2019). ISSN: 0360-0300. DOI: 10.1145/3309551. URL: https://doi.org/10.1145/3309551.

[29] Hongsong Wang and Liang Wang. “Modeling Temporal Dynamics and Spatial Configurations of Actions Using Two-Stream Recurrent Neural Networks”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 3633–3642. DOI: 10.1109/CVPR.2017.387. URL: https://doi.org/10.1109/CVPR.2017.387.

[30] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[31] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

[32] Yong Du, W. Wang, and L. Wang. “Hierarchical recurrent neural network for skeleton based action recognition”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1110–1118.

[33] Chen Zhang et al. “Optimizing fpga-based accelerator design for deep convolutional neural networks”. In: *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 2015, pp. 161–170.

[34] S. Zhang, X. Liu, and J. Xiao. “On Geometric Features for Skeleton-Based Action Recognition Using Multilayer LSTM Networks”. In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2017, pp. 148–157.

Chapter 3

Paper II

S. Ribes, A. Malek, P. Trancoso and I. Sourdis,

Reliability Analysis of Compressed CNNs,

Technical Report.

Reliability Analysis of Compressed CNNs

Abstract

The use of artificial intelligence, Machine Learning and in particular Deep Learning (DL), have recently become a effective and standard de-facto solution for complex problems like image classification, sentiment analysis or natural language processing. In order to address the growing demand of performance of ML applications, research has focused on techniques for compressing the large amount of the parameters required by the Deep Neural Networks (DNN) used in DL. Some of these techniques include parameter pruning, weight-sharing, *i.e.* clustering of the weights, and parameter quantization. However, reducing the amount of parameters can lower the fault tolerance of DNNs, already sensitive to software and hardware faults caused by, among others, high particles strikes, row hammer or gradient descent attacks, *et cetera*. In this work we analyze the sensitivity to faults of widely used DNNs, in particular Convolutional Neural Networks (CNN), that have been compressed with the use of pruning, weight clustering and quantization. Our analysis shows that in DNNs that employ all such compression mechanisms, *i.e.* with their memory footprint reduced up to 86.3×, random single bit faults can result in accuracy drops up to 13.56%.

3.1 Introduction

In recent years, artificial intelligence, machine learning and in particular deep learning have seen a steady growth in popularity thanks to their groundbreaking results. An example of this can be found in Deep Neural Networks (DNN) and in particular Convolutional Neural Networks (CNNs), a special kind of DNN models. CNNs revolutionized the field of computer vision by classifying with high accuracy images from a large set of possible classes. CNNs do so by repeatedly tensor-multiplying an input image with a set of parameters, called *weights*. The resulting tensor can be then further multiplied for the next set of weights. This is process is repeated for a certain amount of steps, or *layers*, each characterized by its own weight parameters. The final result of the data processed throughout the layers is a probability vector that highlights the corresponding class which the image belongs to.

In order to achieve their high accuracy, CNNs go through a process called *supervised training*, which effectively tunes their parameters by aiming at minimizing the error between expected image labels (training images) and the ones produced by the network. Once trained, a CNN can be utilized to classify images never seen during training. The use and execution of trained networks is usually referred as *inference*.

Since CNNs, and DNNs in general, are typically composed of several layers, made of thousands of weights each, running DNNs inference requires a significant amount of memory accesses, eventually making the memory a performance bottleneck. Because of that, accelerating DNNs on general purpose CPUs, GPUs and FPGAs had proved being a challenging task. This drove a lot of research effort into compressing the memory footprint of DNNs in order to improve their performance at inference time in terms of execution time.

However, compressing DNNs, and in particular CNNs, can make the networks more sensitive to faults and attacks. For instance, a CNN can be utilized for classifying objects in images coming from a camera sensor. If the camera is mounted on a self-driving car, the CNN inference would require fast computation and high accuracy in detecting possible objects (like street signs or obstacles) [1]. One straightforward way for improving its performance is to reduce its parameter space, for example by using parameter quantization, *i.e.* moving from a floating point representation to a fixed point one, commonly at 16 or 8 bit. In this scenario, a fault happening in any of the layer parameters of the CNN can eventually propagate, due to the network structure, to the following layers, thus possibly affecting the classification task (like not being able to identify an incoming obstacle). The problem can become even more severe because of the compression in place: a fault, or attack, in a fixed point parameter, that results in a bit flip, can cause a change in its value of a greater magnitude compared to faults taking place in floating point values. This can ultimately increase the chances of the CNN producing wrong outputs, and in turn to a lower accuracy.

In this work, we investigate if and how single and multi bit flip faults can have more severe consequences on the output of compressed DNNs running inference, compared to uncompressed networks. Our hypothesis is that faults happening in the compressed network parameters may cause CNNs to misclassify inputs

at a higher rate compared to their original, uncompressed, counterparts, thus significantly reducing their accuracy score. Based on that, our final aim is to check the conditions for utilizing Odd-ECC from Malek *et al.* [23]. In fact, Odd-ECC can take advantage of the different fault sensitivity of the data to provide efficient and light-weight ECC protection to the different application data regions.

For our analysis, we explore the sensitivity to faults of several DNNs, with a focus on CNNs widely used in research, that have been compressed down to $\times 86$ their original size, *i.e.* the memory footprint of their weights. In order to perform our analysis, we propose a novel framework, named Caffe Macchiato, which we used to compress the networks and then inject single and multi bit faults in specific data regions. To the best of authors' knowledge, this is the first work to analyze and compare the fault tolerance of several CNNs at different levels of compression.

In the remainder of this work, Section 3.2 offers an overview of related works on injecting faults or attacks in deep neural networks. Section 3.3 gives the necessary background knowledge behind CNNs' architectures. In Section 3.4 we describe the implementation of Caffe Macchiato and the methodology we followed to measure the sensibility of compressed CNNs against transient faults. Finally, in Section 3.5 we evaluate the performance of the networks and show the results of our experiments, before concluding in Section 3.6 with a discussion on our findings.

3.2 Related Work

Many authors have investigated the robustness of deep neural networks against faults. DNNs can be executed and accelerated on a variety of computing devices, such as GPUs, ASICs and FPGAs. When focusing on faults in accelerators, faults can happen in the accelerator datapath, such as in MAC units [20, 28, 37], or in buffers [27, 20]. Faults happening in buffers can have a higher impact on DNNs performance than in the datapath counterpart, since buffers are usually used to store partial results of an accumulation and so eventual errors can add up, thereby leading to significant drops in the network accuracy.

Other works focus on studying faults and attacks happening in the network parameters stored in main memory [26, 4]. One example of attacks are trojan attacks on CNNs. Trojan attacks attempt to make the networks misclassify images upon receiving a specific trigger image, while maintaining the same functionality in all the other cases [36, 38, 5, 22]. Other types of attacks include row hammer, laser beam, gradient descend [21] or backdoor [6, 7] attacks.

DNNs are traditionally utilizing single precision floating point representation for their parameters. However, it has been showed that the accuracy of DNNs is not particularly affected when moving to a fixed point representation, *i.e.* after performing a parameter quantization. When testing and analyzing the sensitivity to faults, Reagen *et al.* [27] propose a fault-injecting framework that accounts for this change of precision, while in Li *et al.* [20] single and multi bit faults are injected in different positions within the quantized network parameters. Both works show that quantized parameters are more sensible to faults compared to floating point values due to their reduced bit width. Along

side floating and fixed point values, particular networks called Binarized Neural Networks (BNN) can utilize parameters reduced to binary or ternary values. Khoshavi *et al.* [16] inject cumulative faults in different parts of a BNN that has been implemented as an FPGA accelerator. In their work, they show that 100 single bit faults can cause a drop of 76.7% when injected in the last fully connect layers. However, these works do not account for networks in which multiple compression techniques are applied. In fact, parameter quantization is orthogonal to pruning and weight sharing.

When it comes to fault detection, prevention and correction instead, Li *et al.* [20] selectively apply latch hardening to DNNs accelerators datapath. Another solution from Qin *et al.* [25] is instead correcting the detected faulty parameters by setting them to zero. Along side with that, they also introduce a binary representation for real numbers that is able to limit the effects of faults. Regarding ECC mechanisms targeting DNNs, Guan *et al.* [8] propose to store error check bits in the unused MSB of quantized CNN 8 bit parameters. Their approach is based on a novel training algorithm that regularizes the spatial distribution of large magnitude weights, allowing to exploit unused space for allocating the ECC bits.

Among approaches for protecting DRAM applications that are not tailored to DNNs, Malek *et al.* [23] work, Odd-ECC, dynamically selects and sets the fault tolerance level of different data regions. The ECC bits are in fact stored in separate physical pages, but are physically aligned with the data they protect. This solution allows to access memory efficiently, reducing the energy consumption and significantly improving memory fault tolerance.

Closest to our analysis is the work of Segee *et al.* [30], in which a feed forward neural network is first pruned and then tested for measuring its fault tolerance. Compared to our work, they are not injecting faults by flipping bits, but rather by zeroing out the faulty weights. Moreover, they only focus on one single-input-single-output feed-forward neural network, whereas we analyze a set of more complex networks classifying images. Finally, we not only consider pruning, but also more advanced compression techniques such as weight sharing and quantization.

3.3 Background

3.3.1 Convolutional Neural Networks

Deep neural networks (DNNs) have been extensively applied to many classes of problems like image classification [19, 18], scene labeling [32] or language translation [35]. Figure 3.1 shows an example of a Convolutional Neural Network (CNNs), a particular type of DNNs. DNNs are usually composed of several layers, which are represented by the various blocks and rectangles in figure. Layers are typically connected in a pipeline fashion, where data flows from a layer to the next one. The data produced and consumed by a layer is usually referred as either Feature Maps (FM) or *activations*. A layer can contain a set of parameters called weights, which are used to process the incoming data. Other layers can process incoming inputs without requiring weights. Layers including weights are showed as boxes in the figure, they are, for

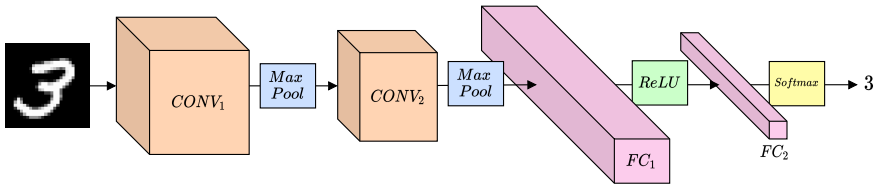

Figure 3.1: An example of Convolutional Neural Network (CNN) called LeNet-5, from the work of Le Cun *et al.* The CNN classifies black and white images of hand-written digits [19].

instance, convolutional (CONV) and fully connected (FC) layers. Weight-free layers are instead pictured as rectangles: Max Pooling, Rectified Linear Unit (ReLU) and Softmax layers are some examples of this kind of layers.

Weights can be learned, *i.e.* finetuned, through a process called training. Training allows a DNN to tune its weights to solve a specific problem, such as image classification.

CONV layers in particular perform a convolution of an input feature map with a weight tensor W of dimension $C_o \times K_h \times K_w \times C_i$. The tensor is divided in kernel matrixes of height K_h and width K_w, as illustrated in Figure 3.2. The terms C_i and C_o represent the number of input and output channels of the input and output data. Given a Feature Map FM_i as input, *e.g.* an image, a CONV layer produces an output Feature Map FM_o whose elements at coordinates (h_o, w_o, c_o) can be obtained following Equation 3.1.

$$FM_o(h_o, w_o, c_o) = B(c_o) + \sum_{k_h=0}^{K_h-1} \sum_{k_w=0}^{K_w-1} \sum_{c_i=0}^{C_i-1} FM_i(h_o \cdot S + k_h, w_o \cdot S + k_w, c_i) \cdot W(c_o, k_h, k_w, c_i), \tag{3.1}$$

where S corresponds to a stride parameter that can further reduce the output feature map dimensions and the amount of operations, while B is a bias term. The idea is to *convolve* the input feature maps with the kernels, *i.e.* weights, saving parameters and preventing overfitting [19]. The usual step that follows the convolution operation is applying an *activation function* to the FM_o. Typically, activation functions are non-linear functions which are applied to the output of a layer before forwarding it to the next one. ReLU is a popular activation function [24] whose behavior is described in Equation 3.2.

$$\text{ReLU}(x) = max(x, 0) + \gamma \cdot min(x, 0), \tag{3.2}$$

where γ represents the negative slope of the function.

Pooling layers are generally placed after CONV layers and are layers responsible for reducing the size of the feature maps thereby preventing the network from overfitting [29], *i.e.* avoids the network learning only the training data. They do so by downsampling the input tensors, as shown in Figure 3.3.

After a series of CONV layers, a CNN generally includes a set of FC layers in its ending part, before concluding with a Softmax layer. FC layers perform Equation 3.3, which consists of a simple matrix-matrix multiplication with a

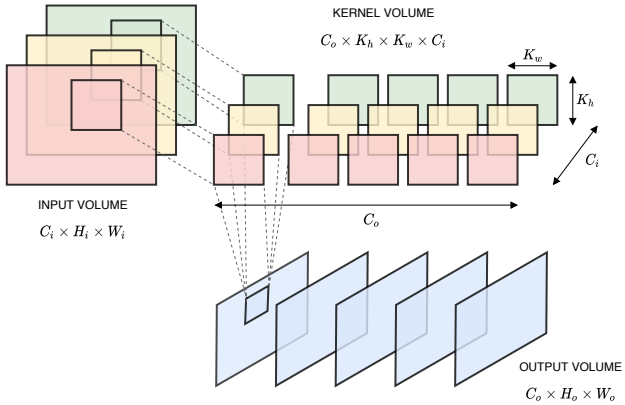

Figure 3.2: Input, output and kernel volumes of a CONV layer.

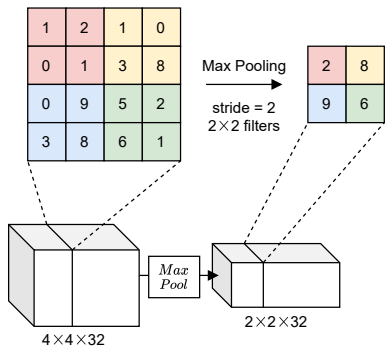

Figure 3.3: Visual representation of a Max Pooling layer. Pooling layers downsample a given tensor reducing the number of parameters and improving the network accuracy.

weight matrix W followed by a bias addition B. Like CONV layers output, the FC layers output is further modified by applying an activation function like ReLU.

$$FM_o = FM_i \cdot W + B \tag{3.3}$$

DNNs including only FC layers are defined as Fully Connected Deep Networks (FCDN). A FCDN typically features two to three FC layers with ReLU activation functions and also terminates with a Softmax layer when solving a classification problem. Finally, a Softmax layer normalizes the output of the last network into a probability distribution over the classes to predict. Hence, the predicted class is identified by picking the class corresponding to the highest probability.

3.3.2 DNNs Compression

Modern neural network architectures generally include a large amount of parameters [31]. However, it has been shown that neural networks can tolerate

a reduction in the amount of parameters without loosing significant accuracy. Such removal of elements is referred as pruning [2]. There exists in literature several ways of pruning a network [9]. One popular and effective technique for pruning consists of iteratively prune and finetune a network to maintain its accuracy [11].

Once most of the weights are zeroed out and do not contribute anymore to the network execution, they can be further compressed by techniques such as weight-sharing and/or quantization (to fixed point or to half-precision floating point representations). Weight-sharing [11], or network clustering, is a technique that attempts to map a limited number of weights in a layer (referred as *centroids*) to all the rest of the weights. In this way, each layer weight is associated to a specific centroid thanks to an index pointing to it. Because of that, only the centroids and the indexes are required during network execution. The centroids can be generated with a clustering algorithm and then finetuned with a backpropagation algorithm to restore the original network accuracy.

Finally, the centroids can be quantized from a floating point 32 or 64 bit representation to 8 or 16 dynamic fixed point representation [10]. The conversion typically causes drops in the network accuracy and therefore requires a retraining phase to calibrate the fixed point parameters.

3.3.3 Soft Errors and Accuracy Degradation in DNNs

Focusing on classification tasks, like assigning a label to a given image, the Softmax layer is a popular final layer for DNNs. The Softmax function is responsible for normalizing the values of the output vector into a probability distribution and for highlighting its maximum value. In practice, it applies Equation 3.4 to all the elements of the output of a DNN.

$$\sigma(x^i) = \frac{e^{x^i}}{\sum_{j=0}^{N-1} e^{x^j}} \ , \ i = 0, 1, ..., N-1 \ , \ x^i \in \mathbb{R}^N \tag{3.4}$$

where N is the length of the vector and so the number of classes. Once the elements are processed, *i.e.* all normalized in $[0, 1]$ and all adding up to 1, the index of the maximum value is selected to identify the class which the input belongs to. This means that having a maximum value at a different index will cause the network to classify the input into another class.

In presence of a fault in one of the weight parameters, the faulty value might generate, in the DNN's output, a different maximum value than the expect one, thereby altering the network prediction. An example of such scenario is depicted in Figure 3.4. Since the FC layer performs a vector-matrix multiplication, a fault in any of its weight matrix parameters can propagate to the layer output and so to the final Softmax layer. In case the fault significantly changes the magnitude of one output parameter, the network will select it as being the class with the highest probability. Because of the fault, if the maximum value results in a different class than the expected one, then the input image is misclassified, thus degrading the accuracy of the DNN.

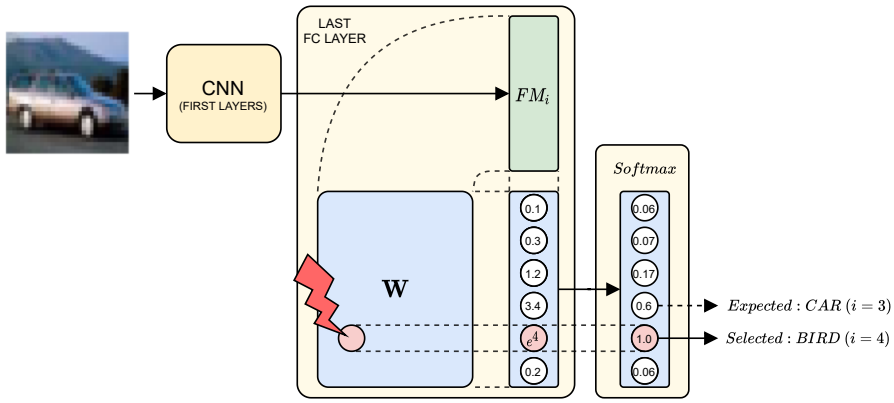

Figure 3.4: How a fault in a weight parameter can affect the classification of an image. A fault in any weight parameter can propagate to the final output and cause a misclassification.

3.4 Design and Methodology

In this section we describe the implementation of our proposed framework Caffe Macchiato, its compression scheme, the application data regions that are sensible to faults and finally how we perform the sensitivity analysis of DNNs.

3.4.1 Caffe Macchiato Framework

We modified Caffe [14] and Ristretto [10] to implement the Caffe Macchiato framework, illustrated in Figure 3.5. In developing Caffe Macchiato, we followed the work of Han *et al.*[11] for designing the steps for network compression without causing significant drops in accuracy. Caffe Macchiato integrates Caffe for training a network given a specification file (in Google Protocol Buffer format [15]), it then performs pruning and clustering before calling Ristretto for the final quantization[1] step. Macchiato also implements a fault injection system for causing bit flips in any compressed network.

Focusing on the implementation details, starting from pruning, we modified the CONV and FC layers in Caffe to include a zero mask to prune the parameters of both weights and bias that are below an adjustable threshold. In order to prune an entire network, Caffe Macchiato first sets the zero masks of all the layers, then retrains the network, *i.e.* finetunes it, while keeping the zero masks fixed.

Once the network is pruned, Caffe Macchiato proceeds applying weight sharing by clustering each layer's weights into a set of k clusters through the K-means algorithm. Caffe Macchiato then stores all the clusters centroids of a layer into a *codebook*, while each weight is substituted by an index, *i.e.* pointer, to its corresponding centroid value. After populating the codebook, Caffe Macchiato retrains the network, thereby finetuning the centroids by following

[1]Han *et al.* in [11] define clustering, *i.e.* weight-sharing, as "quantization". In this work we instead use the term *quantization* for indicating the approximation of the network parameters to fixed point representation.

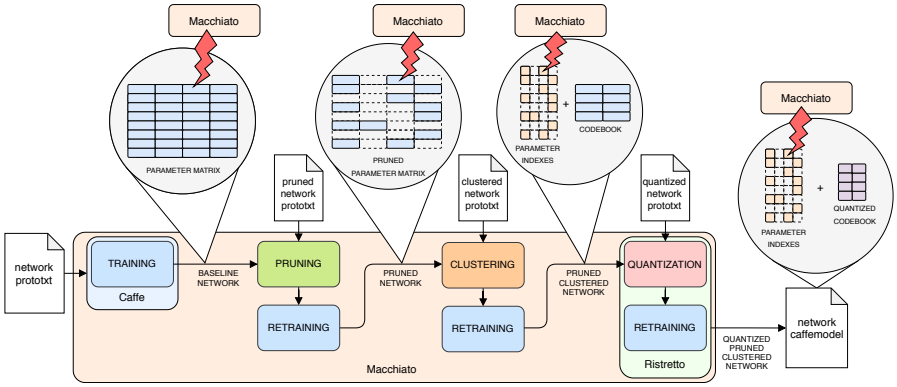

Figure 3.5: The Caffe Macchiato framework. After training in Caffe, Caffe Macchiato prunes and clusters the network while Ristretto finally quantizes it. Besides compression, Caffe Macchiato injects faults in any compression step parameters, as indicated by the red lightnings on the top.

the methodology described in [11], but maintaining the indexes fixed. At run time, the codebook values are used as a substitute of the original weight values, further improving the memory footprint of the network.

Finally, after pruning and clustering, Caffe Macchiato utilizes the Ristretto framework [10] to quantize the network parameters from single precision floating point to a dynamic fixed point representation. We maintain the same non zero parameters and cluster index while quantizing, thus leaving the networks pruned and clustered. In particular, only the non zero values and the codebook values are quantized.

In order to perform our experiments, we first train the networks to achieve a similar accuracy performance as the one reported in literature. We then proceed to prune them to have more than 87% of their values set to zero while maintaining an accuracy drop below 2% after finetuning, *i.e.* retraining. After pruning we cluster the weight values in codebooks of at most 128 centroids, still maintaining a 2% accuracy drop after finetuning.

In order to test the sensitivity of the networks, Caffe Macchiato is able to inject single and multi bit faults in any network layer parameters which has not been zeroed, *i.e.* either weights or biases.

3.4.2 Application Data Regions

A deep neural network is typically composed of several layers that transform an input into either probabilities (classification problem) or real values (regression problem). A layer can include a set of weights and bias parameters that are used to process the layer input activations. During execution, these parameters will need to be available in main memory and so they can be susceptible to faults or attacks. In this chapter we limit our experiments to faults happening only offline in the network parameters, *i.e.* weights and bias values, and not at run time in the activations. The top part of Figure 3.5 provides a simplified view of how weight matrices are compressed and where the faults can happen.

We assume that a fault does not cause the application to crash, but rather that can possibly change the network output, *i.e.* that can affect its accuracy, thus causing Silent Data Corruption (SDC) faults.

For the baseline networks, we marked the layers' weights and bias parameters as a data region for faults to happen. For pruned networks, we further limit the data region to the non-zero parameters only, since the zero values are not required for the computations of a layer. If a network is also clustered, the non zero weights are substituted by a codebook and a list of indexes to the elements of the codebook (referred as *weight indexes*). Hence, for clustered networks, the data region consists of the codebook values, the weight indexes and the non zero bias parameters. We assume the codebook values being in a protected fault-free region of memory and so we are not testing faults happening in the codebook.

For non quantized networks, the weight, bias and codebook parameters are stored as single precision floating point values, whereas the weight indexes (in case of clustered networks) are assumed unsigned integers of bitwidth $log_2(k)$, where k is the number of clusters per layer. For quantized networks instead, we have the parameters quanzited to dynamic fixed point representation [33, 10].

3.4.3 Sensitivity Analysis and Methodology

In order to conduct a sensitivity analysis of DNNs, we follow the methodology described as follows. Given a network specification, we generate in Caffe Macchiato a series of network models which are compressed at different levels. In particular, out of a single network specification we obtain six different network configurations: a baseline network (B), a pruned network (P), a pruned and clustered network (P+C), a quantized network (Q), a quantized pruned network (Q+P) and a quantized pruned clustered network (Q+P+C). All compressed configurations exhibit a reduction in the accuracy of at most 3.1% with respect to the baseline.

Caffe Macchiato is then able to inject single and multi bit faults in any of the above network configurations by flipping bits at random locations in randomly selected parameters. Both the bit position and the targeted parameters for the faults to occur are uniformly distributed.

For the baseline configuration we use Caffe Macchiato for injecting a single-bit fault in a single parameter (either weight or bias) in any network layer. For simulating multi-bit flips instead, the framework swaps the value of two parameters, either two weight or two bias values. We then follows a similar approach for injecting faults in pruned networks, but only targeting a parameter chosen from the non zero ones. In case of pruned and clustered networks instead, we inject single- and multi-bit faults in either the *codebook indexes* or the bias parameters (both aren't zero). For single-bit flips in clustered networks, we flips a random bit in either a random weight index or a random non zero bias value. Multi-bit flips are instead performed by either swapping two weight indexes or by swapping two non zero bias values.

A summary of the types of faults that can be simulated in Caffe Macchiato is reported in Table 3.1. Since Quantization is an orthogonal technique, it can be applied to all the three reported configurations. In case of single bit flips, a

Table 3.1: Description of fault types per network configuration. The faults can be injected in Baseline (B), Pruned (P) and Clustered (C) networks configurations. The location of faults can be either in weights (w) or in bias values (b). For pruned configurations, only the non-zero (NZ) values are selected.

Config	Loc	Single bit fault	Multi bit fault
(B)	(w)	bit flip in random weight	swap 2 random weights
	(b)	bit flip in random bias value	swap 2 random bias values
(P)	(w)	bit flip in random NZ weight	swap 2 random NZ weights
	(b)	bit flip in NZ random bias value	swap 2 random NZ biases
(P+C)	(w)	bit flip in random weight index	swap 2 random weight indexes
	(b)	bit flip in NZ random bias value	swap 2 random NZ biases
(Q)	(w)	bit flip in random quant weight	swap 2 random quant weights
	(b)	bit flip in random quant bias value	swap 2 random quant biases
(Q+P)	(w)	bit flip in random NZ quant weight	swap 2 random NZ quant weights
	(b)	bit flip in NZ random quant bias	swap 2 random NZ quant biases
(Q+P+C)	(w)	bit flip in random weight index	swap 2 random weight indexes
	(b)	bit flip in NZ random quant bias	swap 2 random NZ quant biases

fault can only happen within the bitwidth of the parameters.

3.5 Evaluation

In this section we present and evaluate the results of simulating faults in different networks, compressed in different configurations. For our analysis we chose a series of networks popular in the field of machine learning: a FCDN, LeNet-300-100 [19], and two CNNs: LeNet-5 [19] and CaffeNet [3]. Each network attempts to assign a class to the images from a test dataset. The more test images are correctly classified, the higher is the network accuracy.

The chosen networks have been pruned and clustered without a significant loss of accuracy, as reported in Table 3.2a. The configuration of the compressed networks after quantization is instead reported in Table 3.2b, which illustrates the bitwidth of the quantized parameters and the accuracy of the quantized networks. A summary of the compression ratios achieved for different configurations is showed in Table 3.2c. For our experiments, we report the accuracy drop as the averaged accuracy drop of 1000 fault injection tests. We report a negative drop in cases where a fault is actually improving the original non-faulty accuracy.

LeNet-300-100 on MNIST We injected faults in LeNet-300-100, a network consisting of three fully connected layers classifying the MNIST database, which contains black and white images of hand-written digits. After injecting single-bit and multi-bit faults in both weights and bias in each layer of the baseline network, we do not see any particular drop in accuracy (the drops range from -0.03% to 0.05%). The pruned version of the network is also not affected by random single- and multi-bit faults, showing an accuracy drop between 0% and 0.07%. A similar scenario happens for the clustered network: the accuracy drop is insignificant when injecting single and multi bit flips in both weights

Table 3.2: Different configurations for the analyzed networks and network accuracy scores. The reported configurations are: Baseline (B), Pruned (P), Clustered (C) and Quantized (Q).

(a) The percentage amount of non-zero elements (NZ) for the pruned configurations and the amount of clusters k for FC (k_{FC}) and CONV (k_{CONV}) layers. Notice that we cluster *after* pruning the networks and therefore the pruning percentage is the same in the two configurations.

Network	acc (B)	NZ	acc (P)	k_{CONV}	k_{FC}	acc (P+C)
LeNet-300-100	98.01%	9.7%	98.43%	-	8	97.17%
LeNet-5	99.13%	12.1%	99.09%	64	8|16	98.00%
CaffeNet	81.30%	11.7%	78.51%	128	8	79.11%

(b) Quantized networks accuracy and bitwidth for fully connected (BW_{FC}) and convolutional (BW_{CONV}) layers.

Network	BW_{CONV}	BW_{FC}	(Q)	(Q+P)	(Q+P+C)
LeNet-300-100	-	4	96.97%	96.60%	94.74%
LeNet-5	4	4	97.53%	97.20%	98.17%
CaffeNet	8	8	81.21%	78.20%	81.21%

(c) Original size and compression ratios for the selected networks in different configurations.

Network	Orig Size	(P)	(P+C)	(Q)	(Q+P)	(Q+P+C)
LeNet-300-100	8.14 MB	×10.3	×80.3	×8	×83.1	×86.3
LeNet-5	13.16 MB	×8.3	×80.8	×8	×66.4	×83.0
CaffeNet	2.73 MB	×8.5	×37.3	×4	×34.0	×41.6

and bias, oscillating between 0% and 0.05%. A detailed report of the accuracy drops for single precision parameters can be found in Appendix A.1, Figure A.1.

Table 3.3 shows the result of injecting faults in the quantized network configurations. We can notice that a single bit-flip of the last fully connected layer can cause an average drop in accuracy of 3.95% if injected in weights and 2.68% if injected in bias. For the other layers and fault types we do not see any particular difference instead. A similar scenario happens with the quantized pruned network: a single bit flip fault in the last FC layer can cause a significant drop of 5.12% while a single bit flip in the `fc2` layer reduces the accuracy of 1.03% for injecting in weights and 1.50% in bias. Finally, regarding the clustered, pruned and quantized network, we experience significant drops in accuracy when injecting single bit flips faults in both weights and bias parameters, up to 3.13% and 2.72% for weight and bias respectively.

LeNet-5 on MNIST LeNet-5 is a CNN composed of two CONV layers followed by two FC layers classifying the MNIST dataset, same as LeNet-300-100. We first injected faults in both the baseline, pruned and clustered networks with single precision floating point parameters. In presence of single- and multi-

Table 3.3: Accuracy drop for LeNet-300-100 in configurations: Quantized (Q), Pruned (P) and Clustered (C). The accuracy of the faulty networks is averaged over 1000 tests. The faults are Single bit-flips (S) or Multi bit-flips (M), and are injected in Weights (w) or Bias (b) parameters.

Faulty Layer	Drops: (S in w)	(M in w)	(S in b)	(M in b)
(Q) fc1	0.11%	0.00%	0.26%	-0.06%
(Q) fc2	-0.10%	0.00%	-0.10%	-0.13%
(Q) fc3	3.95%	0.03%	2.68%	-0.02%
(Q+P) fc1	0.18%	0.00%	0.19%	0.00%
(Q+P) fc2	1.03%	0.00%	1.50%	0.06%
(Q+P) fc3	5.12%	0.00%	-	-
(Q+P+C) fc1	2.38%	0.00%	2.38%	0.00%
(Q+P+C) fc2	2.71%	0.00%	2.72%	0.02%
(Q+P+C) fc3	3.13%	0.01%	-	-

bit faults in either weights or bias parameters, we do not see any significant drop in accuracy. All drops remain below 1%, from as low as -0.06% up to 0.11%. More accurate accuracy drops results for single precision parameters can be found in Appendix A.1, Figure A.3.

For the quantized configurations, the results of the fault injection tests are reported in Table 3.4. The quantized networks appear very resilient to multi bit flips, with almost all tests scoring no accuracy drops. We can instead notice significant drops in accuracy when injecting single bit flips, both in weights and bias parameters. For the quantized baseline, injecting faults in the first two CONV layers weights produces high drops of 1.55% and 3.92%, while injecting in the weights of the FC layers does not impact the accuracy. Single-bit faults happening in the bias parameters of any layer greatly affect the accuracy of the quantized baseline, resulting in average drops up to 7.38%.

When analyzing the quantized pruned network configuration, single-bit flips in the weight parameters largely influence the accuracy, causing average drops up to 17.84%. The pruned CaffeNet has most of its bias parameters pruned and so the only layer where we injected faults in bias is `conv2`, causing a non-negligible drop of 2.71%. Lastly, injecting single bit flips in the codebook indexes of the quantized, pruned and clustered network leads to high accuracy drops, as high as 12.37% (except when injecting in the `fc1` layer).

CaffeNet on CIFAR10 We injected faults in different configurations of CaffeNet, a CNN made of three CONV layers followed by a final FC layer. CaffeNet is classifying the CIFAR10 dataset images, a more challenging database of colored images [17]. For the network configurations utilizing single precision floating point values, we could not see any significant accuracy drop after the fault injection tests. The drops are slightly higher compared to LeNet-300-100 and LeNet-5, but still well below 1% (between a minimum of -0.07% and a maximum of 0.53%). All the fault injection results for single precision parameters can be viewed in Appendix A.1, Figure A.5.

We then proceeded to analyze the quantized configurations of CaffeNet.

Table 3.4: Quantized LeNet-5. The accuracy of the faulty network is averaged over 1000 tests. The faults are Single bit-flips (S) or Multi bit-flips (M), and are injected in Weights (w) or Bias (b) parameters.

Faulty Layer	Drops: (S in w)	(M in w)	(S in b)	(M in b)
(Q) conv1	1.55%	0.00%	1.38%	0.00%
(Q) conv2	3.92%	0.00%	6.18%	0.00%
(Q) fc1	-0.81%	0.00%	7.38%	0.00%
(Q) fc2	-0.32%	0.00%	5.27%	0.00%
(Q+P) conv1	1.22%	0.00%	-	-
(Q+P) conv2	13.92%	0.00%	2.71%	-0.12%
(Q+P) fc1	13.87%	0.00%	-	-
(Q+P) fc2	17.84%	-0.01%	-	-
(Q+P+C) conv1	1.72%	0.00%	-	-
(Q+P+C) conv2	1.22%	0.00%	1.48%	0.13%
(Q+P+C) fc1	0.04%	0.00%	-	-
(Q+P+C) fc2	12.37%	0.01%	-	-

We followed the same approach as before and injected single- and multi-bit faults in the parameters of the layers, *i.e.* in weights and bias. The results of the tests are shown in Table 3.5. We can see that multi-bit flips do not cause any particular accuracy drop, similarly to the previous cases (here we have an average drop between -0.01% and 0.55%). However, we can clearly notice a consistent drop in accuracy when injecting single bit flips, both in weights and bias parameters. In particular, for the quantized baseline network, the drop reaches up to 10.39% in weights and 3.65% in bias (the highest drops show up when targeting the last FC layer). We experience a similar trend for the quantized pruned configuration: the drop when injecting in weights is between 4.66% and 40.50%. It appears that random single-bit faults in the last FC layer can halve the original accuracy of the network. Please note that the pruning operation set to zero all the bias parameters and so no faults happening in bias were tested. Finally, the clustered, pruned and quantized CaffeNet shows high accuracy drops when injecting single bit faults in all layers but `conv3` (from 0.77% up to 13.56%). Multi-bit faults do not cause significant drops instead.

3.5.1 Discussion and Limitations

In this work we do not investigate the possible causes that led to the obtained results. In particular, when looking at networks utilizing single precision 32 bit floating point values, we suppose that the wider bitwidth of the values might help in mitigating single bit flips. In fact, the faulty bit position is uniformly distributed and the exponent field, where a bit flip can cause the largest magnitude change, is only 8 bit wide, as specified in the IEEE-754 standard.

Our best speculation regarding the fault injection runs that show low accuracy drops, is that the ReLU and Pooling layers might mask specific single bit-flip faults. In fact, the ReLU function, Equation 3.2, can zero out any negative value when $\gamma = 0$, which was the case for our experiments. Because of

Table 3.5: Quantized CaffeNet-CIFAR10. The accuracy of the faulty network is averaged over 1000 tests. The faults are Single bit-flips (S) or Multi bit-flips (M), and are injected in Weights (w) or Bias (b) parameters.

Faulty Layer	Drops: (S in w)	(M in w)	(S in b)	(M in b)
(Q) conv1	0.36%	0.12%	0.42%	0.01%
(Q) conv2	7.31%	0.09%	0.78%	-0.01%
(Q) conv3	7.31%	0.07%	0.86%	0.00%
(Q) fc1	10.39%	0.05%	3.65%	-0.01%
(Q+P) conv1	4.66%	0.45%	-	-
(Q+P) conv2	14.91%	0.55%	-	-
(Q+P) conv3	13.15%	0.39%	-	-
(Q+P) fc1	40.50%	0.42%	-	-
(Q+P+C) conv1	4.70%	0.11%	-	-
(Q+P+C) conv2	13.56%	0.01%	-	-
(Q+P+C) conv3	0.77%	0.00%	-	-
(Q+P+C) fc1	3.42%	0.00%	-	-

Listing 3.1: Fault masking of `NaN` values through the `max` operation.

```
#define MAX(a, b) a > b ? a : b

MAX(1.0, NaN); // Evaluates to: 1.0 > NaN ? 1.0 : NaN, returns NaN
MAX(NaN, 1.0); // Evaluates to: NaN > 1.0 ? NaN : 1.0, returns 1.0
```

that, all single bit faults leading to a negative real value, even with very high magnitude, can be masked to zero.

If the faulty value turns out to be a `NaN` (either because of a bit flip or as an accumulated result), both Max Pooling, Figure 3.3, and ReLU layers can mask it through their `max` operation. Listing 3.1 provides a simple example of the `max` implementation and the possible outcomes upon processing a `NaN` value.

Effectively, the IEEE-754 floating point standard does not specify the outcome of a comparison operation with a `NaN` [13]. However, our framework is developed in C++11, following the original Caffe implementation, which adheres to the IEC-559 standard [12, 34], which defines `false` as a return value for any comparison involving a `NaN`. Because of this, depending on "which side" the `NaN` value falls in the comparison, it can result in either a regular floating point number or a `NaN` value, thus eventually masking the fault.

Overall, the above conclusions seem to be supported by our findings. In fact, according to the results in Tables 3.3, 3.4 and 3.5, the last FC layer in all of the tested networks, which is never followed by a Max Pooling layer nor a ReLU layer, generally appears to be, on average, the most sensitive to faults, *i.e.* leading to the highest accuracy drops.

3.6 Conclusions

Our experiments suggest that using single precision floating point values ensures a high level of fault tolerance against random single- and multi-bit flips. Even if compressed, DNNs and in particular CNNs are able to correctly classify images in presence of faults and thus do not require any additional protection mechanism. Instead, significant drops in accuracy are happening to the quantized network configurations when injecting single bit flips. For the baseline quantized configuration, the networks are more tolerant to faults happening in their first layer, while are more affected if happening in the last layer. The drops of all pruned networks are higher than the ones of pruned and clustered networks, but both configurations show the highest drops when injecting in the last layer. We do not see a clear difference in drops caused when injecting single bit faults in weights or bias parameters, suggesting that both data regions are highly sensitive to faults. Overall, we observed that the faults causing the highest drops happen at the back of the network, *i.e.* in the last layers, whereas faults effects tend to be mitigated or compensated if happening in the first layer.

We can conclude that quantizing a DNN can significantly lower the fault tolerance of FCDNs and CNNs. In addition to this, our experiments show that further compressing the quantized networks by applying pruning and eventually clustering can lead to even higher losses in tolerance, thus making the networks requiring fault protection mechanisms. Based on our findings, as a future work we will be able to apply and test the Odd-ECC [23] protection mechanisms tailored to the identified more sensitive data regions.

References

[1] Michael Beyer et al. "Quantification of the Impact of Random Hardware Faults on Safety-Critical AI Applications: CNN-Based Traffic Sign Recognition Case Study". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2019, pp. 118–119.

[2] Davis Blalock et al. "What is the state of neural network pruning?" In: *arXiv preprint arXiv:2003.03033* (2020).

[3] *CaffeNet on CIFAR10.* `https://caffe.berkeleyvision.org/gathered/examples/cifar10.html`. Accessed: 2020-07-20.

[4] Wonseok Choi et al. "Sensitivity based error resilient techniques for energy efficient deep neural network accelerators". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6.

[5] Joseph Clements and Yingjie Lao. "Hardware trojan design on neural networks". In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2019, pp. 1–5.

[6] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. "Badnets: Identifying vulnerabilities in the machine learning model supply chain". In: *arXiv preprint arXiv:1708.06733* (2017).

[7] Tianyu Gu et al. “Badnets: Evaluating backdooring attacks on deep neural networks”. In: *IEEE Access* 7 (2019), pp. 47230–47244.

[8] Hui Guan et al. “In-place zero-space memory protection for cnn”. In: *arXiv preprint arXiv:1910.14479* (2019).

[9] Kaiyuan Guo et al. “[DL] A survey of FPGA-based neural network inference accelerators”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12.1 (2019), pp. 1–26.

[10] Philipp Gysel et al. “Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks”. In: *IEEE transactions on neural networks and learning systems* 29.11 (2018), pp. 5784–5789.

[11] Song Han, Huizi Mao, and William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. In: *arXiv preprint arXiv:1510.00149* (2015).

[12] *IEC 559:1989 Binary floating-point arithmetic for microprocessor systems*. `https://www.iso.org/standard/19706.html`. Accessed: 2021-02-19.

[13] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: `10.1109/IEEESTD.2019.8766229`.

[14] Yangqing Jia et al. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678.

[15] Gurpreet Kaur and Mohammad Muztaba Fuad. “An evaluation of protocol buffer”. In: *Proceedings of the ieee southeastcon 2010 (southeastcon)*. IEEE. 2010, pp. 459–462.

[16] Navid Khoshavi, Connor Broyles, and Yu Bi. “Compression or Corruption? A Study on the Effects of Transient Faults on BNN Inference Accelerators”. In: *2020 21st International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2020, pp. 99–104.

[17] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[19] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[20] Guanpeng Li et al. “Understanding error propagation in deep learning neural network (DNN) accelerators and applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–12.

[21] Yannan Liu et al. “Fault injection attack on deep neural network”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 131–138.

[22] Yuntao Liu, Yang Xie, and Ankur Srivastava. "Neural trojans". In: *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 45–48.

[23] Alirad Malek et al. "Odd-ECC: on-demand DRAM error correcting codes". In: *Proceedings of the International Symposium on Memory Systems*. 2017, pp. 96–111.

[24] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *ICML*. 2010.

[25] Minghai Qin, Chao Sun, and Dejan Vucinic. "Robustness of neural networks against storage media errors". In: *arXiv preprint arXiv:1709.06173* (2017).

[26] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. "Bit-flip attack: Crushing neural network with progressive bit search". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 1211–1220.

[27] Brandon Reagen et al. "Ares: A framework for quantifying the resilience of deep neural networks". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE. 2018, pp. 1–6.

[28] F. F. d. Santos et al. "Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs". In: *IEEE Transactions on Reliability* 68.2 (2019), pp. 663–677.

[29] Dominik Scherer, Andreas Müller, and Sven Behnke. "Evaluation of pooling operations in convolutional architectures for object recognition". In: *International conference on artificial neural networks*. Springer. 2010, pp. 92–101.

[30] Bruce E Segee and Michael J Carter. "Fault tolerance of pruned multilayer networks". In: *IJCNN-91-Seattle International Joint Conference on Neural Networks*. Vol. 2. IEEE. 1991, pp. 447–452.

[31] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*. 2015.

[32] Oriol Vinyals et al. "Show and tell: A neural image caption generator". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3156–3164.

[33] Darrell Williamson. "Dynamically scaled fixed point arithmetic". In: *[1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*. IEEE. 1991, pp. 315–318.

[34] *Working Draft, Standard for Programming Language C++*. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`. Accessed: 2021-02-19.

[35] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[36] Jing Ye, Yu Hu, and Xiaowei Li. "Hardware trojan in fpga cnn accelerator". In: *2018 IEEE 27th Asian Test Symposium (ATS)*. IEEE. 2018, pp. 68–73.

[37] Jeff Jun Zhang et al. "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator". In: *2018 IEEE 36th VLSI Test Symposium (VTS)*. IEEE. 2018, pp. 1–6.

[38] Yang Zhao et al. "Memory trojan attack on neural network accelerators". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1415–1420.

Appendix A

A.1 Distribution of Accuracy Degradation

In this section we provide a detailed set of graphs reporting the distribution of the accuracy of the networks in presence of faults. For each experiment we collected 1000 accuracy samples, meaning that we injected 1000 faults per scenario. The results are grouped by layer and are divided according to the network configuration, being: Pruned (P), Clustered (C), Quantized to fixed point (Q) and their combinations. Faults are indicated either as Single or Multi bit flips in Weights (SW or MW) or Single of Multi bit flips in Bias (SB or MB).

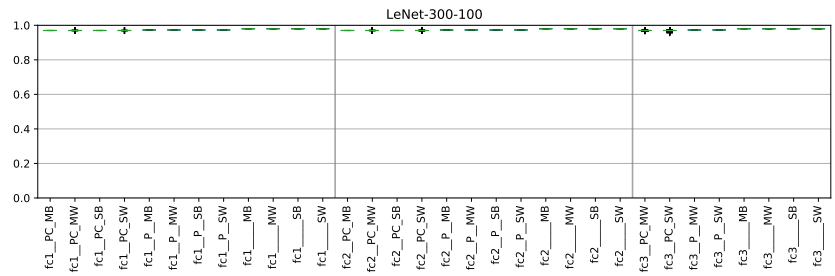

Figure A.1: Accuracy degradation distribution after injecting faults in LeNet-300-100.

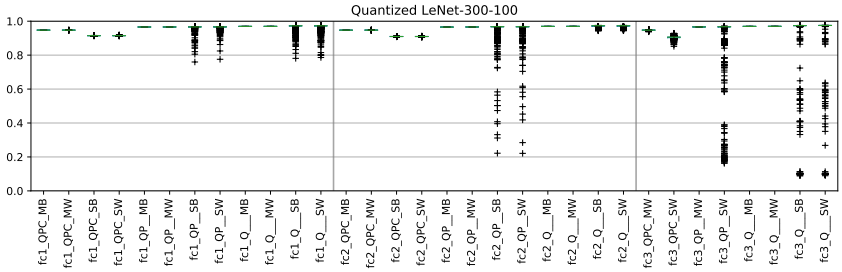

Figure A.2: Accuracy degradation distribution after injecting faults in quantized LeNet-300-100.

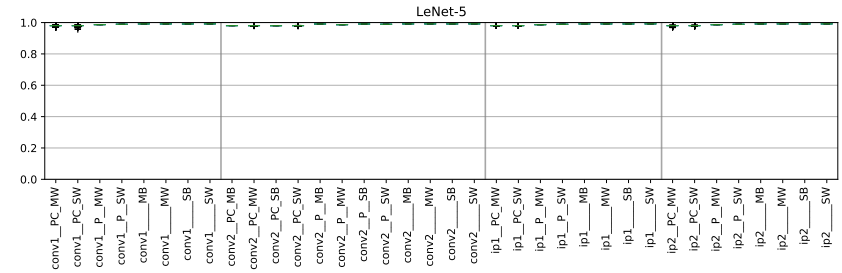

Figure A.3: Accuracy degradation distribution after injecting faults inLeNet-5.

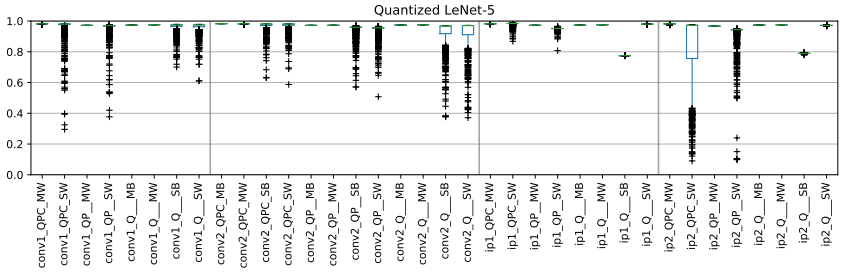

Figure A.4: Accuracy degradation distribution after injecting faults in quantized LeNet-5.

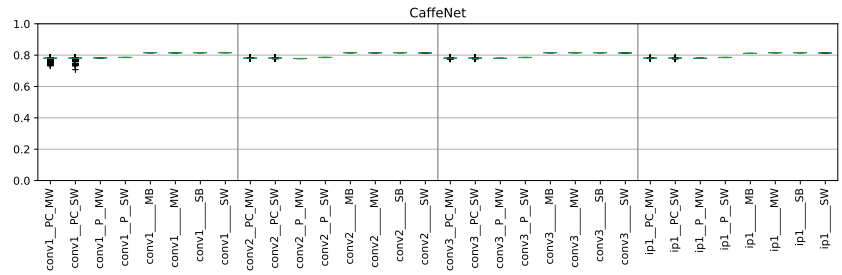

Figure A.5: Accuracy degradation distribution after injecting faults in CaffeNet.

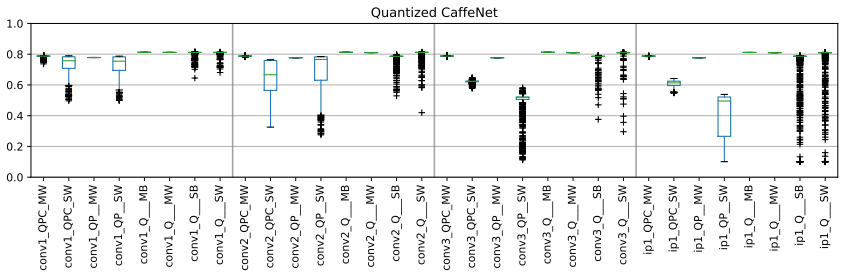

Figure A.6: Accuracy degradation distribution after injecting faults in quantized CaffeNet.