UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática
de Sistemas y Computadores

Fault Tolerance Based on Bit-Level Redundancy for Neural Network Models

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Engineering)

Author

Izan Catalán Gallach

Advisors

Prof. José Flich Cardo
Prof. Carles Hernández Luz

October 2025

Agradecimientos

Uno nunca se imagina que hará un doctorado. A decir verdad, durante el proceso tampoco se vislumbra cuándo se terminará, pero tras varios años y sin creérmelo aún, ese momento al final ha llegado. Ha sido un camino inesperado que me ha permitido aprender innumerables lecciones y mejorar como investigador y como persona, y que siempre recordaré.

Sin embargo, nunca lo habría podido hacer solo. Si he llegado hasta aquí, ha sido gracias a todas aquellas personas que, año tras año, me han acompañado, me han dado ánimos o me han ayudado a desconectar para seguir adelante.

Quiero dar las gracias a quienes han sido mis directores de tesis, Pepe y Carles. Ellos forman parte de este trabajo y, sin su ayuda y guía, no habría podido ni empezar este camino ni acabarlo. Aún recuerdo cuando, al empezar el máster, comencé a trabajar con Pepe; no pensé que ahí iniciaría un proceso que dura hasta hoy. En este tiempo hemos compartido mil y una reuniones —ayudas en mudanzas de despacho incluidas— y almuerzos improvisados con empanadillas donde, indirectamente, han surgido muchas de las ideas y guías para encauzar todo el trabajo. ¿Y cómo no agradecer a Carles por sus enseñanzas y directrices, que aportan claridad con pocas palabras y son capaces de resumir todo lo que necesitaba? Echaré de menos esa cotidianidad que ambos me han transmitido y que, sin duda, me ha ayudado.

Por supuesto, agradezco también a José Cano su ayuda y su recibimiento durante mi estancia en Glasgow, así como su comprensión para gestionar tanto el trabajo de la estancia como el del doctorado.

Gracias a mi familia, que ha estado ahí en todo momento: a mi madre, que siempre ha sido la primera en darme el apoyo y el ánimo necesarios para hacerme ver que todo tiene una salida y que siempre hay una opción; a mi padre, por su toque de tranquilidad, que me permitía descargar con él de manera sosegada las dudas y problemas que he ido teniendo; y, por supuesto, a mi hermano, el "xiquitín", con quien siempre desconectaba hablando de cualquier otra cosa, algo que todos necesitamos a menudo para despejarnos. También a mis tíos y primos, que siempre me han dado ánimos.

Agradecimientos a mis amigos de toda la vida: "del IES Vilamarxant al doctorado", una frase que seguro les saca una sonrisa y que es razón de más para remarcarla. Ellos me han acompañado todos estos años y, seguramente sin saberlo, con su sola presencia me daban la energía y la vida necesarias para seguir adelante. Gracias por las correcciones lingüísticas de Manuel, por una y más *servilletetes* de Carles, por los *abracitos* de Edgar,

por la persona con la mejor barba del mundo, Roberto, porque “el arroz está compacto” es en realidad “ni yo mismo lo haría mejor” a un futuro mejor profe como Ernest, por la ya sabiduría de padre de Coll, la segunda vida de Saal en Seúl, los *carxofs*, Rafa, Papato, el Jiwo, el mejor escalador y ahora runner, Zhon; por el viajecito a Sofía durante la estancia de Manuel y por muchas otras experiencias vividas en este tiempo. Os admiro mucho. Muchas gracias por todo.

A mis amigos de la carrera y del máster: a Manu (y su acento de Aielo) y a Manel, que me ha acompañado también con su doctorado y con quien siempre nos hemos apoyado en las dudas; ellos son mis compañeros de aventuras en Argentina. También a Marcos y su paciencia infinita, ahora convertido en el mejor compañero de piso; a Charlie, ya oficialmente profe; a Juan, con su capacidad de escucha y apoyo; y a David y sus eventos de calendario, de los que nunca se tienen suficientes.

A mis compañeros de laboratorio, que han sido vitales para compartir las investigaciones que estaba realizando, por esos almuerzos en la Tarongeria y por ese *cremaet* que seguro llegará. Gracias por compartir también experiencias juntos.

Y, por último, gracias a todas las personas que he conocido al cursar el doctorado, algunas de ellas en mi misma situación. Son personas maravillosas que merecen todo el ánimo y agradecimiento del mundo y que, al igual que yo, conseguirán terminar con éxito todos los objetivos que se propongan.

Muchas de ellas compaginan trabajos y doctorados complejos, con problemas de tiempo, dudas, momentos difíciles y también alegrías. Sé que muchas veces no se ve el final, pero siempre llega. Gracias por compartir sus vivencias y tiempo conmigo, por su apoyo y ánimos. Ellas también podrán, porque su trabajo se verá recompensado. Sin ellas, no habría sido lo mismo; solo siento admiración. ¡Mucho ánimo!

Contents

List of Figures **ix**

List of Tables **xi**

Abbreviations and Acronyms **xiii**

Abstract **xvii**

Resumen **xviii**

Resum **xix**

1 Introduction **1**
1.0.1 Neural Networks . . . 2
1.0.2 Faults in Neural Networks . . . 3
1.0.3 Compute Platforms: GPUs and FPGAs . . . 3
1.1 Objectives and Motivation of the Thesis . . . 5
1.2 Main Contributions of the Thesis . . . 6
1.3 Thesis Outline . . . 7

2 Background and Related Work **9**
2.1 Background . . . 10
2.1.1 History of NNs . . . 10
2.1.2 Types of NNs . . . 11
2.1.3 Structure and Functioning . . . 12
2.1.4 Training and Inference . . . 14
2.1.5 Convolutional Networks . . . 16
2.1.6 Hardware Faults Affecting NNs . . . 17
2.1.7 Security Attacks Exploiting Fault-Injection in NNs . . . 18
2.2 Related Work in Fault-Tolerant NNs . . . 19
2.2.1 Impact of Faults on NNs . . . 19
2.2.2 Fault-Tolerance Mechanisms . . . 20
2.2.2.1 Hardware Mechanisms . . . 20
2.2.2.2 Software Mechanisms . . . 22
2.2.2.3 Algorithmic/Model-Level Solutions . . . 23
2.2.2.3.1 Positioning of Our Approach . . . 24

3 Bit-level Invariability Analysis 27
3.1 Neural Network Models to be Analysed . . . 28
3.2 Methodology . . . 31
3.3 Bit-Grouping Strategies . . . 33
3.4 Bit-Level Invariability . . . 35
3.4.1 Adapting Strategies to Neural Network Models . . . 36
3.4.2 Bit Invariability Evolution Layer-by-Layer across NN models . . . 38
3.4.3 General Results for FP32 Data Type . . . 40
3.4.3.1 ViT Model . . . 43
3.4.4 General Results for INT8 Data Type . . . 45
3.4.4.1 Conclusions . . . 46

4 Fault Protection Methods 47
4.1 Fixed Protection (FP) Mechanism . . . 49
4.2 Variable Protection (VP) Mechanism . . . 51
4.3 FP and VP Coverage and Overheads . . . 52
4.4 Evaluation . . . 57
4.4.1 Evaluation Methodology . . . 57
4.4.1.1 Fault Model . . . 58
4.4.2 Experimental Setup . . . 60
4.4.3 Results . . . 60
4.4.4 Fault Classification . . . 60
4.4.4.1 Bit Flips Effect on Unprotected Models . . . 63
4.4.4.2 Robustness of VP/FP Mechanisms . . . 64
4.4.4.3 Silent Data Corruption Effect . . . 65
4.4.4.4 Faults Evolution per Convolutional Layer . . . 67
4.4.4.5 Bit invariability vs Model Data Precision Type . . . 71
4.4.5 Comparison with Related Works . . . 73
4.4.5.1 New Methodology and Setup . . . 75
4.4.5.2 Results . . . 77
4.4.6 Impact of Faults Outside Convolutional Filter Weights in CNNs . . . 81
4.5 Summary . . . 84

5 FPGA and GPU Implementations 89
5.1 FPGA-based Accelerator . . . 90
5.1.1 Background and Implementation Procedure . . . 90
5.2 Integration and Overheads . . . 91
5.2.1 Accelerator Architecture Overview . . . 92
5.2.2 Incorporation of FP and VP Mechanisms . . . 93
5.2.3 Impact on Execution Time . . . 95
5.2.4 FPGA Resource Overheads . . . 96
5.3 GPU Integration . . . 97
5.3.1 Background and Implementation Procedure . . . 97
5.3.2 CUTLASS Integration . . . 99
5.3.2.1 CUTLASS Modifications . . . 100
5.3.2.1.1 C++/CUDA Back End . . . 100

5.3.2.1.2 Mask Life Cycle and Execution Policy 102
5.3.2.1.3 Python Front End 102
5.3.2.2 PyTorch Modifications 103
5.3.2.3 User View 104
5.3.2.4 Performance Results 104
5.4 Summary 107

6 Conclusions **109**
6.1 Contributions 110
6.2 Future Directions 112
6.3 Publications 113

References **115**

List of Figures

1.1 Number of people using AI tools worldwide. Source [25]. 2
1.2 Diagram showing the main contributions to fault tolerance in NN. 6

2.1 Biological neuron compared to an artificial neuron in a neural network. . . 10
2.2 NN diagram. 13
2.3 Forwarding and back propagation functions. 15
2.4 CNN structure. 16

3.1 Analysis flow. 32
3.2 Different grouping strategies for weights in a 2D convolution layer. Weights are distributed across four dimensions: $O \times I \times KH \times KW$. 33
3.3 Average invariability across three different grouping strategies simulating FP32 elements. 35
3.4 Bit-invariability across convolution layers per CNN model. 39
3.5 Various grouping techniques analysis in FP32 models. 41
3.6 Various analyses of invariant bits in FP32 models by grouping the sets into 9 elements with ViT model. 44
3.7 Various analyses of invariant bits in INT8 models by grouping the sets into 2D-convolutional filters. 45

4.1 Fixed Protection Mechanism. 49
4.2 Variable Protection Mechanism. 51
4.3 Invariant bits coverage by FP and VP with INT8. 55
4.4 Invariant bits coverage by FP and VP with FP32. 56
4.5 Evaluation flow. 57
4.6 FP32 Data Type Top 1 Predictions Image-by-Image for all ImageNet Dataset. 61
4.7 INT8 Data Type Top 1 Predictions Image-by-Image for all ImageNet Dataset. 62
4.8 Impact (percentage of difference) of bit flips on Top1 and Top5 metrics in FP32 models. 62
4.9 Impact (percentage of difference) of bit flips on Top1 and Top5 metrics in INT8 models. 63
4.10 SDC Ratio with FP32 Data Type. 65
4.11 SDC Ratio with INT8 Data Type. 66
4.12 Top 1 accuracy impact per convolution with FP32 Data Type. 68
4.13 SDC Ratio per convolution with FP32 Data Type. 69

4.14 Extra SDC ratio of VP with respect to FP per convolution with FP32 Data Type. 70
4.15 SDC Ratio with FP32 data type per position. The ViT model was not considered due to its high computational cost. 73
4.16 SDC (positive predictions and 1-bit flip) per class ordered from lowest to highest ratio. 76
4.17 SDC comparison with multi-bit flip injection. Positive predictions and 1 to 20-bit flip injections (BF). 78
4.18 SDC per mechanism comparison on ShuffleNet (positive predictions and 1-bit flip) with no ordered ratio. 79
4.19 Top 1 Accuracy impact with FP32 Data Type. 82
4.20 SDC comparison with multi-bit flip injection. Positive predictions with 1 or 10-bit flip injections (1/10 BF). 83

5.1 HLSInf Baseline. 91
5.2 Accelerator architecture. Modules are represented as boxes, and streams are represented as arrows. Modules can be placed in any order. 91
5.3 Performance of CUTLASS Library using GEMM kernels with different Data Types when run on NVIDIA Blackwell SM100 architecture GPU. Source and details about Data Types can be found in [18]. 98
5.4 Diagram showing the use of the CUTLASS implementation and its integration with PyTorch. The colour green indicates the modifications required to implement the VP fault-protection mechanism. Blue arrows show calls to functions and orange arrows show data flux between those functions. . 99
5.5 Timing overheads with different Image Input Shapes. 105
5.6 Results comparison running Imagenet Dataset Inference using PyTorch with default CuDNN and with CUTLASS: *Unprotected* and *Protected* (with VP). 105

List of Tables

3.1 NN model specifications used in our analysis. 28
3.2 Structural summary of the analysed models. 29
3.3 Filter-based grouping structure for the analysed CNN models. 36

4.1 Running time for the calculation of FP and VP masks in seconds. 53
4.2 Fixed and Variable Protection Analysis. 54
4.3 FP bit coverage with respect to VP. 55

5.1 Accelerator Overheads with Fixed Protection (FP) and Variable Protection (VP) implementation on HLSinf. 96

Abbreviations and Acronyms

AI	**A**rtificial **I**ntelligence
BF16	**B**float **16** (16-bit floating point)
BFA	**B**it **F**lip **A**ttack
CNN	**C**onvolutional **N**eural **N**etwork
CPI	**C**hannels **P**er **I**nput
CPO	**C**hannels **P**er **O**utput
CRC	**C**yclic **R**edundancy **C**heck
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
CUTLASS	**CU**DA **T**emplates for **L**inear **A**lgebra **S**ubroutines
CVT	Convert (module in HLSinf accelerator)
DL	**D**eep **L**earning
DMR	**D**ual **M**odular **R**edundancy
DNN	**D**eep **N**eural **N**etwork
DRAM	**D**ynamic **R**andom-**A**ccess **M**emory
DS	**D**e-**S**tress (mode in accelerator)
DSL	**D**omain-**S**pecific **L**anguage
DVFS	**D**ynamic **V**oltage and **F**requency **S**caling
DW	**D**epth**w**ise (convolution layer)
ECC	**E**rror **C**orrecting **C**ode
FAT	**F**ault-**A**ware **T**raining
FC	**F**ully **C**onnected (layer)
FFT	**F**ast **F**ourier **T**ransform
FMR	**F**eature **M**ap **R**ecovery (averaging duplicated feature maps)
FP	**F**ixed **P**rotection (proposed mechanism)

FP16	**F**loating **P**oint **16**-bit
FP32	**F**loating **P**oint **32**-bit
FP64	**F**loating **P**oint **64**-bit
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
FT	**F**ault-**T**olerant (mode in accelerator)
GEMM	**G**eneral **E**ngine for **M**atrix **M**ultiplication
GPU	**G**raphics **P**rocessing **U**nit
HBM	**H**igh **B**andwidth **M**emory
HCI	**H**ot **C**arrier **I**njection
HLS	**H**igh-**L**evel **S**ynthesis
HLSinf	**H**igh-**L**evel **S**ynthesis **inf**erence accelerator
HP	**H**igh-**P**erformance (mode in accelerator)
HW	**H**eight x **W**idth (Filter dimensions)
img2col	**Im**a**g**e-**to**-**col**umn transform (lowering)
INT4	**4**-bit **int**eger
INT8	**8**-bit **int**eger (quantised data type)
ISO26262	Road vehicles Functional safety (standard)
KH	**K**ernel **H**eight
KR	**K**ernel **R**ecovery (averaging duplicated kernels)
KW	**K**ernel **W**idth
LLM	**L**arge **L**anguage **M**odel
LSB	**L**east **S**ignificant **B**it
MBU	**M**ulti-**B**it **U**pset
MCU	**M**icro**c**ontroller **U**nit
MFC	**M**asked **F**aults **C**orrect
MFM	**M**asked **F**aults **M**isclassified
MLP	**M**ulti**l**ayer **P**erceptron
MMA	**M**atrix **M**ultiply-**A**ccumulate
MSB	**M**ost **S**ignificant **B**it
NBTI	**N**egative **B**ias **T**emperature **I**nstability
NCHW	Batch, Channels, Height, Width (data layout format)
NHWC	Batch, Height, Width, Channels (data layout format)
NLP	**N**atural **L**anguage **P**rocessing

NN	**N**eural **N**etwork
NVFP4	**NV**IDIA **F**loating **P**oint **4**-bit
ONNX	**O**pen **N**eural **N**etwork **E**xchange
PE	**P**rocessing **E**lement
PW	**P**oint**w**ise (convolution layer)
PyTorch	Python Torch (DL framework)
QNN	**Q**uantized **N**eural **N**etwork
ReLU	**R**ectified **L**inear **U**nit
RNN	**R**ecurrent **N**eural **N**etwork
Rowhammer	DRAM disturbance fault attack technique
SA	**S**elf-**A**ttention
SDC	**S**ilent **D**ata **C**orruption
SECDED	**S**ingle-**E**rror **C**orrection, **D**ouble-**E**rror **D**etection
SEU	**S**ingle **E**vent **U**pset
SGD	**S**tochastic **G**radient **D**escent
SRAM	**S**tatic **R**andom-**A**ccess **M**emory
STM	Stream-to-Memory (module in HLSinf accelerator)
TF32	**T**ensor **F**loat **32** (NVIDIA)
TMR	**T**riple **M**odular **R**edundancy
TMRC	**T**riple **M**odular **R**edundancy with **C**hecker (majority voter)
TPU	**T**ensor **P**rocessing **U**nit
VF	**V**isible **F**aults
VFS	**V**oltage **F**requency **S**caling (Attack)
ViT	**V**ision **T**ransformer
VMM	**V**ector **M**atrix **M**ultiplication
VP	**V**ariable **P**rotection (proposed mechanism)
WSC	**W**arehouse-**S**cale **C**omputer
Top-1/Top-5	Accuracy@1 / Accuracy@5 metrics

Abstract

Neural Networks are widely used in critical environments such as healthcare, autonomous vehicles, or video surveillance. To ensure the safety of the systems that rely on their functionality, it is essential to validate and guarantee their correct behaviour in the presence of faults.

This thesis investigates, first, the bit-level redundancy present in state-of-the-art convolutional neural network models by organising weights into structured groups and determining invariant bit positions within those groups.

Building on this analysis, two lightweight model-agnostic protection mechanisms are developed. These mechanisms exploit such redundancy (without requiring retraining of the network or architectural duplication): Fixed Protection (FP), which safeguards consecutive bits in each element of the weights, and Variable Protection (VP), which extends coverage to non-consecutive invariant bits.

To validate the effectiveness of the proposed fault-protection mechanisms, extensive fault-injection experiments involving random bit flips in weights are conducted to quantify the models' sensitivity and their impact on inference accuracy. Unprotected models typically exhibit an accuracy degradation ranging from 1.3% to over 3%. With FP and VP, however, the impact of bit flips is substantially mitigated, down to between 0.0001% and 0.4%, representing an improvement of several orders of magnitude. These mechanisms are designed to be computationally efficient and portable, and can be applied to both convolutional and non-convolutional models.

Finally, we implement support for our mechanism on both GPUs and an FPGA-based accelerator, validating the use of the protection mechanisms in real execution environments. The GPU implementation leverages NVIDIA CUTLASS and PyTorch, while the FPGA implementation incorporates dedicated masking modules in a streaming accelerator without altering the core compute pipeline. Collectively, our results show that exploiting bit-level redundancy provides an effective, efficient, and general approach to enhancing the fault tolerance of modern neural networks.

Resumen

Las redes neuronales se emplean ampliamente en entornos criticos como la sanidad, los vehículos autónomos o la videovigilancia. Para garantizar la seguridad de los sistemas que dependen de su funcionamiento, es esencial validar y garantizar su comportamiento correcto en presencia de fallos.

Esta tesis investiga, en primer lugar, la redundancia a nivel de bit presente en modelos de última generación, en concreto modelos convolucionales, organizando los pesos en grupos estructurados y determinando las posiciones de bit invariantes dentro de dichos grupos.

A partir de este análisis, se desarrollan dos mecanismos de protección ligeros y agnósticos al modelo. Estos mecanismos explotan dicha redundancia (sin requerir reentrenar al modelo ni duplicar la arquitectura): Fixed Protection (FP), que protege bits consecutivos, y Variable Protection (VP), que amplía la cobertura a bits invariantes no consecutivos.

Se han llevado a cabo experimentos extensivos de inyección de fallos, consistentes en cambios aleatorios de bits en los pesos, con el fin de cuantificar la sensibilidad del modelo y su impacto en la precisión de la inferencia. Los modelos sin protección presentan, por lo general, una degradación de precisión comprendida entre 1,3 % y más del 3 %. Con FP y VP, sin embargo, el impacto de los bit flips se mitigan de forma sustancial, hasta valores entre el 0,0001 % y el 0,4 %, lo que supone una mejora de varios órdenes de magnitud. Ambos mecanismos están diseñados para ser computacionalmente eficientes y portables, y pueden aplicarse tanto a modelos convolucionales como no convolucionales.

Por último, se implementamos nuestros mecanismos en GPUs y en un acelerador basado en FPGA, validando el uso de los mecanismos de protección en entornos de ejecución reales. La implementación en GPU aprovecha NVIDIA CUTLASS y PyTorch, mientras que la implementación en FPGA incorpora módulos dedicados de enmascaramiento en un acelerador en flujo sin alterar la tubería de cómputo principal. En conjunto, los resultados muestran que aprovechar la redundancia a nivel de bit constituye un enfoque eficaz, eficiente y general para aumentar la tolerancia a fallos de las redes neuronales modernas.

Resum

Les xarxes neuronals s'utilitzen àmpliament en entorns crítics com la sanitat, els vehicles autònoms o la videovigilància. Per garantir la seguretat dels sistemes que depenen del seu funcionament, és essencial validar i garantir el seu comportament correcte en presència de fallades.

Aquesta tesi investiga, en primer lloc, la redundància a nivell de bit present en models d'última generació, en concret models convolucionals, organitzant els pesos en grups estructurats i determinant les posicions de bit invariants dins d'aquests grups.

A partir d'aquest anàlisi, es desenvolupen dos mecanismes de protecció lleugers i agnòstics al model. Aquests mecanismes exploten aquesta redundància sense requerir reentrenament dels models ni duplicació arquitectònica: *Fixed Protection* (FP), que protegeix bits consecutius, i *Variable Protection* (VP), que amplia la cobertura a bits invariants no consecutius.

S'han dut a terme experiments extensius d'injecció de fallades, consistents en canvis aleatoris de bits als pesos, amb l'objectiu de quantificar la sensibilitat del model i el seu impacte en la precisió de la inferència. Els models sense protecció presenten, en general, una degradació de la precisió compresa entre 1.3% i més del 3%. Amb FP i VP, tanmateix, les fallades de tipo bit flip es mitiguen de manera substancial, fins a valors entre 0.0001% i 0.4%, la qual cosa suposa una millora de diversos ordres de magnitud. Ambdós mecanismes estan dissenyats per ser computacionalment eficients i portables, i es poden aplicar tant a models convolucionals com a models no convolucionals.

Finalment, s'implementa la integració pràctica dels nostres mecanismes en GPUs i en un accelerador basat en FPGA, validant l'ùs dels mecanismes de protecció en entorns d'execució reals. La implementació en GPU aprofita NVIDIA CUTLASS i PyTorch, mentre que la implementació en FPGA incorpora mòduls dedicats d'emmascarament en un accelerador en flux sense alterar el *pipeline* de càlcul principal. En conjunt, els resultats mostren que aprofitar la redundància a nivell de bit constitueix un enfocament eficaç, eficient i general per augmentar la tolerància a fallades de les xarxes neuronals modernes.

Chapter 1

Introduction

Artificial Intelligence (AI) was formally introduced as a research field at the Dartmouth Conference in 1956 [78, 123]; however, its practical impact remained limited for decades due to constraints in hardware, data availability, and algorithmic development. In contrast, the last decade has witnessed a decisive resurgence of AI, driven by advances in computing architectures, the availability of large-scale datasets, and methodological breakthroughs in machine learning and deep learning [63]. The growing volume of people using AI tools (see Figure 1.1) is a consequence of unprecedented advancements in computer vision, natural language processing, robotics, and other domains.

Due to the extensive use of AI and the growing reliance on its applications in real-world environments, it is increasingly important to design AI systems that behave robustly. In particular, when AI is deployed in products with functional safety requirements, these systems must include fault protection mechanisms to ensure the system's safety. For instance, in the automotive domain, failure rates for safety-critical systems are defined in standards such as ISO 26262 [105]. To achieve low rates, protective measures should be implemented in the neural network models, as well as on the platforms that host them. This includes protection against transient soft errors arising from memory upsets, radiation-induced soft errors, internal hardware degradation, and voltage or timing disturbances, manifesting as single or multiple-bit upsets.

Depending on the application domain, such faults may be more or less critical. This doctoral thesis is motivated by precisely this challenge: we investigate the impact of

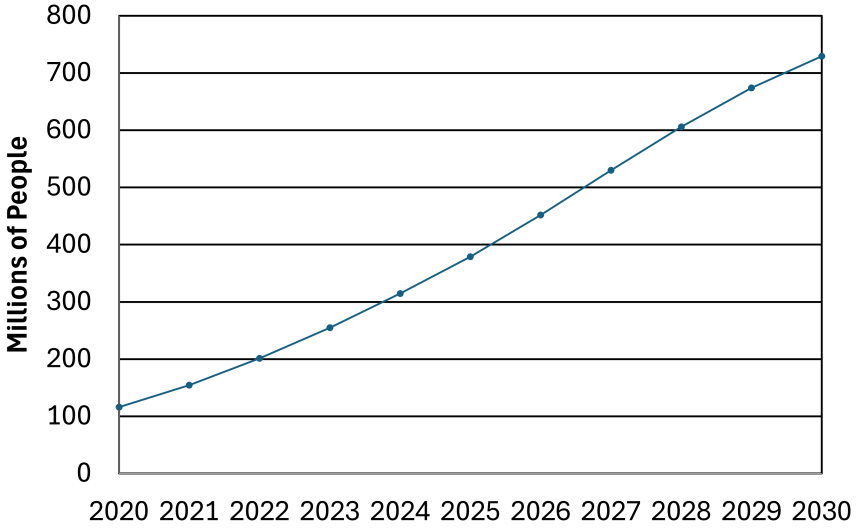


FIGURE 1.1: Number of people using AI tools worldwide. Source [25].

faults in neural network models and, building on the structural properties of the models, we propose lightweight software-based protection mechanisms to mitigate these faults and restore accuracy during inference. We begin our research focusing on Convolutional Neural Networks [127], and we extend our findings to other neural network model architectures.

1.0.1 Neural Networks

AI has transitioned from being an academic pursuit to becoming an integral component of everyday life. Applications now range, among others, from virtual assistants and recommendation systems to predictive analytics, clinical diagnostics, and autonomous driving [103]. To that purpose, neural networks(NN) [131], especially Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) [93], are mainly the models enabling such integration, although these networks require a high volume of mathematical operations.

CNNs have become the standard for visual tasks due to their ability to extract and hierarchically compose spatial features via convolutional filters, enabling state-of-the-art performance in image classification, detection, and segmentation. Other paradigms utilise RNNs and their variants, originally designed for sequential data, and more recently, *Transformer*-based models, which, by employing attention mechanisms, have reshaped

language modelling and are increasingly applied to vision (e.g., Vision Transformers, ViTs [133]).

These models are trained by optimising millions of parameters on large datasets using gradient-based techniques, and then deployed for inference in real-world conditions on real-world images. The inference workloads vary depending on the application and deployment constraints, ranging from cloud servers with high-throughput capabilities to embedded platforms and edge devices with tighter energy and latency budgets. In all cases, the operational viability of these models depends on the reliability of the underlying computations and the integrity of the model parameters.

1.0.2 Faults in Neural Networks

As commented, reliability is critical and NNs are vulnerable to various faults. Bit-level errors and hardware malfunctions can significantly degrade model performance or lead to incorrect predictions [99].

Indeed, once a neural network model is trained and optimised, its accuracy can degrade in real-world deployment due to faults in either the model parameters *(weight faults)* or intermediate neuron states *(neuron faults)*. These faults, often caused by phenomena such as bit flips, can severely degrade the model's performance [48].

Scenarios such as mistaking an animal for a vehicle in autonomous driving [66][46] may occur, demanding the use of protection mechanisms. For such reasons, fault tolerance is an imperative area of research, especially as mentioned, in safety-critical systems like those cited autonomous driving or healthcare. Mechanisms to detect, mitigate, and recover from such faults are essential to ensuring the reliable deployment of neural network models in real-world environments.

In the present thesis, we concentrate on bit-level faults affecting model parameters, principally the weights.

1.0.3 Compute Platforms: GPUs and FPGAs

The training and inference of NNs typically rely on different compute platforms and runtime environments in order to optimise both latency and energy consumption. In

order to facilitate this process, it is necessary to have access to hardware platforms that are both capable and scalable.

The evolution of computing has provided the foundation for the emergence of AI [119, 135]. For many years, the concept of device scaling was predominantly influenced by two theoretical frameworks: Moore's Law [79] and Dennard scaling [22]. Despite the fact that these trends have decelerated due to physical, thermal, and economic constraints, the demand for performance has continued to escalate, particularly in relation to NN workloads. In response to these challenges, highly parallel architectures have emerged as the predominant approach for deployment.

Initially developed for the purpose of accelerating graphics rendering, *Graphics Processing Units* (GPUs) have demonstrated exceptional capabilities in general-purpose parallel computing. The massively parallel execution model of the aforementioned subjects has been proven to be of significant importance in modern AI computation. Indeed, GPUs are predominant in computing facilities for training and at scale [82].

In contrast, *Field-Programmable Gate Arrays* (FPGAs) [4, 129] provide reconfigurable hardware that is adaptable to the specific requirements of the application. This facilitates the customisation of data paths and control flow with a high degree of precision, frequently resulting in a notable enhancement in energy efficiency compared to fixed architectures for specific workloads [43]. As a result, while GPUs typically prevail in raw throughput, FPGAs can offer a more favourable balance of compute and energy for certain deployment scenarios.

In the context of this thesis, both platforms serve complementary roles. GPUs provide a mature software ecosystem (e.g., CUDA libraries and deep-learning frameworks) for rapid prototyping and large-scale evaluation, whereas FPGAs enable the integration of our protection mechanisms within a streaming accelerator without altering the core compute pipeline. Therefore, we implement and evaluate our contributions on two such platforms: GPUs and FPGAs.

1.1 Objectives and Motivation of the Thesis

This thesis aims to investigate and develop lightweight, fault-tolerant mechanisms for deep neural networks (DNNs). The focus is on mitigating the impact of random, hardware-induced bit flip errors in model weights stored in memory components such as DDR and caches memories. Such faults, which may be caused by external factors such as radiation or internal issues like hardware ageing, pose a significant threat to system reliability, particularly in safety-critical applications.

To this end, the research will begin with an analysis of the robustness of state-of-the-art NN architectures, specifically CNNs and ViTs, under fault injection scenarios. The robustness of NN models will be assessed by measuring the degradation in accuracy caused by the presence of errors. The study will encompass two widely used numerical formats for representing model parameters: 32-bit floating-point (FP32) and 8-bit quantised integers (INT8). FP32 is traditionally used due to its high precision and flexibility, making it suitable for training and fine-tuning models. However, it requires substantial memory and computational resources. In contrast, INT8 quantisation is commonly adopted to reduce model size and improve execution speed, particularly on resource-constrained devices. Although quantisation introduces a reduction in numerical precision, it often preserves model performance and is highly effective for inference tasks. Nonetheless, this thesis will explore how bit-level faults impact reduced-precision representations of NNs.

The research in this thesis will also aim to identify the specific layers and weights within these networks that are most vulnerable to bit flip errors, and to discern any underlying patterns or correlations. Special attention will be given to convolutional layers due to their characteristic weight-sharing mechanisms across feature maps, which can cause single-bit errors to propagate and amplify their effects during inference.

Building on this analysis, two lightweight fault protection mechanisms are proposed. Both approaches aim to improve the fault detection and mitigation capabilities of NNs while maintaining their adaptability across various architectures. These mechanisms will exploit the inherent redundancy present in DNNs' weights.

To validate the real-world applicability and effectiveness of the proposed techniques, they are implemented and evaluated on commonly used execution platforms, including

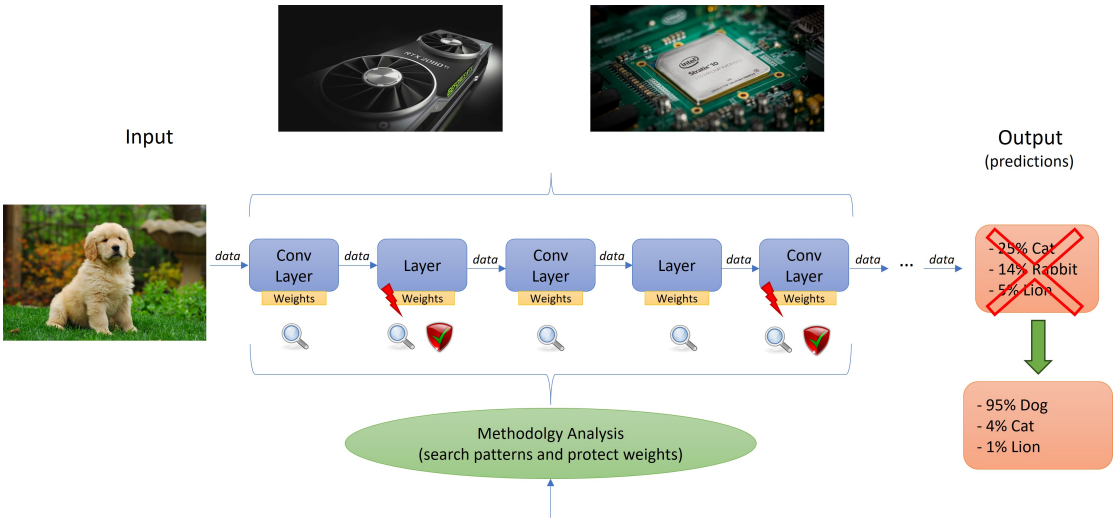


FIGURE 1.2: Diagram showing the main contributions to fault tolerance in NN.

an FPGA-based Accelerator and a Graphics Processing Unit (GPU) with an NVIDIA OpenCore framework. The evaluation will consider critical factors, such as computational overhead, memory consumption, and inference latency, to determine whether the introduced fault-tolerance strategies incur acceptable trade-offs in performance.

A key goal of this thesis is to cultivate the expertise required to analyse, manipulate, and enhance neural network models in the context of reliability and robustness. This includes the development of dedicated software tools for implementing and evaluating fault-tolerance mechanisms, thereby contributing to the broader scientific endeavour of building dependable and resilient artificial intelligence systems.

Figure 1.2 resumes the key objectives of this thesis and its accomplishments. This schematic diagram outlines the use case addressed in this thesis. It illustrates how inference on an image is performed by a NN (deployable on GPUs or FPGAs) comprising multiple convolutional "Conv" layers. In two of these layers, faults are present in the weights; consequently, the affected weights must be identified and corrected to ensure that the final prediction is accurate.

1.2 Main Contributions of the Thesis

The major contributions of this thesis are described below:

- The analysis of bit-level redundancy on CNNs, which is the foundation of our fault-tolerant mechanism.
- The analysis of both 32-bit floating-point models and corresponding quantised INT8 models, showing the higher robustness to bit flip faults of quantised models, albeit they achieve lower performance (accuracy).
- The provision of a study of the impact of faults on current state-of-the-art CNN and ViT models, showing the impact of single-bit flip faults on the model's Top1 and Top5 accuracy and proving if such findings can be extended to other NN architectures.
- The proposal of two fault-tolerant mechanisms, both taking advantage of the bit-level redundancy found on model weights. Both mechanisms show effective fault protection and recovery.
- An implementation on an FPGA-based system, obtaining negligible overheads and no performance impact.
- An implementation on NVIDIA GPUs using the open-source CUTLASS library [18], demonstrating their integration in real-world GPU-accelerated inference pipelines.

1.3 Thesis Outline

This thesis is organised as follows. First, in Chapter 1 we introduce the thesis, outlining its motivation, objectives, and main contributions. Then, in Chapter 2 we review background concepts and related work on CNNs, fault types, and fault-tolerance techniques, setting the foundation for the proposed contributions. Next, in Chapter 3 we examine the bit-level redundancy present across channels and filters in NN weights, using several grouping strategies. This analysis forms the foundation for introducing the fault-tolerance mechanisms investigated in this thesis. In Chapter 4, we present and evaluate the proposed mechanisms through fault injection experiments on various CNNs and vision transformers, analysing their impact on model accuracy and comparing them with other protection schemes. Then, in Chapter 5 we detail the hardware and software implementation of the mechanism on FPGAs and GPUs. Finally, in Chapter 6

we conclude the thesis, summarising key results, highlighting limitations, and outlining directions for future work and publications.

Chapter 2

Background and Related Work

This chapter provides a comprehensive review of the latest achievements and most pertinent contributions related to the subject of this thesis, together with an overview of how to tolerate errors in them.

First, a comprehensive overview of foundational and contemporary work concerning NNs is provided, including their fundamental components, architectural layers, and their core operational stages: training and evaluation.

Second, an examination of existing research focused on fault tolerance within NNs is conducted. This includes a survey of various mechanisms, architectural designs, and methodologies designed to enable fault detection and mitigation.

2.1 Background

2.1.1 History of NNs

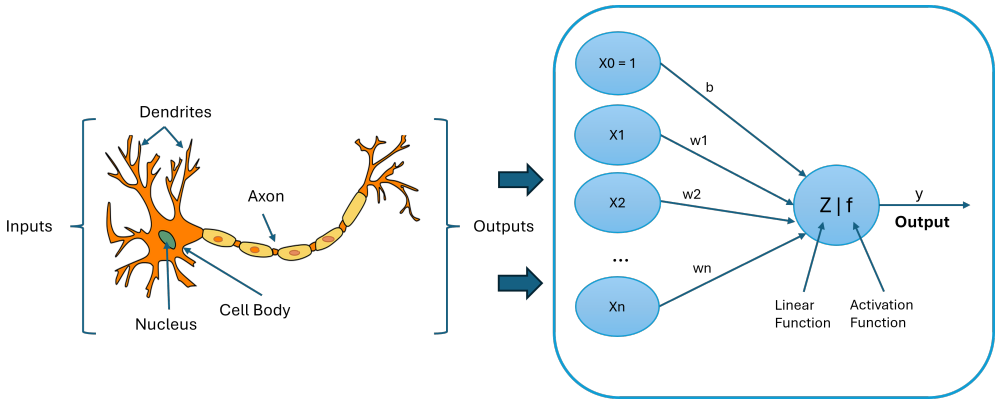


FIGURE 2.1: Biological neuron compared to an artificial neuron in a neural network.

NNs [27] are a class of machine learning models inspired by how the human brain works. They are composed of multiple layers of interconnected processing units, commonly referred to as artificial neurons (see Figure 2.1), which process information in a manner analogous to biological neurons by transmitting and transforming signals.

Each artificial neuron receives one or more inputs, computes a weighted sum, adds a bias term [126], and applies a non-linear activation function [125] to produce an output. This output is then transmitted to the next layer in the network. The activation function introduces non-linearity, which is crucial for enabling the network to capture and model complex relationships and patterns in the data.

NNs have their origins in the fields of artificial intelligence (AI) [26, 124] and machine learning. Initially conceptualised in the mid-20th century, modern NNs have grown significantly in complexity, fueled by advances in computational resources, particularly through the utilisation of Graphics Processing Units (GPUs) [86, 130]. GPUs provide a high degree of parallelism at a relatively low cost, making them ideal for the large-scale training requirements of NNs. These networks typically learn from vast datasets, often stored and processed in warehouse-scale computers (WSCs) [8], which manage and retrieve petabytes of information from the internet and other sources.

A prominent subset of NNs are DNNs [128], which comprise multiple hidden layers between the input and output. DNNs are particularly effective in domains such as image classification, natural language processing, and speech recognition. Within this category, CNNs [127] represent a specialised architecture optimised for spatial data such as images and videos. CNNs apply the convolution operation to input data to identify local features, enabling hierarchical feature extraction across multiple layers. These networks exhibit an impressive capacity to extract and model intricate data patterns through their layered architectures and learning algorithms. As a result, they are widely deployed across a range of real-world applications with remarkable precision and efficiency. Another salient sub-family, also examined in this thesis, is the Vision Transformer (ViT), which presents a different architecture that partitions an image into fixed-size patches and processes them as a sequence using a Transformer encoder [133].

In the following subsections, we will explore the several types of NNs models, training methodologies, specialised types such as CNNs, and the practical use cases that highlight their versatility and impact.

2.1.2 Types of NNs

Various neural network architectures have been developed to address different categories of problems:

- **Multilayer Perceptrons (MLPs):** These are the foundational form of feed-forward [14, 37] NNs, consisting of fully connected layers. MLPs are suitable for problems where input features are structured and do not exhibit spatial or temporal dependencies [93].
- **CNNs:** Designed for visual and spatial data, CNNs employ convolutional layers to exploit local spatial correlations in the input data. They are particularly effective in image classification, object detection, and video analysis [100][122].
- **RNNs:** Equipped with memory elements, RNNs capture temporal dependencies in sequential data, making them suitable for time series forecasting, speech recognition, and natural language processing [93].

- **ViTs:** A recent innovation, ViTs abandon convolutions in favour of self-attention mechanisms. They have demonstrated strong performance in vision tasks by treating images as sequences of patches, analogous to words in natural language processing [23]. Moreover, ViTs can also contain MLP layers, as we show in the following chapter.

- **Large Language Models (LLMs):** Constitute a class of deep learning models that are trained on extensive canons of text to perform a wide range of natural language processing (NLP) tasks, including text generation, translation, summarisation, and question answering. These models, which are based on transformer architectures, learn statistical patterns in language through unsupervised or self-supervised training on datasets comprising billions of words. The emergence of these systems marks a significant milestone in the evolution of artificial intelligence, with implications for research, industry, and society [15].

Each architecture has its strengths and limitations, and the choice depends heavily on the nature of the input data and the task to be performed.

2.1.3 Structure and Functioning

A neural network is composed of three primary layers. These layers are the input layer, one or more hidden layers, and the output layer (a didactic diagram can be seen in Figure 2.2). The input layer receives raw data, while the hidden layers perform computations to extract features and patterns from the input data. Finally, the output layer produces the final prediction or classification. This layered architecture facilitates NNs in modelling intricate relationships in data, rendering them highly efficacious for tasks such as image and speech recognition, natural language processing, and predictive analytics.

The input layer is responsible for receiving the initial data that feeds into the neural network. Each neuron in this layer represents a specific feature of the input data. To illustrate this point, consider a 28x28 pixel image, where the input layer would comprise 784 neurons, with each neuron corresponding to a single pixel.

The presence of hidden layers, situated between the input and output layers, is of crucial importance for the network's learning capabilities. These layers are responsible for

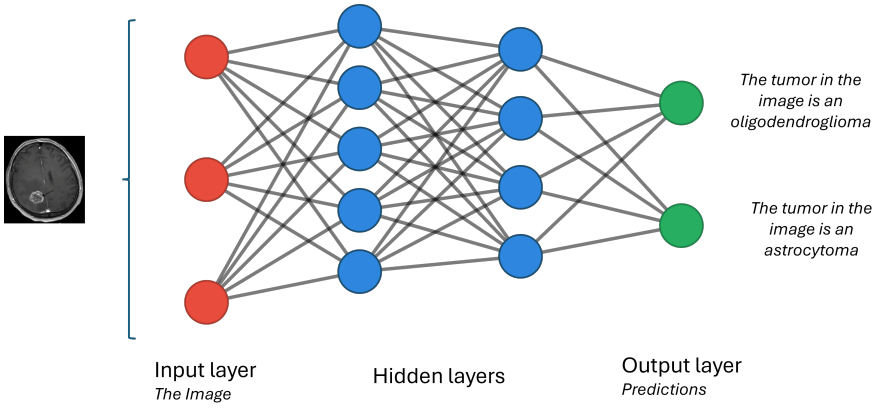


FIGURE 2.2: NN diagram.

implementing non-linear transformations of the inputs, thereby enabling the network to acquire sophisticated representations of the data. The following are examples of the various types of hidden layers:

- **Dense (Fully Connected) Layers**: In these layers, each neuron is connected to every neuron in the preceding and succeeding layers. Dense layers are common in artificial NNs and are effective for capturing global relationships in data [21].
- **Convolutional Layers**: Primarily used in CNNs, these layers apply filters to detect local features such as edges and patterns in images. Each filter slides over the input to produce an activation map that highlights the presence of specific features [20].
- **Pooling Layers**: These layers reduce the dimensionality of activation maps, retaining the most important information. Common methods include max pooling, which selects the maximum value in a region, and average pooling, which computes the average [33].
- **Normalisation Layers**: These layers help accelerate training and improve network stability. Batch normalisation is a common technique that normalises the outputs of a previous layer by subtracting the batch mean and dividing by the batch standard deviation [32].
- **Self-attention layers**: ViT models also feature special layers that are only present in these models. These are the multi-head self-attention layers, which allow them to

capture global relationships in images from the earliest stages of processing, unlike CNNs, which focus on local features through convolutional layers. This feature derives from their transform-based architecture, originally developed for language processing and adapted for computer vision tasks. Research, such as [23], confirms this distinction and highlights the importance of such layers for performance in vision tasks.

The function of the output layer is to provide the final result of the neural network's processing. The structure of the network is determined by the specific task at hand. In the context of binary classification, a single neuron with a sigmoid activation function is typically employed. Conversely, for multi-class classification, a softmax [132] function is utilised to generate a probability distribution over potential classes.

The combination of these layer types enables NNs to model complex data relationships, rendering them powerful tools for a wide range of applications in artificial intelligence.

2.1.4 Training and Inference

The training of an NN is key to its subsequent utilisation. The process necessitates the iterative adjustment of weights until suitable values are identified. Whilst the primary focus of this thesis is on the principles of inference, testing and protecting the NNs and their weights, it is also important to understand the training process by which network parameters are determined.

Training a neural network entails the optimisation of its internal parameters, specifically, the weights and biases, to minimise the discrepancy between the predicted outputs and the actual target values. This optimisation is achieved through iterative processes, commonly employing the back propagation algorithm in conjunction with stochastic gradient descent (SGD) or its variants, such as Adam or RMSprop [2, 38].

The development process is initiated with the delineation of the network's architecture, encompassing the number and categories of layers, the quantity of neurons per layer, activation functions, and the dimensions of the input data. Frequently, established architectures are adopted or adapted for specific tasks, leveraging prior successful implementations.

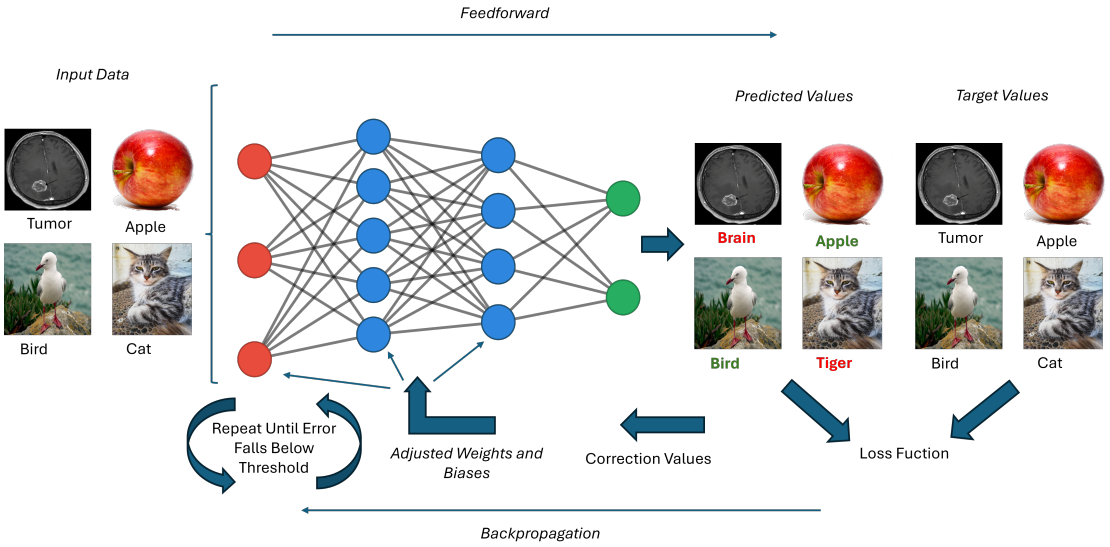


FIGURE 2.3: Forwarding and back propagation functions.

The training process involves adjusting the network's parameters to align with the training data. This process can encompass a range of parameters extending from thousands to hundreds of millions. This stage is computationally intensive, with the duration depending on the complexity of the model and the size of the dataset. The duration can range from several hours to weeks.

Upon completion of the training process, the model transitions into the inference phase. During this phase, the model is deployed to process new, previously unseen data and generate predictions. Inference is optimised for speed and efficiency, typically occurring within milliseconds to seconds.

In the context of supervised learning, the network is trained on a labelled dataset. A well-known example is the ImageNet dataset, which comprises over 1.2 million annotated images across 1,000 categories. The training process involves a forward pass, in which the network makes a prediction, followed by a backward pass (back propagation) that updates the weights based on the prediction error [38] (see Figure 2.3).

Mathematically, the objective is to identify a function $f : X \rightarrow Y$ that accurately maps inputs X to their corresponding output Y over a multi-layer neural architecture. Back propagation computes the gradient of the loss function with respect to each weight, and SGD adjusts the weights in the direction that most effectively reduces the loss [2].

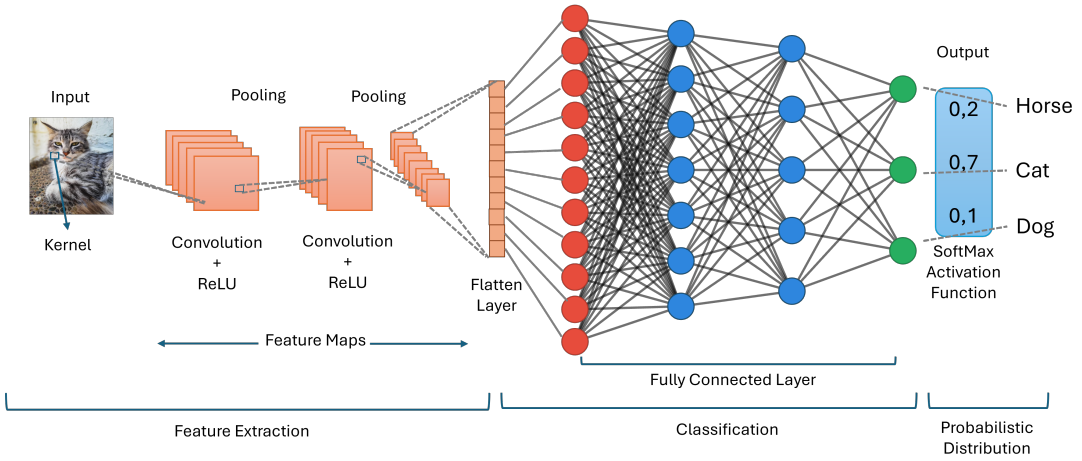


FIGURE 2.4: CNN structure.

2.1.5 Convolutional Networks

As the primary type of network employed in the context of this thesis, we describe the main characteristics of CNNs. These networks will serve as a basis for developing our mechanisms and are highly effective architectures for processing grid-like data, such as images. CNNs comprise multiple convolutional layers, each designed to detect specific local features through learnable filters or kernels.

A convolutional layer receives as input a multi-channel image (e.g., RGB channels) and applies a set of filters that scan local neighbourhoods of pixels. These filters extract features such as edges, corners, and textures. As data passes through successive convolutional layers, the network learns increasingly abstract representations: from low-level features like edges to more complex structures such as shapes and object parts [134].

Each convolution operation results in a two-dimensional feature map, with the number of channels in the output increasing with network depth, often reaching into the hundreds. This increase enables the network to capture a richer set of features at higher levels of abstraction. An overview of the process can be seen in Figure 2.4.

Pooling layers, such as max-pooling, are often included to downsample feature maps, reducing computational complexity and mitigating overfitting by enforcing translational invariance [34].

Finally, fully connected layers aggregate the spatially distributed features into a final classification output. These layers function similarly to MLPs, connecting every neuron in the previous layer to each of the output classes.

One principal advantage of CNNs over traditional MLPs lies in their parameter efficiency. By sharing weights across space and employing local connections, CNNs significantly reduce the number of parameters required, enabling the development of deeper and more complex models without prohibitive both computational and memory costs [134].

2.1.6 Hardware Faults Affecting NNs

Apart from describing the principal properties and types of NNs, it is also important to review the background of some faults that may occur in NNs.

Hardware-level faults may arise randomly from natural sources such as radiation-induced soft errors, manufacturing anomalies, or ageing. Hardware faults can indirectly affect the NN model executed on the underlying platform, manifesting incorrect model predictions in the worst-case scenario. The principal classes of hardware faults are summarised below.

First, there are transient or "soft" errors caused by radiation or electrical noise. In this case, the device is not physically damaged; rather, its logical state is temporarily altered (e.g., bit flips in SRAM/DRAM cells or sequential elements) due to particle strikes (such as cosmic-ray neutrons [138]) or electrical noise. The most relevant phenomena are *single-event upsets* (SEUs), which flip the content of a single cell (SRAM/DRAM/flip-flop), and *multi-bit upsets* (MBUs), which affect multiple nearby cells [16]. Soft errors have a higher impact on memory cells and registers since short-term perturbations in combinatorial logic are usually filtered out (not registered) or masked.

Second, there are permanent or "hard" faults. These stem from physical degradation mechanisms and stress over time (ageing), as well as manufacturing defects, and appear as stuck-at faults, path/timing faults, or defective cells [54]. Additionally, the utilisation of low-voltage supplies to reduce the power consumption of power devices has a significant impact on memory devices, manifesting as a permanent error in the form of stuck-at-1 or stuck-at-0 in memory cell bits [104].

In the context of this thesis, our fault models utilise bit flips to simulate the effects of both soft and hard hardware faults in memory devices where the NN model parameters are stored.

2.1.7 Security Attacks Exploiting Fault-Injection in NNs

Although this thesis concentrates on bit-level random hardware faults affecting the NN models, faults can also be injected intentionally. In this section, we summarise the main attack classes and how they impact NN models:

- **Input attacks:** modify pixels/features at the input to force misclassification (typically transient).
- **Weight attacks:** flip bits in model parameters; their effect persists until repaired (the primary target of this thesis).
- **Memory attacks:** induce *bit flips* in DRAM or other memory hierarchy levels or data paths where weights or activations reside.

Additionally, the execution of these attacks can be achieved through a variety of bit flip attack (BFA) techniques:

1. **Rowhammer Attack:** exploits the physical vulnerability of DRAM by repeatedly activating certain memory rows, causing adjacent rows' bits to flip. It has been demonstrated that Rowhammer can be carried out remotely, making it particularly dangerous [60].
2. **Voltage Frequency Scaling (VFS) Attack:** manipulates the voltage and operating frequency of memory systems or devices running the NN models to induce bit errors. Lowering the voltage increases the likelihood of bit-level faults due to reduced circuit stability [11].
3. **Clock Glitching Attack:** introduces additional clock pulses or alters existing ones to induce instruction-execution errors. Such disruptions can lead to data corruption and, ultimately, bit flips [75].

4. **Laser Injection Attack:** employs high-precision lasers to induce faults in specific bits within memory systems. While highly accurate, it requires physical access and specialised equipment [24].

Rowhammer remains the most commonly employed technique, largely due to its widespread applicability and effectiveness in inducing DRAM bit flips. VFS and Clock Glitching are more appropriate in scenarios where the attacker has limited physical access and must induce errors more subtly. Finally, Laser Injection offers the greatest precision, but its requirement for specialised hardware and physical proximity confines its use to niche environments.

2.2 Related Work in Fault-Tolerant NNs

2.2.1 Impact of Faults on NNs

NNs may be affected by different faults whose impact can manifest during both training and inference. In practical terms, these effects are typically quantified by the degradation in model accuracy and, in the specific case of CNNs or ViTs trained on image datasets, by the proportion of images that become misclassified.

From a functional standpoint, the most relevant fault types are:

- **Weight faults:** persistently alter model parameters. A single bit flip in a high-significant position (e.g. the exponent in FP32) can severely change the numerical value, thereby amplifying the loss of accuracy and increasing the number of images mispredicted.
- **Neuron/activation faults:** corrupt activations during forward propagation; although often transient, their effect may propagate layer by layer, compounding the damage.
- **Input faults:** perturb the original image or feature vector and, despite typically being ephemeral, can still change the prediction outcome.

The impact and consequences of each of these fault types are documented, for example, in [112], which shows that even a single-pixel modification can induce misclassification, evidencing the sensitivity of the input. In [35], the effects of faults on the weights of convolutional layers across different models, as well as on activations, are examined; similar observations are reported in [46]. Both studies report misclassification rates of up to 5% for weight faults and up to 11.8% for neuron faults under a single one-bit flip. This thesis focuses on weight faults, as weights are stored in memory devices that are more likely to exhibit faulty behaviour. In addition, reliability analyses with fault injection on CNNs for safety-critical domains (e.g., automotive, as commented before) confirm that permanent faults can induce non-negligible accuracy losses and reveal layer-operation-dependent vulnerability [10]

2.2.2 Fault-Tolerance Mechanisms

Within the state-of-the-art, a broad spectrum of solutions exists that not only mitigates the effects and consequences of faults in NNs but also addresses their root causes, which, as discussed, often originate in the underlying hardware. For clarity, we classify these techniques into three categories: *hardware*, *software*, and *algorithmic/model-level*.

2.2.2.1 Hardware Mechanisms

Hardware-based mechanisms for fault tolerance primarily centre on replicating either selected components or entire subsystems to varying degrees. Modular redundancy, in its dual or triple forms (DMR/TMR), replicates compute units and applies majority voting to determine the valid outcome [69]. While effective, this strategy incurs significant area and power overheads; moreover, as platform complexity grows (i.e., the number of modules increases), a larger fraction of components becomes critical and potentially replicable. As an alternative, *selective redundancy* protects only the most critical components of the design, thereby reducing overhead [41]. Selective TMR for quantised NN accelerators triplicates only the most *sensitive* output channels (identified via error injection) in convolutional layers of FINN-based FPGA accelerators [120], and applies a per-triplet majority voter (TMRC) [9].

At the architectural level, we can observe that Hybrid/reconfigurable accelerator designs can adapt to faulty resources at runtime. The HyCA approach augments systolic arrays with dot-product units to recompute operations mapped onto faulty PEs, thereby improving fault coverage and scalability compared to localised redundancy [136]. Complementarily, a reconfigurable CNN accelerator exposes distinct FT/HP (high-performance)/DS (de-stress) modes to trade performance for reliability on FPGAs [113].

At the memory level, ECC/CRC, such as Hamming codes or Cyclic Redundancy Code (CRC) [39, 61], are another option for memory protection, though they introduce computational and area costs [81]. Moreover, their protection capability is bound by the code distance. *In the specific context of DNNs*, several works demonstrate practical ECC applications: *value-aware parity insertion* stores parity bits by replacing less critical bits in 8-bit weights to recover errors with minimal storage and no retraining [64]; *functional error correction* selectively protects the most important bits using ECC codes and correct faults in weights with some extra overheads [50]; and recent NN-specific ECC schemes embed codes into weight memories or co-design coding with training to increase tolerance to bit flips [1, 98]. At the same time, studies on GPUs running DNNs report that conventional device-level ECC may not reduce the proportion of *critical* DNN errors, underscoring the need for NN-aware protection [107].

Beyond ECC and architectural replication, non-redundant microarchitectural techniques have been proposed. *Gated-CNN* mitigates NBTI/HCI [58] ageing in on-chip *activation* SRAMs of CNN accelerators by gating activity and balancing stress, improving reliability without replication [80]. *Flip-and-Patch* targets permanent faults due to aggressive voltage underscaling in on-chip activation memories, maintaining baseline accuracy with negligible performance impact [117]. Related studies further explore fault patterns and low-cost protections under underscaling and permanent defects [116, 121].

Other works have examined error resilience at the transistor level [114], proposed 3D die-stacked memory for fault tolerance in CNNs [59], or introduced redundant circuits to enforce data resilience [66]. While these approaches are effective, they come at a high cost in terms of memory and hardware resources. Our solution aims to offer protection with minimal overhead without the need of specific hardware support, ensuring the practical deployment of CNNs in resource-constrained environments.

2.2.2.2 Software Mechanisms

Software-based techniques for fault tolerance in NNs either harden the training process or detect/repair numerical faults during inference. Importantly, these methods can operate independently of the specific hardware state and are therefore portable across platforms. They may be grouped as follows:

- **Training-time hardening:** e.g., binarisation or constrained training to increase robustness [45]; these approaches may degrade baseline accuracy. A prominent line is *fault-aware training (FAT)* from Xilinx, which injects representative hardware faults during training (including layer- or operator-sensitive training and mixed-precision choices) to obtain NNs significantly more tolerant to errors, with improved worst-case accuracy at low cost. It is also reported to have higher sensitivity in convolutional layers than in fully-connected ones [137]. Subsequent radiation tests corroborate the benefits of FAT on quantised models [30] and also improve resilience under realistic fault models [108].

- **Inference-time repair:** e.g., in [67], the authors reconstruct model weights locally during inference, yet this approach also carries the risk of accuracy degradation. Another proposal, [57], suggests backing up MSB bits in LSB positions to detect faulty weights, though this leaves LSB bits unprotected. Similarly, [71] introduces a bit rotation mechanism to obfuscate the MSB bits, which, while innovative, remains vulnerable to random bit flips due to the lack of comprehensive bit-level protection. In quantised accelerators, selective channel replication and fault-aware PE scheduling have shown substantial robustness improvements at significantly lower overhead than full TMR [29]. *Selective kernel/channel duplication* is presented with FT-DeepNets, which proposes to duplicate only the most critical kernels/channels, ranked via weight-sum, second-order (Hessian/gradient-based) and entropy criteria, and to recover either by averaging duplicated *kernels* (KR) or by averaging duplicated *feature maps* (FMR). [7]

- **Machine learning-based error detection:** has also been explored, where a smaller "B" network checks the output of the original "A" network [70]. Firstly, an architecture exploration is performed, which generates a pool of compressed

candidates of the task model (then trained with knowledge distillation to maximise consistency with the original) and selects the design giving the best fault-coverage against overhead. Secondly, a task exploration simplifies the output comparison by selecting easily confused classes under a critical criterion based on three metrics: risk impact, risk probability and inconsistency. However, while this mechanism can identify discrepancies between the two networks, it requires the additional overhead of training and deploying a second model, which may not be feasible in all scenarios.

- **Checksums/Hashes:** Checksum-based protection for weights has also been introduced [68]. In these cases, the larger the group of weights, the higher the detection accuracy; however, it will also increase the false-negative rate, as checksums may fail to detect bit flips in certain positions. Therefore, the efficacy of this approach depends on the size of the weight group being protected. Similarly, hash functions have been applied to specific layers to detect faults by comparing actual outputs with a fault-free reference [53]. While this method is effective at detection, it does not correct errors.

- **Codes:** Several studies are focused on the use of software encodings as a means of protecting against fault injection attacks and on the relationship between such encodings and side-channel resistance. In [12], theoretical limits and trade-offs between the two resistances are established, and it is demonstrated through simulation under typical fault models how a search algorithm enables codes to be designed that achieve an optimal balance between protection against faults and side-channel concealment. In contrast, in [13], an evaluation metric is developed to determine the robustness of a software encoding scheme against bit flips and instruction skips. Furthermore, a code selection mechanism based on user criteria and a dynamic code analysis method are provided to estimate the level of protection with the encoded schemes.

2.2.2.3 Algorithmic/Model-Level Solutions

These mechanisms comprise a representative set of methods that *bound or restore layer outputs* when out-of-range values are detected (e.g., Ranger, Clipper, Fmap-Rescale,

FmapAvg, BackFlip) [35]. Such techniques detect anomalous activations and clip, rescale, or reconstruct them, thereby mitigating the propagation of faults. Their effectiveness depends on threshold selection and the activation distribution, and they tend to be primarily detective or partially corrective.

Weight-sensitivity strategies [72] identify critical layers/parameters and apply value or operator duplication with a lower overhead than TMR, but still with non-negligible costs with model-specific calibration requirements. Complementing these approaches, [115] introduce *Flash-ABFT*, a custom algorithm-based scheme tailored to Transformer attention layers: it performs a single online checksum over the full computation, including the softmax step, thereby removing redundant checks while preserving high error-detection coverage and incurring markedly lower overhead.

Model/algorithm co-design for resilience shows several approaches: (i) *Safe overclocking* for CNN accelerators leverages algorithm-level error detection to unlock voltage/frequency margins while bounding accuracy impact [77]. (ii) *Fault-tolerant NN architectures* at model level can mitigate weight disturbances without expensive retraining [74]. (iii) Cross-layer scheme design explicitly targeting accelerator faults has also been explored from the application side, using CNNs themselves to analyse and alleviate failures in accelerator systems [110].

2.2.2.3.1 Positioning of Our Approach

Unlike threshold-based techniques [35] or more generic coding schemes as seen in [12, 13], our proposal exploits *bit-level redundancy* within groups of weights and protects invariant positions *without* retraining or architectural duplication. In doing so, it achieves both *detection and recovery* at low overhead in a model-agnostic fashion. Moreover, it is compatible with complementary schemes (e.g., ECC in memory, output clipping), enabling more robust combined defences.

In resource-constrained deployments, a reasonable combination of hardware defences with run-time mechanisms and less intrusive algorithmic protections is particularly attractive. In this thesis, we focus on faults affecting model parameters (primarily weights), whether naturally occurring as a consequence of random hardware faults in memory devices or

induced due to fault-injection attacks targeting memory, and analyse their impact on accuracy and the number of mispredicted images due to the fault.

Chapter 3

Bit-level Invariability Analysis

This chapter discusses the empirical foundations necessary for developing our protective mechanisms. These foundations involve an analysis of bit-level invariability within the weights of neural networks, especially convolutional layers. In particular, we search for the coincidence of bits within groups of elements.

3.1 Neural Network Models to be Analysed

TABLE 3.1: NN model specifications used in our analysis.

Model	Data Type	Top 1 Acc %	Top 5 Acc %
MobileNet v2-1.0	FP32	69.45 %	89.23 %
	INT8	62.43 %	84.54 %
ResNet-50 v2-1.0	FP32	75.01 %	92.38 %
	INT8	69.00 %	88.63 %
VGG-16	FP32	72.38 %	91.02 %
	INT8	69.30 %	88.83 %
DenseNet-121-12	FP32	60.94 %	84.49 %
	INT8	59.92 %	83.64 %
ShuffleNet-v2	FP32	66.38 %	86.58 %
	INT8	58.60 %	80.40 %
ViT-L-16	FP32	88.06 %	98.51 %

In this thesis, we examine neural network models that have been extensively employed in image classification tasks and are available both in 32-bit floating-point (FP32) and 8-bit integer (INT8) formats.

Currently, these data formats represent the most common standards, with FP32 offering greater precision at the cost of higher memory overhead. In contrast, integer formats conserve memory but provide lower precision. Additional data types, such as BrainFloat [56] or FP16, will also be discussed.

These models, excluding ViT-L-16, were acquired from the ONNX Model Zoo repository [90], where detailed references to the original publications and model origins can be found. The ViT-L-16 model was initially obtained from the Torchvision repository [118] and converted to the ONNX format [95]. The key specifications of these models are presented in Table 3.1. In addition, we present the internal structure of these models in Table 3.2, describing the number of convolutions (*convs*) within them and the percentage of convolution layers with respect to the total number of layers in the model. Also, the percentage of parameters belonging to convolutions out of the total number of parameters is shown.

Notably, we should acknowledge the high percentage of parameters present in convolutions. Parameters are the weights of the model, which characterise it and, therefore, the

TABLE 3.2: Structural summary of the analysed models.

Model	**Convs**	**Total Layers**	**% Convs**	**Total Params**	**% Params in Convs**
MobileNet v2-1.0	10 standard +9 DW+9 PW = 28	53	53 %	3.5 M	93 %
ShuffleNet-v2	28 pointwise convolutions	243	12 %	2.3 M	95 %
ResNet-50 v2-1.0	48 convs (16 blocks×3)	50	96 %	25 M	94 %
VGG-16	13	16	81 %	138 M	90 %
DenseNet-121-12	121 (dense blocks)	427	28 %	8.06 M	93 %
ViT-L-16	0	24 SA + MLP	0 %	307 M	0 %

part of the model that is more critical, as any change to them affects the general behaviour of the model. Having more than 90% of parameters in convolution layers across all the CNN models to be analysed means that if we manage to protect these layers, we will tolerate failures in nearly the entire model, so their effectiveness will be maximised. On the other hand, in the case of the ViT model, since it does not have convolutions, the objective is to protect all the layers with parameters that can be grouped, i.e., layers with more than one parameter. Therefore, we will attempt to verify whether the results obtained using the CNN architecture are comparable to those obtained using the ViT architecture.

MobileNet [106] is an efficient model designed for resource-constrained environments. Employing depthwise separable convolutions significantly reduces computational complexity and memory usage, making it particularly well-suited to mobile and embedded applications.

MobileNet employs an initial 3×3 standard convolution followed by *inverted-residual bottleneck* blocks: each block contains a 1×1 expansion pointwise layer (PW), a 3×3 depthwise layer (DW) and a 1×1 projection PW layer, yielding 10 standard convs plus 9 DW+9 PW separable convs (28 conv layers) within approximately 53 layers overall

(including batch normalisation and activation layers). This architecture drastically reduces both FLOPs and parameter count (3.5M total, 93% in convs), making it ideal for resource-constrained devices.

ShuffleNet [76] adopts a similar approach, targeting low-power devices and real-time applications, by implementing grouped convolutions combined with efficient channel shuffling. This allows us to effectively compare its weight structure and evaluate its level of similarity relative to other models.

With ShuffleNet, dense bottlenecks are replaced with pointwise grouped convolutions and a channel-shuffle operation. It comprises roughly 243 layers (including non-convolutional layers), 28 pointwise convolutions spread across four stages, plus channel-shuffle interleaving. Only 12% of the layers are convolutions, yet they account for 95% of the 2.3 million parameters.

Conversely, ResNet50 [44] is distinguished by its skip connections, which facilitate gradient backpropagation and address the gradient vanishing issue [47], a scenario where gradient updates become excessively small, hindering training efficacy. ResNet50 remains widely used for complex image recognition and classification tasks and is a fundamental model in this field.

ResNet-50 begins with a 7×7 convolution layer and a max pooling layer, followed by four stages of {3,4,6,3} blocks. Each block has three convolutions (1×1, 3×3, 1×1), totalling 48 convolutional layers out of 50 layers (96%), with 94% of its 25M parameters allocated to convolutional kernels. The skip connections mitigate vanishing gradients, enabling the training of very deep models.

DenseNet's [49] architecture is densely connected, as its name suggests. Its design promotes efficient reuse of features that have been extracted, while reducing the total number of parameters. This structural innovation enhances precision, particularly in specialised medical imaging and detailed object classification tasks. DenseNet employs four dense blocks, each comprising {6, 12, 24, 16} mixed layers, where a 1×1 convolution is used to reduce channels and a 3×3 convolution to extract features. That gives $6+12+24+16 = 58$ composite layers $\times 2$ conv + 5 transition 1×1 convs = 121 conv modules. These blocks are connected via feature concatenation, yielding a total of 427 layers, 28% of which are convolutional (convs), yet they contain 93% of the 8.06 million parameters.

Lastly, among CNN models, VGG16 [73] provides a simple, uniform architecture based exclusively on sequential 3×3 convolutional filters, ideal for transfer learning and feature extraction tasks, renowned for its capability in robust visual representation. Basically, it is organised into five sequential convolution blocks, two convolutions in the first two blocks and three convolutions in the remaining three, totalling 13 convolution layers, followed by three fully connected layers. Convolutional layers occupy 81% of the 16 weighted layers and account for 90% of the 138M parameters.

Furthermore, we have included a recent model, ViT-L-16 (Vision Transformer) [23], which demonstrates a Top1 accuracy of 88%, considerably higher than traditional CNN models, such as ResNet50, which achieves a maximum accuracy of 75%. ViT-L-16, distinctively devoid of traditional convolutional layers, operates on image patches. It divides an image into 16×16 patches, linearly projects them (patch embedding), then processes them through 24 transformer encoder layers (self-attention "SA" + MLP), totalling 307 M parameters with 0 % in convolutional form. This structure of layers [31] leverages global attention for image understanding.

The shapes of the ViT weights are totally different to convolutional weights. Here, no convolutional kernels are used; only plain vectors or two-dimensional matrix weights are employed. Therefore, this transformer-based structure enables us to evaluate our protective mechanisms in various alternative NN architectures. Despite its extensive parameter count, ViT-L-16 was also explicitly chosen to offer the second-highest accuracy among transformers while maintaining roughly half the weights compared to other transformer models, rendering it computationally viable.

All models were pre-trained on the ImageNet dataset, a widely accepted benchmark for NN performance evaluation [102].

3.2 Methodology

We now present the methodology employed to assess the analysis that will be performed. The design approach adopted to obtain the results will be outlined, beginning with an analysis of neural network weights and the identification of potential patterns within them, which serve as the foundation for the proposed protection mechanisms.

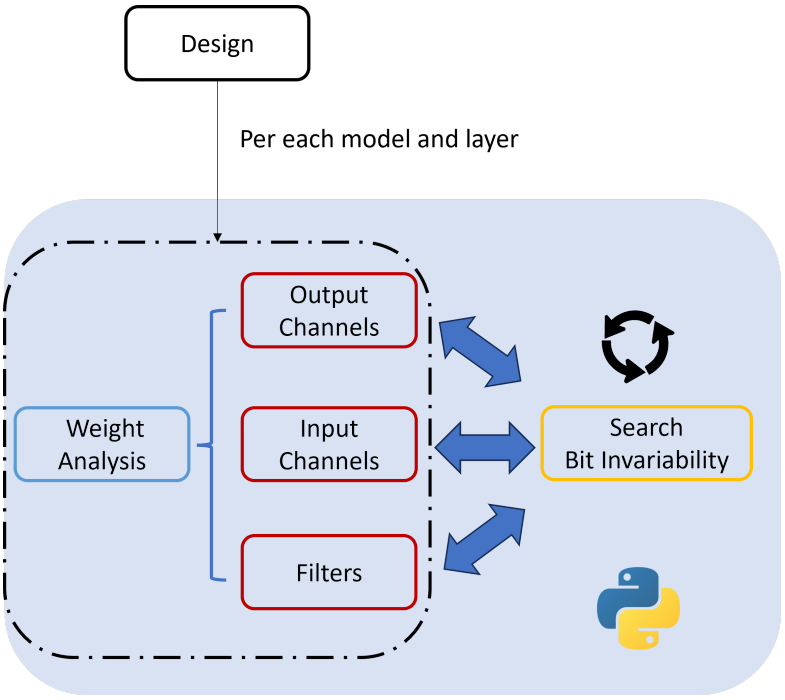


FIGURE 3.1: Analysis flow.

The hypothesis of this thesis is to find the inherent redundancy within the parameters of the NN layers [111] (using as an experimental base convolutional layers) and to show that such redundancy can be exploited to introduce resilience against faults. In other words, we seek stable, repeatable patterns at different levels of organisation (from channels to filters and down to bits) that can serve as a base for protection. In our analysis and results, we refer to this as the invariability of the model.

Figure 3.1 outlines the methodology. We begin with a structural analysis of convolutional layers, where weights are hierarchically organised: output channels encompass input channels, each associated with a specific filter or kernel. We then search for invariability at the bit level across these organisational structures, compare the invariability obtained, and select the one with the highest percentage of detected invariability. We repeat this procedure layer by layer.

A key element of the study is the visualisation of these structures to quantify structural invariability; as we identify such invariability, we enable further protection strategies. Accordingly, we verify the persistence of the identified invariability across all relevant layers to ensure applicability beyond the initial convolutional focus.

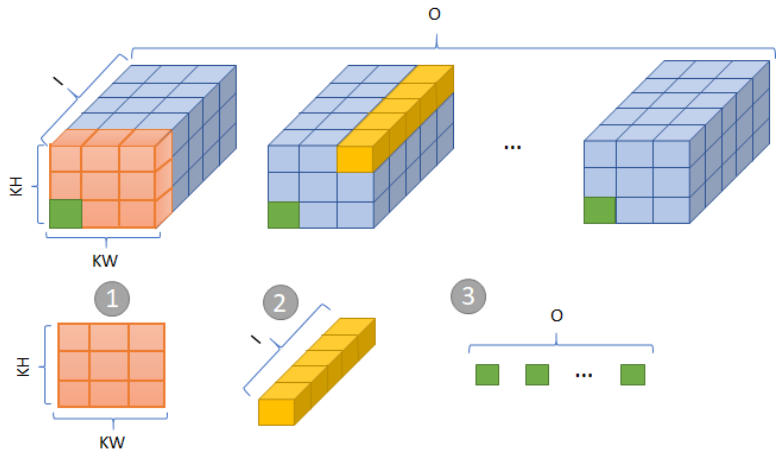


FIGURE 3.2: Different grouping strategies for weights in a 2D convolution layer. Weights are distributed across four dimensions: $O \times I \times KH \times KW$.

For this analysis, all models were converted to the ONNX format [89]. The analysis platform was developed in Python.

3.3 Bit-Grouping Strategies

To assess bit-level invariability in CNNs, we begin by organising the weights of 2D convolutional layers into coherent sets. These sets will follow the typical distribution of weights across four dimensions: output channels (O), input channels (I), kernel height (KH), and kernel width (KW), forming a tensor of shape $O \times I \times KH \times KW$. This multidimensional structure enables the implementation of different grouping strategies as commented in the methodology and illustrated in Figure 3.2.

Three grouping approaches are examined in this work:

- **Filter-based grouping (1):** Each group comprises all weights that constitute a single 2D convolutional filter, i.e. a set of $KH \times KW$ values associated with one input and one output channel. This results in $O \times I$ sets and is based on that filter weights are jointly trained to detect specific features, which should induce intra-group correlation. Additionally, we chose this approach as a strategy that promotes small groups.

- **Input-channel grouping (2):** Sets are constructed by collecting weights that share the same output channel and spatial location but vary across the input channels. This structure may capture inter-channel correlations due to the way information from different channels is aggregated during the convolution operation. We aim to group I elements in total with this approach.
- **Output-channel grouping (3):** Here, sets include weights that share the same input channel and spatial location but belong to different output channels. While expected correlations might be weaker in this configuration, it offers an alternative perspective for evaluating invariability across model dimensions and grouping O elements per set.

These strategies are motivated by the underlying structure of CNNs. By leveraging these potential correlations, we aim to detect invariability at the bit level, thereby allowing the design of efficient fault-tolerance techniques.

Note that the order in which weights are stored and accessed from memory is also important. Typical storage order is CHW, with channels being at the higher dimensions [92, 96]. This implies sequentially reading the weights (filter) for the same I and O channel.

To quantify invariability in each grouping strategy, we analyse the number and position of invariant bits that remain constant across all weights in a group.

This can be observed in Figure 3.3, where each grouping strategy is illustrated with the number of elements combined and subsequently analysed. The invariability refers to the highest number of matching bits in the same position across all elements in a set. For instance, in Strategy (1), we find four invariant bits located in the most significant positions, specifically, positions 30 to 27, which correspond to the exponent and are highlighted in light blue. Therefore, in this set of nine elements, we can state that there are four invariant bits, as all set members maintain identical values in those four bit positions.

The average invariability obtained using Strategy (1) with FP32 precision is thus four bits. However, as depicted in the same figure, Strategies (2) and (3) yield an average of three and two invariant bits, respectively. In these cases, the invariability also arises

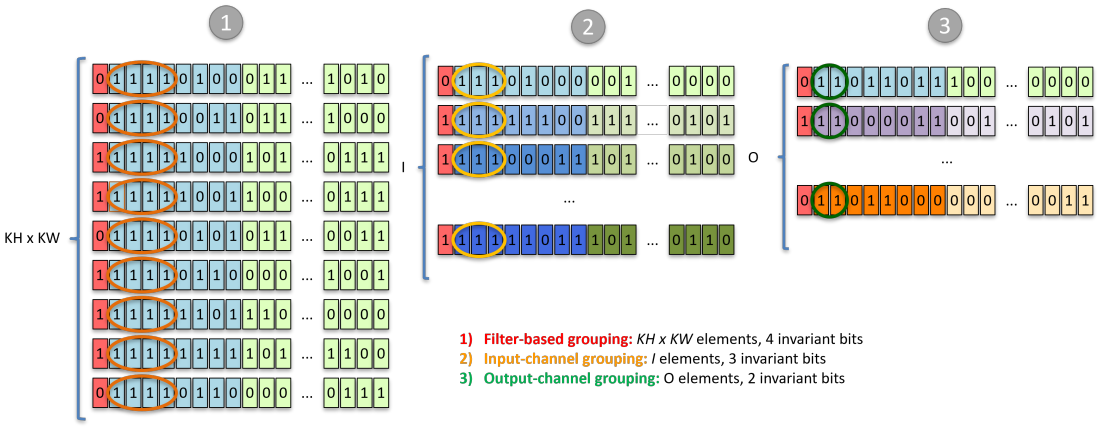


FIGURE 3.3: Average invariability across three different grouping strategies simulating FP32 elements.

predominantly within the exponent, which is shown in various shades of blue to indicate that the elements correspond to the same spatial coordinate and a specific output dimension (O), albeit from different input channels (I). In Strategy (3), the elements are coloured differently to reflect that they belong to distinct output channels.

With these results, we present a detailed analysis of the invariability found by these strategies in all NN models in the next section. We will analyse the results, focusing on Strategy (1), which we have selected as the best case due to its higher invariability.

3.4 Bit-Level Invariability

Building upon the grouping strategies presented above, we now compare their effectiveness in capturing bit-level invariability across different CNN architectures and data formats.

To that end, we shall proceed in a structured order. First, we examine the shape of the convolutional layers in our models under analysis, altogether with their specific characteristics. Next, we present the evolution of bit-level invariability across these layers with each grouping strategy. Finally, we detail the results of each strategy in terms of the number of sets formed, the proportion of such sets, the number of invariant bits detected, and their corresponding bit positions.

3.4.1 Adapting Strategies to Neural Network Models

TABLE 3.3: Filter-based grouping structure for the analysed CNN models.

Model	**Convs**	**Filters** 3×3	**Filters** $\neq 3 \times 3$	**Filters** 1×1	**Filters** 1×1 $+Leftover$	**Bits** 1×1 (%)	**Bits** 3×3 (%)
MobileNet v2-1.0	52	18	0	34	0	97 %	3 %
ResNet-50	53	17	1	36	0	52 %	48 %
VGG-16	13	13	0	0	0	0 %	100 %
DenseNet-121-12	121	59	1	62	0	73 %	27 %
ShuffleNet-v2	56	20	0	36	24	98 %	2 %

Table 3.3 presents the convolutional nodes and their structural distribution within the models under analysis. This information is essential, as the convolutional nodes do not correspond directly to the number of convolutional operations previously reported in Table 3.2.

The discrepancy between the two tables lies in the fact that one reflects the model's conceptual definition as found in the literature, while the other is derived from the actual convolutional nodes observed in implementations following ONNX or PyTorch standards.

These differences arise from the way convolutional operations are accounted for and separated. For instance, in MobileNet v2-1.0, the original definition describes 28 convolutions organised into complete blocks (10 standard, 9 depthwise, and 9 pointwise), whereas the ONNX implementation contains 52 independent convolutional nodes. This increase stems from each convolution type (standard, depthwise, or pointwise) being treated as an isolated operation.

A similar situation occurs with ResNet-50. While its theoretical definition comprises 48 convolutions structured in functional blocks, the ONNX representation identifies 53 convolutional nodes. This is due to the explicit inclusion of the initial convolution operation and additional internal convolutions, such as those within intermediate layers of residual blocks, each recorded as a separate node.

In DenseNet, the contrast is even more pronounced. The literature describes 121 convolutional modules grouped into dense blocks, whereas ONNX identifies each convolution

separately (59 modules with 3×3 filters and 62 with 1×1 filters), leading to a more granular breakdown.

ShuffleNet also exhibits a notable divergence (56 nodes versus approximately 28 convolutions, as originally reported), which arises from the decomposition of grouped convolutions and the explicit representation of additional operations, such as channel shuffling.

Finally, VGG-16 exhibits full correspondence between the two tables. This is due to its straightforward, sequential architecture of standard convolutional layers, with no complex internal subdivisions. In summary, the observed differences are primarily the result of the varying levels of granularity with which each model expresses convolutional operations, more detailed in practical ONNX implementations than in theoretical model descriptions.

We should note that CNNs are the base structure of our work, as convolutional layers have served as the inspiration for designing our element grouping strategies and quantifying bit-level invariability. Consequently, we first present the results obtained exclusively from CNNs and subsequently adapt these grouping strategies to alternative models such as ViT to validate their effectiveness and applicability across various neural network architectures.

Having established this, we must highlight the variety of filter shapes encountered within the tensors of our studied models, as indicated in the columns of Table 3.3.

Most convolutional filters are structured in a 3×3 shape. However, in two particular networks, ResNet-50 and DenseNet, the first convolutional layer employs 7×7 filters. This represents the only deviation from the standard 1×1 or 3×3 filter shapes. In this specific scenario, we apply the grouping strategies in an analogous manner to the 3×3 filters, as previously described in Figure 3.2, except that we group 49 elements instead of nine.

Another distinct case arises with convolutional layers utilising 1×1 filters. Here, each filter consists of a single element, rendering strategy (1) unsuitable, as it would result in groups containing just one element. However, strategies (2) and (3) remain unaffected by this issue, since elements are grouped along dimensions I or O, respectively, regardless of filter size.

These lighter filters, typically 1×1, are extensively employed by networks such as MobileNet and ShuffleNet throughout most of their layers (as evidenced by the percentage of bits in each filter type). DenseNet also predominantly uses 1×1 filters, whereas in ResNet-50 their usage is more balanced.

To ensure consistency in terms of overhead and compatibility with the standard 3×3 convolutions, we adapt strategy (1) for these single-element filters by grouping nine consecutive elements along dimension I, effectively treating this grouping as a single "filter". This approach maintains uniformity in the number of grouped elements across different convolution types. Occasionally, this division of dimension I into groups of nine results in a leftover group containing fewer elements. Regardless of the number of elements, this leftover set is treated as an independent group, an occurrence we observe only in ShuffleNet.

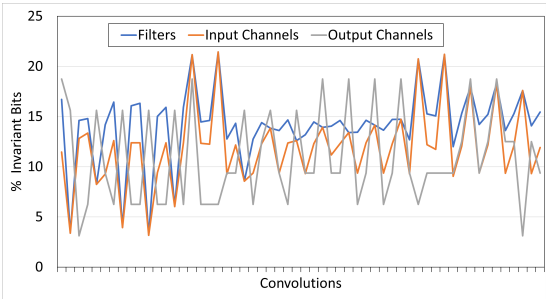
Having clarified these details, the following sections will explore how grouping strategies evolve layer by layer, identify which strategy achieves optimal results, and provide a detailed analysis of its performance, including its impact on ViT models.

3.4.2 Bit Invariability Evolution Layer-by-Layer across NN models

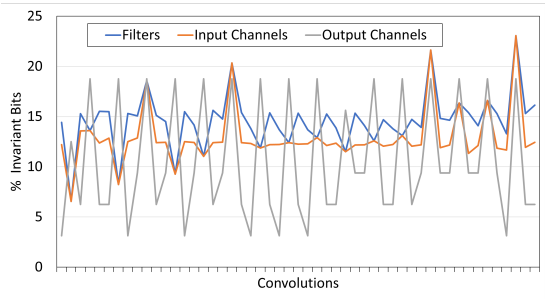
In the first place, as a general study, a layer-by-layer examination across all CNN models was conducted, as depicted in Figure 3.4. Although no consistent correlation was found between layer depth and bit invariability, the grouping-by-filter strategy consistently outperformed other strategies across the model hierarchies.

In the figures, the X-axis represents the total number of convolutional layers per model, while the Y-axis indicates the average percentage of invariant bits for each convolutional layer. Additionally, the graphs illustrate the evolution of the three distinct grouping strategies: Filter-based (blue), Input-Channel (orange), and Output-Channel (grey). This analysis is specifically conducted using models with the data type containing the greatest number of bits and thus the highest probability of bit-level invariability, namely, models employing FP32 precision.

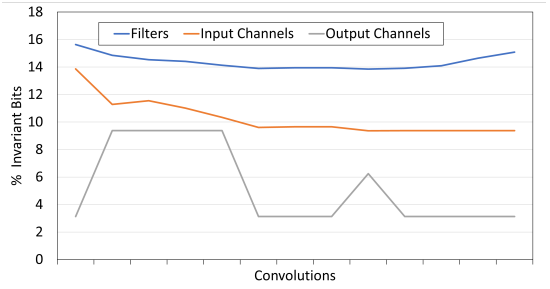
It is likewise noteworthy that invariability remains largely continuous throughout the full stack of layers. This continuity is encouraging, as it indicates that the phenomenon



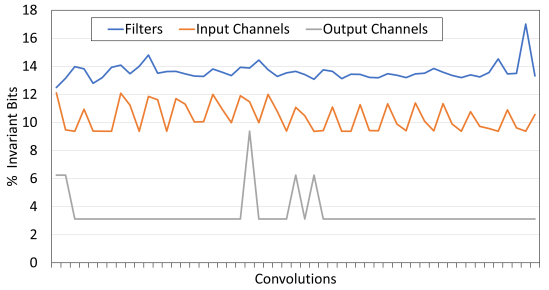
(A) ShuffleNet.



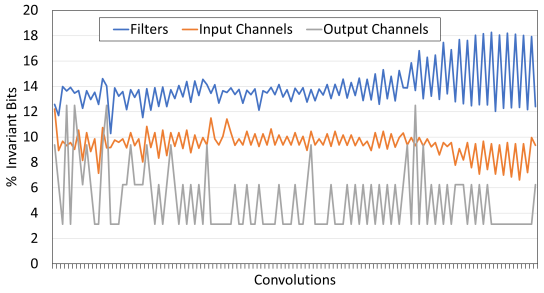
(B) MobileNet.



(C) VGG16.



(D) ResNet50



(E) DenseNet.

FIGURE 3.4: Bit-invariability across convolution layers per CNN model.

can be exploited across the entire network. Although certain architectures, such as ShuffleNet, for example, employ distinct convolutional schemes, these variations do not appear to diminish bit-level invariability. This is, therefore, the most important outcome of these Figures.

Interestingly, lightweight models such as MobileNet and ShuffleNet, designed for constrained environments, exhibit similar trends, albeit with more fluctuation in invariability across layers.

Groupings across I and O, which contain more external dimensions and might therefore be expected to yield lower invariability, sometimes exhibit higher invariability, particularly in certain layers, yet markedly less in others. This variability is likely due to

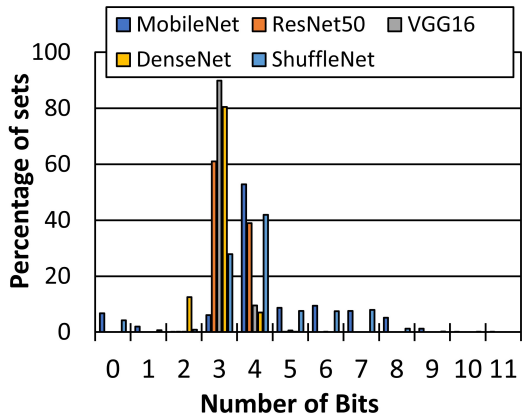
the reduced number of parameters, which in turn limits the consistency and density of correlations across weights.

Overall, the results demonstrate that the Filter-based strategy (1) consistently exhibits the highest average percentage of invariant bits across all layers. As previously noted, ShuffleNet and MobileNet show significant fluctuations, with numerous peaks and valleys in their invariability percentages; nevertheless, the most consistently uniform approach remains strategy (1). Given these findings, this strategy has been selected as the basis for developing our mechanisms, leveraging the invariability it reveals. This invariability is discussed in greater detail below.

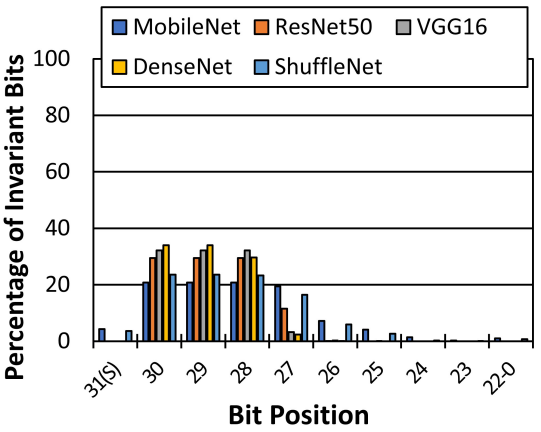
3.4.3 General Results for FP32 Data Type

Based on the previous study, Figure 3.5 collectively illustrates the invariability patterns observed for input-channel (I), output-channel (O) and filter-based ($KH \times KW$) strategies. Moreover, the Figure 3.5 also shows the distribution of invariant bits per set and their respective positions across the 32-bit representation. It is important to note that the analysis includes the sign bit, exponent, and mantissa bits. The aim is not only to confirm the previous layer-by-layer analysis results but also to obtain two statistics that clearly highlight the extent of invariability identified.

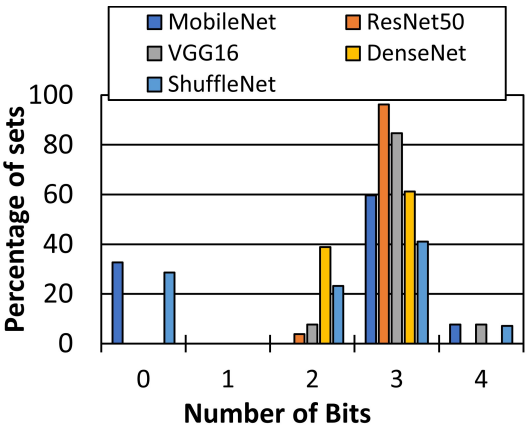
The first statistic refers to the number of invariant bits per set. When expressed as a percentage of the total sets in the network, this figure illustrates the true invariability across the entire neural network model. This statistical measure is presented in Figures 3.5a, 3.5c, and 3.5e (Input-Channel, Output-Channel, and Filter-based strategies, respectively). In these figures, the Y-axis represents the percentage of sets within each model, while the X-axis denotes the number of invariant bits per set.



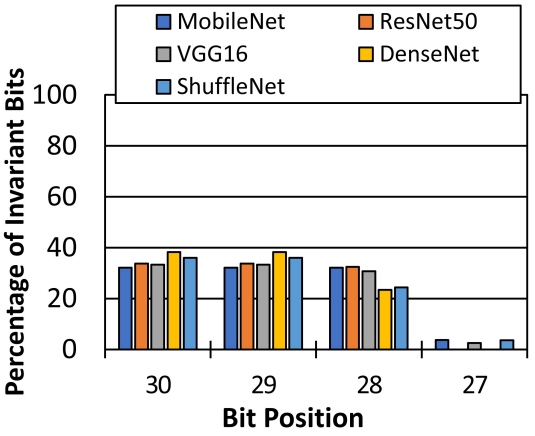
(A) Percentage of sets with a given number of invariant bits in all its elements across dimension I.



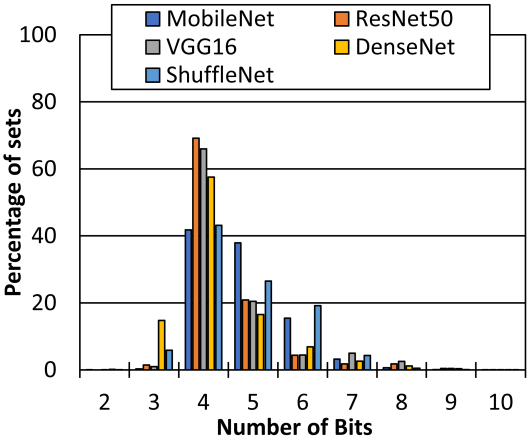
(B) Percentage of invariant bits in each bit position in all the sets across dimension I.



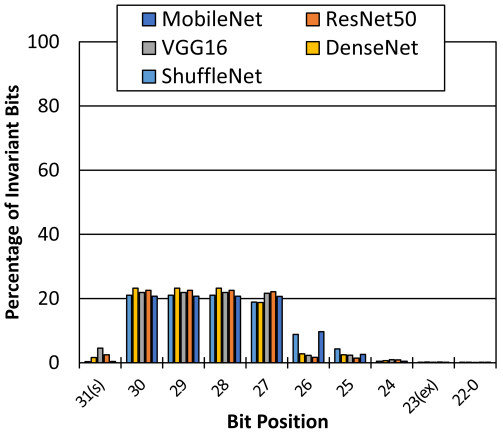
(C) Percentage of sets with a given number of invariant bits in all its elements across dimension O.



(D) Percentage of invariant bits in each bit position in all the sets across dimension O.



(E) Percentage of sets with a given number of invariant bits in all its elements by grouping the sets into 2D-convolutional filters.



(F) Percentage of invariant bits in each bit position in all the sets by grouping the sets into 2D-convolutional filters.

FIGURE 3.5: Various grouping techniques analysis in FP32 models.

The second key statistic involves identifying which bit positions are most invariant. This aspect is critical, particularly for FP32 precision, although it is also relevant to INT8, as not all bit positions have the same significance. For instance, a bit flip occurring in the exponent of a floating-point number has a far greater impact than one in the mantissa. Similarly, in INT8 representations, an error in the sign bit differs substantially from one in the least significant bit. Therefore, understanding which bit positions are most frequently invariant provides essential guidance on what we can, or indeed should, protect. This statistic is illustrated in Figures 3.5b, 3.5d, and 3.5f.

The results clearly indicate that filter-based grouping leads to the highest proportion of invariant bits per set, as also confirmed in the previous sections. In contrast, groupings across I and O dimensions tend to produce sets with a lower number of invariant bits, typically 2 to 4, depending on the model.

The observed trends suggest that inter-channel grouping (dimension I) often produces higher invariability than output-channel grouping (dimension O), likely due to structural similarities across input channels. However, both approaches exhibit relatively modest levels of invariability, typically limited to 2-4 bits per group. With dimension I, as shown in Figure 3.5a, most sets contain either 3 or 4 invariant bits, with VGG16 standing out in particular, exhibiting up to 90% of its sets with 3 invariant bits. Similarly, both MobileNet and ShuffleNet show that nearly 50% of their sets contain 4 invariant bits. In contrast, when using dimension O as the basis for grouping, the majority of sets display only 3 invariant bits. This change is particularly evident in DenseNet, where the proportion of sets with 3 invariant bits drops from approximately 80% along dimension I to just 60% along dimension O.

In addition to the number of invariant bits, their position within the FP32 representation is also informative. As Figure 3.5b shows, many invariant bits concentrate in the most significant positions of the exponent field (bits 30 to 28), and to a lesser extent in bits 26 and 25. These bit positions significantly influence the final value of the weight, and their consistent invariability across sets underscores a potentially exploitable pattern.

This trend is also maintained in Figure 3.5d, indicating that the positions of invariant bits do not differ significantly between these two grouping strategies.

The conservative yet effective strategy of grouping weights exclusively within convolutional filters (without mixing input or output channels) will therefore be justified with the results shown in Figures 3.5e and 3.5f, both in terms of theoretical correlation and empirical evidence. This approach, as we explained, aligns with the functional role of convolution filters in generating a single output activation, inherently promoting weight similarity within 70% of its sets with four invariant bits, whereas across dimension I, it only exhibited 40% FP32 models is shown. The majority of invariability is observed in sets where 4-6 bits remain invariant. That is an improvement of 2-3 more invariant bits.

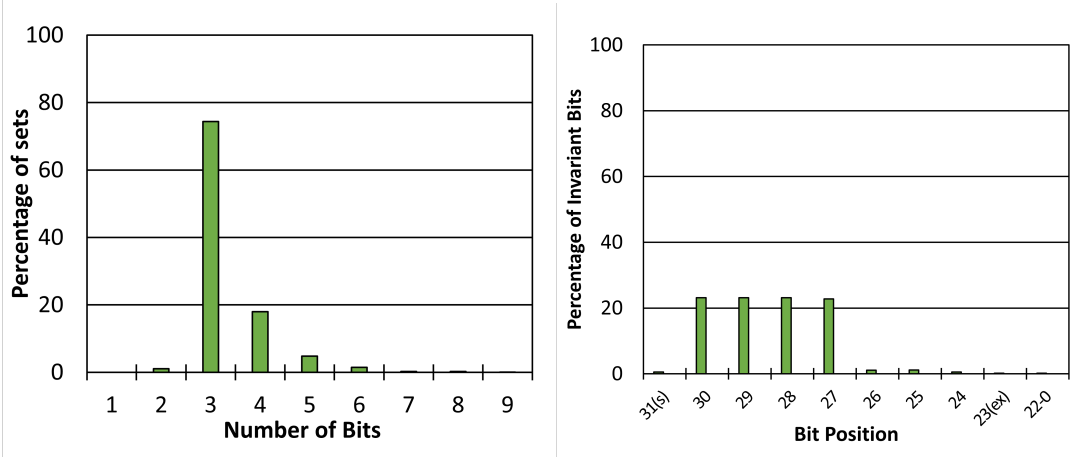
Notably, ResNet50 exhibits up to 70% of its sets with four invariant bits, whereas across dimension I, it only exhibited 40%. Although the absolute count of invariant bits is relatively low (4 out of 32), their specific positions within the weight structure hold significance. Figure 3.5f illustrates the bit positions within the weights where invariant bits are located across all FP32 models. Evidently, as with the other strategies, a substantial portion of invariability occurs within bits 30-27, corresponding to the most significant bits of the exponent; however, this time, we have also found bit position 27 to be uniformly invariant across all CNN models.

Analysing the position of these invariant bits reveals that, on average, approximately half of the exponent bits share the same value within most groups. This outcome aligns with expectations, considering that FP32 models are trained, and weights typically adhere to a Gaussian distribution centred around the value 0.

However, it is crucial to note that invariant bits situated in the most significant bit positions of the exponent bear significance as they determine the sign of the exponent. Consequently, these bits contribute significantly to the final value of each weight. Based on these analyses, there may be potential benefits in safeguarding these bits at the group level, and the best strategy for achieving this goal is to use filter-based grouping.

3.4.3.1 ViT Model

While these grouping strategies were initially conceived for CNNs and their structured filter topologies, they can be adapted to other architectures such as the Vision Transformer (ViT).



(A) Percentage of sets with a given number of invariant bits in all its elements.

(B) Percentage of invariant bits in each bit position in all the sets.

FIGURE 3.6: Various analyses of invariant bits in FP32 models by grouping the sets into 9 elements with ViT model.

In this model, however, predicting bit-invariability is more challenging due to the absence of convolutional layers that could facilitate correlation. Consequently, it is imperative to investigate the applicability of our hypothesis to models that do not incorporate convolutions.

Unlike CNNs, weights are organised in one or two-dimensional tensors (e.g. 1024 or 1024×2048). To maintain analytical consistency, we group these weights linearly into sets of 9 elements, which have the same cardinality as typical CNN filters.

The idea is that, when working with CNN models, certain convolutional layers with 1×1 filters exhibited invariability when grouped within the same I dimension using sets of nine elements. Therefore, we considered it reasonable to apply the same approach to the weights of ViT layers, grouping them along the same dimension. This hypothesis was subsequently validated through experimentation.

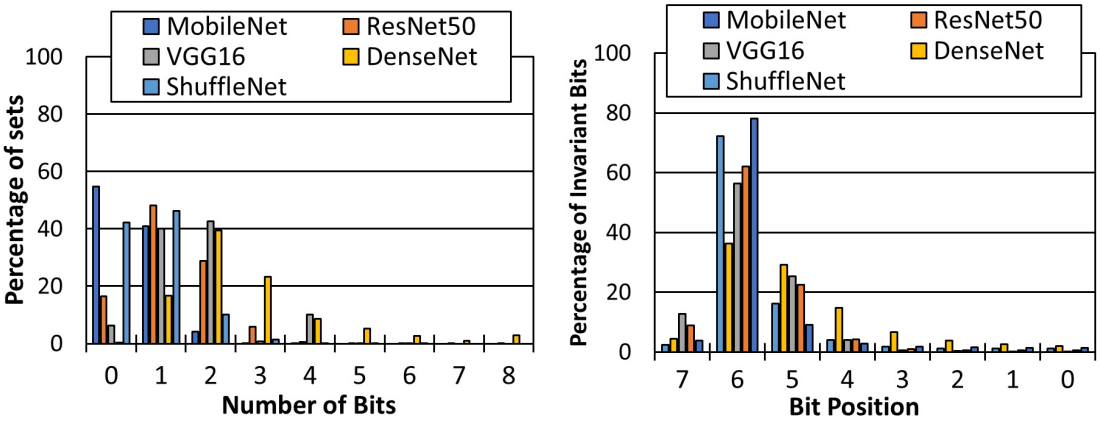
Our approach to selecting this set size is inherently adaptable, depending on the trade-off we wish to optimise. Larger sets typically reduce bit invariability, whereas smaller sets enhance it.

The decision to adopt a 9-element grouping policy for ViT models is motivated by the aim of maintaining consistency in performance benchmarking, timing measurements, and fault tolerance comparisons against CNN models. As illustrated in Figures 3.6a and 3.6b, we confirm that the chosen strategy proved effective.

We observe that even the ViT model achieves a higher proportion of sets with four invariant bits (almost 80%) than any of the CNN models, and nearly 20% of its sets achieve five invariant bits, comparable to the best CNN outcomes. It is also noteworthy that the invariant bit positions in ViT precisely match those of the CNNs, namely bits 27 to 30.

3.4.4 General Results for INT8 Data Type

For the models quantised using 8-bit integers, we have chosen to conduct the analysis exclusively using the 2D convolutional filter-based grouping strategy. This decision is grounded in the results obtained with FP32 models. Given that, even with a higher-bit data type such as FP32, the filter-based strategy consistently yielded the highest invariability, while input-channel and output-channel grouping strategies demonstrated lower levels of bit invariability, a more pronounced reduction in invariability will occur under the INT8 format. This is due not only to the reduced bit width but also to the broader dynamic range of integer weights (i.e., $[-128, 127]$) compared to the typical FP32 range of $[-1, 1]$, where values span the entire decimal spectrum. As such, the likelihood of preserving invariant bits is further diminished under these constraints.



(A) Percentage of sets with a given number of invariant bits in all its elements. INT8 models used.

(B) Percentage of invariant bits in each bit position in all the sets. INT8 models used.

FIGURE 3.7: Various analyses of invariant bits in INT8 models by grouping the sets into 2D-convolutional filters.

Effectively, as illustrated in Figure 3.7a, the extent of bit invariability exhibits variability contingent upon the specific model under consideration. MobileNet, for instance, demonstrates a preponderance of its sets (55%) with no bit invariability. Conversely,

other models exhibit a more moderate degree of bit invariability, typically with 1, 2, or 3 bits displaying invariability per set. This results in a degradation of 2-3 bits with respect to FP32.

Figure 3.7b depicts the most invariant bit positions. Notably, bits 4-6 emerge as noteworthy candidates for protection, as they exhibit up to 80% invariability in certain models and significantly contribute to the final value of the weight. Prioritising the safeguarding of these specific bits at the group level is thus deemed beneficial based on the observed analyses.

3.4.4.1 Conclusions

Taken together, these results validate the use of bit-level analysis as a tool for identifying structural redundancies across neural networks and adapting fault-tolerance strategies accordingly.

It has also been demonstrated that invariability exists not only within the weights of CNN models but also in those of ViT models, including layers that are entirely unrelated to convolutional operations. This finding further supports our hypothesis that such invariability may be present across any layer.

As expected, invariability also depends on the group size. The smaller the group is, the higher the percentage of invariability can be found. Taking advantage of this, we use 2D convolutional filters as the best practice to develop our mechanisms.

Additionally, the fact that this invariability predominantly appears in the most significant bit positions (both in FP32 and INT8 representations) provides further motivation. Should we devise mechanisms to protect these specific positions, we would in effect be safeguarding the bits whose alteration has the greatest impact on the numerical value, thereby enhancing the model's resilience to faults.

Chapter 4

Fault Protection Methods

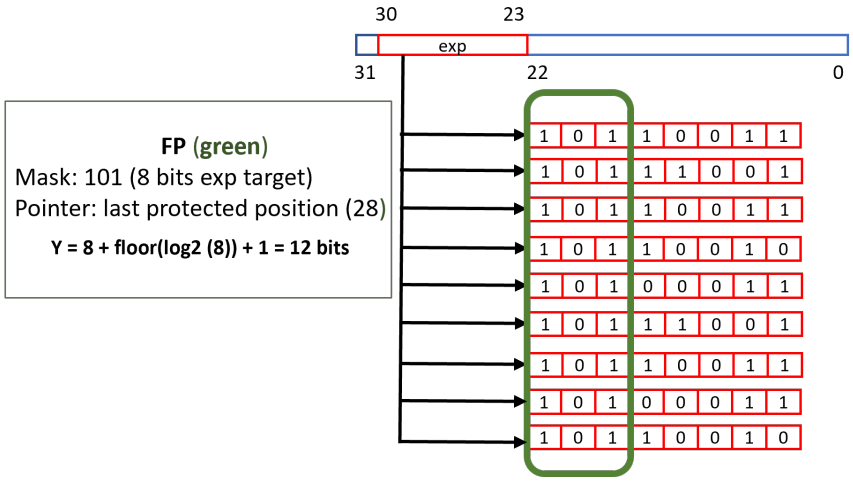
In this chapter, we introduce two fault-protection mechanisms based on insights derived from the preceding analysis. These mechanisms leverage bit-level invariability within sets of filter weights to detect and correct bit flip faults. We hypothesise that exploiting invariant bits within weight sets can substantially mitigate the effects of bit flips. Operationally, we propose storing these invariant bits for each set, which, upon detection of faults, allows restoration of the affected bits to their correct state, thus enhancing the robustness of NN models. Both mechanisms are broadly applicable and can be generalised to from CNNs to any neural network topology and layer type, although their effectiveness decreases with reduced-size data formats.

Following the presentation of these mechanisms, their practical effectiveness must be verified. Therefore, we detail the configuration of the experimental setup, including the methodology and framework used for conducting evaluations, alongside an explicit description of the fault scenarios considered. A series of experiments is conducted to evaluate how CNN models and their convolutional layers, as well as ViT models, respond to faults deliberately injected into their filter weights. Additionally, we discuss the optimal choice of data representation and its relationship with bit-level invariability.

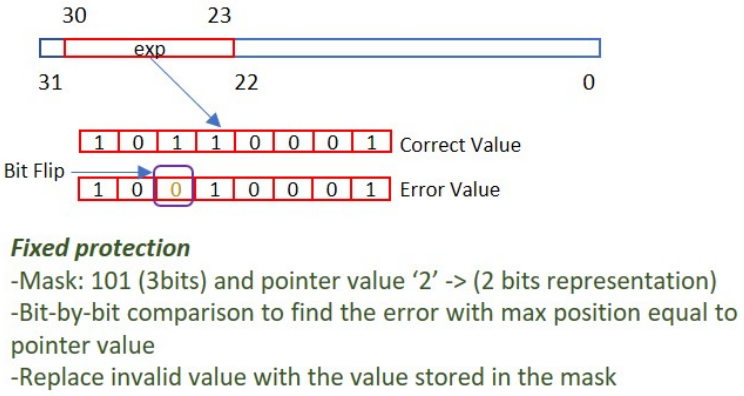
Having demonstrated through these experiments that our proposed mechanisms effectively correct bit flip faults, we then turn our attention to evaluating their effectiveness relative to existing approaches reported in the literature. We select implementations from related works that are straightforward to apply, incur minimal overhead, and do

not require retraining or parallel execution for model inference. To facilitate meaningful comparison, we replicate conditions from referenced works. Lastly, we compare the resulting data and assess the potential advantages of combining our mechanisms with those from the state-of-the-art.

4.1 Fixed Protection (FP) Mechanism



(A) Calculation of bit protection masks for a 32-bit floating point number for FP (green).



(B) FP applied in a recovery procedure.

FIGURE 4.1: Fixed Protection Mechanism.

The Fixed Protection (FP) mechanism safeguards a set of consecutive bits within a group of filter weights. As previously explained, a set is determined by $KH \times KW$ filter weights associated with one (o, i) pair; thus, each convolutional layer contains $O \times I$ sets. In the case of convolution layers with 1×1 filters, we consolidate nine consecutive filters into a set. Within each set, the protection focuses on preserving the bits that exhibit a consistent value across all filter weights, a characteristic identified through the earlier analysis.

For every set, we identify and protect the bits that remain invariant, meaning they retain the same value across all filter weights within the set. If a specific bit position

demonstrates uniformity in value across all filter weights in the set, it is termed an invariant bit within that set.

With the FP mechanism, we derive a bit mask and a pointer value for each set. The mask preserves the values of the invariant bits within the set, while the pointer indicates the last bit position protected. As an illustration, Figure 4.1a shows an example of the application of FP to a specific set.

Initially, we designate a range of bits for protection, such as bits 23-30, in the context of a FP32 format. This range encompasses the exponent and is particularly susceptible to significant alterations in weight value in the event of a bit flip [66]. However, there is a heightened likelihood that these bits exhibit invariability. The FP mechanism exclusively shields sets of consecutive bits commencing from the chosen bit position, which, in this example, is bit 30. The protection is thus applied to bits 30, 29, and 28. For INT8, a different range is selected for protection, specifically bits 6-4. This choice is informed by the previously observed elevated percentage of invariability within this range.

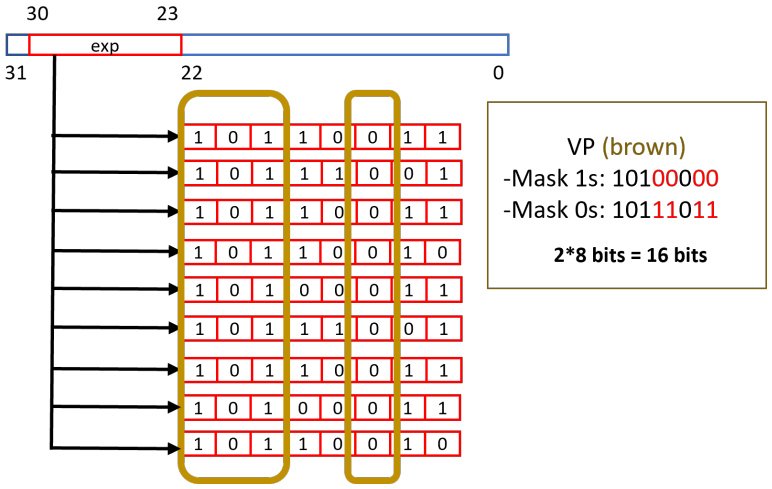
The Fixed Protection mechanism necessitates memory allocation for storing masks and pointers. The number of bits to be used in each set is

$$Y = C_{max} + floor(log_2(C_{max})) + 1 \tag{4.1}$$

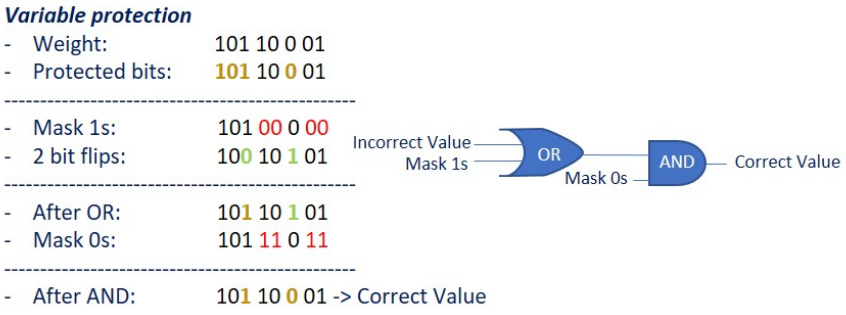
where C_{max} is the maximum number of protected bits (8 bits for the FP32 case and 3 bits for the INT8 case). The FP overhead can be derived from C_{max}. The number of bits required to store an 8-bit mask is 12 (8 bits for the mask, 3 bits for the pointer and one valid mask bit). In a set of FP32 filter weights, this represents an overhead of 4.1% (12 bits of overhead for nine filter weights of 32 bits each). In INT8 models, the number of bits needed to cover the invariant bits at positions 6-4 is 6 bits per set (3-bit mask, the pointer and a valid mask bit), which is an overhead of 8.33% (6 bits of overhead for nine filter weights of 8 bits each). Figure 4.1b also shows how the protection works, comparing bit by bit up to the maximum pointer value, so if a bit is mismatched, it would be replaced by the value in the mask. The combination of bits '101' in bit positions 28-30 is covered by FP. These bits are contiguous and invariant within the set. Moreover, the

pointer needed will have a value of 2 out of 8-bit positions because, with '101', we only need to cover positions 29 and 28, starting from position 30.

4.2 Variable Protection (VP) Mechanism



(A) Calculation of bit protection masks for a 32-bit floating point number for VP (brown).



(B) VP applied in a recovery procedure.

FIGURE 4.2: Variable Protection Mechanism.

The Variable Protection (VP) mechanism is designed to safeguard non-continuous bits within each set. Figure 4.2a illustrates the augmentation of protection by including bit 25 (depicted in brown) alongside the bits already protected by FP. In this instance, the protection encompasses bits 30, 29, 28, and 25.

In this case, we store a mask of 1s (saving the invariant bit positions and putting 0s in the rest of the positions: 27, 26, 24 and 23) and a mask of 0s (same process but with 1s in the positions with no invariant numbers).

The VP mechanism incurs a higher overhead for storing masks. For each set of filter weights, two masks are required. The first mask, referred to as the mask of 1s, retains the bits set to one, while the second mask, known as the mask of 0s, preserves the bits set to zero. Figure 4.2b outlines the recovery procedure following a bit flip event. Initially, an OR operation is applied to the set-bit mask, followed by an AND operation on the unset-bit mask.

To explain it in more detail, we use in Figure 4.2b the same masks, exponent and invariant bits as in Figure 4.2a. Once we have both masks, we show a 2-bit flip insertion: a $1 \rightarrow 0$ bit flip in position 28 and a $0 \rightarrow 1$ bit flip in position 25. Both insertions are successfully corrected. We first perform an OR operation using the mask of 1s shown in Figure 4.2a (10100000). With this, we restore bit 28. Then, we perform an AND operation with the mask of 0s (10111011) to restore bit 25 back to zero.

The overhead incurred to protect any 8 bits in a set of FP32 filter weights amounts to 5.55% (16 bits for nine filter weights of 32 bits each). Correspondingly, the overhead for a set of INT8 filter weights is 8.3% (6 bits for nine filter weights of 8 bits each). It is crucial to note that while VP achieves greater coverage by protecting more bits, this is achieved at the cost of increased memory overhead.

4.3 FP and VP Coverage and Overheads

This section presents an analysis of the overheads and coverage achieved by the Fixed Protection (FP) and Variable Protection (VP) mechanisms. Coverage is the percentage of bits protected within a specified range of positions.

These results were obtained using an open-source tool we developed [36]. The tool is based on Python scripts and provides several ONNX models that serve as examples of how to analyse bit invariability in NN models.

For FP32 models, the targeted range is bits 30-23 (eight bits), while for INT8 models, the focus is on bits 6-4 (three bits). Notably, the VP mechanism extends coverage to all invariant bits within the set, encompassing the specified target bits, whereas FP exclusively covers consecutive invariant bits, starting from the most significant bit that is covered.

Table 4.1: Running time for the calculation of FP and VP masks in seconds.

Model	**Data Type**	**FP**	**VP**
MobileNet	FP32	19	22
MobileNet	INT8	10	11
ResNet50	FP32	210	237
ResNet50	INT8	120	121
Vgg16	FP32	121	142
Vgg16	INT8	65	67
DenseNet	FP32	86	95
DenseNet	INT8	59	61
ShuffleNet	FP32	11	12
ShuffleNet	INT8	6	6
ViT-L-16	FP32	2471	2868

Table 4.1 shows the running times measured for calculating FP and VP masks. It should be noted that these times represent the calculation of all masks for all planned parameters. In the case of CNNs, this refers to the filter weights of the convolutional layers. In the case of a ViT model, it refers to all layers with representative weight tensors.

This process is performed only once and before using the model for inference. Notice that in some cases, the process can take a considerable amount of time (e.g., the ViT-L-16 model with more than 300 million parameters), whereas in other cases, such as ResNet50 or ShuffleNet (with 25 and 1 million parameters, respectively), the time is significantly reduced.

Conversely, Table 4.2 provides an overview of the coverage attained by both mechanisms. In FP32 models, both FP and VP mechanisms achieve coverage ranging from 50% to 57%, indicating that more than half of the target bits are protected. The disparity between FP and VP remains below 2%. In the case of INT8 models, coverage is relatively lower; for instance, MobileNet INT8 achieves 13.62% coverage with FP and 14.91% with VP. Notably, the DenseNet INT8 model exhibits substantial coverage, with 71.84% coverage using FP and 72.67% coverage using VP.

Both mechanisms establish bit masks as a protection method. Those masks are also exposed to faults. Given that a mask affects a set of nine filter weights, a fault in such a mask is potentially more dangerous than a fault in a single weight bit. To protect

Table 4.2: Fixed and Variable Protection Analysis.

Model	**Data Type**	**FP**			**VP**		
		Overhead		% Bits Protected	**Overhead**		% Bits Protected
		%	% plus SECDED		%	% plus SECDED	
MobileNet	FP32	4,10 %	5,9 %	57,02 %	5,55 %	7,2 %	57,41 %
MobileNet	INT8	8,33 %	13,5 %	13,62 %	8,33 %	13,5 %	14,91 %
ResNet50	FP32	4,10 %	5,9 %	51,77 %	5,55 %	7,2 %	52,09 %
ResNet50	INT8	8,33 %	13,5 %	36,08 %	8,33 %	13,5 %	37,27 %
Vgg16	FP32	4,10 %	5,9 %	52,81%	5,55 %	7,2 %	53,15 %
Vgg16	INT8	8,33 %	13,5 %	44,97 %	8,33 %	13,5 %	45,39 %
DenseNet	FP32	4,10 %	5,9 %	50,31 %	5,55 %	7,2 %	50,83 %
DenseNet	INT8	8,33 %	13,5 %	71,84 %	8,33 %	13,5 %	72,67 %
ShuffleNet	FP32	4,10 %	5,9 %	55,54 %	5,55 %	7,2 %	56,79 %
ShuffleNet	INT8	8,33 %	13,5 %	20,02 %	8,33 %	13,5 %	21,81 %
ViT-L-16	FP32	4,10 %	5,9 %	51,06 %	5,55 %	7,2 %	51,45 %

the mask, we can use a SECDED (single-error correction, double-error detection) code with some invariant bits assigned to each mask [40]. This is depicted in Table 4.2, which shows the overhead of protecting masks with Hamming codes with groups of 9 filter weights. With this SECDED protection mechanism, a maximum of five bits are added in the worst case to protect the 16 bits required by the VP mechanism, which utilises a FP32 format (two masks of 8 bits each). The maximum amount of invariant bits for INT8 would be 4 bits to protect the 6 bits made with VP (two masks of 3 bits each).

Table 4.3 shows the FP coverage compared to VP ($VP_{coverage}/FP_{coverage}$ ratio). The coverage achieved by FP in FP32 models is significant, as the coverage is always higher than 98% of total bits covered by VP. However, in INT8 models, FP coverage is lower, being most noticeable in the MobileNet model, where the coverage is reduced to 91,34%.

Comparing the protection of FP and VP throughout the entire models against other correction codes, such as CRC or Hamming SECDED, is also interesting; to do so, we must assume that we aim to protect the same bits as FP and VP.

For protecting a typical 3×3 filter with nine floating-point exponents (8 bits each), applying CRC-7 with a Hamming Distance of 3 adds 7 extra bits per exponent, resulting

TABLE 4.3: FP bit coverage with respect to VP.

Model	**FP32**	**INT8**
MobileNet	99,32 %	91,34 %
ResNet50	99,39 %	96,81 %
VGG16	99,37 %	99,08 %
DenseNet	98,98 %	98,45 %
ShuffleNet	98,52 %	91,79 %
ViT-L-16	99,24 %	-

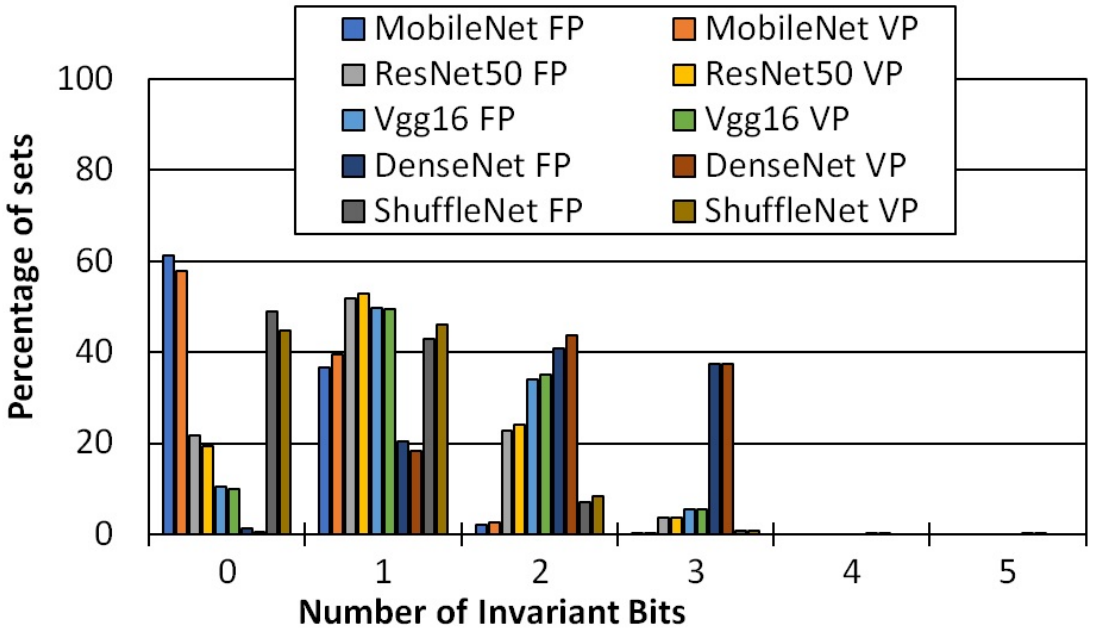


FIGURE 4.3: Invariant bits coverage by FP and VP with INT8.

in a total of 63 extra bits and an overhead of 21.9%, detecting up to two errors. Using Hamming SECDED instead adds five parity bits per exponent, totalling 45 extra bits with an overhead of 15.6%, enabling single-error correction and double-error detection. For INT8 filter weights, protecting the 3 most significant bits requires either 27 extra bits with CRC-3 or Hamming SECDED, both of which have a 37.5% overhead.

The **FP and VP mechanisms** present a significantly lower overhead, typically ranging from **4-5%** for FP32 groups and **8%** for INT8. Unlike CRC and Hamming, which have limitations on the number of detectable and correctable errors, FP and VP can detect and correct **any number of errors** in the protected bits, making them more robust and versatile for fault protection.

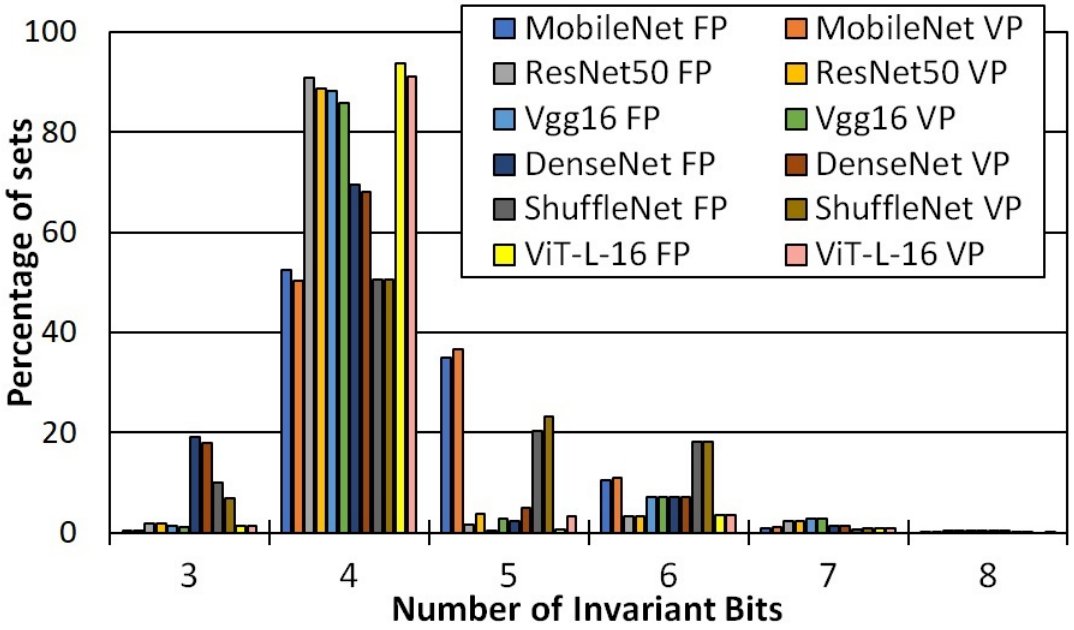


FIGURE 4.4: Invariant bits coverage by FP and VP with FP32.

In addition, Figures 4.3 and 4.4 present the coverage of invariant bits achieved by both the Fixed Protection (FP) and Variable Protection (VP) mechanisms. Remarkably, both methods exhibit a closely comparable percentage of bit coverage across all models. In the case of INT8 (Figure 4.3), a substantial proportion of sets feature one or more invariant bits. Notably, Densenet stands out, with nearly 100% of sets manifesting from 1 to 4 invariant bits using both FP and VP, along with VGG. MobileNet or ShuffleNet demonstrate lower coverage, primarily attributed to the higher number of sets lacking invariant bits.

Conversely, in FP32 models (Figure 4.4), the majority of sets display at least 3 to 6 invariant bits, predominantly in the exponent. It is noteworthy that FP achieves a higher percentage of sets with fewer invariant bits, while VP achieves greater coverage, particularly for cases not covered by FP, and involves a larger number of invariant bits. For instance, we can verify that on average, there is always a higher percentage of sets with 5 to 7 invariant bits using VP compared to FP.

Finally, we also analyse the performance of our mechanisms in a non-convolutional neural network model such as ViT-L-16. Both mechanisms reach 51% of targeted bits. This is confirmed when looking at Figure 4.4, where most sets have four invariant bits out of eight total bits targeted for VP and FP. Despite not having convolutional layers

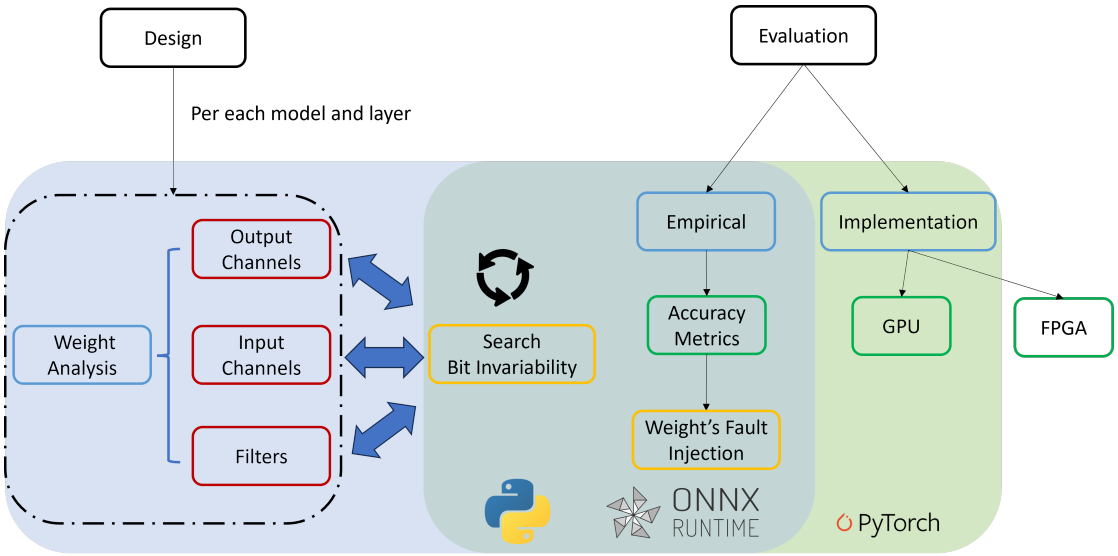


FIGURE 4.5: Evaluation flow.

or filters, its weights are distributed across Matmuls, Gemms (both of which perform matrix multiplications), or Add layers. This demonstrates how invariability can be found in multiple layers that have weights within a specific numeric range, leading to the generalisation of both mechanisms.

4.4 Evaluation

To assess the effectiveness of the implemented mechanisms, we have carried out intensive fault-injection campaigns in several state-of-the-art NN models.

4.4.1 Evaluation Methodology

We use the official ONNX conversion tools [95] to transform models exported from PyTorch and other frameworks into the ONNX format, thereby facilitating uniform manipulation and analysis. Figure 4.5 shows the evaluation diagram, which affects the accuracy of the NN models in this section. Inference is executed using ONNX Runtime [91], a high-performance inference engine developed by Microsoft, which supports execution across CPUs, GPUs, and specialised accelerators. Simultaneously, we continually use PyTorch, with its torchvision library, as a benchmark reference.

To emulate faults, random perturbations are injected into the model weights prior to re-running inference. The evaluation dataset is ImageNet [51], a visual classification benchmark comprising approximately 1.28 million training images and 50,000 validation images across 1,000 categories.

The impact of each fault is quantified using Top1 and Top5 *accuracy metrics*. Top1 accuracy measures the proportion of instances where the most probable prediction matches the right label, whereas Top5 accuracy considers correct any case where the correct label appears within the five highest confidence predictions. Overall model accuracy is calculated as the ratio of correct predictions to the total number of evaluated samples.

The second metric we use involves analysing the Silent Data Corruption (SDC) ratio present in the models. The SDC signifies the percentage of images that are now mispredicted due to a bit flip, whereas they were correctly predicted in the absence of the bit flip.

In conclusion, the same procedure that enables us to measure the impact of faults can also be used to compare our mechanisms with the best state-of-the-art solutions. To establish a fair and valid solution, it is also necessary to perform this comparison. To that purpose, we conduct the same experiments and follow the injection policy with other solutions.

4.4.1.1 Fault Model

The procedure to test our mechanisms will consist of altering the weights of convolutional layers exclusively in CNNs and the general weights across all layers in ViT models. The way of varying these values will be by performing bit flips. We intend to replicate the impact of a memory error (either in DRAM or cache) that would ultimately result in alterations to both bits and in the corresponding weight value. Ultimately, the objective is also to evaluate the performance of the NN models in the presence of weight value faults.

To conduct this evaluation, we introduce a single random bit flip within the set of convolutional filters. Subsequently, we examine the resultant impact on the model's accuracy, utilising the complete test set sourced from ImageNet. This iterative process

ensures the absence of bit flips accumulating over the course of the experiments. The number of experiments carried out was chosen to reach a confidence interval of 99%, with an error margin of 5% by utilising the equation proposed by Leveugle et al. [65]:

$$n = \frac{N}{1 + e^2 \frac{N-1}{t^2 p(1-p)}} \tag{4.2}$$

where: N represents the initial population, p is the estimated probability of failure (by default 50%), e is the chosen margin of error, and t is the required confidence level. Considering that our population size corresponds to the total filter weights across all convolutional layers (a minimum of approximately 1 million for ShuffleNet). This results in 664 different fault injections being needed in all models to reach the confidence interval and error margin targets. Therefore, we conducted 664 fault injections without protection and repeated these injections with FP and VP mechanisms.

To inject a bit flip, we employ a random selection process. Initially, a layer of the model is randomly chosen. Subsequently, an element is randomly selected from the layer's weight tensor, and a random bit from any bit position within that element is flipped. These random selections adhere to a uniform distribution.

As we explained, for CNN networks, our focus is exclusively on 2D convolutional layers, as these layers encompass the majority of the model's filter weights, ranging from 90% for VGG16 to 95% for ShuffleNet. Furthermore, the methodology employed aligns with the approach outlined in [35], which will be compared to our work later.

For the ViT model, we inject faults into all representative layers. Our fault injection policy does not specifically address targeted faults. Our mechanisms are generically designed to address all invariant bits; as such, they will protect against targeted faults and any others. In this study, our primary focus is on faults manifesting in the model filter weights. However, as highlighted in [97], the neurons within a Convolutional Neural Network (CNN) exhibit a resilience 50 times greater than that of their associated filter weights.

4.4.2 Experimental Setup

Experimental procedures are conducted using Python 3.8.10. The ONNX Runtime-GPU framework (version 1.12.0) performs the inference, and results are gathered using the MXNet Library [5] version 1.9.1. These are Python libraries, so compiling them or the Python code is unnecessary. The hardware setup comprises an AMD EPYC 7282 16-core (64 virtual cores) Processor with architecture x86_64 504GB of RAM and 8GB of swap memory. Inferences were executed on the NVIDIA Tesla V100 GPU [19], based on the Volta architecture, which features 5,120 CUDA cores, 640 Tensor Cores, 16GB or 32GB of HBM2 memory, and a memory bandwidth of 900 GB/s. These attributes render the V100 particularly suitable for intensive AI and high-performance computing tasks, ensuring a reproducible and robust environment for evaluating our fault-tolerance mechanisms. CUDA 12.0 is also used to run the experiments.

The GCC compiler version installed is 8.4.0, and the OS Version is Ubuntu 18.04. The introduction of bit flips into the models is facilitated through the ONNX API Reference [88]. It is crucial to emphasise that our focus is exclusively on faults impacting the filter weights, excluding consideration of hardware faults.

4.4.3 Results

From the previous bit-level analysis, it is evident that both the Fixed Protection (FP) and Variable Protection (VP) mechanisms effectively cover the invariant bits within the specified target bits. However, it is crucial to note that merely covering invariant bits does not inherently guarantee effective fault-tolerant performance by either the FP or VP mechanisms. Therefore, this section undertakes an analysis to assess the effectiveness of both mechanisms in mitigating the impact of bit flips.

4.4.4 Fault Classification

Firstly, we delve into the types of faults encountered during our analysis. Since this is an analysis of the entire ImageNet dataset, faults can affect correctly and incorrectly

predicted images. Moreover, faults may have a positive impact, causing previously misclassified images to be predicted correctly or vice versa. To quantify the importance of each effect, we classify the faults into three distinct categories:

- **Masked Faults Correct (MFC):** Correctly-predicted images in which a fault does not have an impact.
- **Masked Faults Missclassified (MFM):** Images always misclassified.
- **Visible Faults (VF):** Images that change the prediction, resulting in a faulty or good prediction.

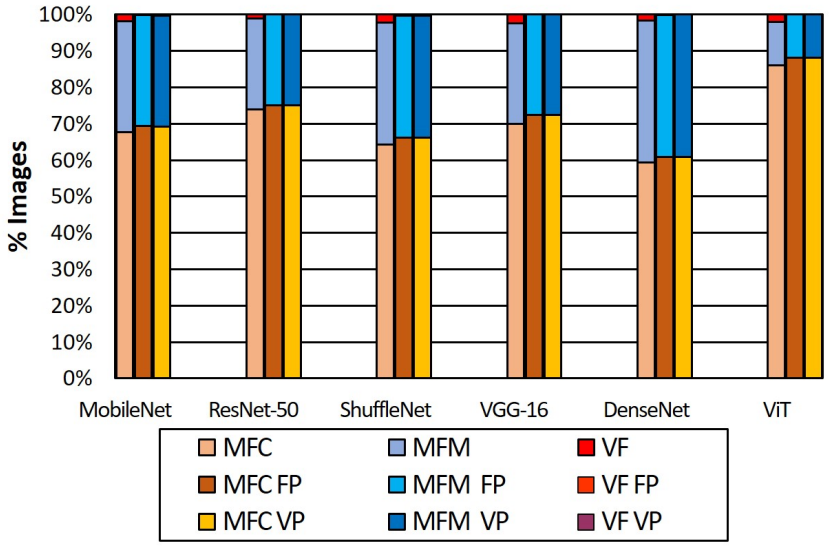


FIGURE 4.6: FP32 Data Type Top 1 Predictions Image-by-Image for all ImageNet Dataset.

In the 32-bit floating point models (Figure 4.6), it is observed that all models without FP or VP have a higher percentage of VF predictions and fewer MFC. However, more MFC are obtained when the protections are applied and the faults are corrected.

In general, the percentage of VF is approximately 2% across all the models analysed, a remarkable observation considering that a single random bit flip in models containing thousands or even millions of parameters can affect the classification of around one thousand images. We will explore this further in subsequent sections; however, it is notable to observe the intrinsic interdependence among all bits within the model, as each bit is potentially critical to its performance.

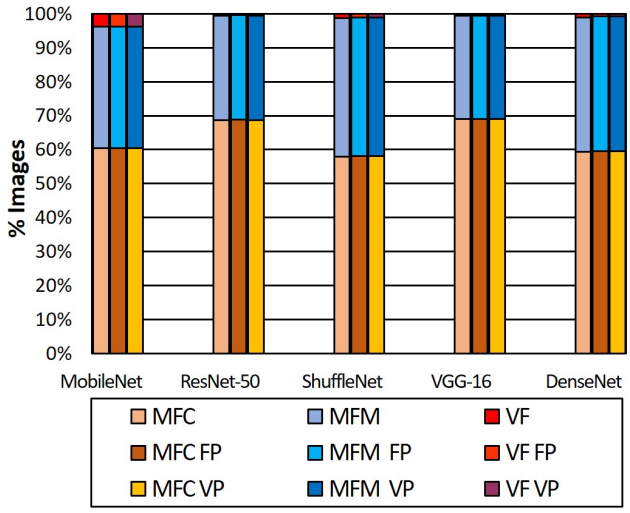


FIGURE 4.7: INT8 Data Type Top 1 Predictions Image-by-Image for all ImageNet Dataset.

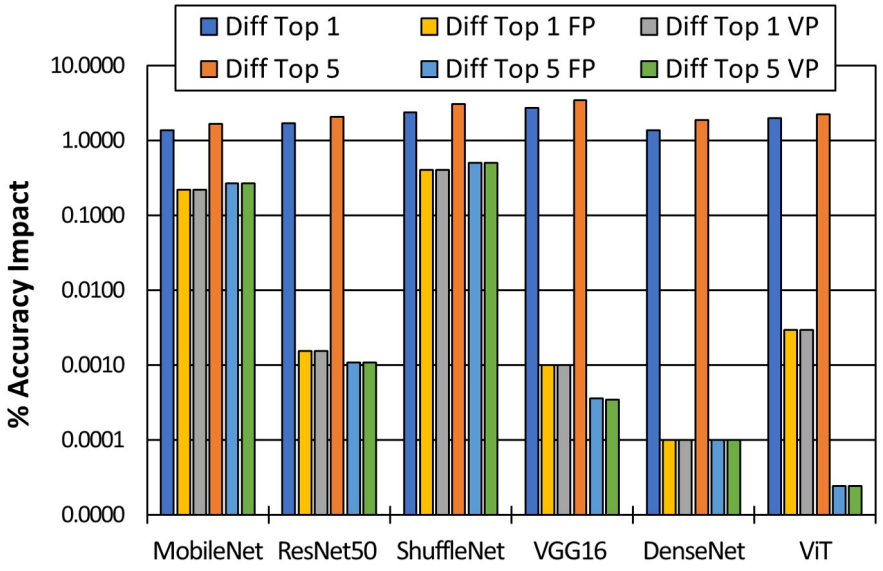


FIGURE 4.8: Impact (percentage of difference) of bit flips on Top1 and Top5 metrics in FP32 models.

On the other hand, INT8 models (Figure 4.7) are less sensitive to faults and maintain a similar percentage of all kinds of them. As we can observe in the figure, ResNet50 and VGG16 exhibit nearly no VF, or at least not to a significant extent. Therefore, applying FP and VP has only a limited effect on these models. In contrast, ShuffleNet and DenseNet exhibit a slight reduction, which will be discussed further in later sections; notably, this decrease is less pronounced in MobileNet.

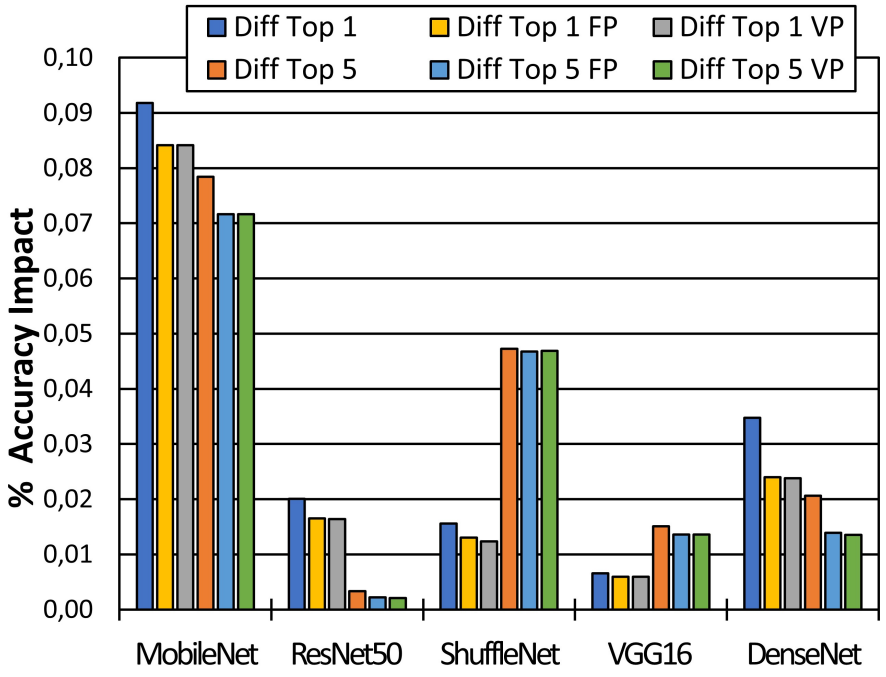


FIGURE 4.9: Impact (percentage of difference) of bit flips on Top1 and Top5 metrics in INT8 models.

MFM consistently maintains the percentage of images; however, by applying our mechanisms, both FP and VP, we effectively prevent the VF from occurring. Therefore, the main shift in image classification happens between VF and MFC (resulting in the Silent Data Corruption metric we will study in the following sections). This indirectly indicates that there is not a significant percentage of images where the prediction is improved due to the fault. Our mechanisms would correct this error, restoring the original negative result even if this were the case. This aligns with our goal of ensuring that the results of protecting NN models closely reproduce the results of models without faults.

4.4.4.1 Bit Flips Effect on Unprotected Models

Figures 4.8 and 4.9 present the influence of bit flips on the Top1 and Top5 metrics of the CNN models, represented on a logarithmic scale. In both figures, the depicted values express the percentage of accuracy impact, signifying the extent of accuracy degradation attributable to bit flips. In other words, we measure how much accuracy we gain or lose due to the impact of bit flips on the model when we perform an inference throughout the entire Imagenet dataset.

The initial observation revolves around the impact of bit flips on unprotected models. It is evident that FP32 models experience an average degradation ranging from 1.35% (DenseNet) to 2.72% (VGG-16) in Top1 accuracy and 1.67% (MobileNet) to 3.05% (VGG-16) in Top5 accuracy. In contrast, the influence on INT8 models is notably lower, with a maximum degradation of 0.092% for Top1 (observed in MobileNet) and 0.078% (also in MobileNet) for Top5.

The ViT model achieves similar results, with a degradation in the model's accuracy of around 1-2% in Top-1, comparable to ResNet50 and ShuffleNet. A similar trend is observed in Top-5, with results closely matching those of the aforementioned models. For INT8 models, the highest degradation observed in VGG16 results in a total Top1 accuracy of 69.001% and a total Top5 accuracy of 88.66%, representing a decrease of 0.3% and 0.17%, respectively. These findings suggest that INT8 models exhibit greater robustness than their FP32 counterparts, although FP32 models demonstrate higher overall accuracy (refer to Table 3.1).

The presented results are averages across all injected bit flips. Notably, there are instances where a single-bit flip renders the model practically unusable, as evidenced by a drop in Top1 accuracy to less than 0.01%, exemplified by the VGG16 model. This scenario occurs when a bit flip on the exponent significantly amplifies the original weight value by 10^{37}.

4.4.4.2 Robustness of VP/FP Mechanisms

Figure 4.8 also illustrates the accuracy impact on FP32 models when employing the Variable Protection (VP) and Fixed Protection (FP) mechanisms. Notably, both FP and VP substantially mitigate the impact of bit flips, reducing the effect to marginal values. The influence on Top1 and Top5 accuracy is minimised, reducing the losses from 1% to only 0.0001% for DenseNet (recovered at 99.9999%). Therefore, FP and VP demonstrate greater efficacy on ViT, ResNet50, VGG16, and DenseNet models, resulting in a reduction in accuracy impact to approximately 0.001% for both Top1 and Top5 accuracy.

In the case of MobileNet and ShuffleNet, a slightly larger margin of bit flip impact on model performance is observed, resulting in a decrease of over 0.1% on both Top1 and

Top5 accuracy when employing FP and VP. Notably, both mechanisms exhibit a similar effect, with VP marginally more effective due to its ability to protect additional bits. In the VGG-16 model, FP yields a 0.0004% Top5 accuracy decrease, while VP results in a 0.0003% decrease.

For INT8 models (Figure 4.9), it is discerned that the protective measures do not have a substantial impact as observed in FP32 models. Despite the potential protection of the most significant bits, a bit flip does not result in significantly larger integer values that could lead to mispredictions, considering the range from -128 to 127. Statistically, there is no discernible pattern, and the application of FP or VP has almost the same impact on accuracy. However, applying these protections to INT8 models remains advantageous, contributing to a slight improvement in accuracy. In essence, INT8 models emerge as more robust, featuring fewer critical bits compared to their FP32 counterparts.

4.4.4.3 Silent Data Corruption Effect

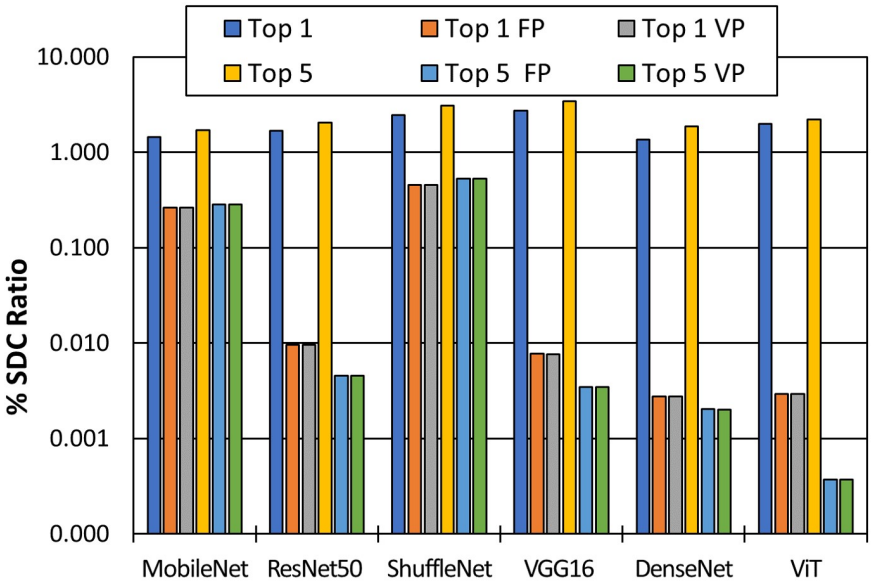


Figure 4.10: SDC Ratio with FP32 Data Type.

Our attention now turns to Silent Data Corruption (SDC) events, our second metric, which denotes the percentage of image predictions (out of 50,000 images from ImageNet) that are misclassified when a fault occurs or images correctly predicted without a fault but incorrectly predicted with a fault.

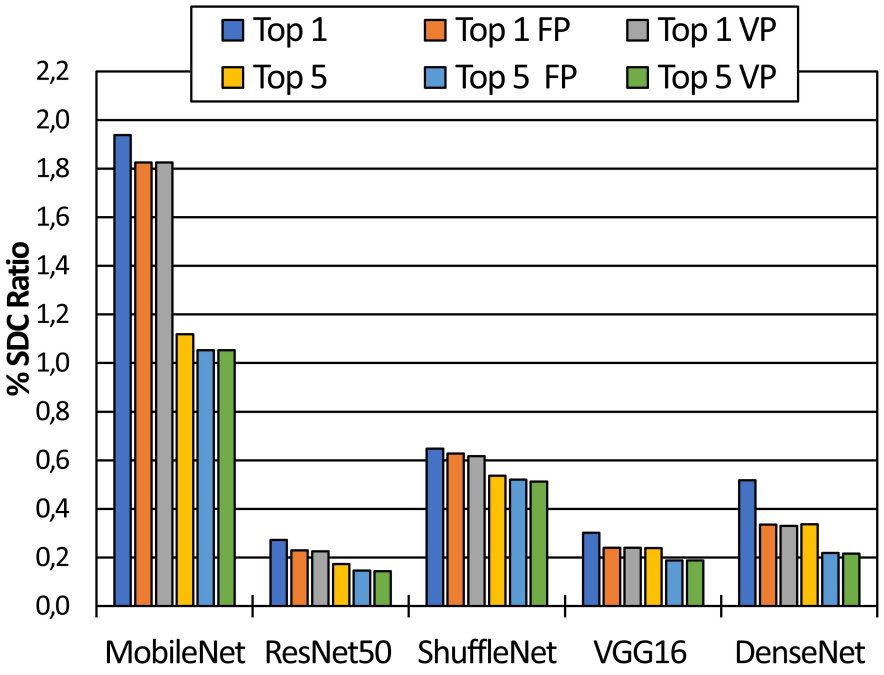


FIGURE 4.11: SDC Ratio with INT8 Data Type.

It is important to note that while accuracy impact and SDC are related, they are not the same. SDC is used to assess the number of images that were previously predicted correctly but subsequently changed the prediction to incorrect due to a bit flip. In contrast, accuracy impact measures the model's accuracy across the entire dataset. This can result in both false positives and false negatives, where a bit flip can alter the result of a previously incorrectly predicted image, making it correctly predicted or not, but still, the fault persists.

Figure 4.10 presents the SDC for FP32 models on a logarithmic scale. It is evident that SDC is higher for unprotected models due to the consequential loss in accuracy caused by a fault. All unprotected models exhibit more than 1% of the images in the dataset experiencing a change in prediction, resulting in failure and a subsequent accuracy loss. With the implementation of FP and VP, this percentage decreases as the number of unprotected errors diminishes, leading to fewer incorrect predictions.

It is also worth mentioning the strong results achieved with the ViT model. Here, the SDC is reduced from over 2% to just 0.003% for FP and VP in Top1, and 0.0004% in Top5. This further confirms that even the filter weights of non-convolutional layers

exhibit invariability when grouped similarly in quantity to the 2D convolution filters, and our protection mechanisms are indeed effective.

A similar trend is observed in the case of INT8 models (Figure 4.11). The SDC ratio remains comparable, irrespective of whether protection is applied. A bit flip is considerably less critical in 8-bit integers than in floating-point models. Given the limited numerical range of 8-bit integers ([-128, 127] as opposed to the 32-bit floating point's range of [-3.4E+38 to +3.4E+38]), a bit flip is unlikely to alter the prediction significantly.

4.4.4.4 Faults Evolution per Convolutional Layer

Once the effects of faults on NN models, especially CNN models, have been analysed, we also consider it pertinent to examine how critical convolutional layers are. Specifically, we aim to determine whether a clear trend or breaking point can be identified across the structure of convolutional layers.

To address this question, we alter our fault-injection methodology slightly. Instead of randomly selecting layers and filter weights within those layers, we now perform targeted injections at specific convolutional layers. As shown in Table 3.3, all CNN models have numerous convolutional layers, each progressively altering and processing inputs for subsequent layers.

This raises the question: which convolutional positions are most vulnerable? Could it be the earliest layers, or perhaps those closer to the final prediction layers? To answer this, we initially considered injecting faults into three layers positioned at the beginning, middle, and end of each CNN. However, we determined it would be more informative to include a larger sample of layers to see if the positional effect diminishes after several intermediate layers.

Consequently, we analysed nine convolutional layers: the first three, three in the middle, and the final three layers. The fault-injection methodology remains identical to that of previous experiments, maintaining a 99% confidence level with a 5% error margin. Only the positional focus of faults has changed for this particular analysis.

To summarise our approach clearly, we inject 664 distinct faults per convolutional layer, maintaining consistency with our previous experiments. Although the population size

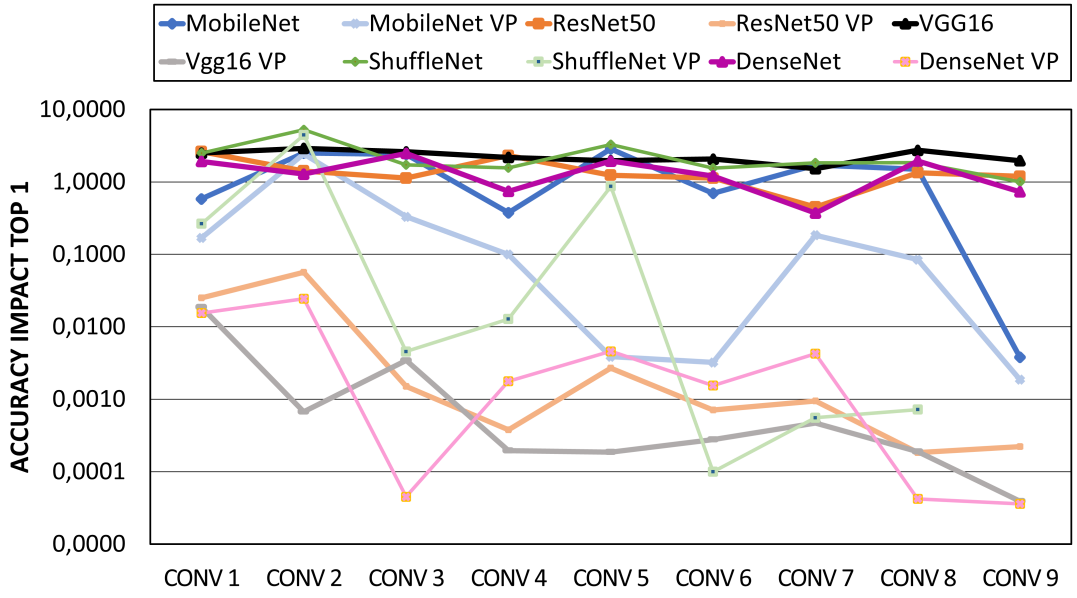


FIGURE 4.12: Top 1 accuracy impact per convolution with FP32 Data Type.

is smaller (consisting only of a single weight tensor), it remains sufficiently large to justify the required sample size of 664 injections. Fault injections are performed without protection and subsequently with FP and VP mechanisms applied to each convolutional layer.

This study is conducted exclusively on convolutional layers and only on CNN models. Furthermore, to examine positional effects clearly, we focus our analysis on models that are more sensitive to faults, specifically FP32 models. This approach ensures that positional differences are more noticeable, whereas with INT8 models, increased resilience would make such differences harder to identify, as previously observed.

Results are presented in Figure 4.12, illustrating the impact on model accuracy, particularly Top1 accuracy. To simplify visualisation, the comparison in this figure is made using only the VP mechanism. A notable initial finding is the variation in results depending on the CNN model analysed. For instance, VGG-16 shows a nearly uniform impact across all layers when they are unprotected, with accuracy degradation consistently around 4-5%. A slight increase is observed in the first three convolutions, but the penultimate layer experiences the highest impact.

Upon applying VP, the results change notably. The protective effect is weakest in the earliest layers but still effective (from 4% accuracy impact to around 0.019%), compared

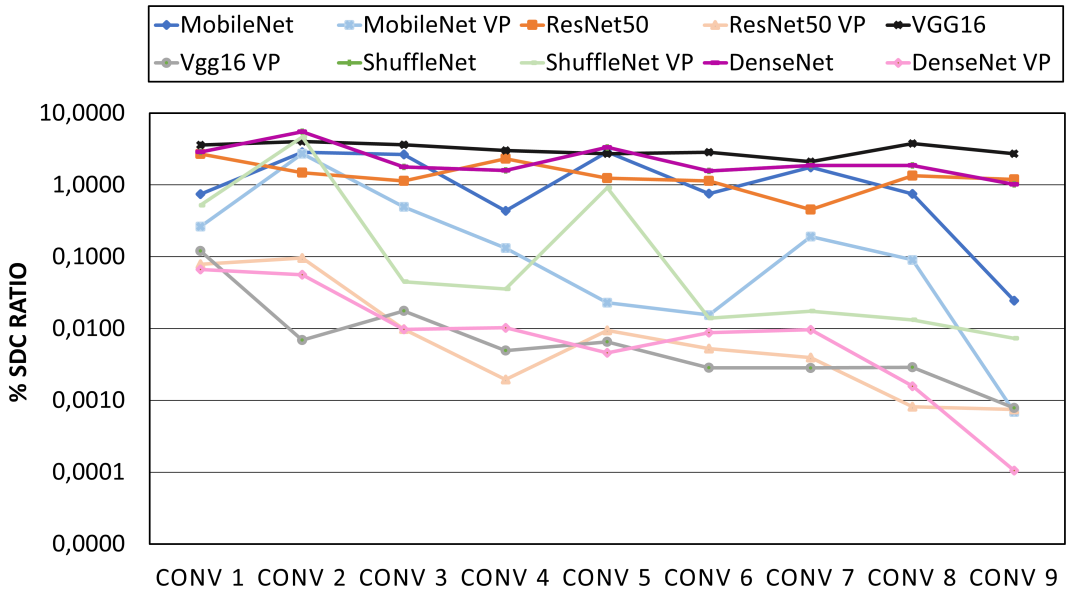


FIGURE 4.13: SDC Ratio per convolution with FP32 Data Type.

to a huge improvement of approximately 0.0001% accuracy impact in the final convolutions (compared to around 5% without VP). From these results, we conclude that invariant and protected bits are more critical and exhibit greater invariability in the later convolutional layers.

ResNet50 and ShuffleNet exhibit similar behaviour. Without protection, there is a slight downward trend in impact from earlier to later convolutions, although this is not highly pronounced. With VP protection, we observe patterns resembling those found in VGG-16. Notably, the middle convolution in ShuffleNet demonstrates a peak in accuracy degradation that VP cannot adequately mitigate, unlike in other layers.

Lastly, MobileNet shows a significant decrease in accuracy impact, specifically at the final convolution, both with and without protection. Counterintuitively, this indicates that the last convolutional layer is the most robust against faults. A plausible explanation is that, following the final convolution, there is a sequence of additional layers: Clip (1), GlobalAveragePooling, Clip (2), Unsqueeze, Concatenate, Reshape, and Clip (3). These layers constrain outputs and mitigate anomalous values, especially the three Clip layers, which maintain values within predefined ranges, likely explaining the lower impact on the last convolutional layer.

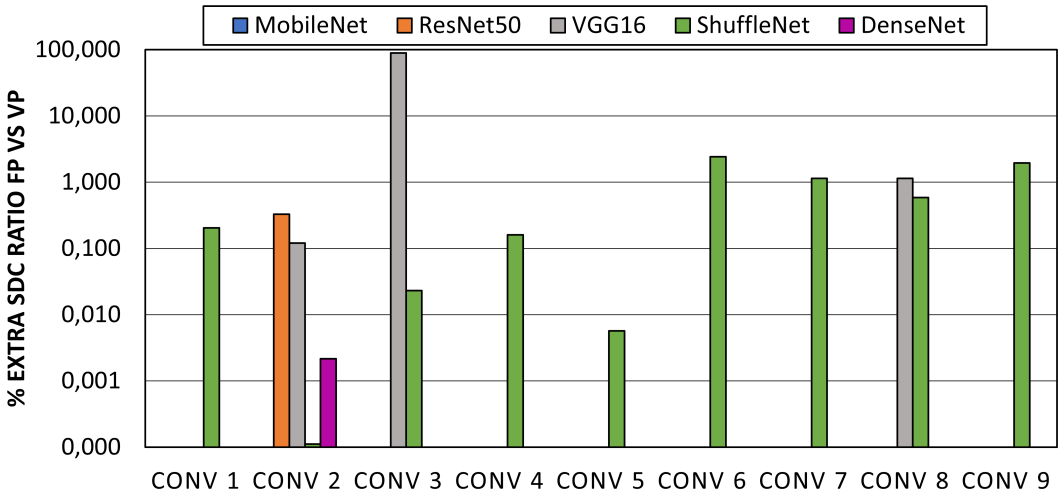


FIGURE 4.14: Extra SDC ratio of VP with respect to FP per convolution with FP32 Data Type.

Figure 4.13 illustrates the evolution of the SDC ratio across convolutional layers. As previously mentioned, SDC correlates with accuracy impact, as most misclassifications due to faults are associated with incorrect predictions. Consequently, the results shown in this figure closely resemble those presented in Figure 4.12.

A noteworthy observation is that ShuffleNet does not proportionally reduce the SDC ratio when VP is applied, unlike the corresponding accuracy impact reduction seen in MobileNet under VP. Specifically, for convolutions 6 and 9, the SDC ratio for ShuffleNet is either similar or even lower than that of MobileNet, despite MobileNet showing greater accuracy degradation in convolution 9. Despite lower accuracy degradation, a higher SDC ratio for one model in a given convolution compared to another does not necessarily imply worse performance. For instance, in convolution 9, MobileNet with VP protection experiences an accuracy impact of 0.0018% compared to 0.0001% for ShuffleNet. However, ShuffleNet has a 0.007% SDC ratio compared to a 0.001% ratio for MobileNet.

These minimal percentages imply that a certain fraction of images can shift from incorrect to correct predictions due to VP protection. This positive impact is captured in accuracy metrics but not in SDC ratios. Similar trends appear in ResNet50 and VGG-16 at convolutions 4 and 8.

Finally, we have also conducted experiments using FP with CNN models. However, given VP's superior performance, we considered explicit FP comparisons unnecessary for this

study. Nevertheless, to measure the relative performance of FP versus VP, we include Figure 4.14.

This figure presents the additional SDC percentage FP produces relative to VP. For example, ResNet50 at convolution 2 shows an additional 0.5%, meaning FP yields an SDC ratio that is 0.5% higher relative to VP. The most pronounced difference occurs in VGG-16 convolution 3, where FP doubles the SDC ratio compared to VP.

Generally, only ShuffleNet exhibits consistent differences across convolutional layers; however, these are minimal, approximately 1% of 0.0175%, equivalent to a mere 0.000175% difference.

4.4.4.5 Bit invariability vs Model Data Precision Type

These findings may prompt questions regarding the relationship between bit invariability and data type considerations. It might be tempting to assume that invariant bits identified in FP32 models could be simply omitted from the model, resulting in the same accuracy with a shortened data type. However, this is not the case, as not all sets of filter weights exhibit the same degree of invariability in identical bit positions. Not all invariant bits prove critical for the accuracy of the model, and not all exponent bits are invariant. Additionally, distinctions should be made between scenarios where all bits within a set are either 1s or 0s; discarding them uniformly from all filter weights is not a viable solution. Our investigation confirmed that merely removing these bits significantly impacts the model's performance, even achieving lower results than equivalent quantised models. Moreover, removing bits necessitates model retraining, leading to a quantised model.

An important consideration arises regarding the relevance of FP32 models when INT8 models demonstrate greater robustness. The debate between floating-point models and integer models is extensive. FP32 models offer increased accuracy at the expense of higher inference time, memory bandwidth, computational power, and energy consumption. In contrast, INT8 models generally offer quicker input processing, requiring less memory, processing power, and energy, albeit at the expense of introducing some degree of inaccuracy that may be unacceptable for specific real-world applications. The

decision to opt for an FP32 model instead of a quantised model hinges on the target accuracy/precision requirements of the model.

For instance, FP32 models may be well-suited for medical computer vision applications, where the advantages of hyper-accurate object detection and classification outweigh the benefit of saving a few milliseconds. Conversely, INT8 models may find applicability in vision-based sensors within the security industry, where alert response time is critical, and the primary concern is determining whether an animal or a human triggered a motion-activated camera. Floating-point models, be they 32, 16 or 8-bit floating-point, can represent a vast range of values with a high degree of mathematical precision. In contrast, INT8 has a smaller numeric range than floating-point data types. Notably, most CPUs and GPUs efficiently handle FP32 operations, leading to the widespread use of FP32 precision in various programs and repositories that employ neural networks, such as PyTorch [118], Intel OpenVINO [52], or the ONNX Model Zoo [90]. Some recent works [55] on using quantised models on Google's TPU have demonstrated that quantisation benefits resource savings, but it should not be compulsory, as it can compromise quality and delay development by requiring months to restore the quality achieved by FP32 using INT8.

Consequently, both data precision models require protection, especially FP32 models, which are the most readily available and commonly used.

As depicted in Table 3.1, FP32 models logically exhibit higher Top1 and Top5 accuracy, making them preferable in scenarios where accuracy is a primary objective. The Top1 performance of INT8 models experiences a downgrade ranging from 1% (DenseNet) to 7-8% (ResNet and ShuffleNet) compared to FP32 models, signifying a substantial difference. Although our protection mechanisms impact the accuracy of FP32 models when faults are introduced, their performance remains superior to the accuracy of fault-free quantised models. This implies that deploying FP and VP in FP32 models is a more advantageous approach than relying on fault-free/robust quantised models.

In addition to the presented results, we also introduced bit flips into 16-bit BrainFloat quantised models [56]. This data type, with 8 bits in the exponent while reducing the mantissa, is considered a prospective data type for Deep Neural Networks (DNNs) due to its balanced tradeoff between FP32 and INT8.

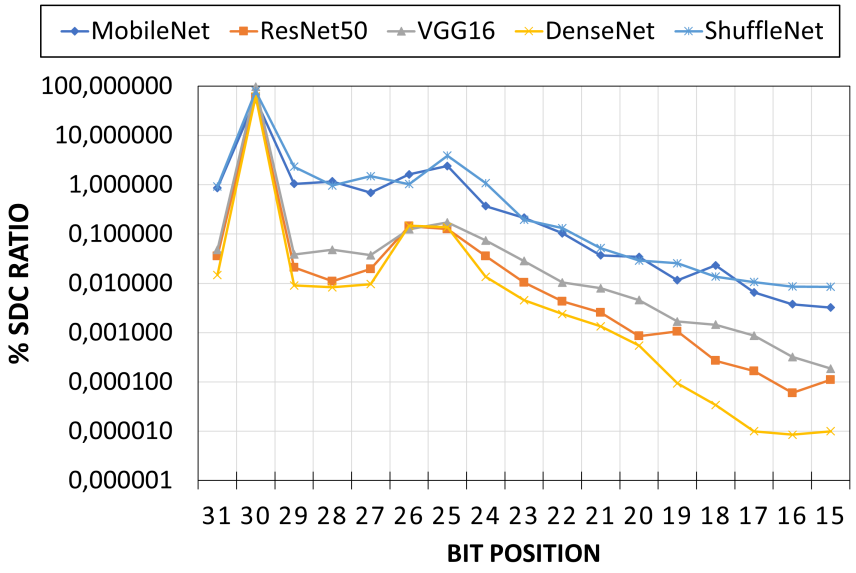


FIGURE 4.15: SDC Ratio with FP32 data type per position. The ViT model was not considered due to its high computational cost.

To evaluate the impact of bit flips with other data types, some models have been converted to Brain Floats by the guidelines set out by PyTorch [94]. Upon closer examination of the conversion process, it becomes evident that the numbers are rounded by removing a portion of the mantissa. However, a detailed bit-by-bit examination of the data reveals that the exponent remains unchanged, indicating that the results obtained by injecting bit flips into the exponent will have the same impact as for 32-bit floating point. Figure 4.15 shows the SDC ratio that affects the remaining bits of the mantissa (23-15). In this experiment, bit flips were injected at each bit position (X-axis), and the average SDC of that position (Y-axis) was computed. This figure reveals that the higher-weight positions are the most important and affected ones, with their significance decreasing as the position lowers. This does not imply that lower-weight positions are unimportant; every position has an impact, though the impact of lower-weight positions is minimal.

4.4.5 Comparison with Related Works

To compare our mechanisms, as explained in Section 2.2.2, there are many approaches targeting the goal of fault tolerance. However, we will conduct a performance comparison between our proposed approach and the one presented in [35]. The techniques outlined in [35] bear the closest resemblance to ours, encompassing software solutions that influence layer outputs, detect bit flips, and markedly mitigate their effects, doing so while the

inference of images is performed on the models. Additionally, these methods can be easily implemented in any architecture, such as ours, without incurring high overhead or requiring specific additional software/hardware.

In contrast, we have reviewed other proposals, such as Hashtag [53] and DeepDyve [70], which solely detect errors without enabling neural network recovery. Moreover, from this comparison, we have left apart other approaches that required re-training the model, replicating it or using more than one network to check the output, the use of ECC/Checksums (as we have already addressed this issue in Section 4.3) or other approaches that differ in similarity.

Various mechanisms are put forth in [35], including Ranger, Clipper, Fmap-Rescale, BackFlip, and FmapAvg. The underlying principle behind these mechanisms is to prevent out-of-bounds values in the filter weights or neurons of the neural network. As elucidated in the context of floating-point numbers, such out-of-bounds values can lead to a substantial loss of accuracy in the neural network.

Firstly, the Ranger approach employs uniform activation range restriction by defining upper and lower threshold values. Activations exceeding these bounds are directly adjusted to the respective threshold values. This method effectively constrains the propagation of anomalously large or small activations induced by hardware faults, maintaining system robustness through straightforward boundary clipping.

Secondly, the Fmap-Rescale technique utilises a linear rescaling approach to restore activations that surpass predefined boundaries. Instead of direct truncation, this method proportionally rescales all out-of-bound values back within acceptable thresholds. By doing so, it partially preserves the original topological information of the feature maps, mitigating the loss of useful discriminative features.

In third place, the FmapAvg method involves reconstruction based on the averaging of unaffected feature maps. Upon detecting out-of-bound activations in a corrupted feature map, FmapAvg replaces these corrupted activations with corresponding values computed as an average from the remaining, uncorrupted feature maps within the same convolutional layer. This method leverages the inherent invariability and similarity typically present among parallel feature maps generated by different convolutional filters.

Finally, Clipper and Backflip yield superior fault-tolerance results (identical for both) among these mechanisms. Clipper employs thresholds and directly assigns 0 to an out-of-bounds value. This strategy prioritises the complete removal of extreme outlier values, thereby preventing fault propagation at the expense of potentially losing some activation information. Meanwhile, Backflip assigns different values based on a formula contingent on the location of the failure.

We opt to use and compare Clipper in our models due to its ease of implementation and its ability to deliver top results. However, a drawback of Clipper is that its thresholds are exclusively derived from activation layers (ReLU). Consequently, our Mobilenet model, lacking ReLU layers, cannot be utilised.

To implement Clipper, we adhere to the guidelines outlined in [35], incorporating Clipper protection layers after activation, pooling, reshaping, or concatenating layers. It is imperative to note that applying this protection indirectly implies that any bit flip on any layer will be affected, as the Clipper thresholds are set at the layer outputs.

4.4.5.1 New Methodology and Setup

In the work presented in [35], the authors select a subset of 1000 images representing 20 random classes from the ImageNet Evaluation dataset. They conduct 500 instances, each corresponding to a different fault configuration, applying each to the 1k images.

In total, this results in 50K distinct fault cases. The fault injections are specifically directed to the exponent field, focusing on critical positions ranging from positions 23 to 30. Additionally, they combine one and 10-bit flip injections to assess the worst-case mitigation scenarios. Subsequently, SDC results are obtained separately for weight faults and neuron faults, exclusively on positively predicted images. Our focus, as explained in Subsection 4.4.2, is on weight faults, and it is noteworthy that we injected faults on all possible bits in our previous results.

To enable a fair comparison, we reproduce the evaluation method in [35] by injecting faults from one to 20-bit flips in the exponent (a multi-bit flip study) and getting SDC results only on positive predicted images.

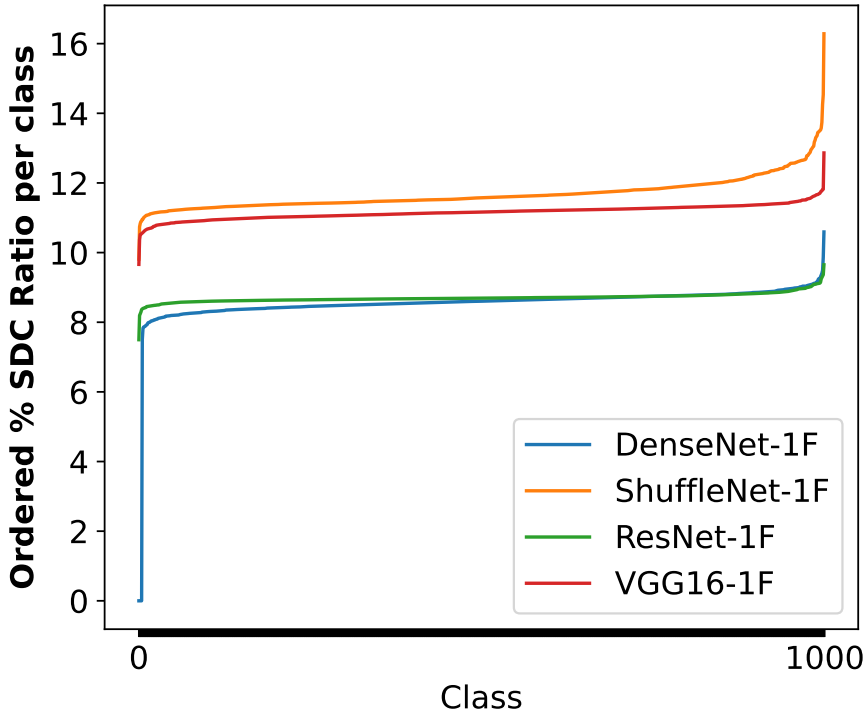


FIGURE 4.16: SDC (positive predictions and 1-bit flip) per class ordered from lowest to highest ratio.

Additionally, we also include the ViT model to determine whether, despite differing in structure, the proposed protection methods remain effective. It should be noted that using a Vision Transformer model prevents the implementation of Clipper, as the model lacks ReLU layers, which would result in differences in how Clipper's thresholds are calculated, as described in [35] (a similar issue applies to MobileNet).

In this case, the experimental setup is identical to that in the previous sections, with the same number of theoretical fault injections: 664. It should be noted, however, that these are 664 injections, each consisting of N number of faults, where $N = 1$, 5, 10, 15, and 20. Therefore, the total number of injected faults will be $664 \times N$. We have adopted this approach to provide greater robustness to the study, as the higher the number of fault injections, the greater the confidence in the results.

Despite this alignment with the evaluation methodology in [35], it is crucial to highlight a difference in our experiments, as discussed in Section 4.4.1.1.

We run the entire dataset for all classes (50K images in total) to maintain a consistent proportion of classes affected by bit flip faults. Figure 4.16 provides evidence of this

approach. In this figure, we introduce a random bit flip into the exponent of the convolutional layer filter weights, repeating the experiments until we achieve a 99% confidence interval with a 5% error rate (resulting in 664 repetitions with 50k images each, or 33.2 million distinct fault cases).

Subsequently, we measure the SDC ratio. Figure 4.16 therefore illustrates how certain classes are more affected than others, displaying a linear increase in the SDC ratio ranging between 8-10% for Resnet and Densenet and between 10-16% for ShuffleNet. Although there are several peaks within these ranges, the key aspect is the linear progression and, more importantly, the proportion within this progression. Therefore, maintaining our 50k image dataset and repetitions based on the confidence interval becomes essential to prevent inaccurate results stemming from the considerable variation in SDC impact across different classes.

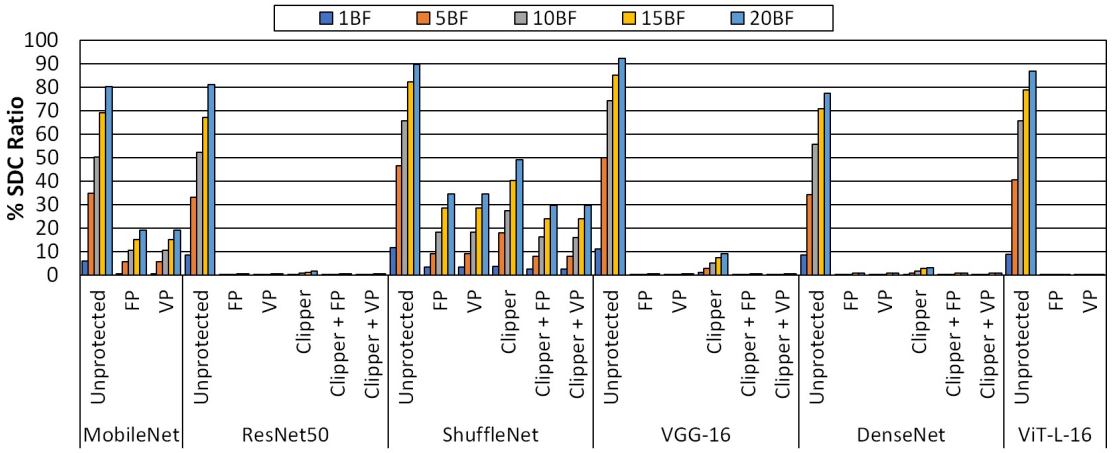
4.4.5.2 Results

In Figure 4.17, we analyse the SDC ratio with multi-bit flip injections in the exponent (from one to 20-bit flips) using FP, VP, Clipper, and a combination of them.

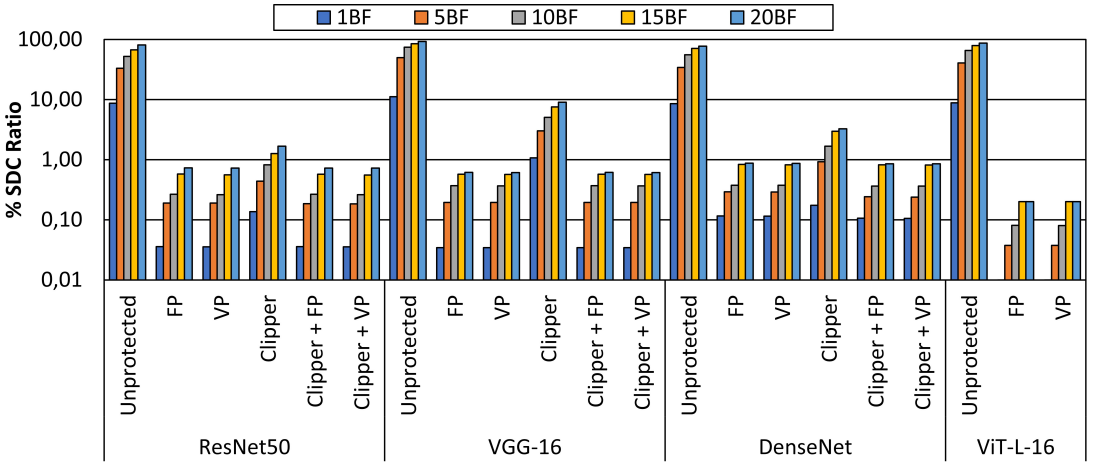
The results reveal that both FP and VP outperform Clipper across all scenarios. Particularly noteworthy are the outcomes for Resnet50 and VGG16 models, predominantly utilising convolutional layers and activation functions. Consequently, Clipper is less frequently employed in these cases.

Here, we observe an improvement factor ranging from 3x to 4x compared to Clipper. Specifically, the SDC rate for Clipper is approximately 0.1%, whereas FP and VP achieve around 0.02% with a single-bit flip. This improvement trend is consistently maintained and evolves proportionally across all cases with 5, 10, 15 and 20 bit flips.

With the ShuffleNet model, the improvement is lower: FP and VP obtain about 3.33% of SDC with 1-bit flip and go from 10% to almost 35% with 5 to 20-bit flips. Clipper achieves a 3.88% to 50% SDC ratio with one to 20-bit flips, respectively. In each case, the results obtained by FP and VP always underperform.



(A) General Overview.



(B) A detailed view of ResNet, VGG, DenseNet and ViT models.

FIGURE 4.17: SDC comparison with multi-bit flip injection. Positive predictions and 1 to 20-bit flip injections (BF).

However, our experiments also involve combining Clipper with FP and VP, resulting in enhanced outcomes, as indicated by reduced SDC ratios from 2.6% to less than 30% for one to 20-bit flips, respectively.

ShuffleNet's model configuration is particularly suitable for Clipper, featuring activation layers followed by pooling, reshaping, and concatenation. For this reason, we apply a layer of Clipper protection after each of these operations. A detailed examination of Figure 4.18, illustrating SDC results for each class, reveals a general trend where all classes exhibit lower SDC ratios with VP (chosen for its ability to detect more bit flips than FP). Additionally, the combination of Clipper and VP (Clipper+VP) consistently improves results across all classes.

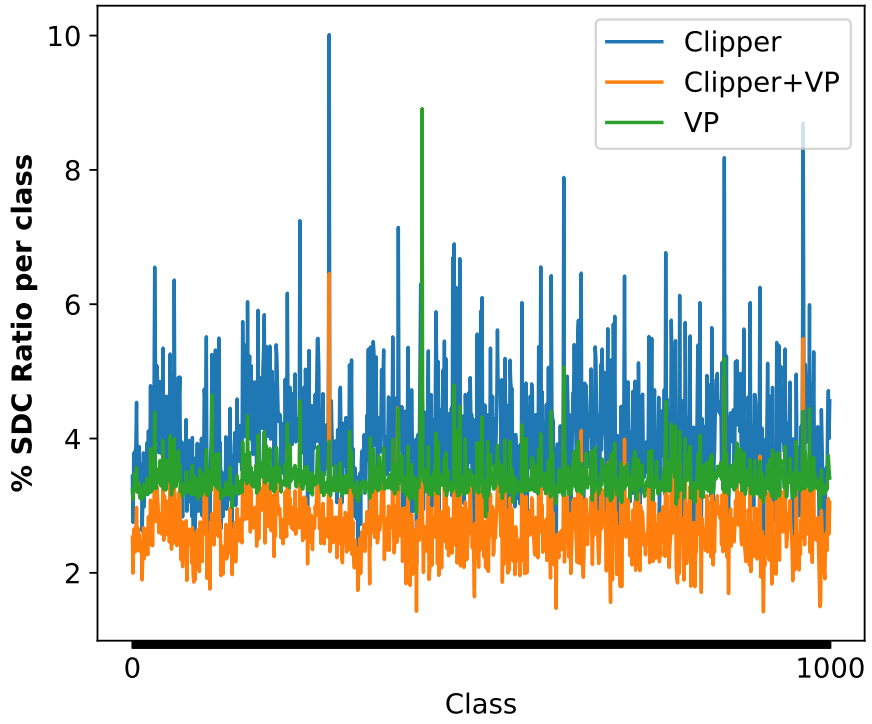


FIGURE 4.18: SDC per mechanism comparison on ShuffleNet (positive predictions and 1-bit flip) with no ordered ratio.

Concerning other results obtained from the multi-bit flip injection, the most significant finding is that from five bit flips onwards, the neural network models become practically unusable. Specifically, we observed Silent Data Corruption (SDC) ratios ranging from 30% to 92% across all evaluated models, rendering any inference performed by these models inherently unreliable.

DenseNet exhibits comparatively less degradation than the other models, showing an SDC ratio of 78% at 20 bit flips, while VGG16 reaches up to 92% under the same conditions. This difference could mainly be attributed to architectural variations: VGG16 contains convolutional and ReLU layers but lacks normalisation layers such as Batch Normalisation, which are employed in all other examined models, including DenseNet.

Two particular models merit further attention. Firstly, MobileNet, which has neither normalisation layers nor ReLU activations, includes native *Clip* layers after certain convolutions designed to constrain outputs within a specific interval, conceptually similar to the Clipper method.

Initially, one might hypothesise that such layers could increase the model's resilience

to bit flip errors; however, in practice, MobileNet shows comparable vulnerability to the other models, reaching similar unprotected SDC levels. Nonetheless, employing FP and VP mechanisms in MobileNet significantly reduces the SDC ratio: specifically, it decreases by approximately 7x (from 35% to 5%) at 5 bit flips, 5x (from 50% to 10%) at 10 bit flips, around 4.7x (from 70% to 15%) at 15 bit flips, and 4x (from 80% to 20%) at 20 bit flips.

Secondly, the ViT model was evaluated to determine both its robustness against substantial bit flip injection and the effectiveness of FP and VP mechanisms. Regarding robustness, the ViT model performs similarly to other models, particularly MobileNet and DenseNet. From this, it can be inferred that, in terms of fault resilience, the critical factor is not necessarily the operational type (ViT has no convolutions) but the parameters themselves. This finding aligns with our second observation: by applying FP and VP in the ViT model, the SDC ratio reduces dramatically by approximately 1000x (from 10% to 0.01%) at 1 bit flip, 80x (from 40% to 0.5%) at 5 bit flips, about 81x (from 65% to 0.8%) at 10 bit flips, approximately 658x (from 79% to 0.12%) at 15 bit flips, and roughly 683x (from 82% to 0.12%) at 20 bit flips, achieving nearly identical results for 15 and 20 bit flips (a phenomenon also observed with DenseNet).

To conclude, Clipper will have more or less effect depending on the variety of layers it applies to. Suppose a combination of layers is formed by activation, pooling, reshaping, or concatenation. In that case, greater protection, time consumption, and implementation overhead are incurred by adding additional layers to the model. In contrast, FP and VP are applied singularly to the convolution layers, consistently yielding superior results irrespective of the model's layer composition. Furthermore, FP and VP offer the advantage of compatibility and adaptability to any type of weight set.

This advantage also translates into using FP and VP in ViT models. As previously observed in convolutional models, the SDC ratio is significantly reduced. This result, in conjunction with that observed in convolutional models, corroborates the initial analysis of invariant bits, in which all floating point models exhibited coverage rates above 50% of the exponent. Consequently, both mechanisms provide comprehensive protection for the most critical bits, demonstrating that regardless of the structure of the neural network model, our mechanisms are effective on any set of elements with numbers within a certain

range and, therefore, sharing bits. This is also the case for the filter weights of neural networks, which represent our target.

4.4.6 Impact of Faults Outside Convolutional Filter Weights in CNNs

Following the investigation and analysis of fault injections into convolutional layers, we consider it beneficial to complete our study by also evaluating non-convolutional layers within CNN models and parameters within convolutional layers, specifically bias, in addition to filters. In summary, this study addresses all parameters, excluding convolutional filters.

As previously established (see Table 3.2), parameters outside convolutional layers represent only a minority. Nonetheless, it is essential to determine the significance of critical faults in these parameters in comparison to faults that occur within convolutional filters.

The experimental procedure, encompassing the choice of models, number of repetitions, fault injections, image dataset, and confidence intervals, remains consistent with the approach described in the preceding section.

However, the target layers now include all layers possessing groupable parameters of more than two elements. Thus, layers such as Dropout, Clip, ReLU, Max/AvgPool, Softmax/LogSoftmax, Concat, Add, Flatten, Reshape, Split, and ChannelShuffle are excluded. Consequently, we inject faults into layers including Batch Normalisation, Linear or Fully Connected (FC), Gemm, GroupNorm/LayerNorm, and Squeeze-and-Excitation (MLP comprising two FC layers).

We note that the ViT model is excluded from this part of the analysis, as it has no convolutional layers, and fault injections have already been comprehensively performed across all its applicable layers. Therefore, we focus solely on the remaining CNN models in the previous section, utilising their FP32 versions.

To ensure a comprehensive analysis, we inject either 1 (1BF) or 10 (10BF) bit flips into the exponent bits of the weights across the model each time. The results are then compared against equivalent fault injections exclusively targeting the convolutional filters analysed earlier.

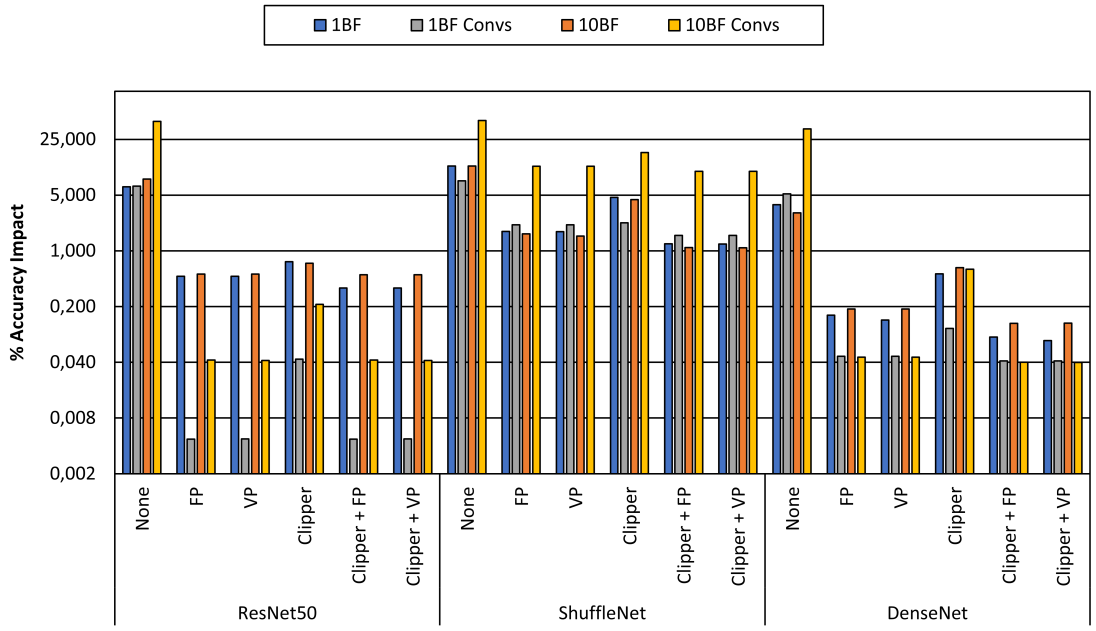


FIGURE 4.19: Top 1 Accuracy impact with FP32 Data Type.

Regarding the protection mechanisms, we evaluate FP and VP alongside Clipper, as previously done, including an additional scenario where FP/VP are combined with Clipper. Clipper's implementation remains unchanged and is positioned immediately after ReLU layers. However, an important difference arises: previously, faults were injected directly into convolutional filters, placing ReLU (and thus Clipper) immediately downstream of them. Some faults occur in parameters farther from ReLU layers, allowing faults to propagate potentially unmitigated for several operations before reaching a Clipper protection point.

Figure 4.19 illustrates the impact on model accuracy. Firstly, it is clear that, consistent with previous sections, FP and VP successfully limit the impact of faults, outperforming Clipper in all cases. Moreover, the combined FP/VP + Clipper approach yields superior protection compared to each method.

Several noteworthy observations emerge from these results. In models such as ResNet50 and DenseNet, the impact of bit flips is significantly lower when faults occur within convolutional filters, especially in the presence of protective mechanisms and particularly with a single-bit flip.

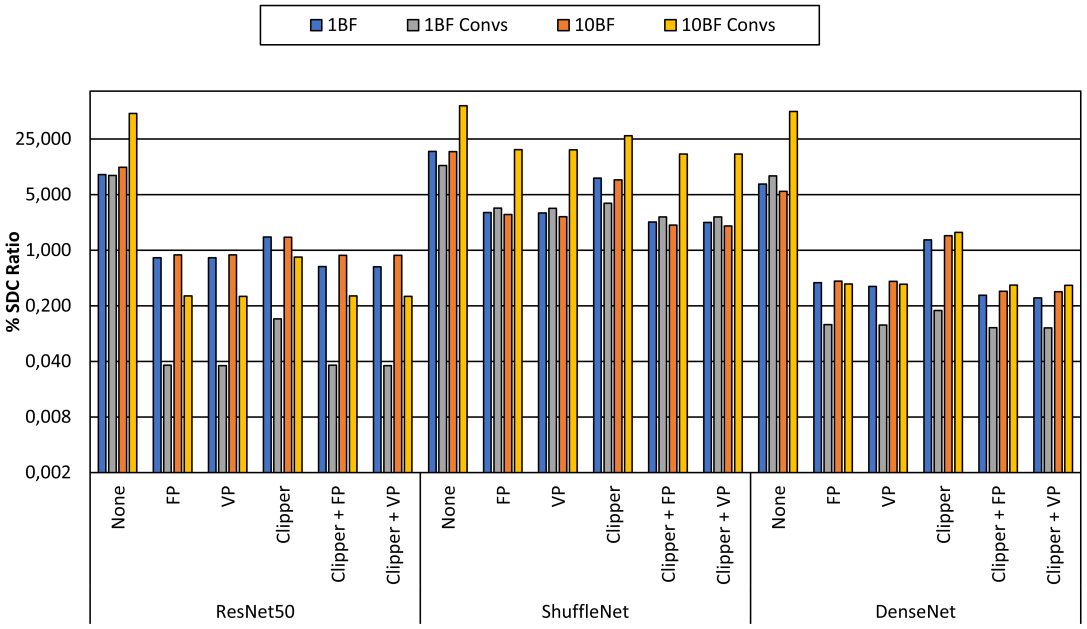


FIGURE 4.20: SDC comparison with multi-bit flip injection. Positive predictions with 1 or 10-bit flip injections (1/10 BF).

For instance, in ResNet50 with one bit flip (1BF), the accuracy impact without protection is approximately 5%, but this reduces markedly to 0.5% with FP or VP, 0.8% with Clipper alone, and further down to 0.3% when combining FP/VP and Clipper. However, when bit flips occur in convolutional filters (1 BF Convs), the accuracy impact diminishes dramatically to 0.005% with FP/VP, 0.04% with Clipper, and 0.004% with the combined method.

Similar proportional reductions occur when injecting 10-bit flips. These disparities suggest two primary factors: firstly, FP and VP protection strategies exhibit greater bit-level invariability and efficacy specifically within convolutional filters; secondly, Clipper's effectiveness diminishes when faults occur at points farther from ReLU layers.

Interestingly, this trend is not universal across all models. ShuffleNet, for instance, demonstrates an evenly distributed or even slightly reversed pattern of accuracy impact, showing a marginally greater impact within convolutional filters (grey) when employing FP/VP or their combination with Clipper with a single-bit flip. This reversed pattern is absent when using Clipper alone, corroborating our hypothesis that Clipper's efficacy is reduced when it is distant from ReLU layers.

Another key finding is observed with 10-bit flips. When injecting faults into convolutional filters (yellow bars), the accuracy impact proportionally increases relative to 1-bit injections. However, injecting ten bit flips outside filters (orange bars) does not significantly amplify the accuracy impact; instead, it remains stable. This stability indicates either a saturation point or limited criticality. We interpret this result as suggesting that convolutional filters represent the most critical parameters: even a single bit flip within these filters may yield similar impact levels as multiple faults outside them, but multiple faults inside the filters lead to a notably greater impact.

Regarding the Silent Data Corruption (SDC) ratio (Figure 4.20), similar overall trends are observed, aligning closely with the accuracy impacts reported in Figure 4.19. It is important to clarify that accuracy impacts are measured across the entire image dataset, while SDC ratios exclusively evaluate images predicted correctly before fault injection. Hence, the percentages between these metrics are not directly comparable.

A significant difference worth highlighting is the notably increased SDC ratio (particularly in ResNet50 and DenseNet) when faults occur in convolutional filters compared to those outside these filters.

This difference arises because accuracy impacts can include images that, despite faults, are still predicted correctly or masked by protection mechanisms, thereby reducing observable impacts. For example, in ResNet50, with a single bit flip, the SDC ratio is 0.9% for faults occurring outside convolutional filters, compared to only 0.04% for faults inside convolutional filters protected with FP, yielding a net difference of 0.86%. Regarding accuracy impact, the external faults result in a 0.5% impact, whereas faults in protected convolutional filters cause merely a 0.005% reduction, a net difference of 0.495%.

4.5 Summary

This chapter thoroughly assessed the effectiveness of our proposed Fixed Protection (FP) and Variable Protection (VP) mechanisms by conducting an extensive comparative analysis against established fault-tolerance methods from existing literature, particularly the Clipper mechanism.

To achieve a rigorous and fair comparison, we implemented the Clipper mechanism in accordance with the guidelines outlined in the literature. We performed comprehensive multi-bit fault injection experiments, systematically injecting from one up to twenty simultaneous bit flips into the exponent bits of model parameters. This comprehensive analysis yielded several key findings.

Firstly, our FP and VP mechanisms consistently demonstrated superior performance over Clipper, particularly in reducing the Silent Data Corruption (SDC) ratio. Notably, FP and VP achieved SDC ratios close to 0.02% in CNN models such as ResNet50 and VGG16 with a single-bit flip, representing a three- to four-fold improvement compared to Clipper's 0.1%. This performance advantage was maintained and proportionally improved as the number of injected bit flips increased to five, ten, fifteen, and twenty.

Our analysis revealed significant vulnerabilities across all neural network models when subjected to multi-bit faults. Specifically, models became unreliable when exposed to as few as five injected bit flips, showing SDC ratios escalating rapidly between 30% and 92%. This highlights the critical need for adequate protection mechanisms, particularly in scenarios susceptible to multiple simultaneous hardware faults.

MobileNet emerged as an intriguing case among the evaluated models due to its unique architecture, which incorporates native clipping layers. Contrary to initial expectations, these layers did not inherently provide additional robustness against bit flips. MobileNet exhibited vulnerability comparable to other models, with high SDC rates under multi-bit faults. Nevertheless, the application of FP and VP significantly mitigated these vulnerabilities, reducing the SDC ratios by approximately sevenfold (from 35% to 5% at five bit flips), confirming the effectiveness of our protection methods even in architectures featuring intrinsic value clipping mechanisms.

Similarly, the Vision Transformer (ViT) model, despite having an entirely different operational structure from convolutional models, displayed comparable vulnerability under fault injection. This underscores the conclusion that parameter values, rather than specific layer types or operations, predominantly determine fault resilience. Encouragingly, FP and VP demonstrated outstanding efficacy with the ViT model, dramatically reducing its SDC ratio from 10% down to as low as 0.01% under single-bit flips and maintaining similar performance across higher numbers of injected faults.

Furthermore, our analysis of faults occurring outside convolutional filters in CNN models reinforced the criticality of convolutional filter weights. While bias terms and non-convolutional layer parameters proved less sensitive overall, our FP and VP mechanisms offered robust protection. Specifically, faults injected outside convolutional filters resulted in significantly lower SDC rates. They reduced the degradation in accuracy compared to faults within convolutional filters, confirming that filter weights remain the most critical parameters requiring robust protection.

In summary, this comprehensive evaluation confirms the superior fault tolerance of FP and VP mechanisms compared to Clipper, highlighting their capability to effectively safeguard critical bit positions irrespective of the neural network's layer composition or structural characteristics. An essential advantage of FP and VP mechanisms is their non-intrusive integration, as they do not require adding extra protective layers into the neural network's original architecture. Unlike Clipper, which introduces additional layers after activation, pooling, reshaping, or concatenation operations, leading to increased computational overhead, memory usage, and inference latency, FP and VP avoid these extra costs.

Moreover, the intrinsic versatility of FP and VP allows them to be effectively combined with Clipper, providing enhanced fault protection when desired. This adaptability enables the development of tailored reliability strategies that depend on the specific deployment scenario. These findings, validated across a wide range of neural network architectures, underscore the robust potential of FP and VP for ensuring reliable neural network inference in the presence of hardware-induced bit flip faults.

Chapter 5

FPGA and GPU Implementations

The preceding chapters have shown the efficacy and validity of the mechanisms in protecting the weights of CNNs and other neural network models, provided that the weights can be appropriately grouped.

The experiments conducted thus far have only focused on the fault-tolerant capabilities provided by VP and FP mechanisms. However, their deployment in a particular hardware platform can incur in performance overheads that need to be quantified.

This chapter presents such an implementation in two distinct environments, along with their associated costs and overheads. Firstly, an implementation within a custom environment is described, in which the mechanisms are integrated into an accelerator based on FPGA systems. Secondly, we consider implementing them for their use in Graphics Processing Units (GPUs) through an open-source Nvidia library.

5.1 FPGA-based Accelerator

In this section, we outline the methodology used to incorporate our strategy into an FPGA-based solution. Next, we will describe the tools and features employed for this purpose.

5.1.1 Background and Implementation Procedure

For the FPGA-based implementation, we employ the **HLSinf** accelerator [28], an open-source platform developed using *High-Level Synthesis* (HLS). HLS is a software tool that generates Verilog logic from high-level programming languages such as C, enabling efficient implementation of customised accelerators for quantised and pruned neural network models on **FPGAs**.

HLSinf addresses the trade-off between the increasing computational demands of contemporary models and the resource constraints typically found in embedded systems. Its architecture follows a modular dataflow model, where each module represents a specific operation (e.g., convolution, activation, or pooling) and supports parallel processing of multiple channels through configurable parameters: **CPI** (channels per input) and **CPO** (channels per output), both of which can be defined during the design phase. It further supports various convolution types, including direct convolution, the *Winograd* algorithm [3], and *Depthwise Separable Convolutions* [109], offering flexibility depending on the model architecture. Figure 5.1 provides a schematic of the platform design.

In terms of experimental evaluation, HLSinf has demonstrated substantial performance improvements over CPUs. Models such as *VGG16* and *SegNet* [6], trained on the *ISIC* [101] dataset for classification and medical image segmentation, have been adapted using structured pruning techniques and post-training quantisation. The results report speedups of up to **94**× compared to conventional CPU implementations, significantly reducing inference time while incurring minimal impact on model accuracy.

In summary, HLSinf offers a practical and flexible solution for executing computer vision models on FPGAs. For our purposes, we will focus on modifying the internal structure of the accelerator to design modules that enable fault-tolerance mechanisms to be applied without altering the dataflow execution. The implementation will also enable the

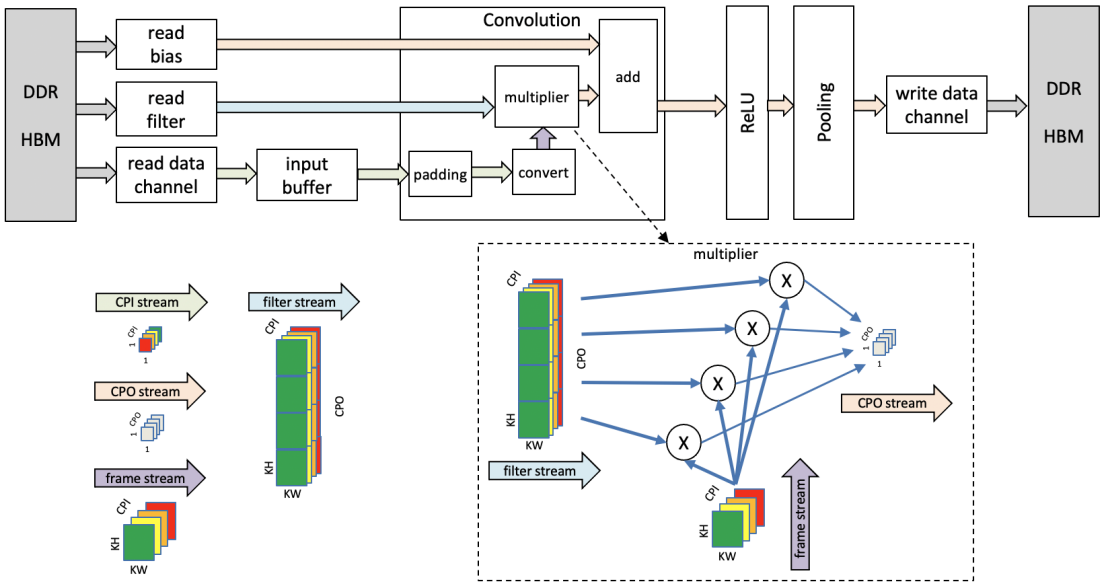


FIGURE 5.1: HLSInf Baseline.

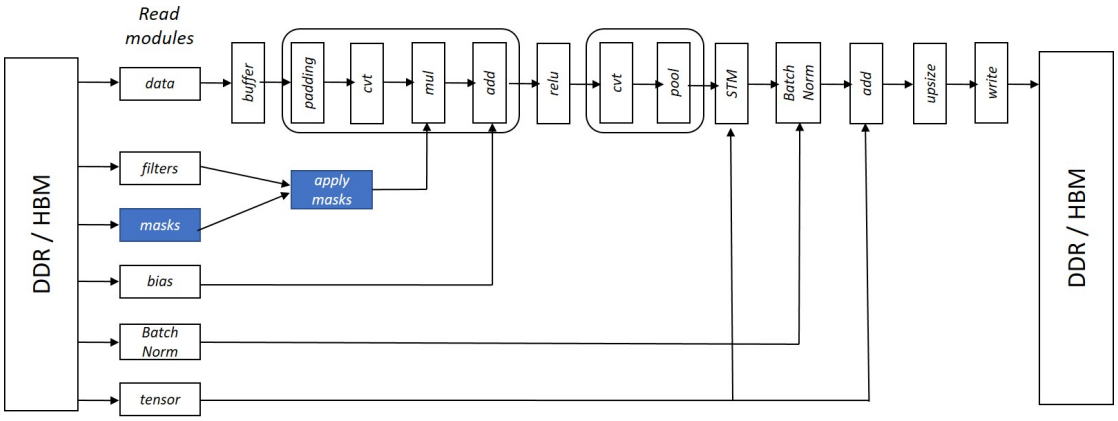


FIGURE 5.2: Accelerator architecture. Modules are represented as boxes, and streams are represented as arrows. Modules can be placed in any order.

measurement of latency and resource overhead introduced by the proposed protection techniques.

5.2 Integration and Overheads

We have instantiated both the Fixed Protection (FP) and Variable Protection (VP) mechanisms on HLSinf[28]. This accelerator facilitates the inference processes of Convolutional Neural Network (CNN) models, allowing us to assess the impact of implementing our protection mechanisms and quantify the associated overheads. Figure 5.2

illustrates the fundamental architecture of the accelerator, comprising modules interconnected through data streams, and highlights the addition of two novel modules specifically introduced to support the FP and VP mechanisms.

The underlying accelerator architecture follows a modular dataflow design, as thoroughly described in [28]. Modules, depicted as boxes, are interconnected via streams (indicated by arrows), allowing for flexible pipeline configurations that are adapted to different CNN model requirements. The accelerator is specifically designed to perform 2D convolution operations, which are central to CNN inference tasks, in an efficient and highly parallelised manner.

5.2.1 Accelerator Architecture Overview

The main operation executed by the accelerator is the 2D convolution, performed in the convolution module comprising various interconnected submodules:

- **Padding and CVT Modules:** These modules preprocess input data by applying horizontal and vertical padding and generating frames (i.e., small data windows) to be used for convolution operations. Specifically, the CVT module organises incoming pixels into frames matching the kernel dimensions ($KH \times KW$), forwarding these frames at each clock cycle.
- **Multiplication (Mul) Module:** This module carries out the convolutional operation by multiplying input frames by their corresponding filters. The operation utilises multiple blocks of multipliers, each handling a set number of input (CPI) and output channels (CPO), performing multiply-accumulate operations in parallel.
- **Accumulation (Add) Module:** This module accumulates the convolution results, adds the corresponding bias values, and outputs feature maps to the subsequent layers or for storage back into external memory.

The accelerator's processing capability is parametrised by two key values: Channels Per Input (CPI) and Channels Per Output (CPO). Adjusting these parameters at design time permits scaling the accelerator's parallelism, significantly influencing performance and resource utilisation.

Algorithm 1 Read Masks Modul for HLSinf.

— **PROCEDURE read_masks** —

procedure READ_MASKS($NumIterations$, $MaskMemStartAddr$, $MaskMemPtr$, $MaskOutStream$)
  $CurrentAddr \leftarrow MaskMemStartAddr$
  for $i \leftarrow 0$ to $NumIterations - 1$ **do**
    Initialize $current\_mask\_packet$ ▷ Packet to hold masks for this iteration
    for $cpo \leftarrow 0$ to $CPO - 1$ **do**
      for $cpi \leftarrow 0$ to $CPI - 1$ **do**
        $mask\_part\_0 \leftarrow MaskMemPtr[CurrentAddr]$
        $mask\_part\_1 \leftarrow MaskMemPtr[CurrentAddr + 1]$
        $current\_mask\_packet.data[cpo][cpi][0] \leftarrow mask\_0s$
        $current\_mask\_packet.data[cpo][cpi][1] \leftarrow mask\_1s$ (in case of VP)
        $CurrentAddr \leftarrow CurrentAddr + 2$
      end for
    end for
    $MaskOutStream$.write($current\_mask\_packet$) ▷ Send packet
  end for
end procedure

5.2.2 Incorporation of FP and VP Mechanisms

To integrate our protection strategies, two dedicated modules have been implemented within this accelerator architecture:

- **Read Masks Module:** This module retrieves the FP and VP masks from the external FPGA memory. Each mask is associated with a specific group of filters and loaded synchronously with the filter weights to ensure the protection data is readily available at inference time without latency penalties. The implementation is illustrated in Pseudocode 1. We implement a loop that progressively iterates over *NumIterations* ($\frac{numMasks}{CPO \times CPI}$) until all masks are processed. In this case, the masks, like the kernels, are stored in arrays of shape $CPO \times CPI$ to allow for subsequent parallel access. Once the masks are read and stored in the array, they are sent to the stream that connects with the *Apply Masks* module.

- **Apply Masks Module:** This module applies the masks retrieved by the previous module directly onto the filter weights. In the event of bit flips occurring in the filter weights, this module utilises FP and VP masks to restore corrupted bits, ensuring the reliability and fault tolerance of convolution operations. The *Apply Masks* module is slightly more complex (see Pseudocode 2). This module consumes the

Algorithm 2 Apply Masks Modul for HLSinf.

```
— PROCEDURE apply_masks —
procedure APPLY_MASKS(NumIterations, KernelInStream, MaskInStream, KernelOutStream)
    for i ← 0 to NumIterations − 1 do
        current_masks ← MaskInStream.read()                    ▷ Read mask packet
        current_kernels ← KernelInStream.read()                ▷ Read kernel packet
        for cpo ← 0 to CPO − 1 do
            for cpi ← 0 to CPI − 1 do
                if IsVariableProtection then
                    mask_VP_info ← current_masks.data[cpo][cpi][0]
                else(Fixed Protection)
                    mask_1s ← current_masks.data[cpo][cpi][0]
                    mask_0s ← current_masks.data[cpo][cpi][1]
                end if
                for p ← 0 to 8 do                              ▷ Loop over 9 kernel elements
                    kernel_element ← current_kernels.data[cpo][cpi][p]
                    if IsDataTypeFloat32 then
                        kernel_bits ← GETBITSFROMFLOAT(kernel_element)
                        if IsVariableProtection then
                            corrected_bits ← doVP(kernel_bits, mask_VP_info)
                        else(Fixed Protection)
                            corrected_bits ← doFP(kernel_bits, mask_1s, mask_0s)
                        end if
                        current_kernels.data[cpo][cpi][p] ← (corrected_bits)
                    else(IsDataTypeINT8)
                        kernel_bits ← GETBITSFROMINT8(kernel_element)
                        if IsVariableProtection then
                            corrected_bits ← doVP(kernel_bits, mask_VP_info)
                        else(Fixed Protection)
                            corrected_bits ← doFP(kernel_bits, mask_1s, mask_0s)
                        end if
                        current_kernels.data[cpo][cpi][p] ← (corrected_bits)
                    end if
                end for
            end for
        end for
        KernelOutStream.write(current_kernels)    ▷ Send corrected kernel packet
    end for
end procedure
```

streams from both the kernels and the masks. It then iterates over the total number of kernels (and thus masks, as there is one per kernel) using $NumIteration \times CPI \times CPO$. This gives us each of the kernels, for which we also access the stored mask. Subsequently, we iterate over each element of the kernel (in this case, the kernel is 3×3, so we iterate over 9 elements). Finally, the protections for both FP32 and INT8 are performed, and the bits of the kernel values are overwritten to ultimately send them through the output stream.

Given the pipelined design and modular dataflow architecture, both the Read Masks and Apply Masks modules operate synchronously at the same clock frequency as the other accelerator modules. This design ensures that mask reading and application processes occur concurrently with the convolutional computations, effectively eliminating any additional execution latency.

It is crucial to emphasise that the accelerator's original architecture remains unchanged. The newly introduced modules dedicated to FP and VP exclusively handle mask-related operations and do not interact with the data path or computations beyond their designated roles, maintaining the integrity and modularity of the original design.

It should also be noted that the accelerator does not calculate the masks, but rather applies them. Consequently, both the masks and the weights are read from main memory. Thus, applying both FP and VP protects against errors that may occur in DRAM memory. Additionally, the accelerator saves both kernels and masks in the on-chip memory, so the errors affecting the on-chip RAM will also be tolerated.

5.2.3 Impact on Execution Time

The execution time of a single CNN convolutional layer is primarily determined by the input data size ($I \times H \times W$), as the convolutional kernels ($KH \times KW$) and weights represent significantly smaller data volumes. The execution latency of the accelerator remains governed by the input data stream and the inherent parallelism provided by the CPI and CPO parameters. Consequently, the incorporation of FP and VP mask processing modules does not affect the overall execution time, as their operation is entirely overlapped with filter preloading and convolutional computation steps. The execution time will be determined by the following equation:

TABLE 5.1: Accelerator Overheads with Fixed Protection (FP) and Variable Protection (VP) implementation on HLSinf.

Configuration	**Resources**		**% Overhead**		**Latency (ms)**
	FF	***LUT***	***FF***	***LUT***	
FP32	246609	137989	-	-	1,31
INT8	84336	80212	-	-	1,31
VP + FP32	263379	149595	1 %	1 %	1,31
VP + INT8	103844	109491	1 %	3 %	1,31
FP + FP32	263759	149788	1 %	1 %	1,31
FP + INT8	105384	115434	1 %	3 %	1,31

$$T_{ex} = \frac{I}{CPI} \times H \times W \times \frac{O}{CPO} cycles \tag{5.1}$$

To calculate the execution time, the equation leverages the accelerator's structural parallelism. For each output channel O, it iterates over $H \times W$ positions, plus the accumulation from each input channel I.

The accelerator processes CPI input channels and CPO output channels, both in parallel and in every cycle. Therefore, the number of work blocks to be sequenced is $\frac{I}{CPI}$ for the input dimension and $\frac{O}{CPO}$ for the output dimension.

5.2.4 FPGA Resource Overheads

Table 5.1 summarises the FPGA resource utilisation overhead associated with the introduction of the FP and VP mechanisms, targeting both FP32 and INT8 data formats. The measurements were conducted on an Xilinx Alveo U200 board. The results indicate an average overhead of approximately 1% for both mechanisms when using FP32 arithmetic, owing to the substantial existing resource consumption of floating-point operations within the accelerator. In contrast, resource overheads for INT8 implementations are slightly higher, around 3%, attributable to the inherently lower total resource usage of INT8 arithmetic operations, making the overhead introduced by mask-processing modules comparatively more significant.

Notably, system latency remains unaffected across all configurations at 1.31 ms, demonstrating that our FP and VP mechanisms do not incur any measurable performance degradation in terms of inference latency.

5.3 GPU Integration

In the context of GPU-based implementation, we have implemented our mechanisms in a more industrial environment, capable of performing convolutions in neural network model inferences, as is PyTorch. To that purpose, CUTLASS is employed. CUTLASS for Linear Algebra Subroutines is an open-source library developed by NVIDIA, providing optimised primitives for high-performance linear algebra operations on GPUs. The library is primarily focused on matrix multiplication (GEMM) and convolution.

The original design of the library was in C++ through modular interfaces. However, recent library versions also incorporate a domain-specific language (DSL) in Python, known as CuTe, to facilitate the development of optimised CUDA kernels.

5.3.1 Background and Implementation Procedure

The CUTLASS system is organised into hierarchical parallel execution levels, including threads, warps, blocks and devices, utilising the underlying CUDA architecture. The design of CUTLASS is such that its components are reusable and adaptable according to the specific requirements of the model or application concerned.

A key feature of CUTLASS is its support for a wide range of data types: from double precision floating point (FP64) and single precision (FP32) to more recent and reduced formats such as TF32, FP16, BF16, INT8, INT4, and even binary formats (1-bit), as well as NVIDIA-specific types such as NVFP4 and MXFPx. This broad compatibility enables the maximum exploitation of Tensor Cores available in Volta, Turing, Ampere, Ada, and Hopper architectures, allowing for adaptation to the accuracy and performance needs of each task. Selected performance data is shown in Figure 5.3, in which it can be seen that CUTLASS offers nearly optimal utilisation.

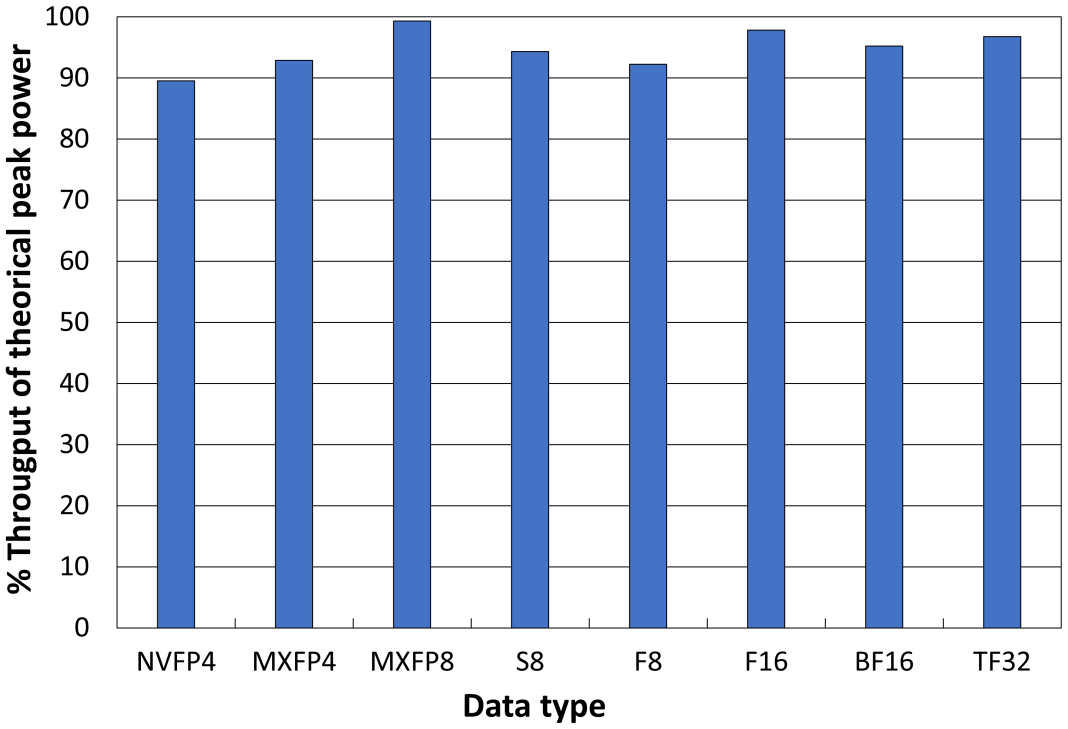


FIGURE 5.3: Performance of CUTLASS Library using GEMM kernels with different Data Types when run on NVIDIA Blackwell SM100 architecture GPU. Source and details about Data Types can be found in [18].

CUTLASS implements convolutions as implicit GEMMs to accelerate neural network operations, leveraging its matrix multiplication pipeline and applying transformations such as *img2col* [17]. It also employs optimisations such as tile loaders, double buffering, and block scheduling to ensure efficient data loading across global, shared, and register memory spaces, minimising latency and maximising GPU occupancy.

The library includes a dedicated tool, *CUTLASS Profiler*, which allows performance evaluation of different kernel configurations based on architecture, data type, and problem size. Moreover, its integration with CMake and compatibility starting from CUDA 9 ensures portability and customisability across various development environments.

Our implementation will leverage the GEMM pipeline offered by CUTLASS alongside the parallelism provided by the GPU for executing inference and convolution operations while simultaneously applying the proposed fault-tolerance mechanisms.

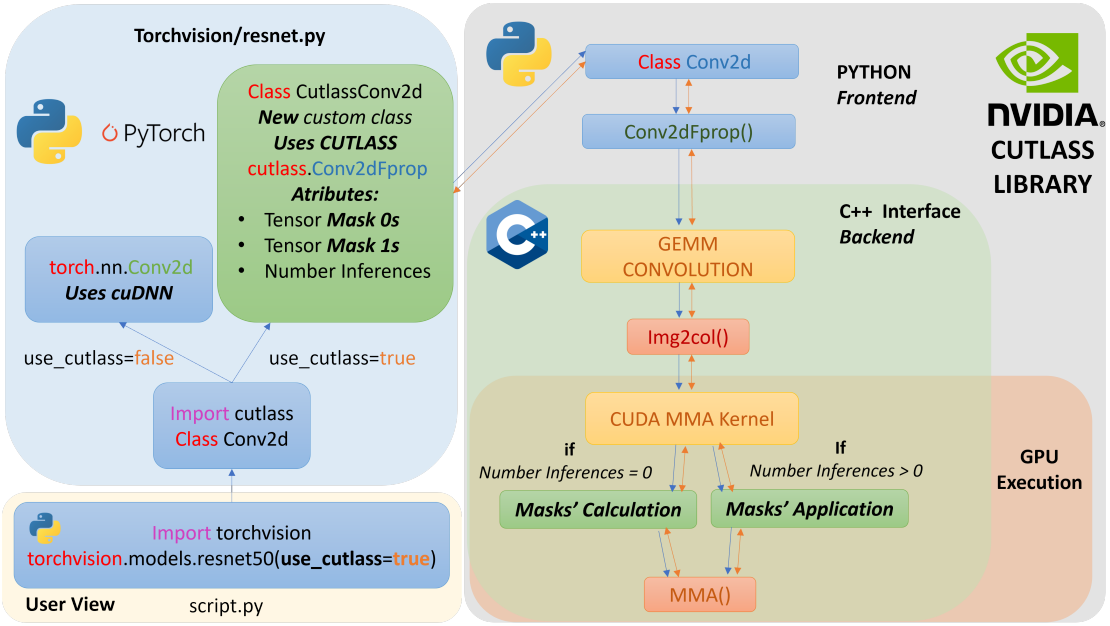


FIGURE 5.4: Diagram showing the use of the CUTLASS implementation and its integration with PyTorch. The colour green indicates the modifications required to implement the VP fault-protection mechanism. Blue arrows show calls to functions and orange arrows show data flux between those functions.

5.3.2 CUTLASS Integration

We implement the VP mechanism in CUTLASS to quantify the impact of our fault-protection mechanisms on performance, as it is the mechanism that provides a higher level of protection and thus potentially has a worse performance impact. However, using one mechanism or the other does not change the integration approach.

This implementation has three parts. The first part describes the changes made to the CUTLASS library itself (the main part of our modifications). The second part explains how CUTLASS is integrated and utilised within PyTorch, specifically within PyTorch's NN models. Finally, the third part describes the user case, outlining what the user needs to do to use it and what the user's feedback is. This process is shown in Figure 5.4.

There are several ways to use CUTLASS. The first and most direct method is to start from one of its C++ examples. These examples show, using a simple C++ program, how to invoke the operation interfaces and how these interfaces, in turn, call the CUDA kernels that perform the computations. However, although this is straightforward, it is not suitable for our use case. Our goal is to perform the full inference of a pre-trained neural network model, so running individual operations in isolation is sufficient for us.

Due to this, we opted for the second option. This approach requires the use of the Python API provided by the library itself. The API defines functions that internally call the corresponding C++ interfaces and execute the corresponding operation. In our specific case, we will need to alter the behaviour of the regular convolution without gradient updates, defined as *Convolution Forward Propagation* or *Conv2dFProp*.

This API is particularly useful not only because it enables the execution of convolutions directly from Python, but also because it greatly simplifies development and aligns with widely used frameworks such as TensorFlow and Keras. Additionally, it allows the convolution function to be invoked from any Python file, which is crucial for our needs. This is important because, when using and downloading the models employed in our experiments, one of the key repositories is PyTorch's `torchvision`, which includes the ViT model alongside the other pre-trained models used. The main difference between `torchvision` and the ONNX Model Zoo (the other source of pre-trained models) is that PyTorch natively defines its models in Python files (which can subsequently be exported to the ONNX standard if required, as in the case of the ViT model).

Following this approach, our objective is to modify these pre-trained neural network models, defined in Python scripts, to replace PyTorch's default convolutional layer, `nn.Conv2d` (which relies on the proprietary CuDNN library), with CUTLASS (an open-source alternative) via its Python API. We can then adapt its interfaces and the kernels they invoke to implement the fault-protection functionality.

5.3.2.1 CUTLASS Modifications

The modifications made in CUTLASS comprise two parts: the C++/CUDA back end (including interfaces and the GEMM-based kernel) and the Python front end.

5.3.2.1.1 C++/CUDA Back End

The CUTLASS back end includes C++ interfaces, which perform data preparation (e.g. `im2col()`) and then launch CUDA kernels on the GPU. Our changes propagate new attributes necessary to perform the protections (*mask 0s, mask 1s and numInferences*) through the CUTLASS call stack; however, the substantive modification lies in the kernel.

Algorithm 3 VP Protection for CUTLASS.

▷ **Main Protection Logic** ◁
Read convolution dimensions: K, C, H, W
Read $numberInferences$
Read $use\_cutlass$
if $use\_cutlass$ **then**
    if *filter_is_1x1*$(H = 1$ **and** $W = 1)$ **then**
        Define *groups* by grouping adjacent channels.
    else
        Define *groups* as individual HxW filters.
    end if

    for each *group* in *groups* (each thread in parallel) **do**
        if is_first_run $(number\_inferences = 0)$ **then**
            CalculateMasks(*group*)
            Store *mask0* and *mask1* for *group*.
        else
            Load *mask0*, *mask1* for *group*.
            ApplyMasks(*group*)
        end if
    end for
else
    Continue.
end if

Here, we exploit thread-level parallelism to compute and/or apply masks efficiently: each GPU thread is assigned a defined group of weights and is responsible for either (i) computing the mask for that group or (ii) applying a previously computed mask, before proceeding with the convolution. We show a representation in Pseudocode 3.

If *use_cutlass* is enabled, the code is executed. Next, depending on whether the convolutional filters have a shape of (1×1) or (3×3), we define either groups of contiguous input channels or the filters themselves as the grouping unit, as explained in Chapters 3 and 4. We then divide the total number of groups among the GPU threads. This way, each thread is assigned to a specific number of groups, which it will process them either to compute the mask or to apply it. We adopt this organisation to avoid overlap and to prevent concurrent read/write memory accesses across multiple threads.

Lastly, if it is the first iteration, the masks are computed and the invariant positions are identified; the resulting masks are stored in GPU memory and also returned as an attribute to the corresponding `CutlassConv2d` in PyTorch. Otherwise, each thread applies the mask to its assigned group.

5.3.2.1.2 Mask Life Cycle and Execution Policy

The policy for mask handling is as follows. On the *first* inference for a given model, the masks are computed on the GPU and stored both in device memory and returned to the host. For *subsequent* inferences in the same execution context, the stored masks are applied directly on the device; no re-computation is required. This minimises overheads and ensures that protection does not impede the convolution pipeline after the initial pass.

We note a practical warning: if a model performs exactly one inference on the GPU (and both the host process and GPU context are then turned down), no protection is applied to that single run, since masks are only computed on the first pass and reused thereafter. However, the masks can be pre-calculated, saved, and used as long as each model is executed as an extension of the model, which is inherent to our mechanisms. However, in this implementation, we don't follow this approach. In typical industrial deployments, neural networks are invoked repeatedly and continuously, meaning the masks are computed once and then applied across many inferences, which aligns with the intended usage and amortises the initial overhead.

Additionally, by calculating the masks during the first inference and applying them in subsequent ones (once per inference), we also effectively overwrite the weight values. This occurs because parameters are passed by reference in CUTLASS. In this sense, we protect the values stored in memory; therefore, we protect against errors in RAM.

5.3.2.1.3 Python Front End

In order to perform the convolution in CUTLASS, we extend the internal call hierarchy so that the three new attributes introduced (the mask tensors, an inference counter, and any auxiliary state) are propagated through the `Conv2d` class and its derived function `Conv2dFprop()`. In this way, these attributes are also instantiated within CUTLASS and, crucially, can be transferred to GPU memory for device-side use, avoiding repeated transfers from the CPU (where PyTorch executes). This design enables the masks to persistently reside on the device across successive inferences.

5.3.2.2 PyTorch Modifications

Secondly, we describe the PyTorch-side modifications required to integrate CUTLASS. PyTorch serves as the framework, but the actual model execution is managed by the Python modules within the `torchvision` repository, where each network is defined by its layers, forward passes, and weight tensors in a Python file.

To invoke CUTLASS, we introduce new convolution wrapper functions and substitute the default calls with these wrappers. As part of this change, we add a configuration flag *use_cutlass* that enables or disables the CUTLASS path (and thus the associated protections).

To deploy the VP functionality, we implement a new class, `CutlassConv2d`, which encapsulates the CUTLASS-backed convolution. This class maintains three additional attributes: (i) tensors storing the protection masks, (ii) an internal counter for the number of inferences executed (useful for bookkeeping and amortising initialisation costs), and (iii) any auxiliary state needed by the CUTLASS front end. During evaluation of a dataset, these attributes are already available (precomputed) and are reused across images, thereby avoiding the re-computation of masks and minimising overhead.

In addition, we provide specialised entry functions calling `CutlassConv2d` using common kernel shapes, specifically for 3×3 and 1×1 convolutions, to allow minor tuning and clearer dispatch. We predefine these calls due to the fact that (i) these entry calls are also defined in the model when using the default *torch.nn.Conv2d*, and (ii) because we can automatically set the bias, stride or other attributes for these kinds of kernel shapes that are also predefined by `torchvision` using CuDNN. We replicate the behaviour for our `CutlassConv2d` class.

Finally, the `CutlassConv2d` layer, defined directly within the Python file of the NN model, invokes the CUTLASS Python API/Frontend in its `forward` method, which ultimately calls the underlying C++/CUDA kernels (e.g. `Conv2dFProp`). In this way, the default `nn.Conv2d` is transparently replaced by a CUTLASS-enabled operator controlled by the *use_cutlass* flag, without altering the high-level model interface or its numerical behaviour.

5.3.2.3 User View

Finally, in our implementation, the user remains agnostic to the internal modifications. Basically, the workflow consists of creating a Python script and importing the `torchvision` library (which requires PyTorch to be installed). The user may then import any of the CNN models described and, following the standard `torchvision` procedure, simply enable a new flag (set to `true`) to activate CUTLASS, as outlined in Figure 5.4.

Enabling this flag activates CUTLASS, and all convolutions in the model are subsequently executed using the CUTLASS library. From an execution perspective, there is no difference in model usage or loss of accuracy. Therefore, at the end of execution, the user will receive the correct value of all required inferences with no further discrepancies.

5.3.2.4 Performance Results

Before analysing the performance of CUTLASS with VP, we conducted a study on the impact of implementing VP protections on convolutions computed for different input shapes. For this experiment, we utilise a custom convolutional kernel in CUDA, we apply VP, and then we execute it on the GPU. This involves defining groups and subsequently having GPU threads iterate over these groups, either for mask calculation or application. For the calculation, a *bitwise AND* operation is performed with each element in the group to detect invariant bits with a value of 0, and a *bitwise OR* operation is used to detect invariant bits with a value of 1. The application process is as described in Chapter 4.

The kernel is launched with as many threads as the number of output pixels required to be computed by a convolution operation with I x H x W input data. We performed O x I x H x W convolutions, with O and I (output and input channels) in the range of 16 to 2048, an image shape (H x W) of *64 x 64*, *128 x 128* and *256 x 256* using 3 x 3 filters (one for each I x O). We show the overheads in Figure 5.5.

Each thread uses all filters only once (as it computes only one output pixel). This is the worst-case scenario for VP since it will apply the mask on every read of filters from memory. We have used the average execution time per convolution (out of 100 operations) on the GPU to calculate the overheads.

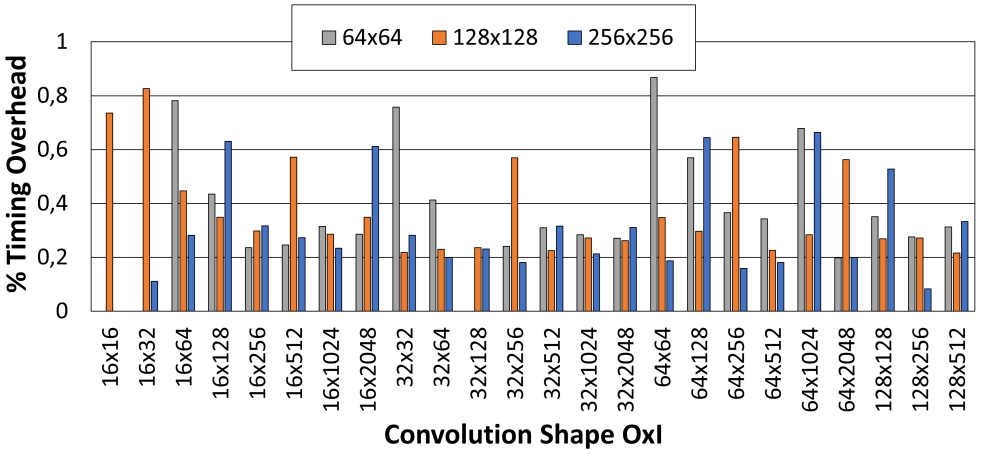
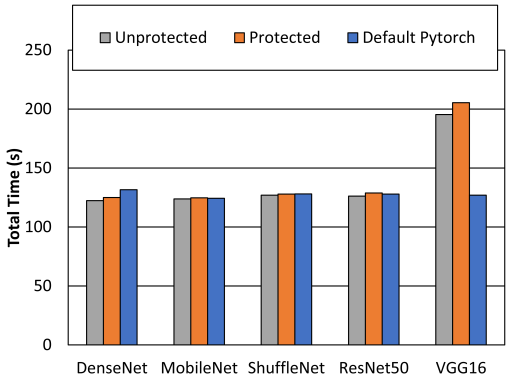
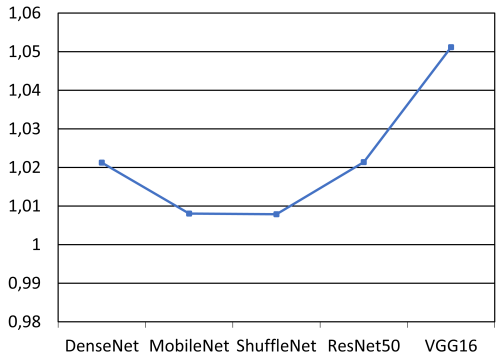


FIGURE 5.5: Timing overheads with different Image Input Shapes.

As can be seen, the overheads vary from 0.01% to nearly 0.9% with convolutions of significantly greater computational demand (shape 64x2048x256x256). It is also worth mentioning that the convolution operation is the most computationally expensive, as it involves multiple multiplications, representing the worst-case scenario. Furthermore, it should be noted that the overheads do not exhibit a clear trend and vary depending on the size of the convolution, leading us to deduce that there is no explicit evolutionary overhead. However, in these cases, the impact of our mechanisms is negligible, even in the worst-case scenario.



(A) Execution time performance.



(B) Slowdown Cutlass Protected vs Cutlass Unprotected.

FIGURE 5.6: Results comparison running Imagenet Dataset Inference using PyTorch with default CuDNN and with CUTLASS: *Unprotected* and *Protected* (with VP).

Now we show performance results of the whole CUTLASS implementation in Figure 5.6. These measurements were repeated 664 times to achieve a confidence interval of 99%, with an error margin of 5%, with the equation described in Section 4.4.1.1. In Figure 5.6a,

we present the total execution time over the ImageNet dataset and the corresponding SpeedUp when comparing default PyTorch against CUTLASS with protection enabled. Figure 5.6b shows the Slowdown comparing CUTLASS with protections activated and CUTLASS execution without modifications and unprotected.

In general, the results indicate that, for all convolutional models except VGG16, the use of CUTLASS does not adversely affect execution time compared to the default PyTorch configuration that uses CuDNN.

First, we compare CUTLASS without protections against the default PyTorch baseline. We can observe some gains in several models, especially in DenseNet, where the improvement increases by a further 6-7% over the default PyTorch execution. In ResNet50, we also have an improvement of 3-4%, more than in ShuffleNet with 1-2% As a consequence of these results, CUTLASS is proven to be a reliable and optimised library to perform convolutions.

Second, when comparing CUTLASS without protections directly against CUTLASS with protections, we use the slowdown metric, which measures the difference in execution time between the default CUTLASS baseline and the CUTLASS-Protected configuration. Values greater than 1 indicate a CUTLASS-Protected configuration with slower execution. This evolution is shown in Figure 5.6b. For VGG16, the slowdown introduced by the protection mechanism is 5%. In comparison, the DenseNet has a rate of around 2%, and the same applies to ResNet50. In MobileNet and ShuffleNet, the performance difference is not significant.

The most notable exception is VGG16, where the execution time was augmented by 68ms, representing a nearly one-third reduction in performance when using CUTLASS, similar to both protected and unprotected, compared to the default PyTorch. This substantial loss is attributed to how PyTorch, through CuDNN, handles convolution operations in this specific architecture.

VGG16 uses filters of shape 3×3 with stride and padding 1 (as we checked in Table 3.3) convolutions across the whole model, precisely the case where CuDNN typically selects fast transform-based algorithms (e.g., Winograd/FFT) [62]that reduce the arithmetic relative to direct/implicit-GEMM methods (fewer floating point operations)[85].

Replacing `torch.nn.Conv2d` (CuDNN) with a CUTLASS GEMM/MMA kernel, therefore loses this algorithmic advantage and tends to be memory-bound in early VGG convolutional layers. This is due to internal considerations; CuDNN chooses the type of convolution that yields the best performance (Winograd, GEMM, or FFT). For example, the input shape in these layers is lower, with activations of 224, 112, 56 or 24 pixels, which creates less arithmetic intensity and more memory traffic when using GEMM/MMA instead of Winograd.

By contrast, ShuffleNet and DenseNet incorporate numerous 1×1 filter convolutions into their designs, which map cleanly to GEMM [84] and thus leverage CUTLASS's implementation strengths, explaining the similar or slightly better runtimes we have described.

Finally, CUTLASS achieves peak performance using tensors with NHWC layout, 128-bit alignment, and channel/filter counts that are multiples of 32 [83]. In contrast, PyTorch and CuDNN maintain the NCHW format and also fuse the conv, bias, and activation paths, which can further widen the gap in VGG16 [87].

In conclusion, the implementation is most effective for mixed models or those with a higher proportion of 1×1 filters. For example, *ResNet50*, which contains only around 53% of convolutional layers with 1×1 filters, attains the same performance. In such models, CUTLASS offers an efficient execution path with no slowdown even when protections are enabled for every inference due to its optimisation for these cases.

5.4 Summary

In this chapter, we have implemented the proposed mechanisms on two distinct platforms: FPGAs and GPUs.

Through the integration of FP and VP mechanisms into an FPGA-based CNN inference accelerator, we have demonstrated a highly efficient approach to enhancing the fault resilience of neural network inference processes without impacting latency. The minimal FPGA resource overhead observed, coupled with the preservation of original execution latency, underscores the practicality and scalability of these mechanisms for deployment in real-world embedded systems where fault tolerance is of paramount importance. The

modular and pipeline-friendly design approach ensures that FP and VP mechanisms can be seamlessly integrated into various CNN architectures, providing flexible and robust solutions adaptable to diverse reliability requirements.

On GPUs, we first integrated VP into CUTLASS to support full neural-network models in a production-oriented environment. This end-to-end pathway confirms the practicality and portability of the mechanisms across heterogeneous execution platforms.

Chapter 6

Conclusions

This thesis addresses the problem of how faults affect the parameter (weights) values of Neural Networks, with a particular focus on CNNs.

We identified a high degree of bit-level invariability within model parameters. The observed invariability motivated the proposal of mechanisms that exploit, preserve, and reuse such invariability during subsequent inferences. FP and VP were implemented and validated in real execution environments, including FPGA-based accelerators and GPU kernels, leveraging industrial frameworks such as PyTorch.

This chapter quantifies and summarises the principal contributions of the thesis and outlines future research directions arising from this work.

6.1 Contributions

In Chapter 3, a systematic study of the weights in several neural network model architectures, including CNNs and ViTs, was conducted. This empirical investigation is carried out through a detailed analysis of the weights present in the convolutional layers of CNN models and in the remaining layers of the model. The study showed that there are bit-level invariability patterns that can be exploited. This invariability is demonstrated by grouping neural network weights according to several strategies, including the 2D filter-based approach ($O \times I \times KH \times KW$), which gives the best results compared to other strategies such as grouping by input channel (I) or output channel (O).

Regarding the data type, in 32-bit floating point, most invariability is concentrated in the exponent (bits 30-27), with around 4-6 invariant bits. This is high but consistent, since the values are centred around zero. In INT8, the degree of invariability decreases, and it also appears in the higher-weight bits, particularly at positions 5 and 6. Another important observation is that invariability is not restricted to CNNs; it is also present in ViT models, where weights are grouped using the same sizes as in CNNs, despite differences in their internal distribution. These groups yield results that are similar to, or even better than, those of CNNs and can therefore be generalised beyond convolutions. This finding is key, as it supports the view that invariability is an inherent property of both CNNs and ViTs, forming the basis of our protection mechanisms.

Based on these findings, Chapter 4 introduces two novel, lightweight, and model-agnostic protection mechanisms: Fixed Protection (FP) and Variable Protection (VP). These mechanisms exploit, analyse and detect bit-level invariability to safeguard invariant bits and, in the presence of bit-flip faults, to compare, detect, and restore the original values. FP protects consecutive ranges (e.g., exponents in 32-bit floating point, bits 30-23), while VP extends coverage to non-consecutive positions.

An extensive evaluation was also carried out through a fault-injection campaign, which demonstrated the effectiveness of the proposed methods. The results show that, whereas unprotected models suffer an accuracy degradation between 1.3% and over 3%, the application of FP and VP reduces this impact dramatically between roughly 7.5x and 30,000x smaller, representing an improvement of several orders of magnitude. In INT8, however,

the effect of bit flips is smaller due to the reduced numerical range, and therefore FP/VP provide a more modest improvement. Furthermore, when compared to state-of-the-art techniques, such as Clipper, FP and VP, achieve better results in all scenarios. Finally, the study also examined the criticality of faults outside convolutional layers and found them to be less severe than faults in convolutional filters, confirming that protecting the latter addresses the majority of the risk.

Finally, in Chapter 5, the application of the FP and VP mechanisms is validated through their implementation in real hardware execution environments, FPGAs and GPUs. First, in the FPGA-based accelerator, it is shown that the protection modules integrate efficiently into the architecture without introducing additional latency, thanks to a pipelined design and the possibility of including specific hardware support. The resource overhead is also minimal, ranging from 1% to 3%.

For the GPU implementation, the VP mechanism was integrated into the high-performance NVIDIA CUTLASS library, demonstrating compatibility within industrial ecosystems such as PyTorch. The results show that, for most models, the protection does not degrade runtime, thereby consolidating the portability and feasibility of our proposal across heterogeneous platforms.

In summary, this thesis has comprehensively studied bit flip errors in various NN models using both 32-bit floating-point numbers and 8-bit integers. Additionally, it has introduced and implemented two types of protection mechanisms: Fixed Protection (FP) and Variable Protection (VP). To the best of our knowledge, our work is the first to propose mechanisms based on bit invariability. Furthermore, we establish the compatibility of FP and VP with other mechanisms, showcasing their adaptability. These protections can be applied to any group of weights for any neural network model, highlighting their versatility. Their success will also depend on how critical the layers containing these weights are. As we have analysed with a structure completely independent of CNNs, such as Vision Transformer models, the percentage of covered bits is similar in both topologies. Therefore, there is no distinction in the use of our mechanisms between them. If there is any, it will depend on how critical the layer or weight is where a bit flip occurs, which is inherent to our mechanism and the topology used. These mechanisms have been successfully tested and applied to hardware environments.

6.2 Future Directions

As future work, we identify two avenues of research. One approach would be to increase the coverage of our mechanisms to enhance their effectiveness, particularly by protecting more bits in INT8, as well as additional mantissa bits in 32-bit floating-point int models. The second is to implement both FP and VP in other hardware architectures that are more susceptible to faults.

To increase the coverage, we propose conducting a comprehensive examination of the training process for these neural network models. Our mechanisms have demonstrated the efficacy of protecting invariant bits. Consequently, the subsequent step would be to train models and enforce the invariability of certain bits across the same sets. A preliminary approach would be to define a set of weights where seven or eight weights have the same bit in a specific position, thereby forcing the final weight to be invariant across the set. Subsequently, the training process would be employed to maintain this bit change. A successful outcome would be maintaining the model's accuracy or reducing the loss incurred by a bit flip in the new invariant bit to a less critical level than the original bit flip.

A second line of work could focus in implementing and integrating both fault-tolerance mechanisms within the SECDA-TFLite environment [42], beginning with the software protection *Variable Protection* (VP) applied to convolutional weights. This is complemented by a hardware protection based on the replication of multiplication operations in VMM (*Vector Matrix Multiplication*) units. This work was carried out during an internship at the University of Glasgow and is currently at an advanced stage. We expect to continue this work in the future.

The SECDA environment operates with *delegates*, which are responsible for each layer and operation in neural network models. The scheduler reads the NNs and directs to each delegate the data or the execution environment in which the operation will be performed, either CPU or GPU.

On the software side, VP has so far been integrated into the delegate responsible for convolution. The protection masks are computed in the driver (CPU) prior to execution, utilising parallel multitasking and threads to accelerate preprocessing, batch the mask

computation, and pack. These masks are sent in *streaming* alongside the weights and are consumed by a dedicated accelerator module that restores bits when discrepancies with the mask are detected. On the hardware side, functionality has been developed to exploit the four VMMs with data replication: two VMMs compute the same operations in parallel, and their results are compared to detect discrepancies in the multiplication operators, accepting the associated performance penalty (approximately $\times 0.5$) in exchange for greater fault-detection capability.

In summary, our goal is to enhance both the utility of the proposed mechanisms and their application in additional projects. Their use within SECDA not only enables the deployment of VP (and, where appropriate, FP) in a new execution environment, but also provides the opportunity to extend both protections to other operations and model types, including LLMs.

6.3 Publications

Below, we present the publications related to the research performed in this thesis. These contributions appeared in journals, book chapters, together at national or international conferences:

- **Exploiting neural networks bit-level redundancy to mitigate the impact of faults at inference**. I. Catalán Gallach; J. Flich Cardo; C. Hernández Luz. *The Journal of Supercomputing*, vol. 81, no. 1, pp. 581-590, 2024. DOI: 10.1007/s11227-024-06693-7.

- **Efficient Inference Of Image-Based Neural Network Models In Reconfigurable Systems With Pruning And Quantization**. J. Flich Cardo; L. Medina Chaveli; I. Catalán Gallach; C. Hernández Luz; A. Bragagnolo; F. Auzanneau; D. Briand. *29th IEEE International Conference on Image Processing (ICIP)*, Bordeaux, France, pp. 2491-2495, 2022. DOI: 10.1109/ICIP46576.2022.9897752.

- **The DeepHealth HPC Infrastructure: Leveraging Heterogeneous HPC and Cloud Computing Infrastructures for AI-based Medical Solutions**. I. Catalán Gallach, R. Tornero-Gavilá, D. Rodríguez-Agut, J. M. Martínez Martínez,

J. Flich Cardo, C. Hernández Luz. In: *HPC, Big Data, and AI Convergence Towards Exascale: Challenge and Vision*, CRC Press, Taylor & Francis Group, 2022, pp. 191-217. ISBN: 9781032009841.

- **Mitigando el Impacto de los Fallos en CNNs con Protección Fija y Variable**. I. Catalán Gallach; J. Flich Cardo; C. Hernández Luz. *XXXV Jornadas de Paralelismo, Jornadas SARTECO 2025*, pp. 151-159, Sevilla, España (2025). Sociedad Española de Arquitectura y Tecnología de Computadores. ISBN/ISSN: 978-84-09-74530-2.

- **Estudio sobre la redundancia a nivel de bit y el impacto de Bit Flips en las Redes Neuronales Convolucionales**. I. Catalán Gallach; J. Flich Cardo; C. Hernández Luz. *XXXIII Jornadas de Paralelismo. Jornadas SARTECO 2023*, pp. 155-160, Ciudad Real, España (2023). Sociedad Española de Arquitectura y Tecnología de Computadores. ISBN/ISSN: 978-84-09-54466-0.

- **Comparison of Convolutional Algorithms on FPGAs Devices**. I. Catalán Gallach; J. Flich Cardo; C. Hernández Luz. *18th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems (ACACES 2022)*, pp. 101-104, Fiuggi, Italia (2022). ISBN/ISSN: 978-88-947027-0-5.

- **Implementación de Algoritmos de Convolución en Plataforma de Inferencia para FPGAs**. I. Catalán Gallach; J. Flich Cardo; C. Hernández Luz. *XXXII Jornadas de Paralelismo. Jornadas SARTECO 2022*, pp. 217-223, Alicante, España (2022). Universidad de Alicante. ISBN/ISSN: 978-84-1302-185-0.

- **HLSinf: Una Plataforma de Aceleración de Procesos de Inferencia en FPGA aplicado a Imágenes Médicas**. L. Medina-Chaveli; I. Catalán Gallach; J. Flich Cardo; C. Hernández Luz; A. Bragagnolo; F. Auzanneau; D. Briand. *XXXII Jornadas de Paralelismo. Jornadas SARTECO 2022*, pp. 185-190, Alicante, España (2022). Universidad de Alicante. ISBN/ISSN: 978-84-1302-185-0.

- **Implementing Fault Protection Mechanisms using SECDA-TFLite toolkit**. I. Catalán Gallach; J. Flich Cardo; C. Hernández Luz; J. Harris; R. Saha and J. Cano. *Design, Automation and Test in Europe Conference 2026* (DATE 2026). *Under Submission.*

References

[1] Syed Tuhin Ahmed, Surendra Hemaram, and Mehdi B. Tahoori. NN-ECC: Embedding error correction codes in neural network weight memories using multi-task learning. In *2024 IEEE VLSI Test Symposium (VTS)*. IEEE, 2024. doi: 10.1109/VTS60656.2024.10538886. Cited in text as 2025; actual venue year is 2024.

[2] Richmond Alake. A data scientist's guide to gradient descent and backpropagation algorithms. *NVIDIA Developer Blog*, February 2022. URL `https://developer.nvidia.com/blog/a-data-scientists-guide-to-gradient-descent-and-backpropagation-algorithms/`. Accessed: 2025-05-28.

[3] Syed Asad Alam, Andrew Anderson, Barbara Barabasz, and David Gregg. Winograd convolution for deep neural networks: Efficient point selection. *ACM Trans. Embed. Comput. Syst.*, 21(6), December 2022. ISSN 1539-9087. doi: 10.1145/3524069. URL `https://doi.org/10.1145/3524069`.

[4] AMD (Xilinx). Fpga architecture (ug1291), 2023. URL `https://docs.amd.com/r/en-US/ug1291-viv/FPGA-Architecture`. Official architectural description; accessed 24 Sep 2025.

[5] Apache. Mxnet library for deep learning, 2023. URL `https://mxnet.apache.org/versions/1.9.1/`.

[6] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017. doi: 10.1109/TPAMI.2016.2644615.

[7] Iljoo Baek, Wei Chen, Zhihao Zhu, Soheil Samii, and Ragunathan Raj Rajkumar. Ft-deepnets: Fault-tolerant convolutional neural networks with kernel-based duplication. In *2022 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 1878–1887, 2022. doi: 10.1109/WACV51458.2022.00194.

[8] Luiz André Barroso, Urs Hölzle, Parthasarathy Ranganathan, and Margaret Martonosi. *The Datacenter As a Computer: Designing Warehouse-scale Machines*. Morgan & Claypool Publishers, 3rd edition, 2018. ISBN 1681734354.

[9] Timoteo García Bertoa, Giulio Gambardella, Nicholas J. Fraser, Michaela Blott, and John McAllister. Fault-tolerant neural network accelerators with selective tmr. *IEEE Design & Test*, 40(2):67–74, 2023. doi: 10.1109/MDAT.2022.3174181.

[10] Alberto Bosio, Paolo Bernardi, Annachiara Ruospo, and Ernesto Sanchez. A reliability analysis of a deep neural network. In *2019 IEEE Latin American Test Symposium (LATS)*, pages 1–6, 2019. doi: 10.1109/LATW.2019.8704548.

[11] Andrew Boutros, Mathew Hall, Nicolas Papernot, and Vaughn Betz. Neighbors from hell: Voltage attacks against deep learning accelerators on multi-tenant fpgas. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 103–111, 2020. doi: 10.1109/ICFPT51103.2020.00023.

[12] Jakub Breier and Xiaolu Hou. Feeding two cats with one bowl: On designing a fault and side-channel resistant software encoding scheme. In *Springer International Publishing*, pages 77–94, 2017. doi: 10.1007/978-3-319-52153-4_5.

[13] Jakub Breier, Xiaolu Hou, and Yang Liu. On evaluating fault resilient encoding schemes in software. *IEEE Transactions on Dependable and Secure Computing*, 18 (3):1065–1079, 2021. doi: 10.1109/TDSC.2019.2897663.

[14] Brilliant Math & Science Wiki. Feedforward neural networks. `https://brilliant.org/wiki/feedforward-neural-networks/`, 2025. Intro definition: forward-only flow from input to output; accessed 30 Sep 2025.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon

Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[16] Sandhya Chandrashekhar, Helmut Puchner, Jun Mitani, Satoshi Shinozaki, Mohamed Sardi, and David Hoffman. Radiation induced soft errors in 16 nm floating gate slc nand flash memory. *Microelectronics Reliability*, 108:113631, 2020. ISSN 0026-2714. doi: https://doi.org/10.1016/j.microrel.2020.113631. URL `https://www.sciencedirect.com/science/article/pii/S002627141930633X`.

[17] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), 2006. Université de Rennes 1, Suvisoft. URL `https://inria.hal.science/inria-00112631`.

[18] NVIDIA Corporation. Cutlass: Cuda templates for linear algebra subroutines. `https://github.com/NVIDIA/cutlass`, 2023. Accessed: 2025-05-29.

[19] NVIDIA Corporation. Tesla v100. `https://www.nvidia.com/en-gb/data-center/tesla-v100/`, 2025. Accessed: 2025-05-29.

[20] Databricks. What is a convolutional layer?, 2019. URL `https://www.databricks.com/glossary/convolutional-layer`.

[21] DataScientest. Dense neural networks: Understanding their structure and function, 2025. URL `https://datascientest.com/en/dense-neural-networks-understanding-their-structure-and-function`.

[22] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi: 10.1109/JSSC.1974.1050511.

[23] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=YicbFdNTTy`.

[24] Mathieu Dumont, Pierre-Alain Moellic, Raphael Viera, Jean-Max Dutertre, and Rémi Bernhard. An overview of laser injection against embedded neural network models. In *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pages 616–621, 2021. doi: 10.1109/WF-IoT51360.2021.9595075.

[25] Edge ai + vision. Global ai adoption to surge 20%, exceeding 378 million users in 2025, 2025. URL `https://www.edge-ai-vision.com/2025/02/global-ai-adoption-to-surge-20-exceeding-378-million-users-in-2025/`. Accessed June 5, 2025.

[26] Encyclopaedia Britannica. Artificial intelligence (ai), 2025. URL `https://www.britannica.com/technology/artificial-intelligence`. Encyclopedic definition; accessed 24 Sep 2025.

[27] Encyclopaedia Britannica. Neural network, 2025. URL `https://www.britannica.com/technology/neural-network`. Encyclopedic definition; accessed 24 Sep 2025.

[28] José Flich, Laura Medina, Izan Catalán, Carles Hernández, Andrea Bragagnolo, Fabrice Auzanneau, and David Briand. Efficient inference of image-based neural network models in reconfigurable systems with pruning and quantization. In *2022 IEEE International Conference on Image Processing (ICIP)*, pages 2491–2495, 2022. doi: 10.1109/ICIP46576.2022.9897752.

[29] Giulio Gambardella, Johannes Kappauf, Michaela Blott, Christoph Doehring, Martin Kumm, Peter Zipf, and Kees Vissers. Efficient error-tolerant quantized neural network accelerators. In *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, page 1–6. IEEE, October 2019. doi: 10.1109/dft.2019.8875314. URL `http://dx.doi.org/10.1109/DFT.2019.8875314`.

[30] Giulio Gambardella, Nicholas J. Fraser, Ussama Zahid, Giuseppe Furano, and Michaela Blott. Accelerated radiation test on quantized neural networks trained with fault aware training. In *2022 IEEE Aerospace Conference (AERO)*, pages 1–7. IEEE, 2022. doi: 10.1109/AERO53065.2022.9843614.

[31] GeeksforGeeks. Vision transformer (vit) architecture. `https://www.geeksforgeeks.org/deep-learning/vision-transformer-vit-architecture/`, 2025. Last updated: 23 Jul 2025; accessed 30 Sep 2025.

[32] GeeksforGeeks. What is batch normalization in deep learning?, 2025. URL `https://www.geeksforgeeks.org/what-is-batch-normalization-in-deep-learning/`.

[33] GeeksforGeeks. Cnn | introduction to pooling layer, 2025. URL `https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/`.

[34] GeeksforGeeks contributors. Cnn | introduction to pooling layer. *GeeksforGeeks*, 2025. URL `https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/`. Accessed: 2025-05-28.

[35] Florian Geissler, Syed Sha Qutub, Sayanta Roychowdhury, Ali Asgari Khoshouyeh, Yang Peng, Akash Dhamasia, Ralf Graefe, Karthik Pattabiraman, and Michael Paulitsch. Towards a safety case for hardware fault tolerance in convolutional neural networks using activation range supervision. *CoRR*, abs/2108.07019, 2021. URL `https://arxiv.org/abs/2108.07019`.

[36] Github. Onnx bit analysis, 2023. URL `https://github.com/IzanCatalan/OnnxBitAnalysis`.

[37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, chapter Deep Feedforward Networks. MIT Press, 2016. URL `https://www.deeplearningbook.org/contents/mlp.html`. Also called multilayer perceptrons (MLPs); accessed 30 Sep 2025.

[38] Google Developers. Neural networks: Training using backpropagation. `https://developers.google.com/machine-learning/crash-course/neural-networks/backpropagation`, 2025. Accessed: 2025-05-28.

[39] Xiaofei Guo and Ramesh Karri. Recomputing with permuted operands: A concurrent error detection approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(10):1595–1608, 2013. doi: 10.1109/TCAD.2013.2263037.

[40] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. doi: 10.1002/j.1538-7305.1950.tb00463.x.

[41] Muhammad Hanif and Muhammad Shafique. Salvagednn: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378:20190164, 12 2019. doi: 10.1098/rsta.2019.0164.

[42] Jude Haris, Perry Gibson, Jose Cano, Nicolas Bohm Agostini, and David Kaeli. Secda-tflite: A toolkit for efficient development of fpga-based dnn accelerators for edge inference. *Journal of Parallel and Distributed Computing*, 173:140–151, 2023. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2022.11.005.

[43] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 9780080556017.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[45] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14083–14091, 2020. doi: 10.1109/CVPR42600.2020.01410.

[46] Le-Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. In *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*, DATE '20, page 1241 to 1246, San Jose, CA, USA, 2020. EDA Consortium. ISBN 9783981926347. doi: 10.23919/DATE48585.2020.9116571.

[47] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998. doi: 10.1142/S0218488598000094.

[48] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraş. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *Proceedings of the 28th USENIX Conference on Security Symposium*, page 497–514, Santa Clara, CA, USA, 2019. ISBN 9781939133069. URL `https://dl.acm.org/doi/10.5555/3361338.3361373`.

[49] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017. doi: 10.1109/CVPR.2017.243.

[50] Kejie Huang, Fei Ye, and Yier Jin. Functional error correction for reliable neural networks. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 2694–2699. IEEE, 2020. doi: 10.1109/ISIT44484.2020.9174137.

[51] ImageNet. Data set, 2023. URL `https://www.image-net.org/about.php`.

[52] Intel. Open vino model zoo repository, 2023. URL `https://docs.openvino.ai/2023.2/model_zoo.html`.

[53] Mojan Javaheripi and Farinaz Koushanfar. Hashtag: Hash signatures for misc detection of fault-injection attacks on deep neural networks. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. doi: 10.1109/ICCAD51958.2021.9643556.

[54] Jinteng Jiao, He Li, Yanzhao Feng, Chengdong Qian, and Qiang Liu. In-depth analysis of the effects of electromagnetic fault injection attack on a 32-bit mcu. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, pages 1–6, 2022. doi: 10.1109/SOCC56010.2022.9908097.

[55] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google tpuv4i : Industrial

product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021. doi: 10.1109/ISCA52012.2021.00010.

[56] Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training. *CoRR*, abs/1905.12322, 2019. URL `http://arxiv.org/abs/1905.12322`.

[57] Yash Khare, Kumud Lakara, Maruthi S. Inukonda, Sparsh Mittal, Mahesh Chandra, and Arvind Kaushik. Design and analysis of novel bit-flip attacks and defense strategies for dnns. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–8, 2022. doi: 10.1109/DSC54232.2022.9888943.

[58] Saman Kiamehr, Farshad Firouzi, and Mehdi B. Tahoori. Input and transistor reordering for nbti and hci reduction in complex cmos gates. In *Proceedings of the Great Lakes Symposium on VLSI*, GLSVLSI '12, page 201–206, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312448. doi: 10.1145/2206781.2206829. URL `https://doi.org/10.1145/2206781.2206829`.

[59] Jae-San Kim and Joon-Sung Yang. Dris-3: Deep neural network reliability improvement scheme in 3d die-stacked memory based on fault analysis. pages 1–6, 06 2019. doi: 10.1145/3316781.3317805.

[60] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014. doi: 10.1109/ISCA.2014.6853210.

[61] P. Koopman and T. Chakravarty. Cyclic redundancy code (crc) polynomial selection for embedded networks. In *International Conference on Dependable Systems and Networks, 2004*, pages 145–154, 2004. doi: 10.1109/DSN.2004.1311885.

[62] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016. doi: 10.1109/CVPR.2016.435.

[63] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015. doi: 10.1038/nature14539.

[64] Seo-Seok Lee and Joon-Sung Yang. Value-aware parity insertion ECC for fault-tolerant deep neural network. In *Proceedings of the 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 724–729. IEEE, 2022. doi: 10.23919/DATE54114.2022.9774543.

[65] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 502–506, 2009. doi: 10.1109/DATE.2009.5090716.

[66] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351140. doi: 10.1145/3126908.3126964.

[67] Jingtao Li, Adnan Siraj Rakin, Yan Xiong, Liangliang Chang, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Defending bit-flip attack through dnn weight reconstruction. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi: 10.1109/DAC18072.2020.9218665.

[68] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Radar: Run-time adversarial weight attack detection and accuracy recovery. *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021. doi: 10.23919/DATE51398.2021.9474113.

[69] Wenshuo Li, Guangjun Ge, Kaiyuan Guo, Xiaoming Chen, Qi Wei, Zhen Gao, Yu Wang, and Huazhong Yang. Soft error mitigation for deep convolution neural network on fpga accelerators. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 1–5, 2020. doi: 10.1109/AICAS48895.2020.9073925.

[70] Yu Li, Min Li, Bo Luo, Ye Tian, and Qiang Xu. Deepdyve: Dynamic verification for deep neural networks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 101–112, 2020. ISBN 9781450370899. doi: 10.1145/3372297.3423338.

[71] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating robust dnn with resistance to bit-flip based adversarial weight attack. *IEEE Transactions on Computers*, pages 401–413, 2023. doi: 10.1109/TC.2022.3211411.

[72] Qi Liu, Wujie Wen, and Yanzhi Wang. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–8, 2020. doi: 10.1145/3400302.3415726.

[73] Shuying Liu and Weihong Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734, 2015. doi: 10.1109/ACPR.2015.7486599.

[74] Tao Liu, Wujie Wen, Lei Jiang, Yanzhi Wang, Chengmo Yang, and Gang Quan. A fault-tolerant neural network architecture. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[75] Wenye Liu, Chip-Hong Chang, Fan Zhang, and Xiaoxuan Lou. Imperceptible misclassification attack on deep learning accelerator by glitch injection. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi: 10.1109/DAC18072.2020.9218577.

[76] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 122–138, Cham, 2018. Springer International Publishing. ISBN 978-3-030-01264-9. doi: 10.1007/978-3-030-01264-9_8.

[77] Thibaut Marty, Tomofumi Yuki, and Steven Derrien. Safe overclocking for cnn accelerators through algorithm-level error detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4777–4790, 2020. doi: 10.1109/TCAD.2020.2981056.

[78] John McCarthy, Marvin L. Minsky, Nathaniel Rochester, and Claude E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Magazine*, 27(4):12, Dec. 2006. doi: 10.1609/aimag.

v27i4.1904. URL https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1904.

[79] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. doi: 10.1109/N-SSC.2006.4785860.

[80] Nicolás Landeros Muñoz, Alejandro Valero, Rubén Gran Tejero, and Davide Zoni. Gated-CNN: Combating NBTI and HCI aging effects in on-chip activation memories of CNN accelerators. *Journal of Systems Architecture*, 128:102553, 2022. doi: 10.1016/j.sysarc.2022.102553.

[81] Adam Neale and Manoj Sachdev. Neutron radiation induced soft error rates for an adjacent-ecc protected sram in 28 nm cmos. *IEEE Transactions on Nuclear Science*, pages 1912–1917, 2016. doi: 10.1109/TNS.2016.2547963.

[82] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781450378451. doi: 10.1145/1401132.1401152. URL https://doi.org/10.1145/1401132.1401152.

[83] NVIDIA. Cutlass documentation: Functionality and performance recommendations. https://docs.nvidia.com/cutlass/media/docs/cpp/functionality.html, . Recommends NHWC layouts and 128-bit alignment for Tensor Core paths. Accessed 8 Oct 2025.

[84] NVIDIA. Implicit gemm convolution — cutlass documentation. https://docs.nvidia.com/cutlass/media/docs/cpp/implicit_gemm_convolution.html, . CUTLASS implements convolution via implicit GEMM. Accessed 8 Oct 2025.

[85] NVIDIA. Convolutional layers user's guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html, 2023. Describes cuDNN convolution algorithms (GEMM, Winograd, FFT). Accessed 8 Oct 2025.

[86] NVIDIA. Gpu performance background user's guide, 2023. URL https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html. Architectural background for GPUs; accessed 24 Sep 2025.

[87] NVIDIA. cudnn graph api: Operation fusion (convolution, bias, activation). https://docs.nvidia.com/deeplearning/cudnn/backend/v9.2.1/developer/graph-api.html, 2024. See also cudnnConvolutionBiasActivationForward (deprecated in cuDNN 9). Accessed 8 Oct 2025.

[88] Onnx. Api reference, 2023. URL https://onnx.ai/onnx/api/.

[89] Onnx. The open standard for machine learning interoperability, 2023. URL https://onnx.ai/.

[90] Onnx. Model zoo github repository, 2023. URL https://github.com/onnx/models.

[91] Onnx. Runtime cross-platform inference and training machine-learning accelerator, 2023. URL https://onnxruntime.ai/.

[92] ONNX. Operator: Conv. https://onnx.ai/onnx/operators/onnx__Conv.html#l-onnx-doc-conv, 2025. Accessed: 2025-07-04.

[93] Aravind Pai. 12 types of neural networks in deep learning. *Analytics Vidhya*, May 2025. URL https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/. Accessed: 2025-05-28.

[94] Pytorch. Empowering pytorch on intel® xeon® scalable processors with bfloat16, 2023. URL https://pytorch.org/blog/empowering-pytorch-on-intel-xeon-scalable-processors-with-bfloat16/.

[95] Pytorch. Export a pytorch model to onnx, 2023. URL https://pytorch.org/tutorials/beginner/onnx/export_simple_model_to_onnx_tutorial.html#export-a-pytorch-model-to-onnx.

[96] PyTorch Team. torch.nn.conv2d — pytorch documentation. https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html, 2025. Accessed: October 20, 2025.

[97] Syed Qutub, Florian Geissler, Yang Peng, Ralf Gräfe, Michael Paulitsch, Gereon Hinz, and Alois Knoll. Hardware faults that matter: Understanding and estimating the safety impact of hardware faults on object detection dnns. In *Computer Safety, Reliability, and Security: 41st International Conference, SAFECOMP 2022, Munich, Germany, September 6–9, 2022, Proceedings*, page 298–318, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-14834-7. doi: 10.1007/978-3-031-14835-4_20.

[98] Mohamed E. Raji, Shimaa A. El-Sayed, and Aly A. Fahmy. An ECC-based fault tolerance approach for DNNs. *arXiv*, 2025.

[99] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1211–1220, 2019. doi: 10.1109/ICCV.2019.00130.

[100] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1137–1149, jun 2017. ISSN 1939-3539. doi: 10.1109/TPAMI.2016.2577031.

[101] Veronica Rotemberg, Nicholas Kurtansky, Brigid Betz-Stablein, Liam Caffery, Emmanouil Chousakos, Noel Codella, Marc Combalia, Stephen Dusza, Pascale Guitera, David Gutman, Allan Halpern, Brian Helba, Harald Kittler, Kivanc Kose, Steve Langer, Konstantinos Lioprys, Josep Malvehy, Shenara Musthaq, Jabpani Nanda, and Peter Soyer. A patient-centric dataset of images and metadata for identifying melanomas using clinical context. *Scientific Data*, 8:34, 01 2021. doi: 10.1038/s41597-021-00815-z.

[102] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115, 09 2014. doi: 10.1007/s11263-015-0816-y.

[103] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597.

[104] Behzad Salami, Osman S. Unsal, and Adrian Cristal Kestelman. Comprehensive evaluation of supply voltage underscaling in fpga on-chip memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 724–736, 2018. doi: 10.1109/MICRO.2018.00064.

[105] Rick Salay, Rodrigo Queiroz, and Krzysztof Czarnecki. An analysis of iso 26262: Using machine learning safely in automotive software. 09 2017. doi: 10.4271/9780768002683.

[106] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks . In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, Los Alamitos, CA, USA, June 2018. IEEE Computer Society. doi: 10.1109/CVPR.2018.00474. URL `https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00474`.

[107] Fernando Fernandes dos Santos, Angeliki Kritikakou, Josie E. Rodriguez Condia, Juan-David Guerrero-Balaguera, Matteo Sonza Reorda, Olivier Sentieys, and Paolo Rech. Characterizing a neutron-induced fault model for deep neural networks. *IEEE Transactions on Nuclear Science*, 70(4):370–380, 2023. doi: 10.1109/TNS.2022.3224538.

[108] Fernando Fernandes dos Santos, Niccolò Cavagnero, Marco Ciccone, Giuseppe Averta, Angeliki Kritikakou, Olivier Sentieys, Paolo Rech, and Tatiana Tommasi. Improving deep neural network reliability via transient-fault-aware design and training. *IEEE Transactions on Emerging Topics in Computing*, 13(3):829–840, 2025. doi: 10.1109/TETC.2024.3520672.

[109] ScienceDirect Topics. Depthwise separable convolution – overview. `https://www.sciencedirect.com/topics/computer-science/depthwise-separable-convolution`, 2025. Reference topic page; accessed 26 Sep 2025.

[110] Jashanpreet Singh Sraw and M. C. Deepak. Using convolutional neural networks for fault analysis and alleviation in accelerator systems. In Sheng-Lung Peng, Cheng-Kuan Lin, and Souvik Pal, editors, *Proceedings of 2nd International Conference*

on Mathematical Modeling and Computational Science, pages 289–304, Singapore, 2022. Springer Nature Singapore. ISBN 978-981-19-0182-9.

[111] Pierre Stock. *Efficiency and Redundancy in Deep Learning Models : Theoretical Considerations and Practical Applications*. Theses, Université de Lyon, April 2021. URL `https://theses.hal.science/tel-03208517`.

[112] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019. doi: 10.1109/TEVC.2019.2890858.

[113] Rizwan Tariq Syed, Marko Andjelkovic, Markus Ulbricht, and Milos Krstic. Towards reconfigurable cnn accelerator for fpga implementation. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(3):1249–1253, 2023. doi: 10.1109/TCSII.2023.3241154.

[114] Olivier Temam. A defect-tolerant accelerator for emerging high-performance applications. New York, NY, USA, 2012. Association for Computing Machinery. doi: 10.1109/ISCA.2012.6237031.

[115] Vasileios Titopoulos, Kosmas Alexandridis, and Giorgos Dimitrakopoulos. Custom algorithm-based fault tolerance for attention layers in transformers. In *Proc. IEEE International System-on-Chip Conference (SOCC)*, 2025. URL `https://arxiv.org/abs/2507.16676`. Proposes Flash-ABFT: a single online checksum covering Q–K–V and softmax for low-overhead fault detection in attention.

[116] Yamilka Toca-Díaz, Rubén Gran Tejero, and Alejandro Valero. Shift-and-safe: Addressing permanent faults in aggressively undervolted CNN accelerators. *Journal of Systems Architecture*, 2024. doi: 10.1016/j.sysarc.2024.103292.

[117] Yamilka Toca-Díaz, Alejandro Valero, and Rubén Gran Tejero. A fault-tolerant technique for on-chip memories of CNN accelerators at ultra-low voltage: Flip-and-Patch. *Microprocessors and Microsystems*, 110:105023, 2024. doi: 10.1016/j.micpro.2024.105023.

[118] Torchvision. Pre-trained models repository, 2023. URL `https://pytorch.org/vision/stable/models.html`.

[119] Transparency Market Research. Embedded system market - global industry analysis, size, share, growth, trends, and forecast, 2022-2031, 2022. URL https://www.transparencymarketresearch.com/embedded-system.html. Accessed June 5, 2025.

[120] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 65–74, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450343541. doi: 10.1145/3020078.3021744. URL https://doi.org/10.1145/3020078.3021744.

[121] Alejandro Valero, Rubén Gran Tejero, and Yamilka Toca-Díaz. Impact on the accuracy of aggressive voltage underscaling in CNN accelerators. Preprint / technical report, 2023. URL https://www.researchgate.net/publication/372459375_Impact_on_the_Accuracy_of_Aggressive_Voltage_Underscaling_in_CNN_Accelerators. Accessed online; venue pending.

[122] A. Vasuki and S. Govindaraju. *Deep neural networks for image classification*, pages 27–49. 12 2017. doi: 10.3233/978-1-61499-822-8-27.

[123] Wikipedia contributors. Dartmouth workshop. *Wikipedia, The Free Encyclopedia*, 2024. URL https://en.wikipedia.org/wiki/Dartmouth_workshop. Accessed: 2025-07-04.

[124] Wikipedia contributors. Artificial intelligence, 2025. URL https://en.wikipedia.org/wiki/Artificial_intelligence. Definition page; accessed 24 Sep 2025.

[125] Wikipedia contributors. Activation function, 2025. URL https://en.wikipedia.org/wiki/Activation_function. Definition of nonlinear activation functions in neural networks.

[126] Wikipedia contributors. Artificial neuron, 2025. URL https://en.wikipedia.org/wiki/Artificial_neuron. Explains the bias term added to the weighted sum before the activation.

[127] Wikipedia contributors. Convolutional neural network, 2025. URL `https://en.wikipedia.org/wiki/Convolutional_neural_network`. Definition and core building blocks of CNNs.

[128] Wikipedia contributors. Deep learning, 2025. URL `https://en.wikipedia.org/wiki/Deep_learning`. Defines deep neural networks within the deep learning article.

[129] Wikipedia contributors. Field-programmable gate array, 2025. URL `https://en.wikipedia.org/wiki/Field-programmable_gate_array`. Definition and fundamentals; accessed 24 Sep 2025.

[130] Wikipedia contributors. Graphics processing unit, 2025. URL `https://en.wikipedia.org/wiki/Graphics_processing_unit`. Definition and overview; accessed 24 Sep 2025.

[131] Wikipedia contributors. Neural network, 2025. URL `https://en.wikipedia.org/wiki/Neural_network`. Definition (biological and artificial); accessed 24 Sep 2025.

[132] Wikipedia contributors. Softmax function, 2025. URL `https://en.wikipedia.org/wiki/Softmax_function`. Definition of the softmax as a normalized exponential mapping to probabilities.

[133] Wikipedia contributors. Vision transformer. `https://en.wikipedia.org/wiki/Vision_transformer`, 2025. Encyclopedic definition of ViT (image-to-patches, Transformer encoder, [CLS] token); accessed 30 Sep 2025.

[134] Wikipedia contributors. Convolutional neural network. `https://en.wikipedia.org/wiki/Convolutional_neural_network`, 2025. Accessed: 2025-05-28.

[135] Wayne Wolf. *Computers as components: principles of embedded computing system design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 155860541X.

[136] Dawen Xu, Cheng Chu, Qianlong Wang, Cheng Liu, Ying Wang, Lei Zhang, Huaguo Liang, and Kwang-Ting Cheng. A hybrid computing architecture for fault-tolerant deep learning accelerators. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 478–485, 2020. doi: 10.1109/ICCD50377.2020.00087.

[137] Ussama Zahid, Giulio Gambardella, Nicholas J. Fraser, Michaela Blott, and Kees Vissers. Fat: Training neural networks for reliable inference under hardware faults. In *2020 IEEE International Test Conference (ITC)*, pages 1–10, 2020. doi: 10.1109/ITC44778.2020.9325249.

[138] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979. doi: 10.1126/science.206.4420.776. URL `https://www.science.org/doi/abs/10.1126/science.206.4420.776`.