Politecnico di Torino

Politecnico di Torino

Mechatronic engineering

Master's thesis

New Techniques for automatic generation of input stimuli for detecting hardware faults in AI-oriented accelerators

Candidate:

Vittorio Turco

Advisor: **Prof. Matteo Sonza Reorda**

Co-advisor: **Prof. Ernesto Sanchez**

Co-advisor: **Dr. Annachiara Ruospo**

Academic Year 2022-2023

Acknowledgments

With the conclusion of this journey, I have the duty and pleasure to thank some people for the achievement of this thesis. Sincere thanks to Professor Matteo Sonza Reorda who allowed me to undertake this project by supporting my curiosity. I would like to thank co-advisor Ernesto Sanchez and especially Dr. Annachiara Ruospo, who helped and supported me both technically and humanely step by step throughout these months of work. Also, I would like to thank my close friends, hanging out and growing with them was very important to be able to achieve results like this. Finally, a big thank you to my family, who has always supported me throughout these years and allowed me to take whatever path I wanted without ever imposing a choice on me.

Contents

List of Tables

List of Figures

Introduction

In recent years, Deep Neural Networks (DNNs) have become increasingly present and used in any field, it is now a fundamental element for all artificial intelligence applications and is expanding more and more. Today these networks are being introduced in the medical, automotive, financial, and industrial fields, all requiring high-security parameters [18]. Hardware faults can compromise the functionality of the DDNs and the safety of the system on which they are connected, the increased use of DNNs in safety-critical domains has led to the need to accompany their exceptional performance with diagnostic systems to ensure safety and reliability. Therefore the issue of network resilience has become very important, it is now mandatory to introduce the ability to maintain standards performance while managing to inhibit, or correct, factors such as hardware-induced faults, wrong inputs, and any type of disturbance.

Considering a DNN for image recognition, this thesis proposes a new experimental technique to generate high-quality test stimuli in the form of automatically generated test images. the goal is to create an Automatic Test Pattern Generation (ATPG)-based approach to generate an Image Test Library (ITL) such as those mentioned in the article [11], with the task of identifying permanent faults. Going into detail, the main idea behind the thesis is to create a virtual test environment in which to simulate hardware-induced faults through a fault injection within various Neural Networks for image recognition. The ITLs are created by an evolutionary algorithm, which receives feedback on the observability of the outputs of the generated images. Therefore the algorithm is driven to try to produce new images which have the effect of propagating the hardware-induced fault up to the output of the Neural Network. During the training of Artificial Neural Networks, the weights are modified and their influence re-balanced, this introduces a natural ability to mask irrelevant input elements. This feature is certainly very powerful because it can allow you to maintain the same performance, but at the same time, it can hide the accumulation of vulnerabilities within the system. Therefore, the main task of the ITLs is that of reducing the masking effect of the model and enhance the detection of potential failures.

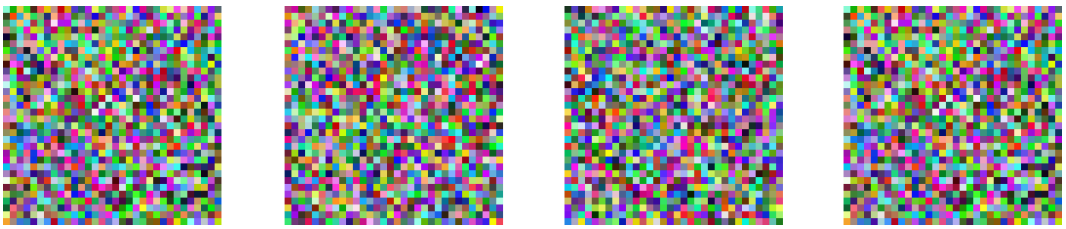

Figure 1: Test images set

In this work, the focus is on Convolutional Neural Networks (CNNs) including ResNet-18, ResNet-34, and DenseNet-161 Networks trained with CIFAR-10, one of the most used datasets in the academic and scientific fields. Regarding the generation of inputs, the evolutionary algorithm used is called µGP, and it is used to generate 32x32 color images like those of CIFAR-10 to then be fed to the neural network. Then there is an evaluator, in this case, a Python code, capable of generating a fitness file that will be delivered as feedback to µGP.

Now the thesis is organized as follows. Chapter 1 describes the backgrounds problem and the notions to know previous the start of the project, in particular, it is discussed the architectures of the deep neural networks, their functioning, and the description of the concept of evolutionary algorithm together with a more accurate explanation regarding the fault injection topic. Chapter 2 and chapter 3 explain first the proposed approach to create the environment and then the actual performed work done to achieve it, chapter 4 contains the results obtained using it as a test bed. In chapter 5 the conclusions are discussed.

Chapter 1

Background

In this chapter, the foundation is laid for the comprehensive exploration of various areas covered in this thesis. Here, it is provided a comprehensive overview of the fundamental and preliminary concepts that form the bedrock of the entire project. By delving into these concepts, the aim is to establish a solid understanding of the diverse domains and disciplines that intersect within the scope of this research. The chapter is divided into 3 sections, the first aims to explain the architecture of neural networks and to underline their characteristics and their advantages, the second focuses on the functioning of the evolutionary algorithm and in particular on the structure and functionality of µGP, finally, the last section concerns more specifically the problem of the problem of the faults within the analyzed case.

1.1 Artificial Neural Networks

At their core, artificial neural networks (ANNs), consist of interconnected nodes, called artificial neurons or units, organized into layers. These networks leverage the mathematical concept of weighted connections to propagate and transform information throughout the layers. Activation functions determine the output of each neuron, providing non-linear transformations that enable ANNs to model complex relationships in the data. The inception of ANNs draws inspiration from the intricate workings of human biology, specifically the remarkable interplay between neurons in the human brain. These networks aim to emulate the collective functionality of neurons in processing and comprehending inputs received through human senses. By mirroring

the remarkable capabilities of the human brain, ANNs strive to unlock new frontiers in understanding and interpreting complex information.

1.1.1 Artificial Neural Network base structure

The base structure of an ANN comprises interconnected layers of nodes, known as artificial neurons or units. These layers are organized in a hierarchical manner and play a fundamental role in information processing and decision-making within the network. In agreement with [13] and [19], the structure of an ANN typically consists of three key components: the input layer, hidden layer(s), and the output layer.

- **Input layer**: It serves as the entry point for data into the neural network. It receives the initial input data, each node in the input layer corresponds to a specific input feature, and the number of nodes matches the dimensions of the input data.
- **Hidden layer(s)**: It(they) sits between the input and output layers and is/are responsible for extracting and transforming the input data through a series of computations. The number of hidden layers and the number of nodes within each layer are design choices that can vary depending on the complexity of the problem at hand.
- **Output layer**: It is the final layer of the ANN and produces the network's predictions or outputs. The number of nodes in the output layer depends on the nature of the task, for example, if the network has to recognize one of the CIFAR 10 labels, the number of output nodes must be 10.

When the architecture contains multiple layers (more than 3) of artificial neurons or units we can refer to a **deep neural network (DNN)**, this structure contains many layers and a vast number of artificial neurons.

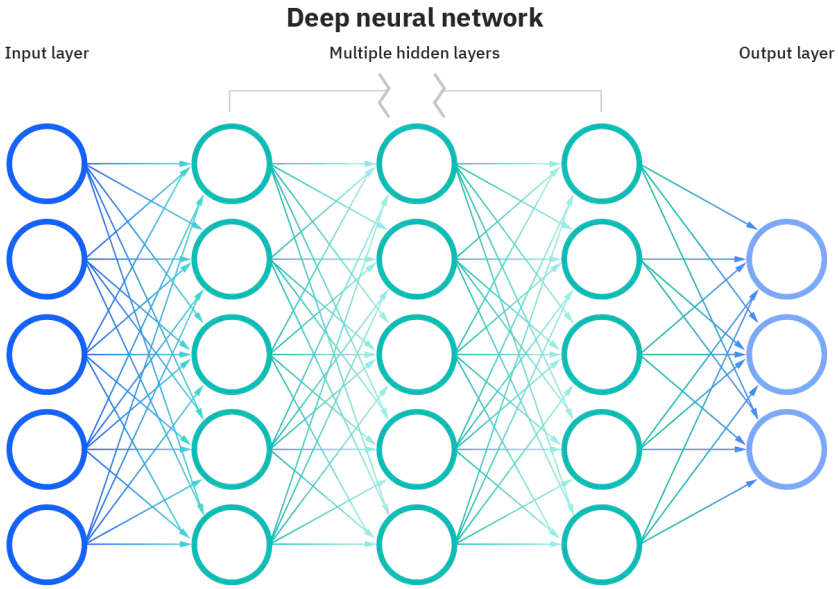

Figure 1.1: Deep neural network structure [4]

1.1.2 How does it work?

In the realm of DNNs, according to [19], each node can be envisioned as a distinct regression model with input data, weights, a bias, and an output. Upon establishing an input layer, weights are assigned to the connection between nodes. These weights serve as indicators of the relative importance of specific variables, with larger weights contributing more significantly to the overall output compared to the inputs. The inputs are then multiplied by their weights and summed together, the resulting value undergoes an activation function, which governs the final output of the node, if the output surpassed a predefined threshold, the node is said "activate" and it passes its data to the next layer. This sequential process of transmitting data from one layer to the next characterizes the DNNs as a feedforward network. This feedforward architecture enables DNNs to efficiently process and analyze complex data, making them highly valuable tools in various domains, including pattern recognition, regression, and classification tasks.

1.1.3 Deep Neural Networks typologies

DNNs encompass various typologies that are designed to handle specific types of data and tasks. This overview doesn't cover every single type, by consulting [26] or [6] it is provided a glimpse into the most prevalent neural network variants frequently encountered:

- **Feed-forward NN (FNN)**: It is widely used for facial recognition technologies, it is designed to process input data in a unidirectional manner, flowing from the input layer through hidden layers to the output layer, without any loops or feedback connections.
- **Recurrent NN (RNN)**: This artificial neural network is designed to process sequential or time-dependent data. The key feature of an RNN is its ability to maintain an internal memory (hidden state) and update it to propagate information through time. it is popular in text-to-speech applications.
- **Convolutional NN (CNN)**: One of the most used in the field of processing and analyzing data like videos and images. The idea is to leverage the concept of convolution, which is a mathematical operation that involves filters in the input data able to capture local patterns or features to create feature maps.

1.1.4 Advantages and Disadvantages

These structures offer several advantages and they can be revolutionary in many fields, but they can also have certain disadvantages such as those described in [5] and [8]. Here I list some o the pros and cons of using them:

Pros:

- **Ability to learn complex patterns**: They excel at learning complex patterns and relationship data to solve highly intricate problems.
- **Ability to process large-scale data**: DNNs can handle huge datasets with high dimensional features, furthermore they can process and be trained on modern computer architectures.

- **Transfer learning and fine-tuning**: Pre-trained models can be easily used as a foundation or starting point for solving different but related tasks, these features save significant time and computational resources.
- **Non-linearity**: The non-linear activation function enables non-linear relationships in data, this flexibility is very useful to deal with real-world problems.

Cons:

- **Need for large amounts of data**: DNNs require a significant amount of labeled training data, this information can be time-consuming, or in certain cases almost impracticable.
- **Computational Complexity and training time**: Training a DNN is computationally speaking intense, generally deeper is the architecture more time and hardware resources are necessary.
- **Over-fitting**: Proper regularization techniques are mandatory to mitigate overfitting and improve generalization because these structures are very susceptible.
- **Data dependence**: Quality and diversity of data is a very important topic to not lead to biased predictions end potentially dangerous.

1.2 Evolutionary algorithm

Evolutionary algorithms are a family of computation techniques inspired by biological evolution and natural selection. They mimic the process of natural evolution to iteratively search for optimal or near-optimal solutions to complex problems, this approach is called Genetic programming, a field discovered in the 1960s by J. H. Holland, who was the first to notice similarities between environment system adaptability and biological evolution. These algorithms are particularly effective when the problem lacks a well-defined mathematical formulation or when traditional optimization methods are computationally expensive or infeasible. There are generally defined stages, such as those described in [15] and [2], involving evolutionary algorithms:

1. **Initialization**: A population of **individuals** is generated, each of them representing a potential solution to the problem.

2. **Evaluation**: Each individuals is described by a **fitness** value. This is usually one o more numbers that represent how well the individual satisfies the objective of the problem.

3. **Individual selection**: A group of individuals is chosen in a deterministic or stochastic way based on their fitness scores, then a specific selection method can be applied.

4. **Crossover**: This is the reproduction phase, also called **recombination**, where the individual's genetic material is shared between parents to generate an offspring with different features from the previous population.

5. **Mutation:** This part introduces diversity and explore new regions of the solution space with random changes, this phase is crucial to prevent premature convergence to suboptimal solutions.

6. **Replacement**: The new offspring is integrated with some old individuals and create the next generation, here replacement strategies can be applied.

7. **Termination**: The algorithm continues to iterate until a termination criterion is met. Termination criteria may be related to the number of generations, time, fitness, and many other things.

In summary, evolutionary algorithms are powerful optimization techniques, they can search a large solution space, handle complex problems and offer robustness, however, they come with high computational complexity.

1.2.1 Micro Genetic Programming

Citing [17], µGP was initially created to generate assembly-language programs by a CAD group at Politecnico di Torino, but then its use has widened to create test programs of Bayesian networks, mathematical functions, real-values optimization, and many other fields. The algorithm is capable of generating different solutions and the process for doing it mimics modern evolutionary principles. The first step is always to generate random solutions, then it uses a heuristic algorithm combined with information introduced by the user to explore the possible solutions better and better until the optimal or sub-optimal one is found. In the system, the candidates' solutions are represented internally as directed multigraphs, these multigraphs must adhere to a set of rules defined by the user to ensure sensible structures (constraints). The individuals are converted into text files and then passed to a user evaluation program (evaluator) which ultimately has the task of generating a numerical value to evaluate them (fitness), the basic goal of the algorithm is to increase these values with the next generation.

1.2.2 µGP structure

The typology of the µGP architecture is stratified, to divide in the most concrete way possible the internal structure of the individuals and the characteristics of the constraints to the concepts linked purely to the evolutionary method. As described in [22], the basic structure is a tagged graph structure, which therefore allows each individual to be represented with his phenotypic features. Furthermore, there is a library that contains all the instructions and descriptions to which individuals are bound, this means that the structure is transformed into a Constrained tagged graph. Within this structure there is the concept of the individual represented as a class, modifiable by the user, however, the individual class can directly influence the population through selection and mating, or the death of the individual himself due to aging. A set of individuals is represented by the population class, this too modifiable and subject to genetic operators. Lastly, the evolutionary algorithm uses the population class to generate new ones and determine their structure, size, and genetic operators type to apply. In figure 1.2 is possible to see a scheme that summarizes the basic structure of the software.

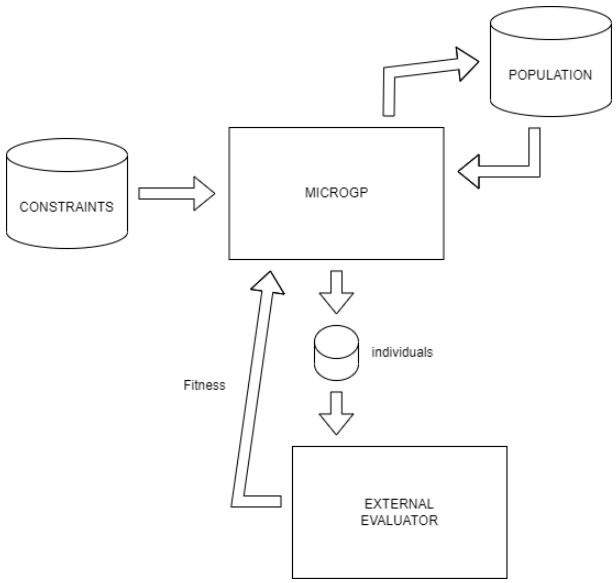

Figure 1.2: µGP structure

To sum up, µGP is composed by 3 main blocks [20]:
To sum up, µGP is composed by 3 main blocks [20]:

1. **Evolutionary core:** Place where computation and selection are performed. Here population and individuals are managed by the algorithm.

2. **External evaluator:** Tool to which individuals are given as input and which produces a post-elaboration report as output to provide the necessary feedback to the evolutionary core to close the evolution loop.

3. **Instruction library:** It provides and applies rules to individuals so that they can always be generated or modified correctly.

1.2.3 Settings for optimization process

The personal modifications that can be made to set the optimization process are various, as described previously, they can concern the algorithm in general, the single individual, or the population to which it belongs, they can be divided into 3 categories:

1. **Main settings**: These options are represented in the file ugp3.settings.xml, inside can be added several parameters tied to the general evolution. This file contains the lists of all the populations with which the algorithm will have to interact during the next run of the program. Furthermore, there are references to the related files for the recovery of the interrupted experiments and the description of the entire evolutionary process. There are also optional parameters, one of the most interesting is called 'RandomSeed', which allows you to set the internal random number generator of µGP is seeded with, useful if you want to reproduce an experiment [14].

2. **Population's constraints**: This file is called by default constraints.xml and describes the individual structure the experiment is trying to run. The great flexibility can reproduce multitudes of individuals and at the same time impose constraints that each element must rigorously respect. The structure is composed of any number of types of definitions and sections, each one containing a prologue and an epilogue with the possibility to add an arbitrary number of internal sub-sections [16]. The sub-sections are characterized again with a prologue and an epilogue but now can contain one of the most powerful tools in µGP, the **macros** [23].

    - Expressions: It's the text of the macro, everything that is written will be printed, moreover, this is where the references to the parameters that the algorithm will fill with the appropriate values are declared.
    - Parameters: This section allows to specify the type of the parameter µGP can use, the type can be numerical(integer, float), a string(bit array), and so on.

3. **Population's settings**: This is a very important settings file, where the most important population parameters are described, such as the

population typology (enhanced or multiObjective) [24], the link to the individuals constraints file or to more general parameters like:

- Population size.
- Number of genetic operators applied at each step.
- Strength of the genetic operator's regulators(sigma).
- Tuning and self-adapting µGP mechanism like the inertia or the resistance of the system to change.
- The tournament selection to compare individuals.
- Stopping conditions related to the number of generations, the fitness value, the steady state condition, or the elapsed time.
- Aging options to remove old individuals or to create an elite.

Always in this settings file it is necessary to declare which types of genetic operators are to be used, then they are randomly selected to be able to generate new individuals and introduce variability to the new populations. Furthermore, they follow a basic rule which increases the use of the operators that give rise to the best individuals, but without ever excluding the use of the less performing ones. The operators are described by 3 classes [21], **standard mutations** or single-parent operators, they can repeat the mutation on a single parameter more than once depending on the sigma value, **scan mutations**, they select macro's variable in the population's constraint and generate child for all the possible parameters, the last is the **crossovers operators**, able to cut and swap slices of subgraphs between individuals.

1.3 Reliability issues

Much research has been done on the performance of DNN accelerators and their application, however, the topic of reliability is still not well understood.

The fundamental problem is that DNNs, due to their great potential for data analytics in industrial applications, need to adhere to the safety and reliability requirements of modern industries. The consequences of soft errors in DNN can be catastrophic, especially in safety-critical scenarios. By consulting [7], a simple example can be an object miss-classification for self-driving cars. There are numerous cases where soft errors can cause the miss-classification of a truck as a bird and the braking action cannot be applied in time to prevent the collision (figure 1.3).

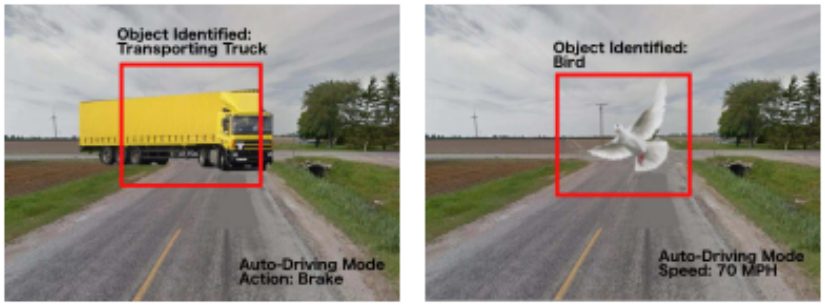

Figure 1.3: Miss-classification [7]

So this topic remains challenging still today, there is a lack of common systematic methods and tools for evaluating fault tolerance and more difficulties arise from the involvement of various architectural and functional features within different conceptual frameworks.

1.3.1 Faults concept

The first important thing to know is to understand the difference between the 3 basic concepts in fault-tolerant systems such as those described in [25]: fault, error, and failure.

- **Fault:** It refers to an abnormal physical condition in a system that leads to an error.
- **Error:** It is the manifestation of a fault within a system and represents a deviation from the expected output.

- **Failure:** It refers to the system's inability to perform its intended functionality or behavior due to errors

It's important to note that a fault in a system does not necessarily result in an error or failure if it remains inactive. Faults can be further classified based on their temporal characteristics:

- **Permanent faults:** They are continuous and stable over time, typically resulting from irreversible physical damage.
- **Transient faults:** They are temporary and often caused by external disturbance, furthermore if they start to occur within a fixed period are called intermittent faults.

Fault detection and error correction is a research field where new models are developed to understand all the different types of faults. These studies aim to major requirements like accuracy, replicate realistic faults, and tractability, to ensure fault modeling feasibility.

One of the main concerns is the occurrence of soft errors, which occur when the electronic device is hit by high-energy particles, or by its degradation and many other external factors that cause malfunctions. Hardware faults can arise from various sources such as electrical noise, manufacturing defects, thermal effects, and external environmental factors. The most common hardware faults in AI accelerators are:

- **Stuck-at faults**: A circuit node gets permanently stuck at a specific logic level due to manufacturing defects or physical damage.
- **Transient faults**: Temporary and non-permanent faults caused by external factors such as radiation or electric noise.
- **Power-related faults**: Power supply issues, such as voltage fluctuations or power spikes.
- **Overheating**: Overheating can cause performance degradation or even lead to the accelerator shutting down to protect itself.
- **Wear and Tear**: Continuous usage of AI accelerators can cause wear and tear on the hardware components, such as connectors, sockets, or solder joints. These physical faults can degrade performance or lead to complete failure over time.

The listed faults in addition to permanently not working the hardware can also only interact with the binary codes installed on the hardware and potentially lead to problems. Here are a few examples of issues that can arise:

- **Bit flips**: In this case, a bit that was supposed to be 0 may be flipped to 1 or vice versa.
- **Data corruption**: This corruption can affect the binary representations of the software's data, for example, when a bit is flipped during a memory write operation.
- **Register errors**: If a fault occurs in a register, it can result in the corruption of the stored binary values.
- **Arithmetic and logic errors**: A fault in the arithmetic logic unit(ALU) can cause incorrect mathematical operations.

1.3.2 Fault models

Several faults models have been used to accurately represent an abstraction of physical defects within electronic systems and devices, in particular, sources such as [10] and [25], explain that the 2 most proposed models over the years are:

- **Stuck-at:** This model represents a data or control line that appears to be consistently held at high (stuck-at-1) or low (stuck-at-0) state, basically, in a stuck-at-faults model, individual elements of the electronic device are tied to a logical state. The model has been extensively researched and remains popular due to its ability to model permanent faults at the logic level, capturing defects in transistor and interconnections structures.
- **Random bit-flip:** This model captures the occurrence of incorrect and random values in data or memory elements, it is designed to model transient faults that typically occur in registers or memory elements due to external disturbances. The main characteristic of this model is that only data are affected or corrupted, the circuit itself is completely undamaged. Usually, this model involves a register bit that randomly

switches, resulting in the memory element holding an incorrect logic value.

It is worth mentioning that these faults models may not fully cover the newer fault mechanisms found in deep-submicrometer technologies, however, over the years these 2 models have been shown to allow effective investigation of fault tolerance, even at the application level, so they are widely used for reliability studies in the field of deep learning. The ultimate goal of these failure models is to analyze the dependability or trustworthiness of the system, these terms are described in [25] by analyzing various properties such as:

- **Reliability:** High probability to perform correctly under specified conditions and for a specified period in the presence of faults.
- **Fault tolerance:** Property guarantees a proper operation also in the event of faults. It can be **passive**, where the system doesn't react in any special way to the internal faults and try to apply fault-masking mechanisms, or it can be **active**, in which a dynamic error recognition manages a process of adaptation or self-repair of the system.
- **Graceful degradation:** System's ability to exhibit low sensitivity to occurring faults to prevent a complete failure.
- **Robustness:** Property used to describe the ability to continue operating in the right way despite noise.
- **Error resilience:** Tolerance to wrong computations inside the system.

1.3.3 DNN faults

One notable property of deep neural networks is their ability to still perform their overall functions even if some neurons or synapses are not functioning properly. These systems don't have a minimum number of neurons required to solve a task, so any neuron or synapse can fail independently. This feature can provide a natural robustness and fault tolerance, but at the same time enable the problem of masked soft errors, hard to fix because hard to see. The errors can occur in various elements like the communication channels

resulting in faulty interconnections or disturbances between neurons, synaptic weights, the values that represent the strength of the connections, or the neuron body, the nucleus of the neuronal cell.

By consulting [7], four general parameters impact the effect of soft errors in DNNs :

- **Topology and Data type:** All the different typologies of DNNs have their unique features and different data types, these 2 things affect a lot the error propagation of the system.
- **Bit position:** Different data types interpret each bit differently, and understanding how errors in specific bits of each data type affect the soft errors detection is fundamental.
- **Layers:** Many layers in DNNs are different from each other, so it's important to understand how errors propagate in all possible types of layers.
- **Data Reuse:** Data reuse implementations in the dataflows of DNNs affect the soft error detection probability.

1.3.4 Fault injection methods

The fault injection (FI) methodologies are techniques used in software and hardware testing to simulate all the vulnerabilities, weaknesses, and faults in a system, the idea is to introduce faults to evaluate the system properties. As [12, 10] explains, the typologies can be categorized as follows:

- **Simulation-based:** Injection process is conducted without a physical device executing, in our case, a DNN. The abstraction level is the highest.
- **Platform-based:** Analyses are performed directly on a physical device that emulates the final implementation.
- **Radiation-based:** It's an accelerated radiation test that mimics external electromagnetic interference.

Analyzing in particular the simulation-based, one can still make a further distinction. First the **software-level simulation-based FIs**, where the aim

is to inject a high-level model of DNN, independent of any specific hardware, to identify only the weakness in the DNN. The second distinction is the **hardware-level simulation-based FIs**, here the abstraction level is lower and the model tries to simulate the target hardware architecture and allow a more realistic fault injection.

Finally, it is important to list the qualitative parameters [10] to be able to compare the various methods with their advantages and disadvantages:

- **Cost:** The resources that are involved to conduct the analyses.
- **Development effort:** Level of effort required to develop the FI method.
- **Exactness:** Degree of accuracy in replicate reality
- **Controllability:** The simple ability to control the process
- **Observability:** Capacity to identify events
- **Repeatability:** Number of times the process can be repeated using the same framework
- **Fault injection time:** Time to execute a single cycle of injection

Chapter 2

Proposed Approach

In this chapter, the goal is to explain the approach and methodology of the system created for the realization of tests. The task of the tests is to be able to create new input stimuli that allow the recognition of a fault injected into a CNN by simply displaying the output. The first decision was on which networks to create the environment to experiment on, the choice fell on networks based on image recognition, in particular, we are talking about:

- **ResNet18 and ResNet34:** Short for "residual network", they are 2 CNNs that are fundamental for taking a step forward in the field of image recognition as they can use residual links to add information to the data stream, these technique allows to learn much more representations deep.
- **Densenet161:** "Densely connected network", another CNN which, unlike traditional ones, has a direct connection between all levels, creating a densely connected structure.

2.1 Proposed flow

It is necessary to explain that the test environment can be divided into 2 blocks, the first is the **evolutionary algorithm** (µGP), set so that it auto-generates a population of individuals represented by 32x32 color images of the same type to be fed to the network, corresponding to the ITL first introduced in [11], and the second is the **evaluator**, a python code where the individuals are shipped and the networks, data converters, fault injection and

fitness conversion of the output evaluation are integrated. It is important to underline that the software only sends to the evaluator one individual at a time, and an individual always means one or more images collected in a batch. The steps that are explained below always refer to the single individual taken from the generated population and are repeated until fitness is obtained for each of them.

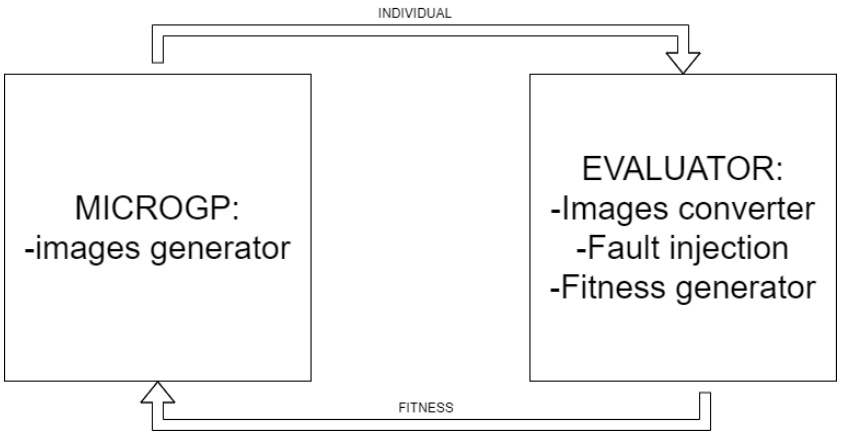

Figure 2.1: 2 blocks environment structure

2.1.1 Images conversion

The first step inside the evaluator is to convert the images produced by the evolutionary algorithm, the conversion is necessary because the format in which they are generated is not the classic one used in Cifar-10, but as explained in 1.2 is a text file that describes the images, but this topic will be covered in more detail in the next chapter. So the individual, represented by a text file, must be converted into a NumPy array that holds the contents with a structure that corresponds to an 8-bit unsigned integer, so each element in the array is represented by a single byte from the file. This step is crucial because it allows us to reshape the data to subdivide the created images and extract and rearrange the color channels of the pixels. Finally, it is necessary to transform the images into PyTorch tensor form for one last time to be used by the network.

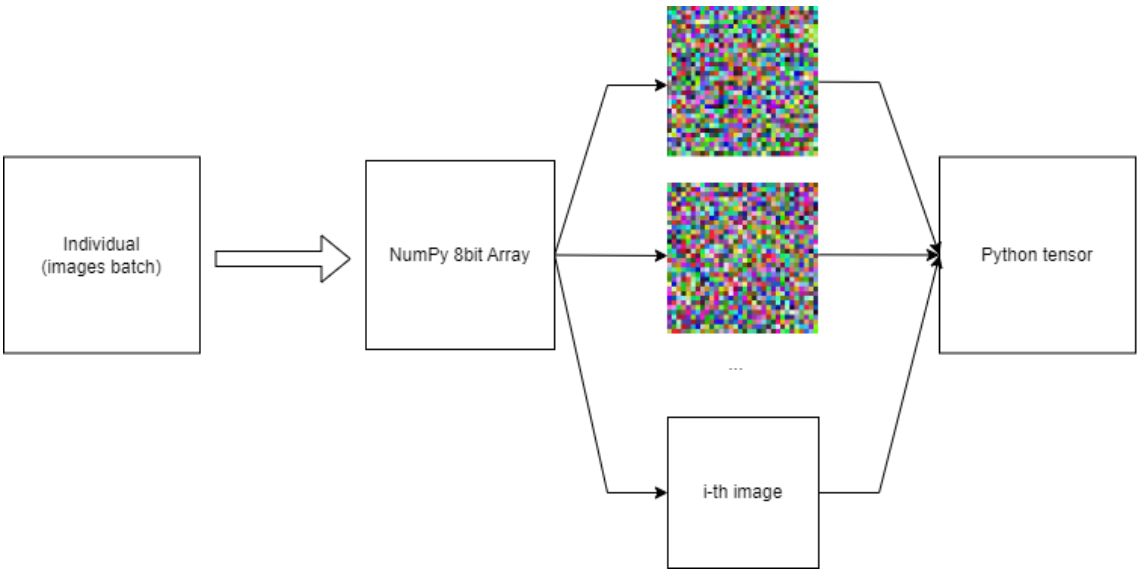

Figure 2.2: Images conversion and transformation

2.1.2 Fault injection and data collection

In this moment of the process, the individual is inserted as an input in the CNN for the first time, the network is in optimal conditions, and no faults have been injected into it yet. The collected output, which in our experiment is represented by as many vectors of 10 values as there are images in the batch, represents the system's predictions and is saved to then be used in comparison with corrupted outputs. Now the integration of the faults injector with the neural network takes place. Inside the code there is a loop capable of inserting and correcting faults in the form of bit-flips in the weights of the CNNs, these faults are previously selected and written with a certain format inside an external text file faults.txt. On the first cycle, the injector tool takes the first error in the list and inserts it into the network, reloads the same individual we used before, and stores the corrupted output values. In this moment of the process, both the original and the corrupted output are stored and it is possible to extrapolate information by comparing them with each other. Once all the necessary data has been collected, the tool corrects the network, inserts the next fault in the list, generates and stores by overwriting the old corrupted output with the new one, and collects information again by comparing it with the non-corrupted one. This loop repeats until all the faults in the list have been used. It is important to underline that the experiment is focused on understanding how the network behaves with the injection of a single fault and not a hypothetical accumulation of these. The summary of this flow can be seen in figure 2.3.

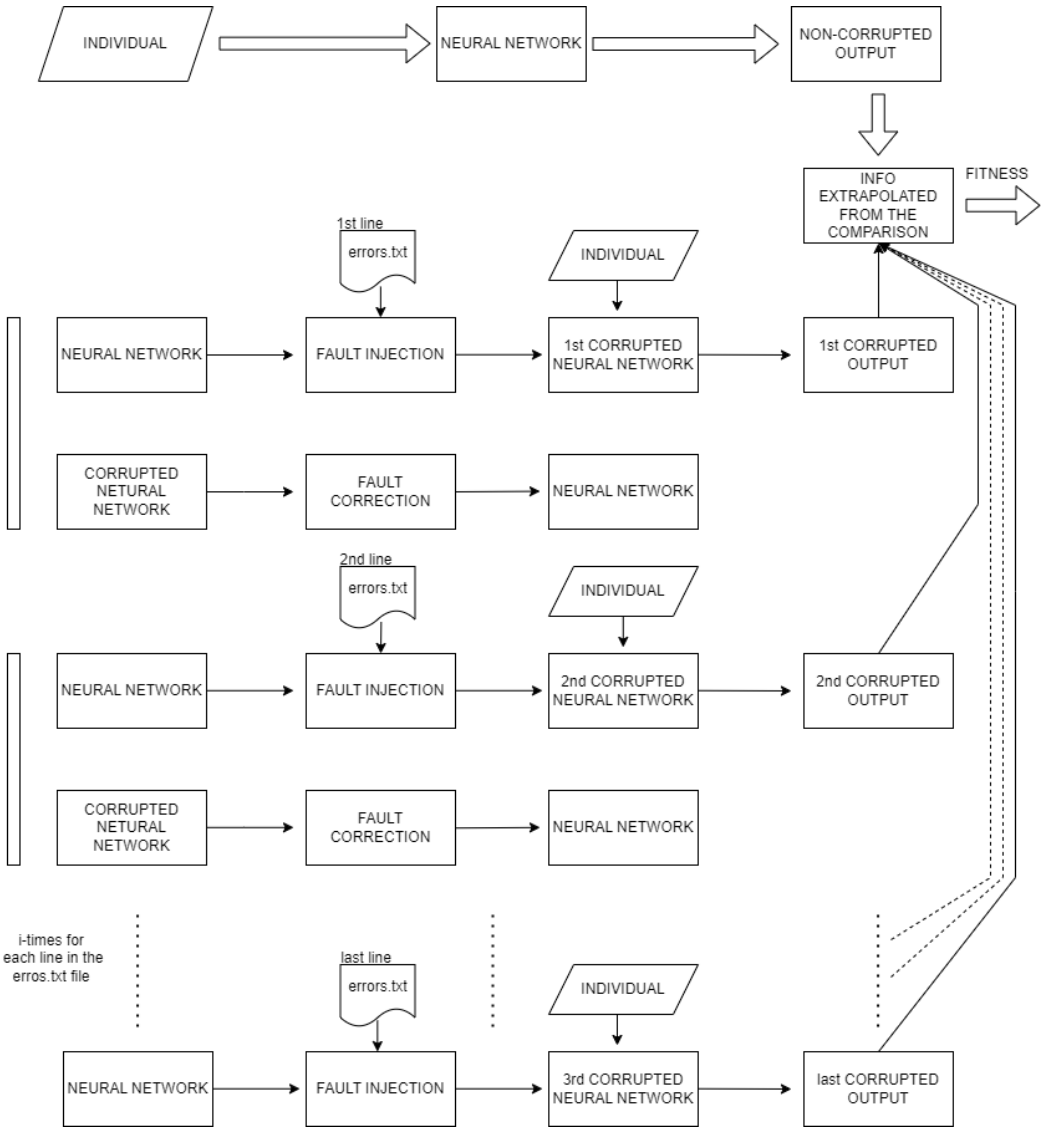

Figure 2.3: Images processing and fault injection loop

2.1.3 Fitness

At the end of the flow, outside the loop, there is the calculation of the fitness, the value that describes the differences between the outputs, it represents the quality and performance of each individual. The type of evaluation chosen is based on the assignment of higher values to individuals who perform better, the function used for the calculation is:

$$fitness = \frac{Observed\ faults}{Total\ Injected\ Faults} \tag{2.1}$$

The ratio between the observed faults and the total number of injected faults, the higher this value is, the more the individuals generated will be able to identify and unmask the injected faults. The observed faults are calculated when the 2 collected outputs are compared, the original one and the corrupted one, if they are different a counter takes note and signals the fault. Instead, if there is no difference between the two, it means that the fault is masked and the individual has failed to unmask it. Therefore fitness always has a value between the 2 limit cases, 0 and 1, the first when the evaluator has not been able to recognize any faults in the individual, and the second when it has identified them all. The information collected as explained in 2.1.2 can be of 2 types based on the experiment to be carried out:

1. **Logit observation:** A more expensive approach where the data is represented by all the information within the output.
2. **Max-Logit observation:** A cheaper approach is to use only, one per image in the batch, stored value to make a comparison between the outputs, the highest of all 10 of the vector score.

Once the fitness of the individual has been collected, the software's task is to load the next individual in the population into the evaluator and repeat all the previous steps. When all the individuals have been evaluated, the evolutionary algorithm uses all the collected fitness scores to generate the next generation. The higher the score, the more likely it is that the corresponding individual will be selected for reproduction. In conclusion, this

process described has the final objective of generating new images that are increasingly capable of showing faults, in particular, to increase sensitivity to masked ones; the criteria for interrupting the simulation are better described in the next chapter. Here ends the explanation regarding the concrete approach that has defined this thesis, now the next chapter focuses on the precise definition and description of all the components discussed so far.

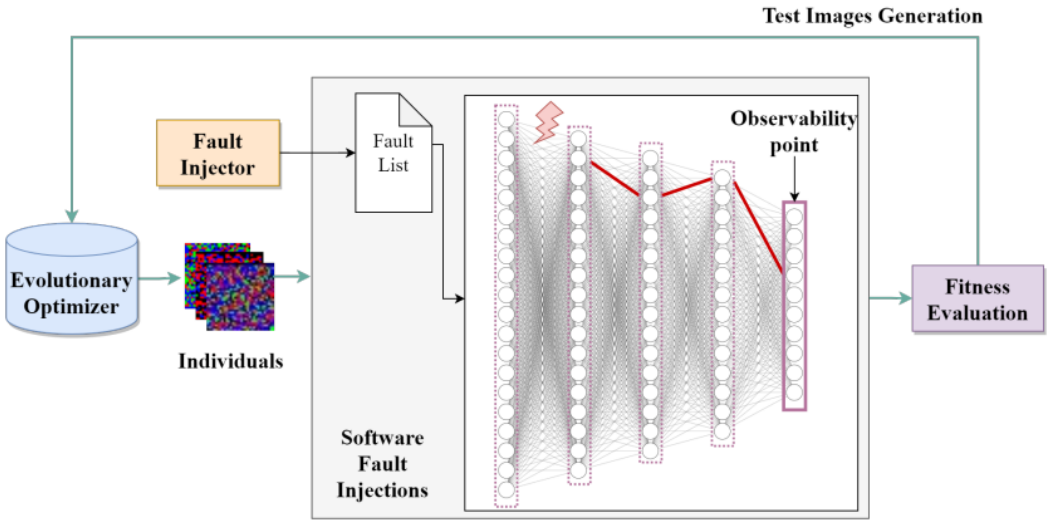

Figure 2.4: Proposed approach representation

Chapter 3

Work environment

The purpose of this chapter is to go into more detail about each component of the working environment, there are step-by-step descriptions of each element within the software with associated reasons and problems.

3.1 Evolutionary algorithm setup

The first step is to compile the µGP settings files, which are three, as explained in 1.2.3. These files are important because they affect all the parameters of the populations and generations, they must be chosen carefully to obtain the best possible result. The three files are all scripts in XML and differ in the area and level of the software they modify.

3.1.1 µGP settings

The **setting.ugp3.xml** file is the one with the fewest options directly related to the evolutionary algorithm, in this case, it contains the name of the generated population P1, the reference file population.settings.xml and also a reference to the statistics.csv file which stores data on fitness and generations. There is also a section on recovery options, which allows an experiment to be interrupted and restarted via the status.xml file, which contains essential information about the populations.

```
<?xml version="1.0" encoding="utf-8" ?>
<settings>
<context name="evolution">
<option name="randomSeed" value="0" />
<option name ="populations">
<population name="P1" value="population.settings.xml" />
</option>
<option name="statisticsPathName" value="statistics.csv" />
</context>
<context name="recovery">
<option name="recoveryOutput" value="status.xml" />
<option name="recoveryOverwriteOutput" value="true" />
<option name="recoveryDiscardFitness" value="true" />
</context>
<context name="logging">
<option name="std::cout" value="info; brief" />
<option name="debug.log" value="debug; brief" />
</context>
</settings>
```

Listing 3.1: ugp3.settings.xml code

3.1.2 Constraints settings

The 2.1.1 section indicates that the output of the evolutionary algorithm will be text files representing batches of images, the rules for generating which are defined in the **constraints.xml** script. The purpose of this element is to introduce a suitable structure for the generation of 32x32 RGB images such as those of Cifar-10. The image batches of the dataset have a well-defined binary structure and it is necessary to reproduce it correctly. It is made up of 3073 bytes for images, of which the first byte represents a number between 0 and 9, i.e. the label to which the image is assigned, then the remaining 3072 bytes are the color channels of the pixels, 1024 bytes each for red, green and blue. For example, in the Cifar-10 database, there are five training batches, each one with 10000 images, but in the case of the thesis, there are much fewer to make the process manageable and functional. The idea is to create a text file with the same format, where the colors of the pixels represented by a byte are described by a number between 0 and 255, all non-negative integers being describable with 8 bits.

```
<1 x label><3072 x pixel>
...
<1 x label><3072 x pixel>
```

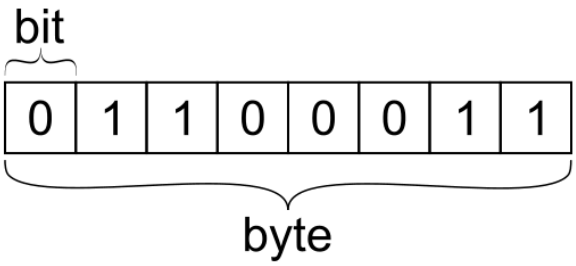

Figure 3.1: 8-bit representation [3]

The structure of constraints has already been briefly explained in 1.2.3. The code 3.2 defines a type called "byte_type", when this term is invoked within a macro, a number with the characteristics listed above is inserted. The script is composed of a main section called "byte_string", which represents the generated image, with the parameters "maxOccurs" and "minOccurs" it is possible to decide how many times to repeat the sequence of 3073 numbers that define the image. Further down there is a sub-section called "main" which contains a prologue and an epilogue. In the former, the image label is printed, in the latter, all the macros that represent the colors are declared. In line 36 we see how the expression <expression>, which is the command capable of printing on the file, contains all the 3072 macros linked to the "byte_type" definition in the <parameters> section immediately following it. Thanks to this code, µGP will never be able to generate images with characteristics other than those desired; it also allows the modification and mutation of each of the macros in the reproduction process and the use of genetic operators.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <?xml-stylesheet type="text/xsl"
3   href="http://www.cad.polito.it/ugp3/transforms/constraintsScripted.xslt"?>
4  <constraints
5   xmlns="http://www.cad.polito.it/ugp3/schemas/constraints"
6   id="One-Max" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7   xsi:schemaLocation="http://www.cad.polito.it/ugp3/schemas/constraints
8
9   http://www.cad.polito.it/ugp3/schemas/constraints.xsd">
10
11  <typeDefinitions>
12  <item name="byte_type" type="integer" minimum="0" maximum="255" />
13  </typeDefinitions>
14
15  <commentFormat><value/></commentFormat>
16  <identifierFormat>n<value /></identifierFormat>
17  <labelFormat><value/>: </labelFormat>
18  <uniqueTagFormat><value /></uniqueTagFormat>
19  <prologue id="globalPrologue"/>
```

```
20 <epilogue id="globalEpilogue"/>
21 <sections>
22 <section id="byteString" prologueEpilogueCompulsory="true" maxOccurs="1" minOccurs="1" maxReferences="1">
23 <prologue id="sectionPrologue"/>
24 <epilogue id="sectionEpilogue"/>
25 <subSections>
26 <subSection id="main" maxOccurs="1" minOccurs="1" maxReferences="1">
27 <prologue id="stringPrologue">
28 <expression><param ref="byte"/> </expression>
29 <parameters>
30 <item name= "byte" type="integer"  minimum="0" maximum="9" />
31 </parameters>
32 </prologue>
33 <epilogue id="stringEpilogue"/>
34 <macros maxOccurs="1" minOccurs="1" averageOccurs="1" sigma="0">
35 <macro id="byteString">
36 <expression><param ref="byte1"/> <param ref="byte2"/> ... <param ref="byte3072"/>
   </expression>
37 <parameters>
38 <item type="definedType" ref="byte_type" name="byte1" />
39 <item type="definedType" ref="byte_type" name="byte2" />
40 ...
41 ...
42 <item type="definedType" ref="byte_type" name="byte3072" />
43 </parameters>
44 </macro>
45 </macros>
46 </subSection>
47 </subSections>
48 </section>
49 </sections>
50 </constraints>
51 </constraints>
```

Listing 3.2: constraints.xml code

3.1.3 Population settings

The next and final file is the **population.settings.xml**, which contains all the population parameters. The most important ones concern the size, the way, and the number of modifications applied to each generation and the conditions of interruption of the experiment, they are very easy to visualize in the code **population.settings.xml** code. The values concerning the size of the population were decided after numerous experiments, the criteria used for the choice were mainly related to the general computational cost of the process, instead the most important decisions were related to the genetic operators. 3 genetic operators are used:

- AlterationMutation
- OnePoinCrossover
- TwoPoinCrossover

The first can change the value of the macros to introduce differentiation between populations, the second and third are quite similar, they cut slices of values between individuals in the same position and of the same size to mix them and produce children with cross-genetic inheritance. Numerous macros are applied to the generated files, printing thousands of numbers, so the choice of these 3 operators is important, they can work with individuals with high genetic content and mix them efficiently.

```
<?xml version="1.0" encoding="utf-8" ?>
<parameters type="enhanced">

<!-- BASIC POPULATION PARAMETERS -->
<!-- the initial size of the population -->
<nu value="10" />
<!-- the maximum size of the population -->
<mu value="10" />
<!-- the numbers of genetic operators applied at every step of the evolution -->
<lambda value="15" />
<!-- the inertia for the self-adaptating parameters [0,1] -->
<inertia value="0.9" />
<!-- the number of dimensions of the fitness -->
<fitnessParameters value="1" />
<!-- the strength of the mutation operators (0,1) -->
<sigma value="0.9" />
<cloneScalingFactor value="0" />
<maximumTime hours="74" />
<maximumAge value="10" />
<maximumEvaluations value="1000000" />
<maximumFitness value="1" />
<maximumSteadyStateGenerations value="100" />
<invalidateFitnessAfterGeneration value="0" />
<!-- parents selector parameters -->
<selection type="tournamentWithFitnessHole" tau="2" tauMin="1" tauMax="4" fitnessHole="0" />
<!-- the definition of the constraints of the problem -->
<constraints value="constraints.xml" />
<!-- evaluator parameters -->
<evaluation>
<concurrentEvaluations value="1" />
<removeTempFiles value="true" />
<evaluatorPathName value="python3 batch_evaluator.py" />
<evaluatorInputPathName value="individual_%s.in" />
<evaluatorOutputPathName value="fitness.out" />
</evaluation>

<!-- operator statistics -->
<operators default="none">
<operator ref="alterationMutation" />
<operator ref="twoPointCrossover" />
<operator ref="onePointCrossover" />
</operators>
</parameters>
```

Listing 3.3: population.settings.xml code

Once the software has been started, thanks to these 3 scripts, the algorithm can create the first population with its individuals; once this step has been completed, it will come into contact with the evaluator. µGP, once the population is finished, does not immediately create all the files containing

the values of all the individuals to be given to the evaluator, but it prints the first one together with another file called "individuals_to_evaluate" in which the name of the file is present. The second individual and its name are only overwritten when the fitness of the first individual is delivered. This cycle continues until all individuals have gone through the evaluator and all fitness reports have been collected. The fitnesses are analyzed, the genetic operators are applied and the new population is born to which the same process is applied, this is the role of the evolutionary algorithm. The conditions for stopping the experiments vary according to the objective to be achieved, it can be linked to performance or time, but these criteria are discussed in the chapter 4 where the experiments performed are described and the data collected.

3.2 Evaluator

3.2.1 1st block: image conversion

The evaluator collects the individuals, i.e. the sequence of numbers generated by µGP, the general task is the one explained in 2, converts the individual images, and evaluates the outputs of the CNNs. In the beginning, the numerical values must be transformed into images of the format [x, 3, 32, 32], where x is the number of images in the individual.

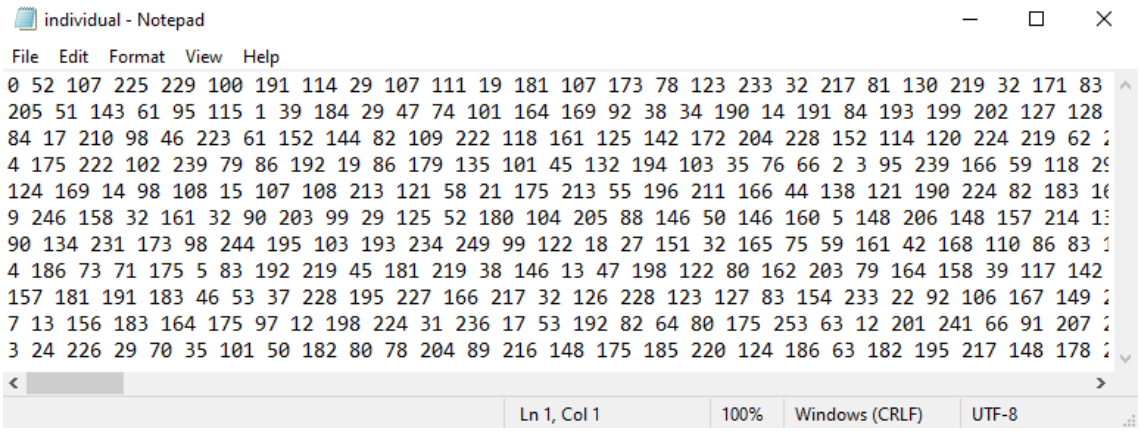

Figure 3.2: Individual.txt file

An example of an individual's text file is shown in figure 3.2, the color values are extracted from the text and converted to a NumPy vector.

```
individual_list = open('individualsToEvaluate.txt', 'r')
individual_name = individual_list.readline().split()

numbers = []
with open(individual_name[0], "r") as numbers_file:
    for line in numbers_file:
        data.extend(int(x) for x in line.strip().split(
```

The values are converted one by one into the vector called "data", with this element it is now possible to transform the data and extract the colors of the different images. First, the data is decomposed and reassembled to obtain a matrix [n. of images, 3073], then the 3 channels can be isolated to obtain the actual images.

```
with open(batch_path, 'rb') as f:
    data = np.fromfile(f, dtype=np.uint8)

# Reshape the data into (num_images, 3073) and split it into labels and pixels
num_images = len(data) // 3073
data = data.reshape(num_images, 3073)
labels = data[:, 0]
pixels = data[:, 1:]

# Reshape the pixel data into (num_images, 3, 32, 32) format
red_channel = pixels[:, :1024].reshape(num_images, 32, 32)
print(red_channel)
green_channel = pixels[:, 1024:2048].reshape(num_images, 32, 32)
blue_channel = pixels[:, 2048:].reshape(num_images, 32, 32)
images = np.stack([red_channel, green_channel, blue_channel], axis=1)
images = np.transpose(images, (0, 2, 3, 1))
```

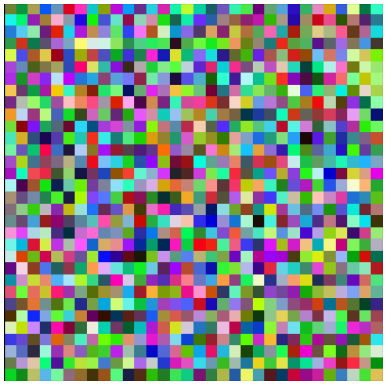

Figure 3.3: 32x32 RGB generated image

The result is a batch of images like the one in figure 3.3, which are not yet ready to use because they need to be transformed into PyTorch tensors to be uploaded to the network. PyTorch is an open-source machine learning library that is widely used for deep learning tasks. It provides a high-level interface, known as torch.nn, which offers a wide range of pre-built functions and classes for building neural networks, and is a popular choice among researchers and practitioners in the field of artificial intelligence.

```
transforms_list = [transforms.ToTensor()]
transform = transforms.Compose(transforms_list)
pil_images = [Image.fromarray(img) for img in images]
tensor_images = torch.stack([transform(img) for img in pil_images])
```

3.2.2 2nd block: fault injection loop and data storage

To proceed with the experiment, it is necessary to decide which type of fault to inject and where. The idea is to simulate bit-flips on the weights of the CNN caused by malfunctions of the hardware on which it is installed. To be precise, not all bit-flips are interesting, only those that are difficult to visualize, those that don't change the weights very much, and those that are more susceptible to masking. The image 3.4 shows the 32-bit structure of each weight inside the mesh, the format is the single precision floating point format. The formula to calculate which value it represents is

$$value = (-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

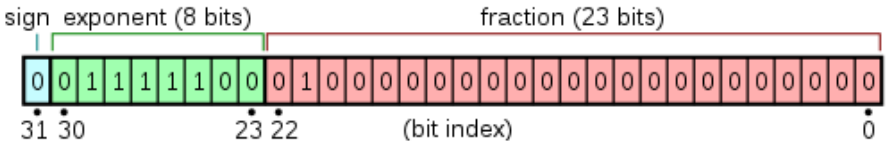

Figure 3.4: 32-bit floating-point structure [27]

The first bit is the sign, the next 8 bits are the exponent, and the last 23 are the mantissa. The most interesting bit flips for the thesis tasks are those on the mantissa, because these values, if modified, slightly distort the weight value, which leads to difficult fault propagation, in fact, the experiments will be done on a pool of faults distributed over it. Furthermore, the faults are all injected into the first layer of all the CNNs tested. Understanding error propagation is one of the most important things, the goal and the main focus of the thesis can be summarised exactly in trying to move the observability from the first layer to the last one.

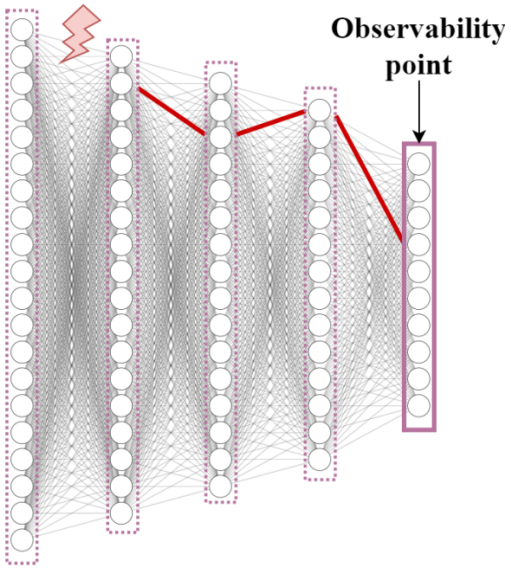

Figure 3.5: First layer fault injection

To proceed with the evaluation process, before fault injection is activated, the model of the CNN to be used must be activated and a "state_dict" of it created, a Python dictionary object that maps each parameter. "state_dict" is used throughout the process as a checkpoint for saving and loading model data, an uncorrupted copy of the CNN is immediately made, and network outputs are collected.

```
# Load the ResNet model
model = resnet18(pretrained=True)

# Set the model to evaluation mode
model.eval()

# Get the state dictionary of the model
state_dict = model.state_dict()

# Pass the images through the model
Vector_score = model(tensor_images)
```

Now the fault list "faults.txt" has to be opened and each error has to be injected one by one. For practical reasons, an external customized error generator has been realized for the experiment:

```
import random

rows = set()
while len(rows) < 1000:
    a = random.randint(10,32)
    b = random.randint(0, 63)
    c = random.randint(0, 2)
    d = random.randint(0, 2)
    e = random.randint(0, 2)
    row = f"{a} [{b},{c},{d},{e}] conv1\n"
    rows.add(row)

with open("faults.txt", "w") as f:
    f.writelines(rows)
```

The format in which the code generates the faults.txt output file is made up of 3 elements for each line, each of which describes the characteristics and the location where the fault is injected. The first element is the bit of the mantissa that is modified, the distribution as previously explained is practiced randomly with the last 23 bits. The second element is the coordinate of the weight inside the layer, stored in "state_dict", the third is the name of the first layer of the CNN being worked on. The formats and names of the first layers of the 3 neural networks used can be found in the table 3.1.

Table 3.1: Neural Networks First Layer Format

Model	first layer name	f.l. format
ResNet18	conv1.weights	[64, 3, 3, 3]
ResNet34	conv1.weights	[64, 3, 3, 3]
Densenet161	features.conv0.weights	[64, 3, 3, 3]

Each line of the printed file is a fault that is fed into the CNN, each randomly generated and different from the others. An example of how a fault list might be presented in a ResNet-18 might be:

```
14 [56,0,2,1] conv1
15 [44,1,2,1] conv1
27 [42,1,1,0] conv1
14 [51,1,2,2] conv1
...
```

Each of these faults is entered one by one by reading the file line by line, and it is also necessary to introduce a counter to note the number of times a fault is injected into the loop, which is essential for calculating fitness.

```
# open the file
with open('errors.txt', 'r') as f:
    # read each line in the file
    for line in f:

    num_faults += 1
    ...
```

Inside the loop, the 3 elements are divided and saved for use.

```
# extract the first, second, and third terms
bitflip_pos = int(parts[0])
coordinated = parts[1]
Layer = parts[2]
```

Now it is necessary to take advantage of 'state_dict' again, to enter it and extrapolate the weight given in the coordinate. The weight is not stored in binary form, but in decimal form, so a conversion is required to inject the fault. When the binary value is changed from 0 to 1 or vice versa, it is converted back to decimal and printed on the previous state_dict coordinate, replacing the old weight with the faulty one. Once the modified state_dict has been loaded into the model, the faulty output can be collected and the bit-flip can be corrected so as not to accumulate faults for the next cycle.

```
# use these terms to access the corresponding value in the state_dict
value = state_dict[f'{Layer}.weight'][tuple(map(int, coordinated.strip('[]')
.split(',')))].item()

# convert the float to binary representation
binary = bin(struct.unpack('!I', struct.pack('!f', value))[0])[2:].zfill(32)

# simulate a bit flip
bit_to_flip = bitflip_pos - 1
flipped_binary = binary[:bit_to_flip] + str(int(not int(binary[bit_to_flip])))
+ binary[bit_to_flip + 1:]

# convert the flipped binary back to a float
flipped_f = struct.unpack('!f', struct.pack('!I', int(flipped_binary, 2)))[0]

# modify the state_dict of the model and load it
state_dict[f'{Layer}.weight'][tuple(map(int, coordinated.strip('[]')
.split(',')))] = flipped_f

model.load_state_dict(state_dict)

# Pass the images through the model
```

```
Corrupted_vector_score = model(tensor_images)

# restore the injection
state_dict[f'{Layer}.weight'][tuple(map(int, coordinated.strip('[]')
.split(',')))] = value
model.load_state_dict(state_dict)
```

Now both the uncorrupted and corrupted output are stored in the variables 'Vector_score' and 'Corrupted_vector_score' so that the necessary data can be collected. The comparison methods as described in 2.1.3 are two, the first, the "logit observation" check if the 10 output values of all images are equal with a decimal digits of precision of 7.22, in case even one value is different a counter signals that the injected fault has been detected.

```
# compare the 2 vectors' score
for i in range(Vector_score.shape[0]):

        if not torch.equal(Vector_score, Corrupted_vector_score):

     critical_faults += 1
```

The second method of comparison does not compare all 10 values of each image in the individual, but selects only one per image, the highest one, to be able to test and analyze results with another method with a lower computational cost.

```
# compare the 2 vectors score
for i in range(Vector_score.shape[0]):

        max_values = torch.max(Vector_score, dim=0)

        max_values2 = torch.max(Corrupted_vector_score, dim=0)

    if max_values == max_values2:
        critical_faults += 1
```

At this point, the for loop moves on to the next line and repeats all the steps described concludes the faults to be injected, closes the errors.txt file, and calculates the fitness from all the information collected. One has to

pay attention to the fact that the critical faults are injected into each of the individual's images and the comparison is made on all images together, so the total faults are only those within the list, and should not be multiplied by the number of images within each individual.

```
injected_faults = num_faults

fitness = critical_faults / injected_faults
```

Fitness is calculated, so now it only remains to create the output file to send to the evolutionary algorithm.

```
# Generate fitness.out file
output_ugp3 = open("fitness.out", "w+")

output_ugp3.write(str(fitness))

output_ugp3.close()
```

With these last lines, the code of the evaluator ends, and fitness data were collected by µGP. Another individual from the population is loaded, to which the same transformation processes are applied again, and the same faults as the previous individual are injected, with the same bit-flip and the same coordinate. Once all the individuals have been evaluated and all the fitness values have been collected by the algorithm, the new population is generated to start a new cycle of evaluations. In the next chapter, it will be shown how this simulation environment was used and what data was collected during the experiments.

Chapter 4

Results

This chapter has the task of explaining the last chosen setups about the application of the µGP evolutionary algorithm and the faults injector, and the results obtained from the generation of ITLs to increase the observability of the faults at the output of the networks. the type of experiments carried out are two, those already discussed in 2.1.3, each on a different network among the three chosen, ResNet-18, ResNet-34, and DenseNet161. The mentioned networks have been trained and tested with CIFAR-10 [1]. The author [9] by changing the number of classes, the size of the filters, the stride, and the padding figure, modified the versions of the networks implemented in the official version of PyTorch. The changes applied made it possible to adapt the networks to the images of the CIFAR-10 data set, with which they were then trained and tested. In table 4.1 is possible to check the accuracy rate and the number of parameters of the three networks.

Table 4.1: Neural Network details

Model	Accuracy	No. Params
ResNet18	93.07 2	11.174 M
ResNet34	93.34	21.282 M
Densenet161	91.07 8	26.483 M

Furthermore, it is important to emphasize a feature of these experiments that concerns observability at the network output. The values to be observed are usually normalized through the SoftMax function, which transforms the output vector into a probability vector, but in the case of the framework used, PyTorch, this does not happen unless applied by the user. All the analyses and data collected do not make use of the SoftMax function, the motive being due to its non-linear characteristics, which introduce a loss of information when a fault is injected and consequently lower its observability.

To enable a direct and functional comparison of the application of the evolutionary algorithm and the different networks, all experiments were performed by setting certain parameters, such as genetic and population size. these parameters were chosen after numerous experiments aimed at balancing the parameters, the goal was to obtain the right balance between results and computational cost. Also, the amount of execution time is equal between all networks in the 2 experiments. The Logit experiment lasts 3 days, while the Max-Logit experiment lasts 24 hours.

4.1 Experimental Results

In the 4.2 and 4.3 tables it is possible to check all the parameters fixed during the experiments. For each generation, a population of 10 individuals each represented by an image is evaluated and 15 genetic operators among the three listed in 3.1.3 are applied. Furthermore, it is important to emphasize that all network analyses were made using the first 20 generations as a reference, the reason being to make not only a temporal but above all a generational comparison.

Table 4.2: Population settings table

Population settings:
Initial size of the population = 10
Maximum size of the population = 10
Genetic operators applied at every step of the evolution = 15
Inertia for the self-adapting parameters = 0.9

Table 4.3: Detail about µGP and evaluator settings and generations

Networks	**Observed element**	**Time [h]**	**Injected faults**	**Generations**
ResNet-18	Logit	72	657	41
ResNet-32	Logit	72	657	22
DenseNet-161	Logit	72	662	20
ResNet-18	Max-Logit	24	657	30
ResNet-32	Max-Logit	24	657	23
DenseNet-161	Max-Logit	24	662	19

The results of the final experiments are shown in figures 4.1 and 4.2. For both experiments, the x-axis describes the number of generations, while the y-axis describes the fitness percentage rate. In the figure, the fitness is expressed as described in 2.1.3 and the results are representative of the trend of the best score of the images in all the generated populations.

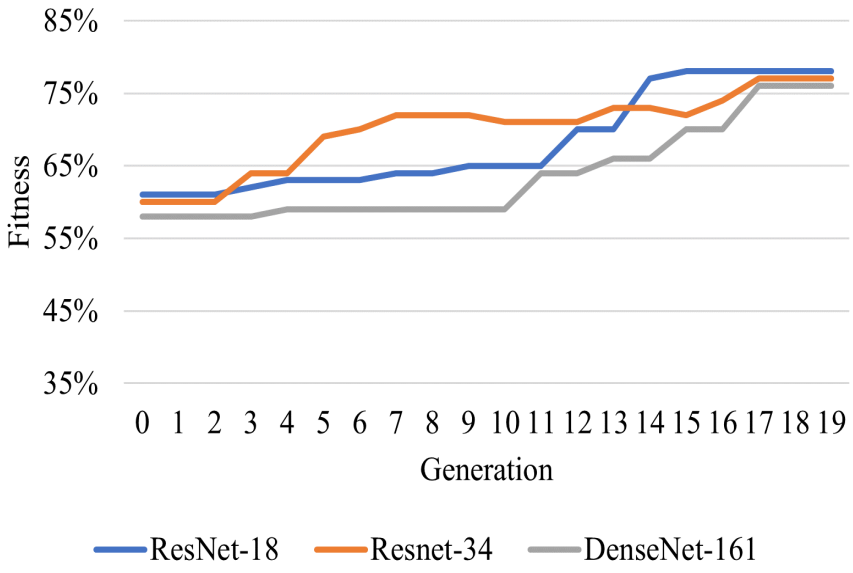

Figure 4.1: Logit experiment results

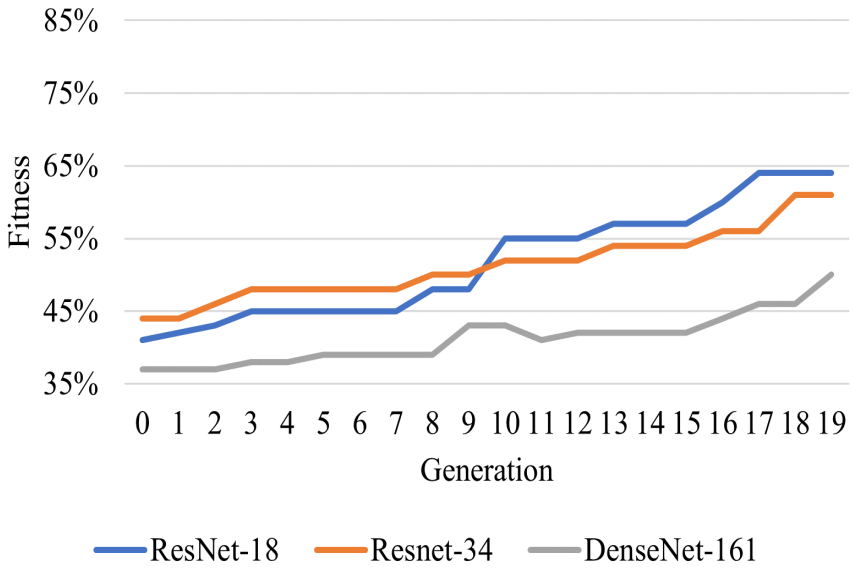

Figure 4.2: Max-Logit experiment results

Figure 4.1 shows how the various image populations increased the ratio of observable faults at the network output in the first experiment. The results show that in all 3 networks, there is an increase. The first completely randomly generated images of generation 0 are able to detect between 57% and 62% injected errors in all 3 networks. During the 20 generations, there is a modest but significant increase in fitness, which increases by at least 10 percentage points in all networks. It is interesting to note that at certain times, for several generations, fitness has great difficulty in increasing, such as between generation 0 and 10 in DenseNet-161, or as the ResNet-34 between generation 7 and 15. It is intuitive to think that during those generations µGP continues to insert genetic operators, without being able to find the right strategy to increase fitness. This fact, in addition to the generally increasing trend of observable faults, suggests that a larger population, the inclusion of more genetic operators, and a longer duration of the experiment can lead to ever greater improvements.

Figure 4.2 shows the data collected from the second experiment, the one on the Max-logit. The first noticeable thing is the sudden increase in fitness compared to the first experiment, especially in the case of the ResNet-18 network, where there is even a 25% improvement over the 20 generations. Only the DenseNet-161 struggles to scale up, reaching the maximum of 50% total fault coverage. Here too it is clear that using other parameters at the cost of a computational request and more time can lead to further improvement.

If the 2 experiments are compared, it is immediate to think that the second one does not meet the minimum fault observability requirements, the best result is a 65% of the ResNet-18, much lower than the corresponding one in the first experiment. However it is necessary to underline that 65% of identified faults are higher than that of all randomly generated images (generation 0) by observing the entire Logit, furthermore, the execution time linked to the increase in fitness is representative of 24 hours, a third compared with the best results. In the 4.4 table it is possible to check the data relating to the first and last fitness achieved by both experiments with the actual improvement.

Table 4.4: Experiments fitness details

Networks	**Observed element**	**First fitness**	**Last fitness**	**improvement**
ResNet-18	Logit	61%	78%	17%
ResNet-32	Logit	60%	77%	17%
DenseNet-161	Logit	58%	76%	18%
ResNet-18	Max-Logit	41%	64%	20%
ResNet-32	Max-Logit	44%	61%	17%
DenseNet-161	Max-Logit	37%	50%	13%

Chapter 5

Conclusions

This thesis describes a different methodology to generate and produce Input Test Images (ITL) to stimulate the observability of faults injected into the output of the neural networks such as those mentioned in the article [11]. The approach is based on the automatic generation of images by means of an evolutionary algorithm capable of finding optimal or suboptimal solutions. The search for new safety strategies using neural networks is constantly evolving. With the arrival of numerous companies and realities that are integrating these new technologies, the old methodologies and approaches are no longer sufficient to guarantee the standards required today. The evolutionary algorithm, used to generate 8-bit RGB images, is µGP, a very versatile tool capable of achieving numerous objectives thanks to its structure. The neural networks used for testing are of the convolutional type, networks that are very adept at image recognition due to their ability to encode image-specific features in their architecture. The injected faults simulate permanent hardware faults in order to create a virtual working environment capable of achieving fast, reliable, and inexpensive results. The faults induced are of the bit-flip type and change the values of the network weights. The aim is to amplify the effect of these faults by attempting to increase the observability at the output of the networks, thereby reducing the natural masking characteristic. All the test structure has been implemented, experiments have been done, and data have been collected to understand whether the proposal is functional for the problem. Experimental results show that in all cases there was always an improvement in observability on the output of the networks. In particular, the improvements in max-logit, which show increases of several percentage points over the use of randomly generated test images, support

this thesis. In conclusion, the data collected shows that this methodology is functional for the main objective on which this thesis is based. In addition, numerous improvements can be made, both in the evolutionary algorithm and in a more careful selection and analysis of the injected faults. A great advantage of the project is that it can be reproduced on any CNN with the right setup, and in addition, an attempt can be made to extend the proposed approach to other types of networks. In the future, more powerful analyses can be carried out, along with other possible improvements. In addition, more specific studies can be carried out on the characteristics and effects of bit flips, with particular attention to their location and distribution.

Bibliography

[1] Vinod Nair Alex Krizhevsky and Geoffrey Hinton. *The CIFAR-10 dataset*. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[2] Thomas Bäck and Hans-Paul Schwefel. "An overview of evolutionary algorithms for parameter optimization". In: *Evolutionary computation* 1.1 (1993), pp. 1–23.

[3] Giuseppe Carichino. *BIT E BYTE*. URL: https://www.youmath.it/domande-a-risposte/view/6866-bit-byte.html.

[4] IBM. *Cosa sono le reti neurali?* URL: https://www.ibm.com/it-it/topics/neural-networks.

[5] Michael I Jordan and Tom M Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245 (2015), pp. 255–260.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

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

[8] Gary Marcus. "Deep learning: A critical appraisal". In: *arXiv preprint arXiv:1801.00631* (2018).

[9] Huy Phan. *PyTorch models trained on CIFAR-10 dataset*. URL: https://github.com/huyvnphan/PyTorch_CIFAR10.

[10] Annachiara Ruospo et al. "A survey on deep learning resilience assessment methodologies". In: *Computer* 56.2 (2023), pp. 57–66.

[11] Annachiara Ruospo et al. "Image Test Libraries for the on-line self-test of functional units in GPUs running CNNs". In: *28th IEEE European Test Symposium 2023*. IEEE. 2023.

[12] Annachiara Ruospo et al. "Pros and Cons of Fault Injection Approaches for the Reliability Assessment of Deep Neural Networks". In: *2021 IEEE 22nd Latin American Test Symposium (LATS)*. 2021, pp. 1–5. DOI: 10.1109/LATS53581.2021.9651807.

[13] Amazon Web Services. *What Is A Neural Network?* URL: https://aws.amazon.com/what-is/neural-network/.

[14] Giovanni Squillero. *Main Settings*. URL: https://sourceforge.net/p/ugp3/wiki/Main%20Settings/.

[15] Giovanni Squillero. *Natural and Artificial Evolution*. URL: https://sourceforge.net/p/ugp3/wiki/EAs/.

[16] Giovanni Squillero. *Population's Constraints*. URL: https://sourceforge.net/p/ugp3/wiki/Population%5C%20Constraints/.

[17] Giovanni Squillero. *µGP (MicroGP)*. URL: https://sourceforge.net/p/ugp3/wiki/MicroGP/.

[18] International Organization for Standardization (ISO). *26262:Road vehicles - Functional safety*. URL: https://www.iso.org/standard/68383.html.

[19] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[20] Alberto Tonda. *A brief summary of µGP's structure*. URL: https://sourceforge.net/p/ugp3/wiki/Structure%5C%20Summary/.

[21] Alberto Tonda. *Genetic operators*. URL: https://sourceforge.net/p/ugp3/wiki/Genetic%5C%20operators/.

[22] Alberto Tonda. *The Evolutionary Core*. URL: https://sourceforge.net/p/ugp3/wiki/Core/.

[23] Alberto Tonda and Giovanni Squillero. *Macros*. URL: https://sourceforge.net/p/ugp3/wiki/Macro/.

[24] Alberto Tonda and Giovanni Squillero. *Population's Settings*. URL: https://sourceforge.net/p/ugp3/wiki/Population%5C%20Settings/.

[25] Cesar Torres-Huitzil and Bernard Girau. "Fault and error tolerance in neural networks: A review". In: *IEEE Access* 5 (2017), pp. 17322–17341.

[26] SAS Viya. *Artificial Neural Networks What they are why they matter*. URL: https://www.sas.com/en_us/insights/analytics/neural-networks.html#technical.

[27] Wikipedia. *Single-precision floating-point format*. URL: https://en.wikipedia.org/wiki/Single-precision_floating-point_format.