António Manuel Delgado Morais

Robust Neural Networks

1 2 9 0 UNIVERSIDADE Ð COIMBRA

1 2 9 0

UNIVERSIDADE Ð
COIMBRA

António Manuel Delgado Morais

Robust Neural Networks

Dissertation in the context of the Master in Informatics Engineering,
Specialization in Intelligent Systems, supervised by Professors Raul Barbosa
and Nuno Lourenço and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2021

This page is intentionally left blank.

Acknowledgements

I would like to thank my supervisors, Professors Raul Barbosa and Nuno Lourenço, for the continuous assistance, for the availability to answer questions, and for the suggestions which were crucial for the success of this work.

I am also deeply grateful for the support given to me by my family and friends, which motivated me to continue working when I stumbled upon a problem, and encouraged me when I managed to overcome it.

This work is partially funded by Portuguese Foundation for Science and Technology, I.P., within the project CISUC - UID/CEC/00326/2020, by European Social Fund, through the Regional Operational Program Centro 2020 and by the project AI4EU – "A European AI On Demand Platform and Ecosystem", project funded by the European Commission, H2020-EU.2.1.1., Grant agreement ID: 825619.

This page is intentionally left blank.

Faculty of Sciences and Technology

Department of Informatics Engineering

Robust Neural Networks

António Manuel Delgado Morais

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems supervised by Professors Raul Barbosa and Nuno Lourenço and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering

September 2021

1 2 9 0

UNIVERSIDADE Ð COIMBRA

This page is intentionally left blank.

Abstract

The growing usage of Machine Learning (ML) based systems in safety-critical contexts has prompted increased concerns over the reliability of the models and algorithms used. Despite their effectiveness, these models can make mistakes with serious consequences. These failures are often attributable to some sort of defect in the model architecture or lack of training data. Other times, however, these errors happen due to random hardware faults. To limit the effects of the latter, several methods have been developed and applied to ML models with the goal of increasing their fault tolerance. Models based on Deep Neural Networks (DNNs), particularly Convolutional Neural Networks (CNNs), are especially significant due to their applications in safety-conscious tasks such as autonomous driving or medical environments.

In this work, we study the effectiveness of existing methods in improving the fault tolerance of CNNs, such as Dropout, Redundancy, Ranger and Stimulated Dropout. We use four datasets of varying complexity that represent diverse applications of ML models, one of which in a safety-critical context. In addition, we combine some of these fault tolerance methods into hybrid approaches.

To measure the fault tolerance of ML models, we devise and implement a model-agnostic experimental process that uses the *ucXception* framework to inject faults during the testing phase.

Our evaluation of the tested methods shows that only Ranger and Stimulated Dropout consistently improve the fault tolerance of CNN-based ML models. Of these two methods, Stimulated Dropout shows the largest improvement in fault tolerance; however, the high computational costs of this method make its use a challenge for modern architectures in its current form, and further research is required to improve its performance.

Keywords

Safety-Critical Systems, Convolutional Neural Networks, Fault Tolerance, Fault Injection, C++, PyTorch, Dropout, Redundancy, Ranger, Stimulated Dropout

This page is intentionally left blank.

Resumo

A utilização crescente de sistemas baseados em Aprendizagem Computacional (AC) em contextos seguros-críticos tem levado a um aumento da preocupação associada à fiabilidade dos modelos e algoritmos utilizados. Apesar da sua elevada eficiência, estes modelos podem cometer erros com consequências graves. Estas falhas são com frequência atribuíveis a algum tipo de defeito na arquitetura do modelo ou a falta de dados de treino. Contudo, existem ocasiões em que estes erros acontecem devido a falhas aleatórias de *hardware*. De forma a limitar os efeitos destes tipos de falhas, vários métodos foram desenvolvidos e aplicados a modelos de AC com o objetivo de aumentar a sua tolerância a falhas. Modelos baseados em Redes Neuronais Profundas (RNPs), particularmente Redes Neuronais Convolucionais (RNCs), são especialmente significativos devido à sua utilização em contextos sensíveis como a condução autónoma ou aplicações médicas.

Neste projeto, estudamos a eficiência de métodos existentes para melhorar a tolerância a falhas de RNCs, como Dropout, Redundância, Ranger e Stimulated Dropout. Utilizamos quatro conjuntos de dados de complexidade variável que representam aplicações diversas de modelos de AC, uma delas num contexto seguro-crítico. Para além disto, combinamos alguns destes métodos de tolerância a falhas em abordagens híbridas.

Para medir a tolerância a falhas dos modelos de AC, idealizamos e implementamos um processo experimental utilizável com qualquer modelo que utiliza a framework *ucXception* para injetar falhas durante a fase de testagem.

A nossa avaliação dos métodos testados mostra que apenas o Ranger e Stimulated Dropout melhoram de forma consistente a tolerância a falhas de modelos de AC baseados em RNCs. Destes dois métodos, Stimulated Dropout mostra uma maior melhoria na tolerância a falhas; contudo, o elevado custo computacional deste método torna a sua utilização desafiante em arquiteturas modernas na sua forma atual, e mais investigação é necessária para melhorar o seu desempenho.

Palavras-Chave

Sistemas Seguros-Críticos, Redes Neuronais Convolucionais, Tolerância a Falhas, Injeção de Falhas, C++, PyTorch, Dropout, Redundância, Ranger, Stimulated Dropout

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

ANN Artificial Neural Network. xii, 4, 6, 7, 9–11, 21–26, 28, 33, 43, 65

API Application Programming Interface. 21

BN Batch Normalisation. 10

CCI Code Change Injection. 19, 21

CNN Convolutional Neural Network. iii, 1, 2, 7–10, 29, 37, 38, 60, 65, 66

CS Critical System. 14

DEI Data Error Injection. 19, 20

DL Deep Learning. 7

DMR Dual Modular Redundancy. 17

DNN Deep Neural Network. iii, 1, 7, 10, 13, 14, 17, 22, 23, 33

FC Fully-Connected. 10, 39

FIT Failure-in-Time. 14

HFI Hardware Fault Injection. 18

IEI Interface Error Injection. 19–21

LN Layer Normalisation. 10

LRN Local Response Normalisation. 10, 22

ML Machine Learning. iii, 1, 2, 7, 11, 13, 20, 28, 29, 33, 43, 63, 65, 66

MSE Mean Squared Error. 6

OS Operating System. 21

PL Programming Language. 32

PSD Power Supply Disturbance. 18

RGB Red, Green and Blue. 35

RNN Recurrent Neural Network. 10

SCIFI Scan-Chain Implemented Fault Injection. 20

SCS Safety-Critical System. 1, 14, 17, 28, 32, 65, 66

SDC Silent Data Corruption. xii, 22, 30, 32, 33, 44, 45, 55, 57, 62, 63, 65, 66

SEU Single Event Upset. 14

SFI Software Fault Injection. 18–20, 29

SoC System on Chip. 14

SOTA State-of-the-art. 30, 34

TMR Triple Modular Redundancy. 17

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

Chapter 1

Introduction

The adoption of Machine Learning (ML) has been expanding in recent years [57]. Recent advances in ML algorithms, especially Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs), as well as growing availability of large amounts of datasets and the reduced cost of computational capabilities, have been the catalysts for this expansion. As a consequence, ML is transforming a wide range of domains, from finance and health care, to science and technology [73]. Some of these domains have strict safety requirements; in these cases, the systems used are called Safety-Critical Systems (SCSs). SCSs are defined [62] as systems whose failure can result in loss of life, significant property damage, or damage to the environment.

The usage of ML in safety-critical contexts comes with particular concerns, one of which is the possibility that the model will make an unintended mistake, resulting in adverse consequences. Some of these mistakes are intrinsic to ML models since it is unfeasible to achieve an accuracy of 100% in every possible scenario in real-world circumstances. However, there are failures which are not related to the accuracy of the model and can occur at random in any computerized system. These mistakes happen due to hardware faults [110], which are caused by external factors that are difficult to control and predict. Because of these characteristics, these failures can compromise the security of these systems and reduce user's trust.

For the use of ML models, particularly those that use DNNs, to become achievable in safety-critical scenarios, the consequences of these events have to be reduced as much as possible. In this context, it is vital to create approaches that increase the fault tolerance of these types of models.

The objectives of this work are to examine, compare and discuss the impact of several techniques on the fault tolerance of CNN-based ML models in a data-driven way. These techniques include regularisation with Dropout [106], redundancy [110], modified learning through Stimulated Dropout [15], and Ranger [28]. In addition to applying these techniques to baseline models, we also combine these approaches into hybrid models.

1.1. Contributions

This work aims to investigate techniques for improving the fault tolerance of CNN-based ML models. In this context, our main contributions are detailed in the following paragraphs.

We implemented a well known CNN architecture [67] and created ML models that could be used as a baseline for comparing properties. We applied existing fault tolerance methods [15, 106, 110] as well as our own implementation of a recent method [28], to the baseline models. We selected some of these fault tolerance techniques and combined them, creating hybrid models. In addition, we trained and tested the models on four datasets of varying complexity.

We designed a model-agnostic experimental process to evaluate the fault tolerance of any ML model. This was achieved by using the *ucXception* [111] framework to inject faults in the models during testing.

Finally, we performed an analysis and comparison between the implemented ML models using several metrics. We quantified the impact of the selected fault tolerance methods in the performance and fault tolerance of ML models, and came to the conclusion that Stimulated Dropout and Ranger provide the most improvement in fault tolerance.

1.2. Structure

This document is organized as follows: in Section 1, we introduce the topic and problem statement and summarise the contributions; in Section 2, we provide the necessary theoretical knowledge to further understand the project topic and implementation; in Section 3, we describe the motivation, objectives and our planned approach to solve the problem; in Section 4, we detail the experimental process, including justifications for the selected tools and resources and explanations for the implementation; in Section 5, we present the experimental findings, analyse and compare the results on different metrics; and finally in Section 6, we present a summary of the work and specify future directions.

This page is intentionally left blank.

Chapter 2

Background and Related Work

2.1. Artificial Neural Networks

The development of Artificial Neural Networks (ANNs) was inspired in part by biological learning systems [85, p.82]. Even though there are some differences between them, ANNs can be thought of as loosely analogous to the neural systems in the brain. ANNs are built out of a dense and interconnected set of simple units, where each unit takes a number of inputs and returns a single output value. These inputs can be either the outputs values of other units, or external data points. Because of the similarity they share with their biological counterparts, these units are interchangeably called neurons or artificial neurons, and these networks can also be referred to as neural networks.

In order to understand how each unit works, we first need to learn about its history. The first representation of an artificial neuron was created by McCulloch and Pitts in 1943 [79], in what is commonly called the McCulloch-Pitts (MCP) neuron. It is described as a simple logic gate with binary outputs. As we can see in figure 2.1, there are multiple input signals which are integrated into the body of the neuron. If the accumulation of the values of those inputs exceeds a certain threshold, an output signal is generated and passed on through the axon.

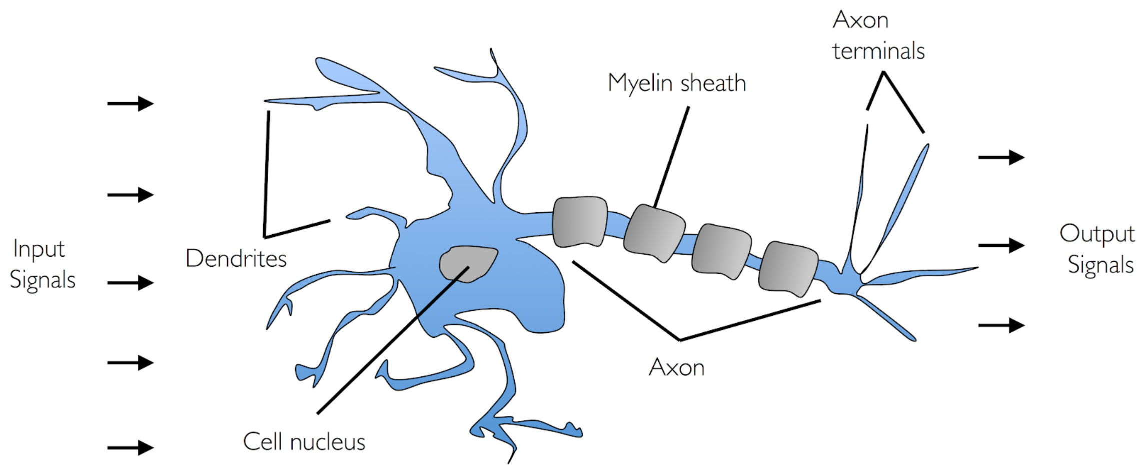


Figure 2.1: Model representation of biological neuron

Based on this work, Rosenblatt developed the perceptron in 1957 [100]. A perceptron takes in several inputs *x1, x2, ... xk* and produces a single binary output.

For the purpose of generating this output, Rosenblatt developed a simple rule. It starts by introducing weights *w1, w2, ... wj*, which are real numbers that are associated with a specific input and represent the importance the respective input has on the output. The binary output is then determined by whether the weighted sum of the inputs is greater than or less than a given threshold.

This threshold can be transformed into a variable called bias, which is equal to the negative threshold. Bias represents how easy it is to activate the given unit, which in this case means to turn its output from 0 to 1. We can see the expression for this rule in equation 2.1.

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{2.1}$$

Perceptrons can be joined together into networks. An example of a possible network [88] is represented in figure 2.2.

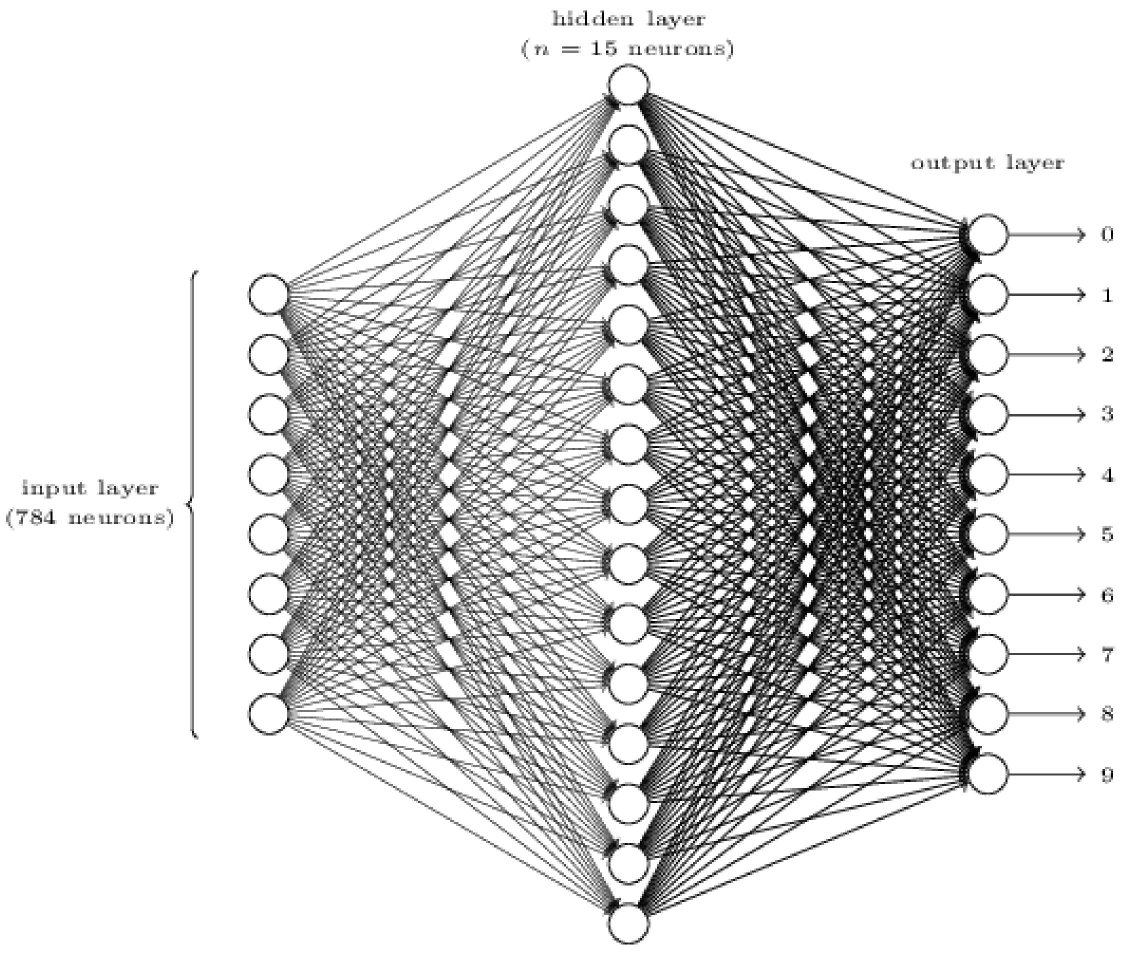


Figure 2.2: Example Artificial Neural Network

This structure can be classified as a feedforward neural network, since the output of one layer is used as input to the next. As we can see, each neuron of one layer is connected to each neuron of the next, which means that the layers are fully-connected. Since this is a perceptron network, each neuron receives and outputs a binary value. We can separate this network into sections with different roles: one input layer, one hidden layer and one output layer.

The input layer receives information from an input sample. Input samples can be any encoded information in the format of real or integer values. One example of an input is a grayscale image with dimensions of 28x28 pixels. In this case, each neuron in the input layer would represent one pixel, with the value of each neuron being the intensity of that pixel in binary, either 0 to 1.

The first hidden layer receives the weighted outputs of the input layer. There is usually more than one hidden layer, in such a way that the output of the first hidden layer is the input of the second, the output of the second is the input of the third and so on. Each of these layers can have any desired number of neurons. The total number of layers is called the depth of the network and the number of neurons of a layer is called the layer width.

We can intuitively think of each additional hidden layer as incrementally increasing the complexity of the network. This means that as a general rule, increasing the depth and width of a network will enable it to learn more abstract patterns.

The output layer receives the weighted outputs of the last hidden layer and generates the final return values for the network. The decision that the network makes depends on the neuron of the output layer with the highest value.

Using Figure 2.2 as an example, we can see that there are 10 neurons in the output layer. If the output of the first neuron is 1 and the outputs of the rest are 0, then the decision of the network is that the input image belongs to class “0”.

As previously mentioned, in perceptron networks, all neuron values are binary. This means that learning complex patterns is realistically impossible. Because of this, the function that generates the output of each neuron, called the activation function, is modified in most ANN architectures. One widely used activation function is the sigmoid function, which receives and returns real values.

In the context of ANNs, learning means adapting the weights and biases of the network to a given training dataset. This means that given a set of examples and their respective labels, the network should be able to output values that correctly classify each of the training inputs. To quantify how well we are achieving this goal, we utilise cost functions. One example of a cost function is Mean Squared Error (MSE), which simply measures the average of the squares of the errors. In this case, the errors are the difference between the expected labels of a given input and the actual labels that were returned by the network. We can see an example cost function that utilises MSE in equation 2.2, where w represents the weights, b corresponds to the biases, n is the number of elements in the training dataset, y is the correct label for sample x and $\hat{y}$ is the predicted label for that sample.

$$C(w,b) \equiv \frac{1}{n} \sum_x \|y - \hat{y}\|^2. \tag{2.2}$$

The objective is therefore to minimise this cost function. This can be achieved in many ways, one of which is by using the gradient descent algorithm [88]. This algorithm works by iteratively taking small steps in the direction that decreases the cost function. In practice, this means that the algorithm calculates the gradient for the weights and biases of the network, and then moves in the direction specified by the negative gradient. This process continues until a local minimum is reached.

The most common way for computing the gradient of the cost function for one input and output set is through an algorithm called backpropagation [101]. This computation is done one layer at a time, starting from the output layer and ending on the input layer. This is accomplished by applying the chain rule in order to avoid redundant calculations. This method is more efficient than traditional, naive approaches that calculate the gradient separately for every layer.

2.2. Deep Neural Networks

Now that we have explained in broad terms how ANNs work, we can introduce specific implementations that are relevant to this thesis.

Deep Neural Networks (DNNs) are ANNs that have multiple layers between the input and output layers. Deep Learning (DL) is a term that encompasses all Machine Learning (ML) methods that utilise DNNs. In recent years, DL has taken a dominant role [74] in an immense amount of fields, both in terms of state-of-the-art performance and real-world applications.

It is generally agreed upon that DL was officially introduced in 2006 when Hinton proposed a novel deep structured learning architecture called Deep Belief Network [47]. Since then, growing attention has been given to this field, resulting in numerous breakthroughs, for instance in 2012 [53] when a team led by Hinton won the ImageNet image classification competition.

2.3. Convolutional Neural Networks

Convolutional Neural Network (CNN) is a type of DNN that is especially well suited for the task of image classification [42]. To explain how CNNs work, it is helpful to start with the ANN structure from Figure 2.2.

CNNs are basically ANNs which have two particular types of layers: convolutional and pooling. There are three important concepts to understand how CNNs work: local receptive fields, shared weights and pooling.

In order to understand the concept of local receptive fields, we can visualise the input neurons in the format of the input image. For instance, if we have an input image of 28x28 pixels, the input neurons are represented as a 28x28 square. Now instead of connecting each neuron of the input layer to every neuron of the first hidden layer, we will only connect a localised region of the input space. In Figure 2.3 we can see an example with a region of 5x5 input neurons, which is connected to one hidden neuron. This region is called a local receptive field.

The next step is to iteratively slide this region through the input neurons, from left to right and top to bottom, until the region collides with the bottom right corner of the image. The length of this slide is a modifiable parameter called stride. In most cases, the value of the stride is 1.

In CNNs, the weights and biases are shared between all of the hidden neurons of a given

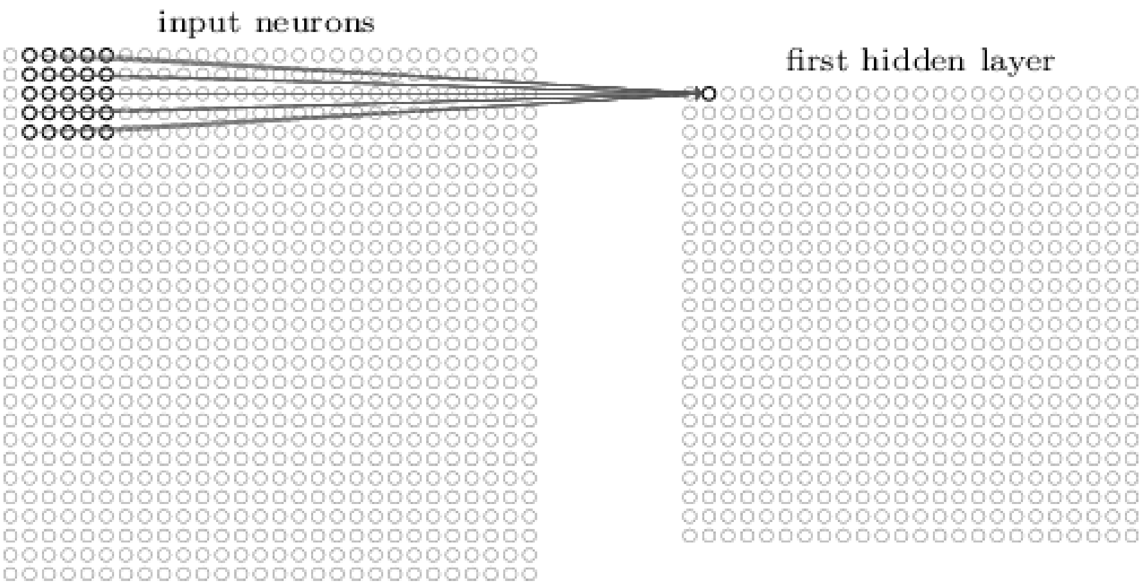


Figure 2.3: Mapping of 5x5 region to hidden neuron

hidden layer. This means that all of the neurons that belong to the same hidden layer are detecting the same pattern or feature, only for different regions of the input space. The mapping between the input layer and a given hidden layer can also be called a feature map. Convolutional layers are essentially an aggregation of a predetermined number of feature maps, as we can see in figure 2.4. Each of these feature maps represent the capacity to detect one feature at every possible location of the input space. The dimensions of this feature are the same as the dimensions of the region in the input space.

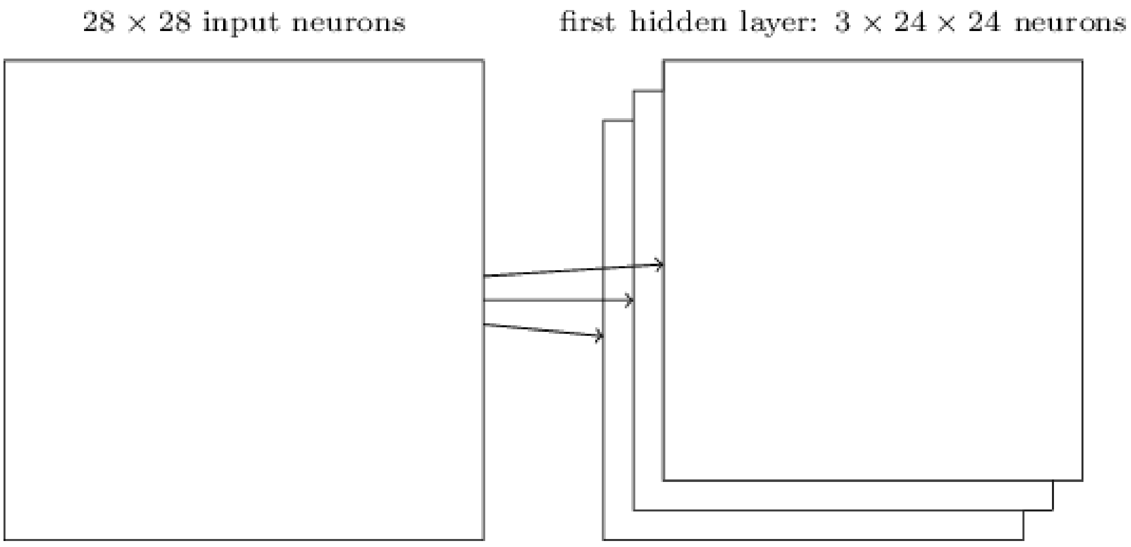


Figure 2.4: Feature maps in hidden layer

Finally, CNNs have an additional type of layer called a **pooling** layer. These layers are usually placed directly after a convolutional layer and are meant to summarise each of that layer's feature maps.

Pooling layers are composed of pooling units. These units contain the output of a pooling operation that is performed on a region of a given feature map. This region is necessarily smaller than the region used to create the feature map. In figure 2.5, since the previous region had dimensions of 5x5, the pooling region has dimensions of 2x2.

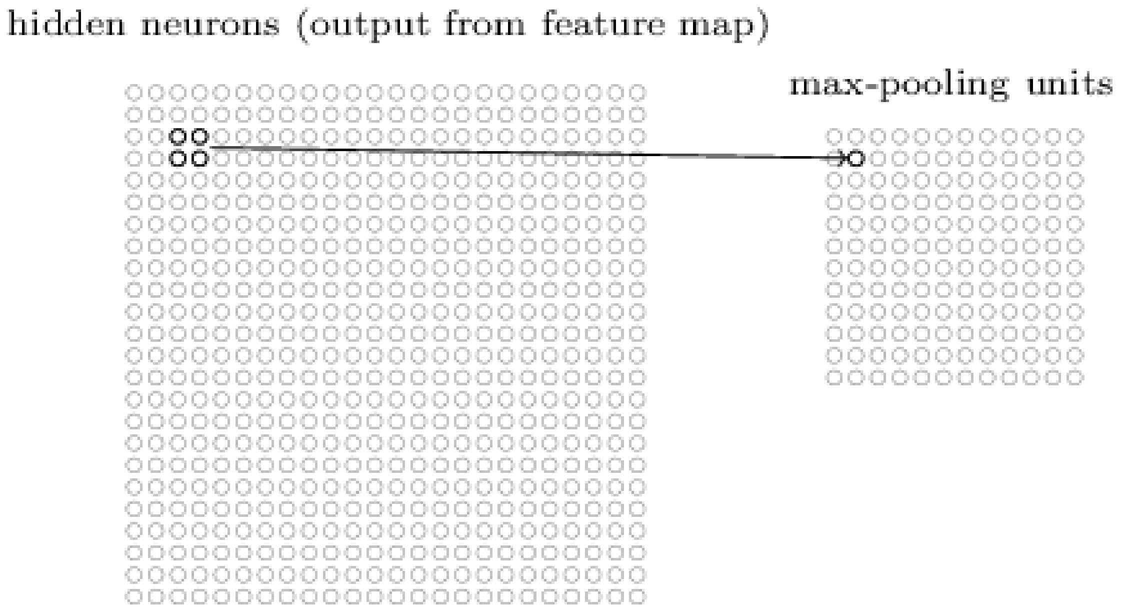


Figure 2.5: Max-pooling example

There are several different pooling operations. Some of the most commonly used include max-pooling, which simply returns the maximum activation value out of a given region, and average-pooling, which calculates the average in the region.

The final scheme of an example CNN can be viewed in Figure 2.6.

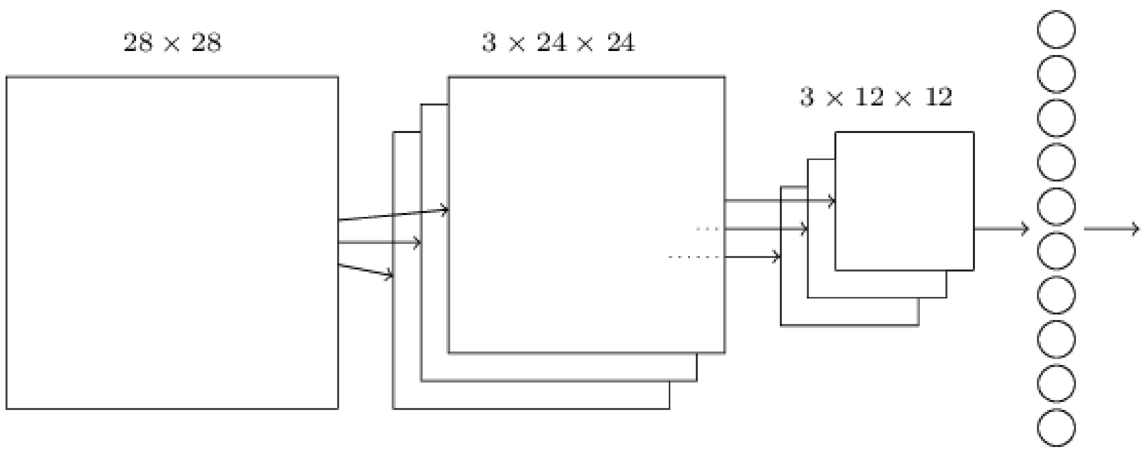


Figure 2.6: Example Convolutional Neural Network

In this case, the pooling layer is fully-connected to the output layer. Similarly to the previously mentioned ANN architecture, we can now train the CNN using, for instance, backpropagation and gradient descent.

2.4. Normalisation

Normalisation is the process of standardizing data, i.e. putting data in the same range. The inputs of DNNs are usually normalized as part of the pre-processing stage. Even with this initial process, the activation values of neurons in the hidden layers of the network can begin to deviate from their usual magnitude, both between different layers, across neurons in the same layer, and over time,[54, 118], which can negatively impact training accuracy. In order to diminish this issue, normalisation can be added to the network as a layer. Normalisation layers have two purposes: speeding up training time and improving generalization accuracy [108].

Local Response Normalisation (LRN) is a type of normalisation first introduced in the AlexNet architecture [65]. It was inspired by a concept from neurobiology called lateral inhibition, where excited neurons should subdue their neighbours. In ANNs, this means that neurons with higher outputs inhibits adjacent neurons, creating local maximums and increasing the significance of the high value neurons. In CNNs, LRN can be applied within the same channel, or across different channels. Later architectures such as VGGNets [105] removed LRN since its effect was not significant in CNNs with a high number of layers [72].

Batch Normalisation (BN) [54] is a normalisation technique that is applied to individual layers. It works by first normalising the inputs of the specific layer. This normalisation is achieved by subtracting the mean and dividing by the standard deviation, where both values are obtained from the current minibatch. Afterwards, we apply a scale coefficient and scale offset. BN has a different implementation depending on the type of layer it is applied to; if it is a Fully-Connected (FC) layer, BN is inserted after the affine transformation (e.g. summation) and before the activation function, whereas in convolutional layers, BN is applied after the convolution and before the activation function.

BN has two major drawbacks. First, it produces inconsistent results in training and testing, which makes it inadequate for complex architectures such as Recurrent Neural Network (RNN) [50]. Second, the error increases dramatically when using small batch sizes [115]. This means that in order to use BN, a sufficiently large batch size needs to be selected (e.g. 32, 64), which can be memory-consuming depending on the task. Despite these issues, BN is one of the most common normalisation techniques for DNNs [3] and is an integral part of several prominent architectures [45, 49, 109].

Another widely used normalisation technique is **Layer Normalisation (LN)** [50], which aims to correct or improve some of the drawbacks of BN. This technique works similarly to BN, with the exception that it estimates the normalisation statistics (mean and variance) from the summed inputs to the neurons within a hidden layer. This is done individually for each training sample, which makes it so that the normalisation does not introduce new dependencies between training cases.

2.5. Overfitting in Artificial Neural Networks

Overfitting is an issue in ML performs well in the training set but fails to generalise to unseen data [118]. In ANNs this happens because the weights are being tuned to fit particularities of the training examples which are not representative of the general distribution of samples [99, p. 73]. This process is particularly apparent during later iterations of the training algorithm, as the decision surface becomes overly complex [85, p. 111].

There are two important concepts related to overfitting: training error and generalization error. The training error is the error of the model as calculated on the training set, whereas generalization error is a measure of the model performance on previously unseen data. Generalization error is estimated based on a validation set (hence often being referred to as validation error), which is randomly sampled from the training set [118]. Methods that prevent overfitting aim to reduce the generalization error.

2.5.1. Data augmentation

One of the most common ways of preventing overfitting is by using a larger training dataset, either by collecting more data, or through data augmentation. Data augmentation [39, 103] is a set of techniques that aims to enhance the size and quality of training datasets. In image data augmentation, two of the most relevant techniques are data warping and oversampling.

Data warping involves transforming existing images while keeping the same label. Example transformations include altering geometric or color properties [104], erasing parts of the image at random [119] or adversarial training [41].

Oversampling is another data augmentation technique. It is based on creating artificial samples using methods such as mixing existing images [61] or utilising generative adversarial networks (GANs) [23].

These techniques are not suitable for every scenario. Smaller datasets can have inherent biases [104] which cannot be prevented or reduced with Data Augmentation.

2.5.2. Regularization

Regularization is defined [42, sec. 5.2.2] as any modification that is made to a learning algorithm that is intended to reduce overfitting. Regularization techniques can be used instead of or in addition to the previously mentioned overfitting prevention methods. There are various regularization methods, of which we will mention Early Stopping, L1/L2 Regularization, Dropout and Stimulated Dropout.

2.5.3. Early Stopping

Early Stopping is one of the most commonly used forms of regularization in ANNs [95]. As we have previously mentioned, after a certain point in the training process, the generalization error starts to increase. We have also noted that this can be seen as an indication that the model is overfitting.

The principle behind early stopping is that we can obtain a model with better generalisation capacity by returning the state of the model at the point in time at which it had the lowest generalisation error. The learning algorithm is therefore terminated whenever there has been no decrease in the generalisation error over a predefined number of iterations. This process can be visualised in Figure 2.7.

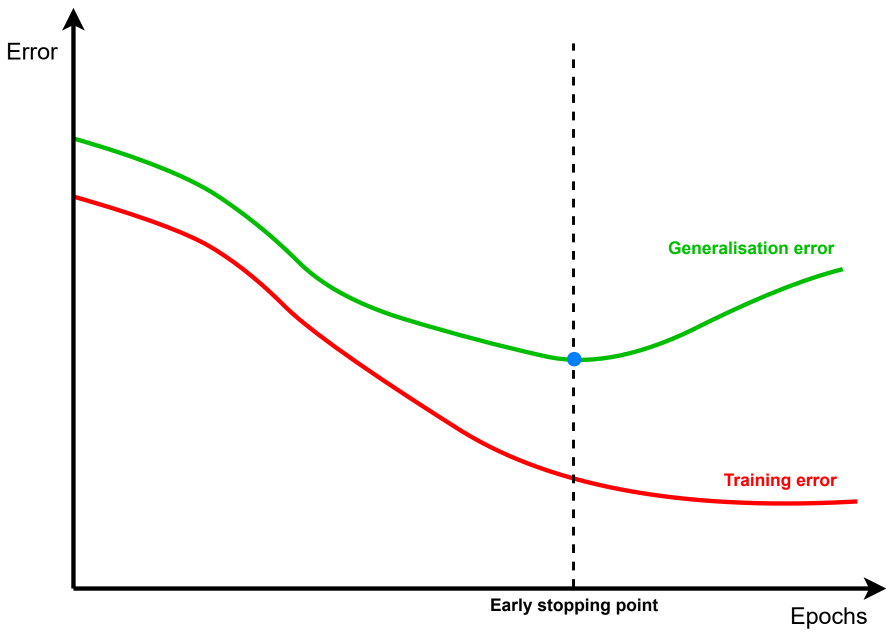


Figure 2.7: Early Stopping when there is an increase in generalization error

2.5.4. L1 and L2 Regularization

L1 and L2 regularization are other commonly used techniques. The basic concept behind these methods is that they penalise models that have large weight values, such that these weights are driven towards the origin (zero). They do so by subtracting a penalty term to the cost function, which is calculated differently for both types. The penalty term for L1 is the sum of the absolute values of the weights, while for L2 it is the sum of the squared values of the weights.

The motivation for L1 and L2 regularization is that if the weight values are kept small, the learning algorithm will become biased against complex decision surfaces, resulting in a simpler model that has a better generalization capacity.

L1 regularization often results in a model that is sparse [42, sec. 2.3.2]. Sparsity in this sense means that most model parameters are zero. For this reason, L1 is utilised in feature selection.

2.5.5. Dropout

Dropout [106] is another regularization method. It provides a simple and computationally efficient way of preventing overfitting. During each training iteration, each neuron has a predetermined probability p of being omitted. In the original version of dropout, both input and hidden layer neurons can be omitted and the probability p can be different between layers (e.g. 0.2 in the input layer and 0.5 on the hidden layers). If a neuron is omitted, it will not be considered during that iteration, which leads to a continuous changing of the network at every iteration.

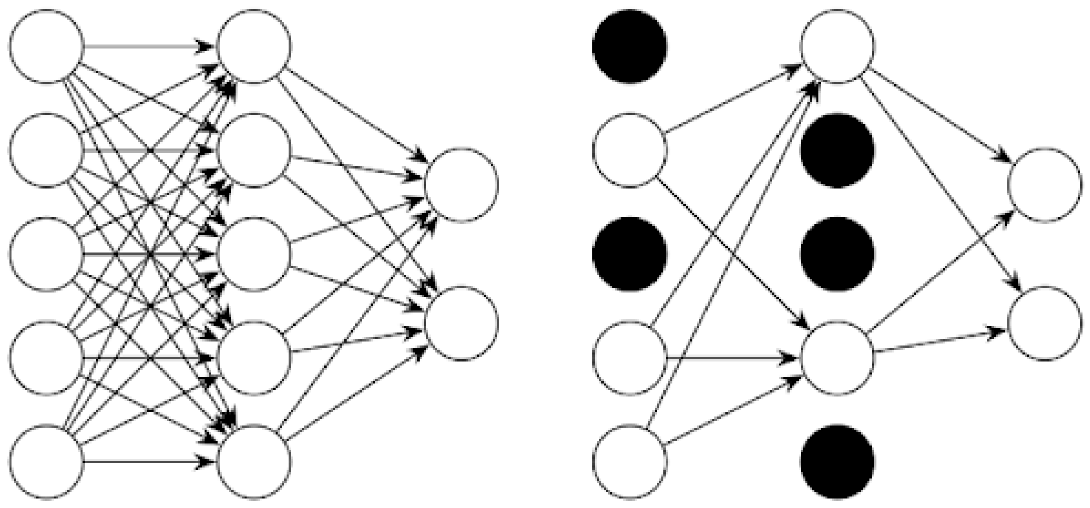


Figure 2.8: Example network with dropout

In the testing phase, the full network is utilised, including the neurons that were omitted. However, every neuron output is multiplied by the probability p of its corresponding layer. This serves to compensate for the larger size of the network, since the outputs of each layer will be larger than in the training phase. This can be interpreted [66] as the averaging of the several different networks that were generated during training. This averaging reduces the testing error, leading to better results than previously mentioned methods such as L2 regularization [92].

2.6. Dependable and Secure Computer Systems

In the past decade, ML has been increasingly used to solve complex problems, particularly through DNN models [17, 117]. There are some problems, such as the classification of skin cancer [38] or the detection of arrhythmia [98], among others, that are achievable by these kinds of models with similar accuracy to specialists in those fields. Recently there have been major advances in the usage of DNNs for object detection and image classification purposes in autonomous vehicles with promising results [30]. Nowadays, there is a higher circulation of autonomous vehicles than ever before, something which is expected to grow in the future [24, 117].

Since these types of models are being used more and more in situations that are sensitive in terms of safety, there is a discussion to be had about the possibility of these models making mistakes, as well as the negative impact of those mistakes.

Every computer-based system delivers a service to its users [18]. When that service corresponds to the function of the system (i.e. what the system is intended to do) it is said that the system delivered a correct service. On the other hand, if the service deviates from what is expected, this is referred to as a system failure. A deviation between expected and actual service is called an error.

Errors are caused by faults, which are anomalous physical conditions in the system. Faults can occur for several reasons, such as thermal cycling, transistor variability or Single Event Upsets (SEUs) [90]. SEUs happen when highly energetic particles (e.g. alpha particles) strike sensitive regions of a microelectronic circuit [35]. These events occur arbitrarily in natural environments.

In autonomous driving, one example of a failure is misclassifying an object, for instance by mistaking a vehicle for a bird [28], resulting in a delayed response that can potentially cause an accident. Autonomous driving systems that are reliable have a low probability of such misclassifications happening or at least affecting the correct functioning of the system.

Because of the potentially harmful consequences of failures in sensitive domains, software solutions are required to meet strict standards such as IEC 62304 [55] for medical and ISO 26262 [56] for automotive applications.

In order to set limits for the failures in different domains, failure rate metrics such as Failure-in-Time (FIT) are widely used [70]. FIT represents the number of failures that can be expected in one billion hours of operation. Standards such as ISO specify the maximum FIT rate of devices in that domain (e.g. ISO 26262 limits the FIT rate to less than 10 [56]).

The FIT rate is calculated for the entire system in question [70], called the System on Chip (SoC), and not just for the part of the system that is running specific processes such as a DNN. This makes the fault tolerance of DNNs even more important, since they need to be minimised in order to make sure the total FIT rate does not exceed the limits.

2.7. Critical Systems

Critical Systems (CSs) are systems where a failure can have a significant negative impact at a human or economical level [46]. There are three types of CSs: Safety-critical, Mission-critical and Business-critical.

Safety-Critical Systems (SCSs) are systems in which failure can lead to loss or injury to life, property damage and/or damage to the environment [63]. This designation encompasses a wide-ranging selection of fields, such as:

- Infrastructure, for instance in emergency services dispatch systems, electrical distribution and generation and telecommunications;
- Medicine, in devices like mechanical ventilation, dialysis or robotic surgery, as well as in medical imaging devices such as X-rays, computerized tomography (CAT) or magnetic resonance imaging (MRI);
- Nuclear Engineering, in nuclear reactor control systems;
- Transport, from aviation, railway and spaceflight control systems to the automotive industry.

Mission-critical systems are essential to the success of a specific task or the survival of a business or organization [46]. Failures in these types of systems result in the termination of a specific operation, at the cost of time, resources or equipment. Examples of this category include spacecraft navigation systems (where a failure can cause the spacecraft to get destroyed), or database systems with no backup (where power outages can lead to loss of important data).

Business-critical systems [46] are systems in which failure leads to significant economic losses, at a tangible (e.g. money) and/or intangible (e.g. reputation) level. If these systems fail, the business or organization can continue to function or partially function, but there is a loss of time or resources. One example is a customer accounting system in a bank, where if it fails, the data is not lost, and the business can continue operating without it.

2.8. Faults

Faults can be classified according to their characteristics [110]. When a fault is constant through time (which usually happens in cases where there is persistent physical damage in the microelectronic components) it is called permanent. When the fault only lasts for a brief interval, it is referred to as a transient fault or soft error. When transient faults are recurring, they are called intermittent. Most of the faults that occur in systems with current semiconductors are transient or intermittent [110].

In modern devices, the decreasing size of computer chips and the adoption of certain energy management techniques (e.g. dynamic voltage scaling) increases those chip's vulnerability to transient faults [22, 91, 102]. Existing problems such as flaws in the design/manufacturing of the components or using those components beyond their expected lifespan further worsens this vulnerability. These characteristics contribute to the fact that transient faults are much more common than permanent faults, by a rate of up to 100:1 [94]. Transient faults have also gathered attention in recent years because of their impact in cloud services [36].

Fault models describe the system components that become defective, as well as the conditions in which this occurs [21]. Two widely used [33] fault models are:

- **Stuck-at**, where data or a control line is stuck at a high or low value or state (also referred to as stuck-at-1 and stuck-at-0, respectively);

- **Random bit-flips**, where a data or memory element is assigned an incorrect, randomly generated value.

The stuck-at fault model intends to mimic defects at the transistor and interconnection structures level. These defects are examples of permanent faults [110].

The random bit-flip fault model [110], on the other hand, simulates faults that happen at the register or memory level. These are examples of transient faults. A bit-flip operation takes place when one bit of a binary value is flipped, i.e. switches to the opposite of its current state. This alteration changes the logic value of the corresponding memory element, affecting the processes which are accessing that memory element at the time. A diagram for this operation can be seen in Figure 2.9.

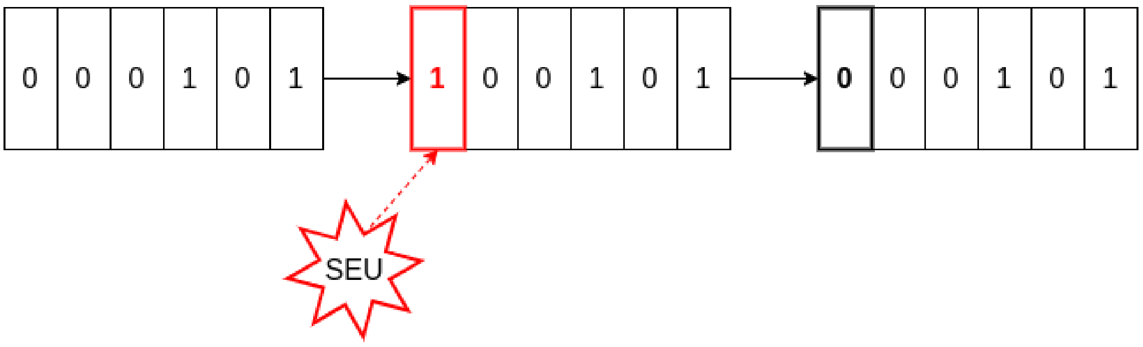


Figure 2.9: Example of a Random bit-flip caused by an Single Event Upset

2.9. Fault tolerance

Fault-tolerant systems have been studied in depth in research [110]. This implies several key issues, from understanding how faults are originated in the first place, to categorising faults based on similar characteristics into types [14, 18, 80] and lastly to evaluating the behaviour that systems have when faults occur. The ultimate goal of this area of study is understand the impact that faults have on systems, and develop methods to prevent or mitigate those effects.

There are several concepts [18] which are important to define in order to understand this field:

- *Dependability* is defined as the capacity of a given system to deliver trustworthy service, i.e. to keep the amount of failures at a reasonable level.
- *Trustworthiness* is the degree of assurance that a system will perform as expected.
- *Availability* is the readiness of correct service at any given time.
- *Reliability* is the ability of a system to perform its required functions under certain conditions for a given time interval [11]. Highly reliable systems have a low probability of failure, which often translates to an increased trustworthiness of that system; this concept is therefore very important when deciding on whether to use computer systems to solve a certain task.
- *Safety* refers to the absence of harmful consequences on users, property and environment in the case of a system failure.
- *Integrity* is the ability for a system to prevent information modification or destruction, while at the same time ensuring properties such as non-repudiation and authenticity.
- *Maintainability* is the ability for a system to be modified or improved.
- *Robustness* can be defined as dependability with respect to external faults (e.g. hardware-related faults). Robust systems continue to operate as intended regardless of noise or variation of internal values.

Avizienis et al. [18] define several means to improve the dependability of a system.

- *Fault prevention* refers to preventing faults or fault introductions (e.g. through fault injection) from happening.
- *Fault tolerance* is the ability of a system to provide correct service despite the occurrence of faults.
- *Fault removal* refers to means to reduce the number and severity of faults.
- *Fault forecasting* refers to means to estimate the current number, the future incidence, and the likely consequences of faults.

Traditionally, the fault tolerance of a system was assured by exploiting redundancy at the hardware level, through methods such as replication, where multiple systems perform the same operation and the final output is decided by majority-voting. Examples of this technique include Dual Modular Redundancy (DMR) [71, 113] and Triple Modular Redundancy (TMR) [51, 76, 116], where two and three systems are used, respectively.

There are major downsides to this type of approach, such as overheads in cost, performance and energy consumption [71]. This is further worsened by the fact that DNN models often have applications with energy or spatial constraints (e.g. embedded devices). In addition, replication-based techniques require synchronization between the systems, which may not be trivial. For these reasons, these methods are not suitable for systems that require swift and efficient functioning, such as most SCSs.

2.10. Fault injection

The occurrence of faults in a natural environment is arbitrary and rare. This makes it troublesome to study their effects. Because of this, techniques have been developed to either create or replicate faults at will.

Fault injection is defined [19] as the deliberate introduction of faults into a system. After the injection, the system is usually examined in some way in order to identify any possible errors or effects of that injection. A sequence of fault injections in the same system is referred to as a fault injection campaign.

There are several characteristics which are important to have in fault injection tools:

- Repeatability: repeated fault injections should get the same results;
- Controllability: ability to control the temporal and spatial components of the fault injection;
- Intrusiveness: degree of the unintentional effects that the fault injection tool has on the system at a temporal and spatial level;
- Observability: how easy it is to observe and measure the effects of fault injection;
- Reachability: ability to access locations in the processor on which to inject faults;
- Reproducibility: ability to replicate the results of a fault injection campaign.

One concern in fault injection is that testing every single possible fault bears tremendous computational cost, even for moderately complex systems [64]. Consequently, it is common to test only a fraction of the possible faults for controllability and reproducibility purposes. The results are approximate estimates of those obtained by exhaustive testing [29].

The earliest methods for fault injection, known as **Hardware Fault Injection (HFI)**, worked by introducing physical disturbances into the hardware components of the system. These types of tools can employ different techniques [19], such as pin-level fault injection, Power Supply Disturbance (PSD), electromagnetic interference or particle radiation (e.g. heavy-ion radiation).

Pin-level fault injection can have two types: Active probe or Socket insertion. Active probe utilises specialised probes which are connected to the integrated circuits or hardware components of the target system. These probes inject faults in the attached components by altering their electrical current [48]. One downside of this approach is that if the electrical current is too high, the target hardware is likely to become damaged.

On the other hand, in socket insertion, a socket is added between the target hardware and its corresponding circuit board, allowing for the injection of complex faults into the pins of the target hardware.

Some examples of tools that use pin-level fault injection include RIFLE [77] and Messaline [16].

PSD is another method of HFI. It consists of dropping the supply voltage of the circuits below the expected range [19]. This affects several nodes in the circuit at once and causes multiple transient faults. It represents what might happen in case of a supply outage that affects a computer system in a real-life scenario [60].

Transient faults can also be generated by performing electromagnetic bursts on part or the entirety of the integrated circuits. This can be achieved by placing the target hardware near an electromagnetic field [48]. This is considered a relatively imprecise method [48] because it is difficult to control the moment at which the electromagnetic field is created, and therefore it is also hard to regulate the time and location of the fault injection.

Lastly, particle radiation [35] can also be used to inject transient faults without physical access to the hardware components. It is representative of what can happen in a real-life situation, where neutron and alpha particle hits can cause errors. As the size of modern integrated circuits becomes smaller, the probability of these particle hits increases, making this a problem of increased importance [91].

Software Fault Injection (SFI), on the other hand, uses software tools that emulate hardware faults by perturbing the state of memory or the hardware registers. SFI tools can only inject faults on the resources that the hardware allows them to access.

The main advantages of SFI approaches are that they do not require expensive specialized equipment [40], they can be applied to most computing devices [93] and they can be programmed to inject faults automatically without supervision. They are also capable of simulating an extensive range of fault models, from hardware to software (e.g. data, interface and code) faults [19]. In general, these tools also offer more controllability and repeatability when compared to HFI methods [64].

There are several properties which are important in SFI tools [87]:

- Representativeness is the ability to simulate the actual faults that a system might have in operation. This property depends on the realism of the fault models. Realistic fault models emulate the types of faults that are expected to occur in the system. They should also have the frequency distribution of normal circumstances. These characteristics can be obtained by studying the failures that occur spontaneously on the target system or in a similar system, a process called field failure data analysis. Failures can also be compared with other fault injection methods that have demonstrated accurate failure representation.

- Usability is the ability to apply the SFI tool on any given system. This property depends on the following aspects:
  - Portability: SFI tools should be functional on different operating systems, applications and hardware components;
  - Intrusiveness: the SFI tool should not affect the results of the experiments (e.g. by impacting the performance);
  - Flexibility: the SFI tool should cover a diverse range of fault models, as well as have the capability to add new ones;
- Efficiency is the ability to produce experimental results with an adequate amount of resources and time. The efficiency of an SFI tool depends on the effort that is put in to obtain useful findings. Common indicators for efficiency include the number of fault injections, and the probability of a fault injection causing an error. The optimal type, location and timing of the injection should be determined by analysing the system.

We can differentiate between SFI tools based on the timing of the injection: **compile-time** and **runtime** [48]. In compile-time injection, the program instructions are modified before the program is loaded and executed. This results in errors being injected into the source code or assembly code of the program, which allows for the emulation of different types of faults (e.g. hardware, software, transient and permanent) without any auxiliary software program. One limitation of this approach is that the user needs to have access to the code of the target program.

Runtime injection uses triggering mechanisms to activate the fault. There are different types of triggering mechanisms, such as timers, exceptions/traps, or code insertions.

Timers expire at an exact instant, generating a *time-out* event. This event creates an *interrupt* signal that is used to trigger the fault injection. Hardware timers need to be associated with an interrupt handler which injects the fault. This technique is adequate for emulating both transient and intermittent faults.

Instead of only injecting faults at an exact instant, exceptions/traps can be used to perform fault injection before, during or after specific events occur. In the case of software traps, the fault can be injected after a specific program instruction, whereas in hardware exceptions faults are injected when a hardware event occurs (e.g. accessing a specific memory location).

Another runtime fault injection technique is code insertion, which works by inserting certain instructions on the source code of the target program just before an event occurs [75]. These instructions inject the fault and then the program continues. This technique differs from compile-time injection in that it adds instructions instead of modifying existing instructions.

There are three main classes of SFI [87], divided according to the types of faults they are intended to simulate: **Data Error Injection (DEI)**, **Interface Error Injection (IEI)** and **Code Change Injection (CCI)**.

DEI works by corrupting the values in a memory location or hardware registers. Though originally intended to simulate the errors caused by hardware faults, DEI tools can also be used to emulate software faults [20]. Different fault models can be used, including *stuck-at* or *bit-flip*. When injecting faults in memory, the location of the injection can be either deliberately selected (e.g. a specific variable) or randomly chosen from a memory area, whereas injection in registers can be done in any of the registers that are accessible through the software.

FIAT [20] is a DEI tool that emulates both hardware and software faults. Injections are performed at compile time, by a fault injection library which is linked to the target process. It is possible to control the experiments and analyse the results through a simple user interface. One limitation of this approach is that it requires access to the source code of the target process.

Xception [26] is another tool that injects faults at runtime. it takes advantage of existing debugging and performance monitoring capabilities of modern CPUs in order to ensure minimum interference in the target system, thus decreasing its intrusiveness. This tool is able to time faults with great accuracy because it uses performance counters instead of the less accurate system clock. One disadvantage of *Xception* is that it is reliant on features that are only available in modern CPUs, which invalidates the usage of this tool on older hardware.

GOOFI [13] is another example of a DEI tool. It is implemented in Java and includes a SQL database that stores information about the experiments. Since it uses programming languages which are available in a wide range of systems, it is easy to adapt this tool to new systems. Because of its object-oriented approach, this tool also provides flexibility in the types of faults that can be injected. Besides single and multiple transient faults, it allows for new types of faults to be implemented easily.

In addition to providing support for compile-time SFI, *GOOFI* can also perform Scan-Chain Implemented Fault Injection (SCIFI), which uses components that are present in modern VLSI circuits to inject faults directly into the pins and internal state elements of the integrated circuit.

TensorFI [69] is an example of a recent SFI tool that is built onto TensorFlow [10], a popular ML framework. It modifies the TensorFlow graph and allows for the injection of both hardware and software faults.

Since *TensorFI* is meant to be used with TensorFlow, the user only needs to add a small number of lines in the Python code in order to use this tool. It is also supported by previous versions of TensorFlow, enabling the programmer to use it in existing programs without modifications. Finally, there is a minimal overhead to the execution, which means *TensorFI* is a very efficient tool.

BinFI [29] is an adapted version of *TensorFI*. This tool searches for the critical bits in a process (i.e. the bits in which the fault injection causes errors). In comparison with other methods that also identify the critical bits, *BinFI* produces similar results while ensuring a speedup of 5x. This is because instead of performing an exhaustive search like existing methods, it uses binary-search, which is more efficient.

IEI works by corrupting the input or output values that are exchanged between the target component and the environment or other software/hardware components. This exchange is done through the target component's interface.

When the inputs are corrupted, this tests the target components ability to tolerate faults that occur in external components, whereas when the faults are injected in the output values, we are able to evaluate the effects of internal faults on the external components. Testing with fault injection in the input values has particular importance in programs that include an Application Programming Interface (API), since they need to be able to handle faults that might occur in a wide range of devices.

IEI tools can be based on two types of techniques: *test-driver*, which uses a special program that is linked to the target component and tests it with corrupted inputs; and *interceptor*, which corrupts the inputs that are sent from other components to the target component.

Fuzz and **Ptyjig** [83] are two of the earliest IEI tools. They send random data to the target process (in this case, a UNIX utility program) in order to check if the altered data induces errors. The results were that the tools were able to crash or hang between 24% and 33% of utility programs on three UNIX systems. Over the years, these tools were tested on modern Operating Systems (OSs) [82, 84] and the conclusion is that there is some improvement; in 2020, the failure rates were between 12% and 19% using the original methods.

Code Change Injection

CCI [87] is used to emulate software faults by injecting code that causes programming bugs. The injection can be achieved by simply modifying a small set of instructions in the source code or the binary executable. This is often done before execution [32] in order to make the faults permanent. Some studies inject faults during execution [37] in order to have more control over the timing of the injection.

FINE [58] is the earliest tool to employ CCI. It is able to inject hardware-induced software errors as well as software faults. *FINE* injects faults into the binary code of a target process in systems with a UNIX kernel. This tool was later extended with **DEFINE** [59], which adds the capacity for executing experiments in a distributed environment and expands the available fault types. Using these tools often results in a high number of inactivated faults, since the inputs required to activate the faults are frequently very specific.

2.11. Effects of faults in Artificial Neural Networks

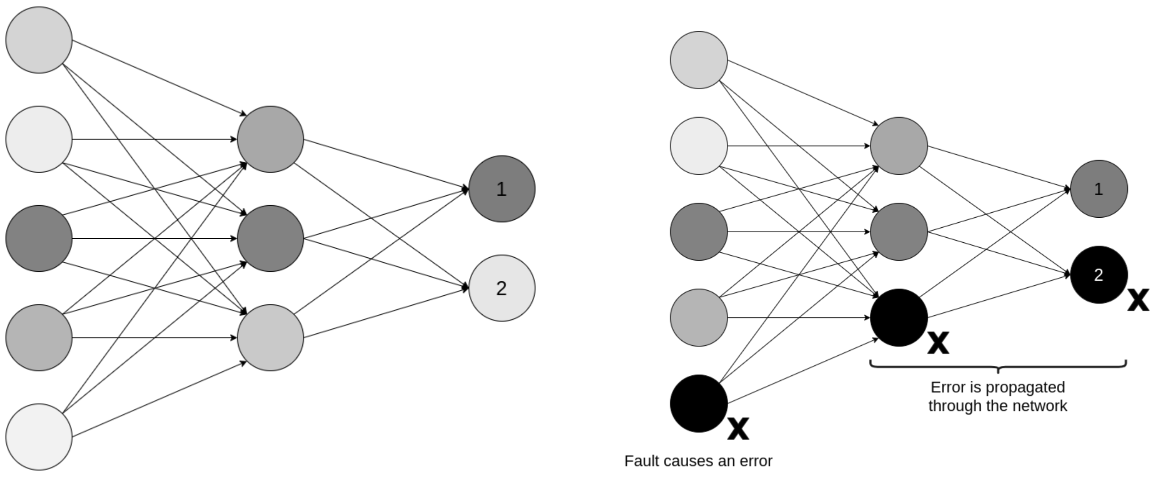
Any process that is running on a computer-based system can be affected by faults. As mentioned in the previous section, transient faults do a bit-flip on one of the CPU registers of the computer. In practice, this means that the process that is using those registers at a given time will use defective values, which can lead to errors in internal values. If that process happens to be an ANN, these types of errors can impact it in several different ways:

- Input layer: change in the values of the input neurons;
- Weights: change in the values of the weights or in the values of mathematical operations involving the weights;
- Neuron body: change in the value of the summation or the output of the activation function. If the activation function has fixed bounds (e.g. *tanh* has a range of $[-1, 1]$), changes in the summation often cause the neuron to become *saturated*, i.e. stuck at

the upper or lower bounds. In the cases where the activation function is unbounded (e.g. *ReLu* has a range of $[0, +\infty]$), these alterations can result in

It is commonly held that ANNs have some intrinsic tolerance to the impact of faults [110]. In other words, if a fault causes a fluctuation in the value of some neurons in the network, this deviation can evened out by the remaining neurons. This is attributed to ANNs distributed structure and over-provisioning [81] (i.e. having more neurons than the minimum number required to perform a computation).

In most cases, this intrinsic capacity makes ANNs capable of withstanding the impact of faults. In rare occasions, however, these faults can overwhelm the network's fault tolerance capacity and affect the output of the neural network resulting in a misclassification. We can see an example of this in Figure 2.10b.



(a) ANN with no faults. The output class is **1**.

(b) ANN with fault and error propagation. Because of the error, the output class is altered to **2**.

Figure 2.10: Visualisation of the effect of transient faults in ANNs. Darker color means higher values.

Nowadays, most research in applying ANNs to the visual domain utilises DNNs. There has been some research into the characteristics of DNNs that impact their fault tolerance the most [68].

According to [68], one of the most important characteristics is the topology of the network. Special emphasis is given to the type of layer that is used. Normalisation layers [65], for instance, are a type of layer that is used to increase the generalization accuracy of the network by bounding the output of specific layers to a predetermined range. They are often applied before unbounded activation functions (e.g. applying LRN before *ReLU* in AlexNet [65]) in order to prevent the output values of the activation layer from exploding. In addition to their intended purpose, LRN layers also increase the fault tolerance of the network since they normalize faulty values according to their adjacent fault-free values.

Another important characteristic is the type of data that is used in the implementation of the DNN. When a fault occurs in a specific bit, the vulnerability of that bit to Silent Data Corruptions (SDCs) is proportional to the dynamic value range of the data type. This means that data types with higher value ranges have proportionally higher deviations due to faults, which in turn results in a higher probability of there being an SDC. In [68], it is stated that, when possible, it is preferable to choose a data type with an adequate dynamic value range according to the needs of the specific network. An added benefit of using data types with smaller dynamic value ranges is increased energy efficiency, which is a requirement for some safety-critical DNN systems such as those used in the automotive domain.

It has been proven [68] that in an autonomous driving scenario, the DNNs which are used for object detection might output the wrong class for a stop sign, or fail to detect it. This type of mistake can provoke serious safety violations in the form of loss of life or damage to property. Since traditional DNNs have no intrinsic way of detecting and handling this type of situation, it is essential to research possible solutions to this problem.

2.12. Fault tolerance in Artificial Neural Networks

There have been significant efforts to improve the fault tolerance capabilities of ANNs. A diagram with the main categories of current fault tolerance techniques can be seen in Figure 2.11.

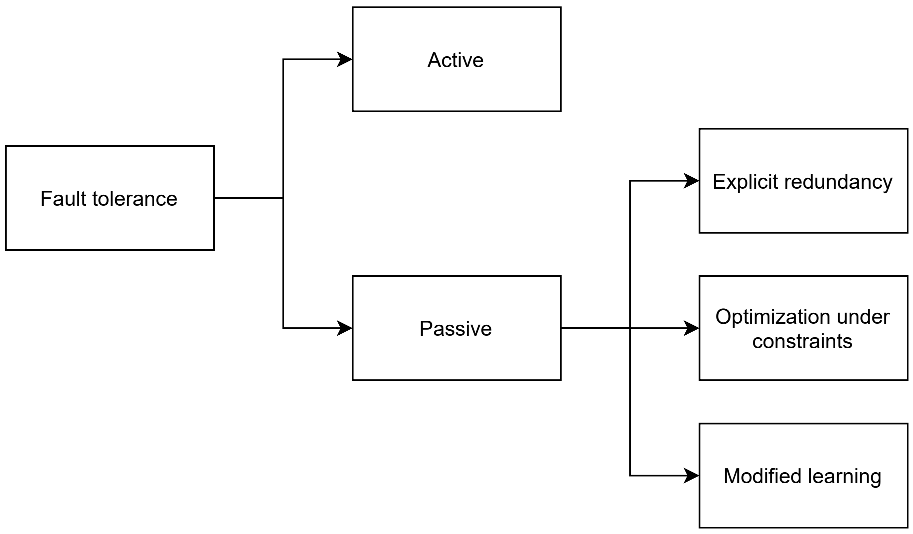


Figure 2.11: Diagram with types of fault tolerance applied to ANNs

There are two generally accepted types of fault tolerance [110]. The first is **active** fault tolerance. Systems that employ this type have two components, one for detecting faults and another for controlling them. The main idea is that these systems detect faults as they appear and handle the effects using mechanisms that compensate for those effects. This compensation is achieved by taking the tasks that were being carried out by the faulty components of the neural network and assigning them to non-defective ones.

The second type is **passive** fault tolerance. In contrast to the previous type, systems that use passive fault tolerance do not directly detect and manage faults. Instead, these systems make architectural changes to the neural network that increase its redundancy, thus indirectly compensating for the fault effects. These changes ensure that the neural network outputs are as expected even when faults occur. In comparison to active fault tolerance, this approach incurs low overhead, since no additional components have to be in place in order for it to work.

There are several categories of passive fault tolerance [110], of which we will mention the three most relevant ones.

The first is **explicitly augmenting redundancy**. The starting point for this method is usually an unmodified ANN which is trained on a given training dataset. After this stage, there is a selection phase that results in the works by inserting redundancy in hidden neurons (those belonging to the hidden layers of the ANN) and their respective weights.

After this stage there is a selection of neurons based on a ranking of their sensitivity. The sensitivity of a neuron can be understood as a measure of the change in the output of the network whenever that neuron is changed. Changing the value of a neuron has an impact on the output of the network that is proportional to that neuron's sensitivity. Highly sensitive neurons are also called critical neurons.

Several techniques are then applied to the neurons with the highest sensitivity. These techniques include spatial redundancy (i.e. duplicating critical neurons) or evenly weight distribution with neuron pruning and removal of unnecessary weights.

The second category is referred to as **optimization under constraints**. In this category, the training process is turned into an optimization problem, where the constraints are the fault tolerance capacity of the model and the model's ability to perform the task that is assigned to it. Optimization algorithms are then used in order to find an ANN architecture and parameters that satisfy these restrictions.

Initially, classical optimization problems such as *minmax* were used [34]. In these cases, conventional methods were used to find the solution. One issue with this was that these methods used up too many resources.

One example of research in this category is using genetic algorithms as an optimization algorithm to adapt the network's architecture, for instance in the work done in [120]. In this case, each network is an individual and the fitness function depends on the global error of the network i.e. the difference between the actual output and the expected output. The authors conclude that the evolved networks had improved fault tolerance and generalisation capacity compared to networks trained conventionally.

The final category is **modified learning**. As the name suggests, alterations are made to the model during training that make it more tolerant to faults in the testing phase. There are two subcategories of modified learning.

The first one is inserting regularization into the learning algorithm. Regularization is defined as any modification that is made to a learning algorithm that intends to reduce overfitting.

The second subcategory is characterised by adding noise, perturbations or fault injections to the learning algorithm during the training phase. The reasoning behind this is that when some of the internal values of the ANN change, other neurons will tend to compensate for this change. This results in a more evenly distributed set of weights, which leads to an improvement in the fault tolerance of the network.

In Section 2.5.5 we discussed the Dropout technique, which belongs to the first type of modified learning. In the following sections we will detail Stimulated Dropout, a technique that fits in the second type of modified learning, and also one recent technique that does not fit within either subcategory, called *Ranger*.

2.12.1. *Ranger*

Constraining the weights of ANNs has proven to be useful for improving their fault tolerance in works such as [114], where the magnitude of the weights in the same layer are bound to a predetermined range.

One recent technique employing this idea is *Ranger* [28]. As previously mentioned, one of the possible consequences of transient faults is a large deviation in a neuron output value. If the network cannot intrinsically handle this deviation, the error can be propagated through the network and possibly affect the output. *Ranger* aims to increase the fault tolerance capability of ANNs by restricting the output values of neurons at certain layers in the network, thus preventing errors that occur before those layers from affecting the following. The selection of the layers where *Ranger* is applied is essential for this technique to have a significant effect. It has been proven empirically [28] that *Ranger* provides the most effect if it is used directly after activation layers.

An example application of *Ranger* can be visualised in Figure 5.10. This is an adapted version of a diagram in the original paper [28]. We are using the example fault propagation given in Figure 2.10b, and demonstrating the effect of *Ranger* when applied to the network between the input and hidden layers.

As a reminder, when faults cause the output value of a neuron to increase dramatically, this error can propagate through the network and affect neurons in subsequent layers. By restricting the range of values that pass from the input layer to the hidden layer, *Ranger* dampened the error that occurred in the input layer and stopped it from being propagated to the following layers. This prevented the network from misclassifying the sample. If *Ranger* was applied between every layer, this protection would extend to errors that occur all throughout the network, not just in the input layer.

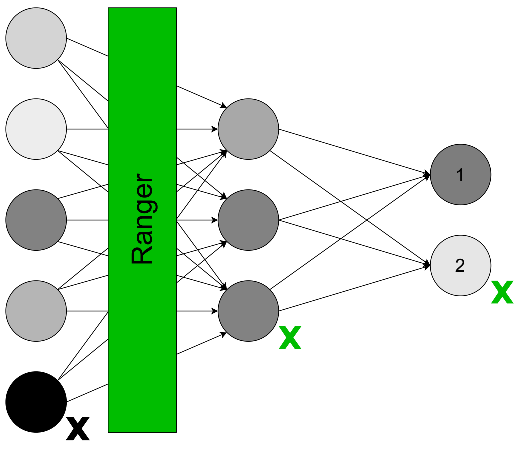


Figure 2.12: Ranger

In order to apply *Ranger*, one first has to derive the restriction bounds. In cases where the activation function of the selected layer is already bounded (e.g. tanh has a range of $[-1, +1]$), those bounds are used. In the cases where the activation function is unbounded (e.g. *ReLu* has range $[0, +\infty]$) the restriction range is obtained by sampling the values of the activation layer when testing the model on *20%* of the training data. Afterwards, the upper bound is obtained by calculating the *99.9%* percentile of that set of values.

2.12.2. Stimulated Dropout

Stimulated Dropout is a novel approach first introduced as part of the project that this thesis is a part of [15]. The implementation of this technique is similar to dropout with one important distinction: instead of being omitted, neurons in the applicable layers have their output values modified arbitrarily. This numerical manipulation consists of a random bit-flip operation that is made on the output value. To help visualise this operation, we have the representation of an example output value in Figure 2.13. Stimulated Dropout is an example of a fault tolerance technique and can be characterised as modified learning by adding noise.

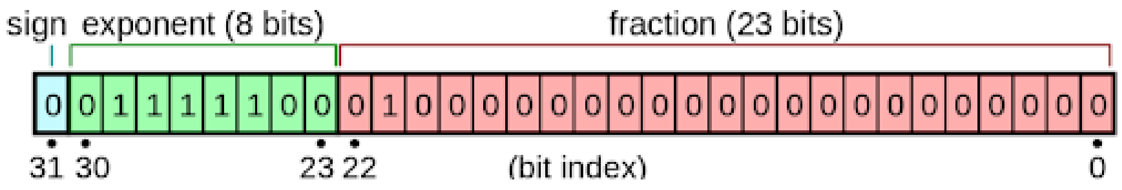


Figure 2.13: 32 bit representation of neuron output value

The internal values of ANNs can have one of several data formats. The most common is the single-precision floating point format with 32 bits from the IEEE 754-2019 standard [52], which is represented in the image. This format can be divided into three parts: the sign which corresponds to one bit, the exponent which has eight bits and the significand which has 23 explicitly stored bits and one implicit leading bit.

Usually the bit-flip operation will choose one of these bits at random and switch its value, thus changing the value of the number that is stored. This manipulation mirrors the random bit-flip fault model mentioned in subsection 2.3.3. The impact of this bit-flip depends on the original number and on the bit that is switched. For instance, if the value of the number is 2 and there is a bit-flip on the 29th most significant bit, the value will increase to 3.689e+19. On the other hand if the value of the number is 0.1 and the same bit is switched, the value decreases to 5.421e-21.

The difference between that fault model and the procedure in Stimulated Dropout is that the bit-flip operation does not affect the sign. This is because it was empirically verified that switching the sign would reduce the accuracy of the network prohibitively, thus invalidating the technique.

This numerical manipulation is meant to recreate the effect that faults would have in the neural network if the chosen neurons were to be affected. By repeating this across a predetermined percentage of neurons, the neural network is trained while taking into account the possibility of these types of errors in the output values of neurons, which in turn leads to an added robustness of the final model. This can also be thought of as adding noise to the neural network, an example of modified learning which was mentioned in Subsection 2.9.

The hypothesis that was being proven was that the regularization capabilities of Stimulated Dropout could be advantageous in improving the resilience of neural networks. This means that since the generalization error reduces when using dropout, this added generalization capability could translate into a more resilient model. The results of the thesis show that there is some decrease in the incidence of soft errors when using stimulated dropout.

This page is intentionally left blank.

Chapter 3

Approach and Methodology

3.1. Motivation and Objectives

In the previous chapter, essential concepts related to Artificial Neural Networks (ANNs) and dependable systems were introduced. The main takeaway from that chapter is that even though faults are uncommon, there is a significant possibility of a failure occurring. Because of this, measures should be taken to prevent such events from affecting systems. As we have previously discussed, this is particularly true in Safety-Critical Systems (SCSs), since these faults can result in safety hazards.

With this issue in mind, the objectives of this work are:

- Implement and validate existing proposals such as Dropout and Stimulated Dropout on additional datasets with varying complexity;
- Implement current methods for improving the fault tolerance of ANNs, such as Redundancy and Ranger;
- Combine current methods into hybrid techniques;
- Perform a critical comparison between all implemented techniques, by injecting faults on the models at inference time and observing the impact in performance metrics.

3.2. Approach

In order to accomplish the goals that were set out, the work was divided into several steps. The guidelines for each step are described in the following sections.

3.2.1. Implement a baseline model on relevant datasets

The first step is to create a Machine Learning (ML) model that can perform image classification on a custom dataset. The model should act as a starting point on which more specific techniques will be added. To achieve this, we used the PyTorch C++ Frontend [96], a ML framework for the C++ programming language.

The code that was used was based on the documentation for this framework, particularly the example that classifies digits from the *MNIST* dataset [97]. In Chapter 4 we detail the additional datasets that were used and their purpose. Since the aim of this project is to improve models intended for safety-critical systems, one of the selected datasets has traffic signals [43], representing the application of these models in the automotive industry.

This step is crucial since this work intends to produce results in a safety-critical context.

3.2.2. Add fault injection during testing

To evaluate the behaviour of the model when faults occur, we resorted to a fault injection tool. For the reasons mentioned in the literature, we opted for a software-based tool. From those available, we selected *ucXception* [111].

Firstly we needed to verify if the Software Fault Injection (SFI) tool had the intended effect on the models. To achieve this, we conceived a testing process that is detailed in Subsection 4.5.2. After validating this process (by checking if the testing accuracy changes after injecting faults), we proceeded to use it in other models and datasets.

This step is mandatory since it will enable us to study the effects of faults.

3.2.3. Dropout and Stimulated Dropout

After setting up a Baseline Convolutional Neural Network (CNN) model and obtaining a way of measuring its fault tolerance, we moved on to implementing current methods of reducing the effects of faults.

The first one was Dropout. This technique was implemented by adding dropout to every layer of the previously mentioned CNN model. The second was Stimulated Dropout, which was similarly applied to each layer. The difference is that instead of ignoring a certain percentage of the neurons in each layer, the output values of the selected neurons are changed by a random amount.

This stage is very important because it will enable us to validate the work done in this project and compare it to other established techniques.

3.2.4. Redundancy

Another type of fault tolerance technique to be analysed results from the application of redundancy techniques to the Baseline CNN model. These techniques are aimed at improving the fault tolerance capacity of the network by altering the number of neurons or feature maps in the fully-connected or convolutional layers, respectively [110].

The implementation of redundancy-based methods in a safety-critical context is mandatory for this work.

3.2.5. Ranger

Ranger [28] is a fault tolerance technique with a good balance between Silent Data Corruption (SDC) coverage and overhead compared to other techniques in the State-of-the-art (SOTA). Since it was originally implemented in TensorFlow using Python, we utilised the descriptions in the paper to produce our own implementation in PyTorch and C++.

Implementing this method was a bonus objective for this thesis.

3.2.6. Creating hybrid methods

After implementing the previously mentioned methods, we considered the most relevant combinations in order to verify if the resulting models have improved fault tolerance.

3.2.7. Perform experiments, compare results and draw conclusions

Finally we injected faults while testing these models in the way described in Section 3.2.2, in order to obtain metrics that allowed us to compare the models. For this comparison we used two types of metrics. The first is related to training, testing and experimentation, and includes:

- Classification Accuracy: number of correctly predicted data points divided by the total number of data points;
- Loss: value calculated by the cost/loss function;
- Time: including loading, training, testing and experiment time.

The second is related to the fault tolerance of the model. The metrics of this type are percentages calculated by dividing the number of times that a specified event happened by the number of fault injections made during a single experiment. The events can be SDC, Hang/Crash or No Effect:

- Silent Data Corruption: when the testing accuracy is lower than expected, we can say that the fault injection caused an SDC, also known as soft error;
- Hang/Crash: when the testing freezes or crashes;
- No Effect: when there is no discernible effect in the testing accuracy.

This is the most important step of this work and is highly dependant on the other stages for it to succeed.

This page is intentionally left blank.

Chapter 4

Experimental Setup

4.1. Environment

In order to study the effects of faults in neural networks, we need to define the conditions on which our experiments will be conducted.

4.1.1. Programming Language

The first consideration was the choice of the programming language. There were two categories of languages that were considered. The first is interpreted languages, that is, languages that are executed without previous compilation. Examples of this type include Lua, JavaScript and Python. The second is compiled languages, which are explicitly processed by a compiler. Examples are C, C++ or Rust.

A recent study on error resilience [27] compared different programming languages in regards to the amount and type of failures that occur when faults are injected. These failures are categorised as No Effect, Silent Data Corruption (SDC) or Crash/Hang. According to [27], interpreted languages are more likely to encounter failures compared to compiled languages. However, these failures are less likely to become SDCs. In contrast, compiled languages have an inferior amount of failures but show a higher percentage of SDCs.

When choosing a programming language, one has to take into account the dependability requirements of the system. If the availability of a system is of paramount importance, compiled languages are preferable since they lead to fewer failures (therefore the system will be available more often) [27]. If the preciseness of the calculations is a bigger concern, then interpreted languages should be chosen due to the fact that they are less likely to cause SDCs.

In Safety-Critical Systems (SCSs) there is often a need to balance these two characteristics. For example, an autonomous driving system should continue to function despite faults occurring, but it should also perform exact calculations so as not to cause an accident due to misclassifications.

In addition, there are differences in speed and resource efficiency between Programming Languages (PLs), with interpreted languages often being slower than their compiled counterparts. This is particularly troublesome in SCSs, since many of these systems require efficient methods due to limited resources (e.g. energy or hardware).

Taking these requirements into account and since we are concerned with the impact of SDCs on Artificial Neural Networks (ANNs), we selected the C++ [25] compiled programming language.

4.1.2. PyTorch

One additional specification is the choice of a Machine Learning (ML) framework that allows for the design, development and manipulation of models with low restrictions and impediments. The framework must also have a C++ implementation, since this is the programming language that was selected. Finally, the framework should preferably be widely used in research to produce relevant implementations that can be readily understood and expanded.

As of August 2021, PyTorch [96] is the most used ML framework in academia [6]. It also provides a C++ API, which is a version of the original PyTorch Python frontend that allows users to design and build models in C++ while sharing similar component structure and nomenclature. Since this framework meets all our requirements, we selected PyTorch.

4.1.3. *ucXception*

Faults in Deep Neural Networks (DNNs) can occur randomly [90]. Because of this, to consistently study their effects, it is necessary to inject faults in the neural network models. For accessibility, only software-based fault injection methods [110] were considered. From those methods, the selected tool should be able to emulate transient hardware faults that happen during runtime. With that purpose, the fault injection framework *ucXception* was used.

This framework causes faults by altering the CPU registers of the computer while a process is running. If these registers are being accessed by the process, this modification can cause deviations in the internal values that the process is using. In this context, the internal values of the neural network that is executing can be altered, which can possibly lead to a misclassification. Since the process needs to access the CPU registers for the fault injection to be successful, the neural networks had to be executed on the CPU.

The faults are also injected exclusively during the testing of the model. The model classifies every image in the test dataset and at the end of each iteration the outcome can be one out of three possibilities: No Effect, Hang/Crash or SDC. This last possibility is returned when the accuracy of the model is lower than the expected test accuracy. By storing these outcomes, we can measure the fault tolerance of the model.

The emulation of soft errors by this tool is achieved by doing bit-flips in a specified bit in a given CPU register. This replicates what happens when an actual hardware fault occurs, while not needing any additional equipment.

Faults are much more likely to occur in testing than in training [90]. This is due to the fact that testing is often not done in a controlled environment. Moreover, the point of this work is to study the effects of faults in real-life scenarios where there are consequences to errors in the DNN. Because of this, the fault injection capabilities of *ucXception* are utilised during testing.

4.1.4. Relevant libraries

Additional libraries were utilised for various reasons.

CMake [31] is a set of tools designed to manage the building, testing and packaging of software in a compiler-independent way. In our project, it was used to compile the *C++* source files.

NumPy [5] is a *Python* library that allows for the manipulation of multi-dimensional arrays as well as applying mathematical functions. In this project it is used in general purpose array manipulation, and also in more specific applications such as calculating the percentile for our implementation of Ranger.

We use the *Matplotlib* [78] and *Seaborn* [9] *Python* libraries for creating graphical representations of data such as stacked or grouped plots.

OpenCV [89] is used in *Python* and *C++* for image processing, such as reading images, manipulating color channels or altering dimensions.

4.2. Datasets

Several datasets were considered to train and validate the model. We used datasets with varying complexity in order to verify if the results could be generalised to different use cases.

4.2.1. *MNIST*

MNIST [86] is a collection of handwritten digits, divided into 60000 training and 10000 testing samples.

The images are grayscale, they have dimensions of 28x28 pixels, and there are 10 classes which correspond to a digit from 0 to 9. These numbers are also size-normalized and centered, such that minimal pre-processing is required. The images have an *IDX* file format which was read using existing functions from the PyTorch framework.

MNIST is also commonly used in research papers, including some that study fault tolerance in neural networks [15, 28]; so it is a relevant benchmark that can be used to compare our techniques with the State-of-the-art (SOTA). A sample image from the dataset can be seen in Figure 4.1, and the training and testing data frequency distributions are shown in Figure 4.2.

4.2.2. Fashion-MNIST

Fashion-MNIST [12] is a dataset that consists of Zalando article images of clothing and shoes. It was developed with the purpose of being used as a more complex alternative to *MNIST*. Because of this, it has the same image dimensions, number of classes and structure (number of training and testing samples) as the *MNIST* dataset. Additionally, it is also stored in the same file format, which means we can use the same functions we used earlier to read the files.

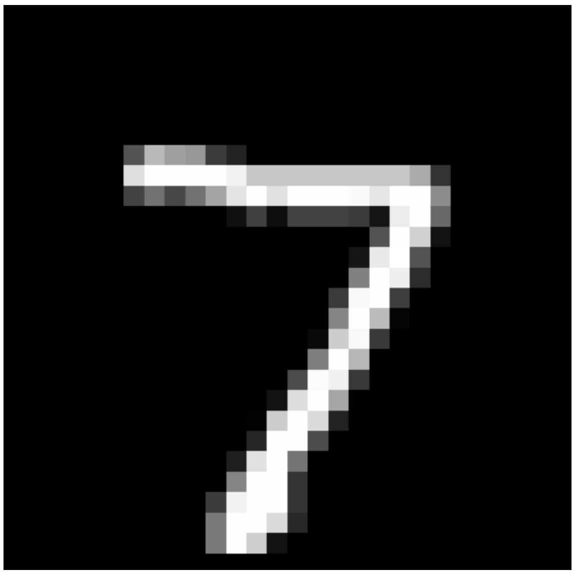
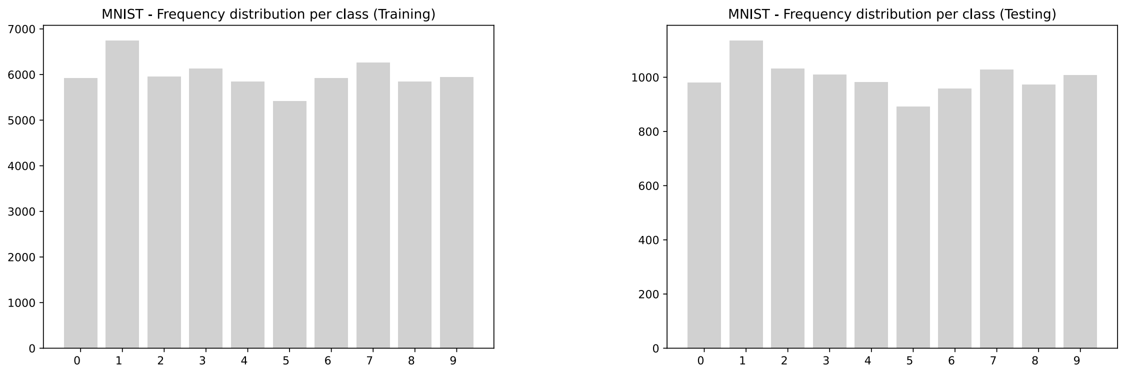


Figure 4.1: Example 28x28 image with label "7" from the MNIST dataset



(a) Frequency distribution of MNIST (Training)

(b) Frequency distribution of MNIST (Testing)

Figure 4.2: Frequency distribution for Training and Testing sets of MNIST

An example image for this dataset is shown in Figure 4.3. In the frequency distribution, shown in Figure 4.4, we can see that all classes are balanced (have the same number of samples), in contrast to the *MNIST* dataset (Figure 4.2).

4.2.3. GTSRB

The German Traffic Sign Recognition Benchmark (GTSRB) [44] dataset contains photographs of traffic signals (example in Figure 4.5). It has 51839 images divided into 43 classes, with each image having 3 channels (Red, Green and Blue (RGB)). The dimensions of the images range from 15x15 to 250x250. Within each class, the images differ in the angle, brightness and other characteristics, in order to provide some variation and encompass plenty of possible real-life circumstances. Since the images did not have a standard size, we resized them using the OpenCV library. The final dimensions were 28x28, in order to preserve the most features possible while not increasing the training time prohibitively.

The motivation for choosing this dataset came from the need to verify if the results that were obtained with the *MNIST* dataset could be replicated in a dataset that represents

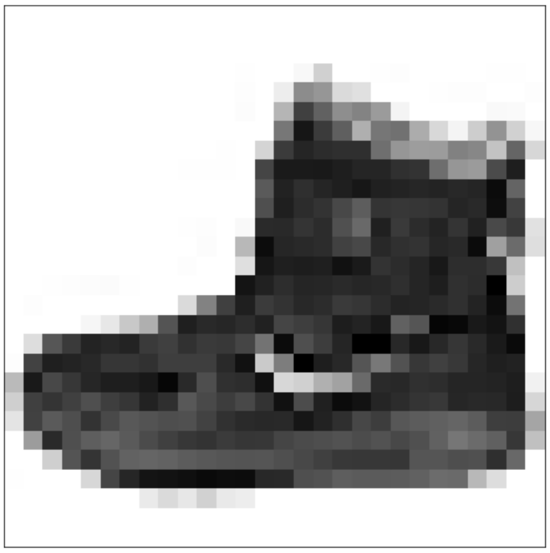
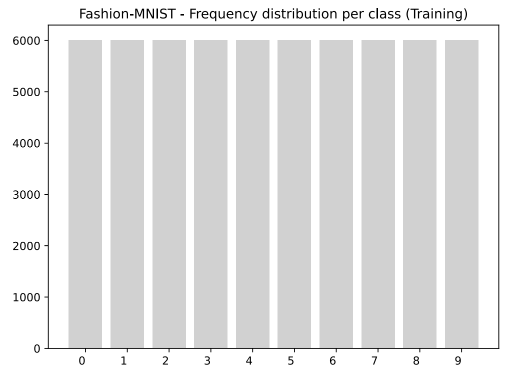
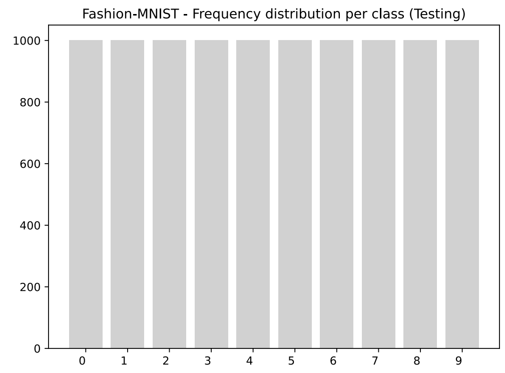


Figure 4.3: Example 28x28 image with label "Ankle boot" from the Fashion-MNIST dataset



(a) Frequency distribution of Fashion-MNIST (Training)



(b) Frequency distribution of Fashion-MNIST (Testing)

Figure 4.4: Frequency distribution for Training and Testing sets of Fashion-MNIST

the real-life usage of the model in a safety-critical context, in this case, in autonomous driving.

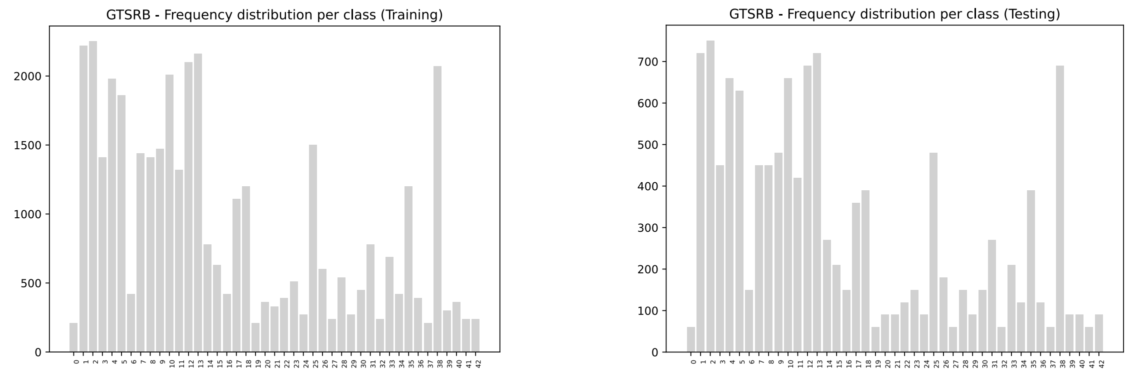
We created a text file with the paths to the images, named *info_traffic.txt*. This file contains the path, label and dataset (training or testing) of every image, and was generated with Python. It is used to load the dataset in C++.

The class distribution for the GTSRB can be seen in Figure 4.6.

4.2.4. CIFAR-10

The CIFAR-10 [1] dataset is composed of a subset of the *80 million tiny images* dataset. There are 50000 training images and 10000 testing images, which are labelled as one of 10 classes. The classes range from animals (e.g. bird, cat, ...) to means of transport (e.g. airplane, truck, ...). The images have dimensions of 32x32 and 3 channels (RGB). An example image can be seen in Figure 4.7 and the class distribution is shown in Figure 4.8.

Figure 4.5: Example 28x28 image of a traffic signal with label "21" from the GTSRB dataset



(a) Frequency distribution of GTSRB (Training)

(b) Frequency distribution of GTSRB (Testing)

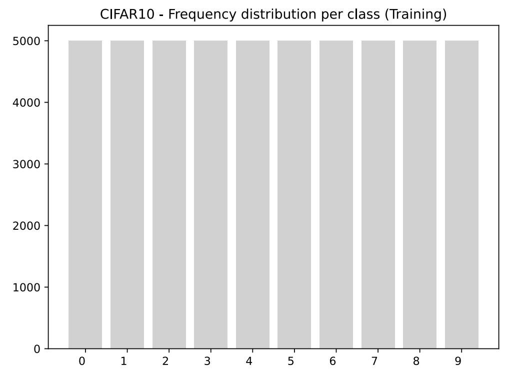
Figure 4.6: Frequency distribution for Training and Testing sets of GTSRB

4.3. Model architectures

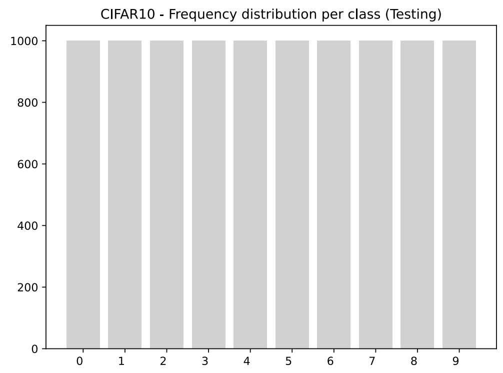
First we need to select a network to serve as a baseline, so that we can add the techniques and see their impact on several metrics. The context for this work is the study of fault tolerance in Convolutional Neural Networks (CNNs), and as such the primary requirement is that this network must have convolutional layers.

In the following subsections we will specify the neural network architectures that were used for the experiments.

Figure 4.7: Example 32x32 image with label "Horse" from the CIFAR-10 dataset



(a) Frequency distribution of CIFAR10 (Training)



(b) Frequency distribution of CIFAR10 (Testing)

Figure 4.8: Frequency distribution for Training and Testing sets of CIFAR10

4.3.1. LeNet

LeNet or LeNet-5 [67] is a CNN architecture that is widely used in academia. It is composed of convolutional, pooling and fully-connected layers. Even though it was initially developed for the *MNIST* dataset, it can also be used in other datasets.

Since LeNet is simple and easily replicable, we can use it as a baseline on which to add layers and techniques, allowing us to evaluate their impact on the fault-tolerance of the network in an unbiased way. Additionally, some relevant works in the literature already use LeNet to evaluate fault tolerance [15, 28] which makes comparisons between existing techniques easier.

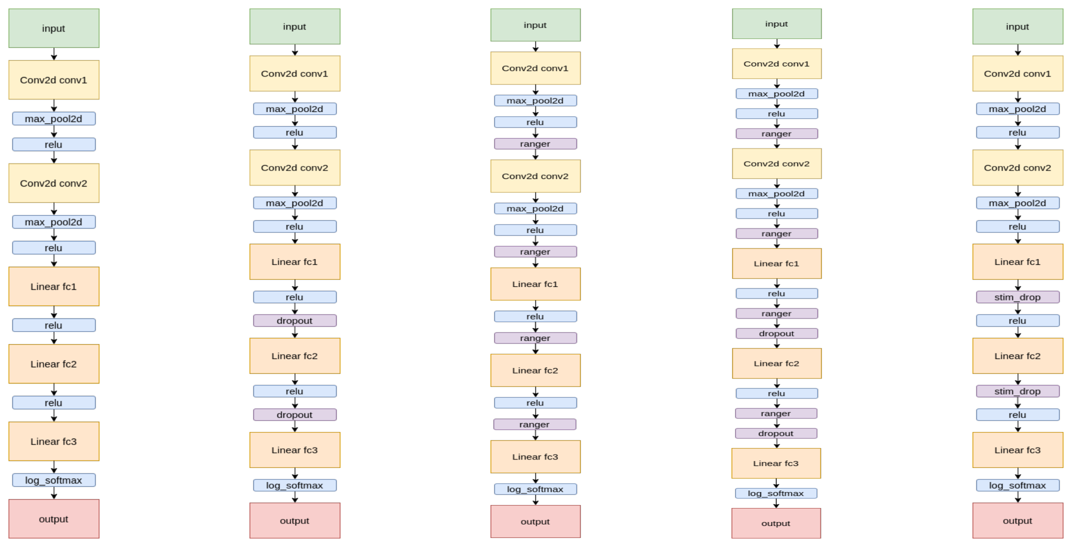
A diagram for the baseline network can be seen in Figure 4.9a. Details of this and other networks that will be mentioned, including the size of the convolutional layers and the number of neurons on the fully-connected layers, can be found in the *network.h* file of the project repository.

We implemented the original LeNet architecture for all datasets. This architecture has a few differences depending on the dataset, namely: the input size (32x32 for *CIFAR10* and 28x28 for the remaining datasets); the number of input neurons of the first fully-connected layer (400 for *CIFAR10* and 256 for other datasets); the number of channels (one for *MNIST* and *Fashion-MNIST*, 3 for *GTSRB* and *CIFAR10*). We identify this architecture as (1). A diagram for architecture (1) can be seen in Figure 4.9a.

Additional techniques were applied to the network, such as Dropout after the *ReLU* activation function in the Fully-Connected (FC) layers, as can be seen in Figure 4.9b. Ranger was applied after every *ReLU* function, including in convolutional layers.

Figure 4.9d shows a combination of the previous techniques, by placing Ranger directly after *ReLU* in layers *conv1* through *fc2*, and applying Dropout to the output of the Ranger layers in *fc1* and *fc2*.

Finally, the architecture for LeNet with Stimulated Dropout is represented in Figure 4.9e. In this case, the technique is applied before the *ReLU* activation function, since this was the order in the original work [15].



(a) LeNet (b) LeNet with Dropout (c) LeNet with Ranger (d) LeNet with Ranger and Dropout (e) LeNet with Stimulated Dropout

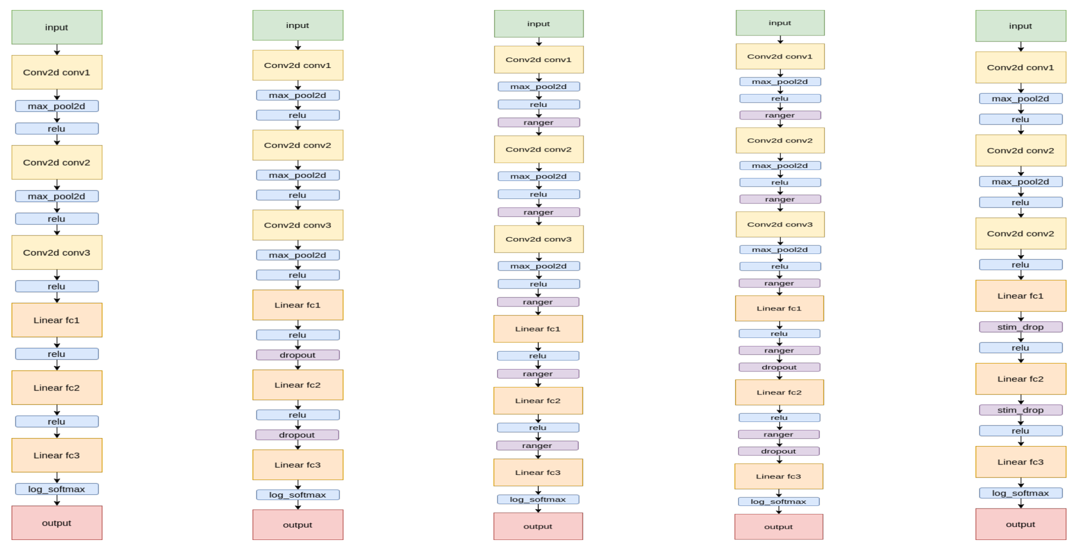
Figure 4.9: Diagrams for LeNet networks (1)

In order to evaluate the impact of redundancy on LeNet when combined with the other techniques that were proposed, we altered some properties of architecture (1). We added one convolutional layer (*conv3*) and increased the input and output dimensions of the convolutional layers, as well as the number of neurons in the fully-connected layers. We called the resulting network architecture (2), and used it in every dataset as a baseline in a similar way to architecture (1). We intended to evaluate the changes in fault tolerance when more layers and neurons were added to the network.

As an additional technique, we changed the number of neurons, but only in the fully-connected layers, with the purpose of verifying if doubling or halving the number of neurons would have an effect on the fault tolerance. This was only done in the *MNIST* dataset, because it did not provide satisfactory results.

Every technique that was applied to architecture (1) was also applied to architecture (2). However, Stimulated Dropout was not used in experiments. This is because the training time for this method increases greatly with larger networks, with makes it impossible to train this network configuration for every dataset due to time constraints.

Diagrams for networks with architecture (2) can be seen in Figure 4.10.



(a) LeNet

(b) LeNet with Dropout

(c) LeNet with Ranger

(d) LeNet with Ranger and Dropout

(e) LeNet with Stimulated Dropout

Figure 4.10: Diagrams for LeNet networks (2)

4.4. Implementation

In this section we describe the implementation process for all models.

4.4.1. Directory tree

The project directory is structured as shown in Figure 4.11.

The source files for training and testing each model (represented here as *sourcefiles.cpp*) are located in the main directory.

The build folder contains the model executables, which are generated after compiling the project. It also includes folders for storing all datasets, models and results, organised in subfolders according to the dataset.

script.sh and *script2.sh* are also in the build folder, and they are executed from this working directory during the experiments.

The network folder contains files that describe the network and dataset classes as well as changeable parameters. *cifar.h* and *cifar.cpp* are files that define the *CIFAR10* dataset class and necessary loading functions.

network.h has class implementations for every network, including shared parameters for every dataset such as number of channels, number of output classes, Dropout probability and Stimulated Dropout probability. It also contains the code implementation for Stimulated

```
thesis
├─ build
│  ├─ datasets
│  │  ├─ CIFAR10
│  │  ├─ Fashion-MNIST
│  │  ├─ GTSRB
│  │  └─ MNIST
│  ├─ models
│  │  ├─ CIFAR10
│  │  ├─ Fashion-MNIST
│  │  ├─ GTSRB
│  │  └─ MNIST
│  ├─ results
│  │  ├─ CIFAR10
│  │  ├─ Fashion-MNIST
│  │  ├─ GTSRB
│  │  └─ MNIST
│  ├─ script.sh
│  ├─ script2.sh
│  └─ modelexecutables
├─ network
│  ├─ cifar10.h
│  ├─ cifar10.cpp
│  ├─ network.h
│  └─ options.h
├─ utils
│  ├─ graphs.py
│  ├─ resizing.py
│  └─ utils.py
└─ sourcefiles.cpp
```

Figure 4.11: Directory tree for the project repository

Dropout and Ranger.

options.h has the *Options* class and Options objects for every model. This class has important parameters for training and testing, as we can see in Figure 4.12.

Finally, the utils folder has several utility files in Python, which have diverse functions from drawing and saving graphs (*graphs.py*) to pre-processing images (*resizing.py*) and general utilities such as reading metrics, generating text files with information about the datasets, or the sampling function used by Ranger (*utils.py*).

4.4.2. Baseline and techniques

The baseline model for architecture (1) was adapted from the code available in the PyTorch documentation [97]. This was the basis for every other model, and as such, any alteration or additional technique was added on top of it.

The project was structured in a modular way, for the sake of simplicity and to allow further improvements or expansions. Making alterations to the baseline model, such as adding layers, changing the number of neurons, or adding methods between layers was facilitated since all networks are in the *network.h* file.

The procedure to create a new model involved duplicating the source code, network class and options object of the baseline model, and then refactoring the code (e.g. changing the path of the dataset or changing the name of the results file). Since we are using CMake, the model executable and source code file were included in the *CMakeLists.txt* file before compilation.

```
static Options options_MNIST = {
    28,  // image_size
    10, // num_classes
    1,  //num_channels
    64,  // train_batch_size
    200,  // test_batch_size
    60000, // train_size
    10000, // test_size
    10,  // iterations
    20,  // log_interval
    "./build/datasets/MNIST/raw", //dataset_path
    "build/info_mnist.txt", //info_file_path
    torch::kCUDA, // train_device
    torch::kCPU, // test_device
    "build/models/MNIST/modelMNIST.pt", //model_path
    "build/results/MNIST/resultsMNIST.txt", //results_path
};
```

Figure 4.12: Options class

The only technique that had to be implemented from scratch was Ranger. Redundancy was achieved by modifying properties of the network (e.g. number of neurons, number of layers), and Dropout already has an existing function in PyTorch [2]. The code for Stimulated Dropout was available in the project repository and just needed to be placed in the *network.h* file and adapted to our networks and datasets.

4.4.3. Ranger

In the original paper, Ranger [28] is implemented using TensorFlow in Python. In order to use this technique, it was necessary to implement it in PyTorch and C++ based on the descriptions in the paper. Although we attempted to recreate the original technique faithfully, there are some differences, namely the fact that the upper bound needs to be calculated manually using an utility function instead of automatically.

A diagram for the process of applying Ranger can be seen in Figure 4.18.

We start this process by testing the model on randomly selected 20% of the training dataset (1). Every time the model is used to predict the label of a batch of images, the forward function in that model's class (defined in *network.h*) is called. This means that the input tensor will be passed forward through the network.

Then, we use a function to save the output values of a predetermined *ReLU* activation function and append those values to a Tensor vector (2). Only one *ReLU* can be selected at the same time. For demonstration purposes, we decided to use the *ReLU* function that is applied after the first fully-connected layer, *fc1*.

When the testing is finished, the Tensor vector has the output values of the selected *ReLU* activation function for every batch. After testing, this vector is saved in the *pt* file format (3) using the *torch::save* function.

The next step is to read the vector in Python (4), so that we can calculate the 99% percentile. This is done using the *tensor_sampling* function, which uses PyTorch's *JIT* library to read *vector.pt*. After loading the values into a NumPy array, we use that library's percentile function to calculate the 99% percentile.

We then use a function that is native to PyTorch, *torch::clamp*, in order to restrict the output values of the selected *ReLU* activation function to a given range (5). Since we are dealing with *ReLU*, the lower bound of that range is 0, and the upper bound is the percentile value calculated in (4).

Finally, we repeat the process from (1) to (5) for every *ReLU* function in the network (6).

4.5. Training and testing

4.5.1. Training and testing

In order to maintain a stable and unbiased environment, the models were trained and tested on a single machine with the following technical specifications:

- *Operating System*: *Ubuntu* 20.04.1 LTS
- *Processor*: *AMD Ryzen* 7 3700X 8-Core Processor
- *Random-access memory*: 16GB DDR4 System memory
- *Graphics card*: *Nvidia GeForce RTX* 2060
- *CUDA*: Version 11.2.67

This machine has installations for two PyTorch versions, 1.10 (Nightly) and 1.4 [8]. Version 1.4 is only used for models with Stimulated Dropout, since it is the most recent version that is compatible with the code implementation of that technique.

GPU acceleration (CUDA) was used in order to speed up training and testing in all models, with the exception of those that use Stimulated Dropout. This is due to incompatibilities between the drivers used by the graphics card and version 1.4 of PyTorch. As a result, models that use Stimulated Dropout had to be trained and tested using the CPU.

4.5.2. Experimenting with fault injection

Hardware faults are rare and arbitrary. To reliably study their impact on systems, researchers have developed software-based fault injection programs or methods. We used *ucXception* [112] to inject faults and developed an experimental process so that we can check the effect those faults have on ANN-based ML models.

A diagram for the experimental process can be seen in Figure 4.19. We use *testModel* as a placeholder for the name of the actual model executable we choose (e.g. testMNISTDropout will test a model with Dropout on the MNIST dataset). The names of all executables are available in the project repository.

We developed two bash scripts for this process. The first, *script.sh*, runs the model executable (*testModel*) while the runtime is less than 60 seconds (1). The model executable will load the testing data and test the model on that data continuously (2). This script resumes execution of the model executable whenever that process crashes or freezes. In the diagram, each rectangle with the label *test* represents one test run, i.e. finished testing on the entire testing dataset. The testing accuracy is calculated at the end of each run and is written in a text file (3). This text file will later be used to calculate fault tolerance metrics.

The next step was to inject faults during the testing. Firstly we cloned the *ucXception* repository (https://github.com/ucx-code/ucXception) and followed the configurations in the *README* file. After setting up the continuous model testing, we run *script2.sh.* his script uses *ucXception* to inject a transient hardware fault into the *testModel* executable. It achieves this by searching for the testing executable by its name, retrieving its process id and performing fault injection on that process id. By default, this is done five times per bit, 64 bits per register, for the 27 available registers (8640 times in total). The fault injection is repeated 5 times on the same bit to make sure that this injection actually happens.

If this fault causes *testModel* to crash, we leave the *while* loop in (1) and write *SEGMENTATION FAULT* in the results file (5). Similarly, if the fault injection results in a hang (*testModel* freezes), we write *HANG* in the results file (6). In both cases, after dealing with the error, *script.sh* is restarted (7).

An additional Python function (*count_tolerance_metrics* in *utils.py*), was used for measuring the results of these fault injections. This function increments a counter every time it measures the outcome of a bit and returns the relative frequency of each outcome for the entire experiment.

This measurement is done by reading the results file line by line and assessing the outcome of each bit-flip. It can be one of the following:

- Silent Data Corruption, if at least one of the testing accuracies of that bit-flip is lower than the true testing accuracy of the model;
- Hang/Crash, if *HANG* or *SEGMENTATIONFAULT* is written at least once and none of the testing accuracies is lower than the true testing accuracy;
- No Effect: if all testing accuracies are equal to the true testing accuracy of the model.

The outcome of each bit-flip is then grouped for each bit. By default, 5 bit-flips are done per bit, so we join 5 outcomes to evaluate the outcome of a bit. This outcome can be one the following:

- Silent Data Corruption, if at least one of the bit-flip outcomes is an SDC;
- Hang/Crash, if at least one of the bit-flip outcomes is Hang/Crash and none is SDC;
- No Effect: if all bit-flip outcomes are No Effect.

An example outcome for a bit can be seen in Figure 4.13. The true testing accuracy in this example is 0.94. The outcomes of the first and fourth bit-flips are SDCs, while the third is Hang/Crash, and the second and fifth are No Effect. By grouping these outcomes, we conclude that the outcome of bit 9 of register 14 is an SDC.

```
...
14---9
0.72
0.94
0.94
14---9
0.94
0.94
0.94
14---9
HANG
14---9
0.68
0.68
14---9
0.94
0.94
...
```

Figure 4.13: The outcome for bit 9 of register 14 is SDC

Initially, we intended to perform the fault injection experiments in parallel on different machines, in order to maximise the amount of experiments done in a given time period. We found that the results differed depending on the machine that was used, including with machines that shared the same *PyTorch* and *Linux* version. Because of this, tests were executed on a single computer.

The scripts are executed in the background by using the *nohup* command [4]. This allowed us to keep the scripts running even after closing the terminal.

script.sh is executed with the following command line arguments:

- $1: *Name of the executable file*;
- $2: *Path to the results text file*, in which the testing accuracy will be written whenever a test run ends.

script2.sh has the following command line arguments:

- $1, $2, $3: *Number of register, bit and repetition* from which to start injecting faults. The register values are limited from 0 to 26 and the bit values are limited from 0 to 63. For instance, if these numbers are *2 24 3*, the injection will start at register no. 2, bit no. 24 and at the third repetition of the fault injection
- $4: *Name of the executable file*
- $5: *Path to the results text file*
- $6: *Accuracy of the model* that is being tested
- $7: *Maximum number of fault injections per bit*
- $8: *Loading time of the model* (ms)

- $9: *Testing time of the model* (ms)

The interval between fault injections is calculated using the loading and testing times of the model (parameters $8 and $9). The reasoning behind this calculation is that there should be a minimum number of test runs before the fault injection, in order to make sure that the testing accuracy of at least one test run is printed on the results file.

The calculation is done by generating a random integer number between 1 and 3, then multiplying that value by the time it takes the model to do a single test run, and finally adding the resulting number to the time it takes to load the dataset. The section of *script2.sh* where this calculation is performed can be seen in Figure 4.14.

```
...
r=$((( RANDOM % 3)+1))
interval=$(loading_time +  testing_time * r)
...
```

Figure 4.14: Section of *script2.sh* where the interval value is calculated (milliseconds)

The experiments had to be done separately, since it was observed during the experiments that the testing metrics differed when more than one test was being done at the same time. This is because *ptrace*, the Linux command that is used by *ucXception* to stop the execution of the process and perform the bit-flip, can only be used in one process at a time [7]. Additionally, more tests have a significant impact on the performance of the computer, which can lead to an increase in inference time. This leads to not only increased testing time, but also an inability to perform fault injection during testing in certain circumstances, since the model has to finish a test run (testing on the full dataset) before the accuracy of that run is written on the results file. In other words, if the testing takes too long, the fault injector will move on to the next bit while the previous has not output its testing accuracy. An example of this situation can be seen in Figure 4.15.

```
...
2---51
0.9888
2---51
0.9888
0.9888
2---51
2---52
0.9888
0.9888
2---52
0.9888
...
```

Figure 4.15: Testing for the last bit-flip in bit 51 of register 2 takes too long

To start an experiment, we first need to run the commands in Figure 4.16. These instructions will enter root mode and allow the fault injector to attach to the target process. Afterwards, *script.sh* is run in the same terminal. This script continuously tests the model on a specific testing dataset, and writes the testing accuracy in a text file with results.

Whenever the model crashes or hangs, the script re-runs it, allowing the testing to continue.

```
sudo -i
echo 0 > /proc/sys/kernel/yama/ptrace_scope
exit
./script.sh testMNIST Results/MNIST/resultsMNIST.txt
```

Figure 4.16: Commands to setup *ucXception* and run *script.sh*

Finally, *script2.sh* is executed in a separate terminal. An example execution can be visualised in Listing 4.17.

```
./script2.sh 0 0 0 testMNIST Results/MNIST/resultsMNIST.txt 0.989
5 80 850
```

Figure 4.17: Example command to run *script2.sh*

This script will continuously inject faults on the *testMNIST* process, and will write the register and bit of that injection on the *resultsMNIST.txt* file.

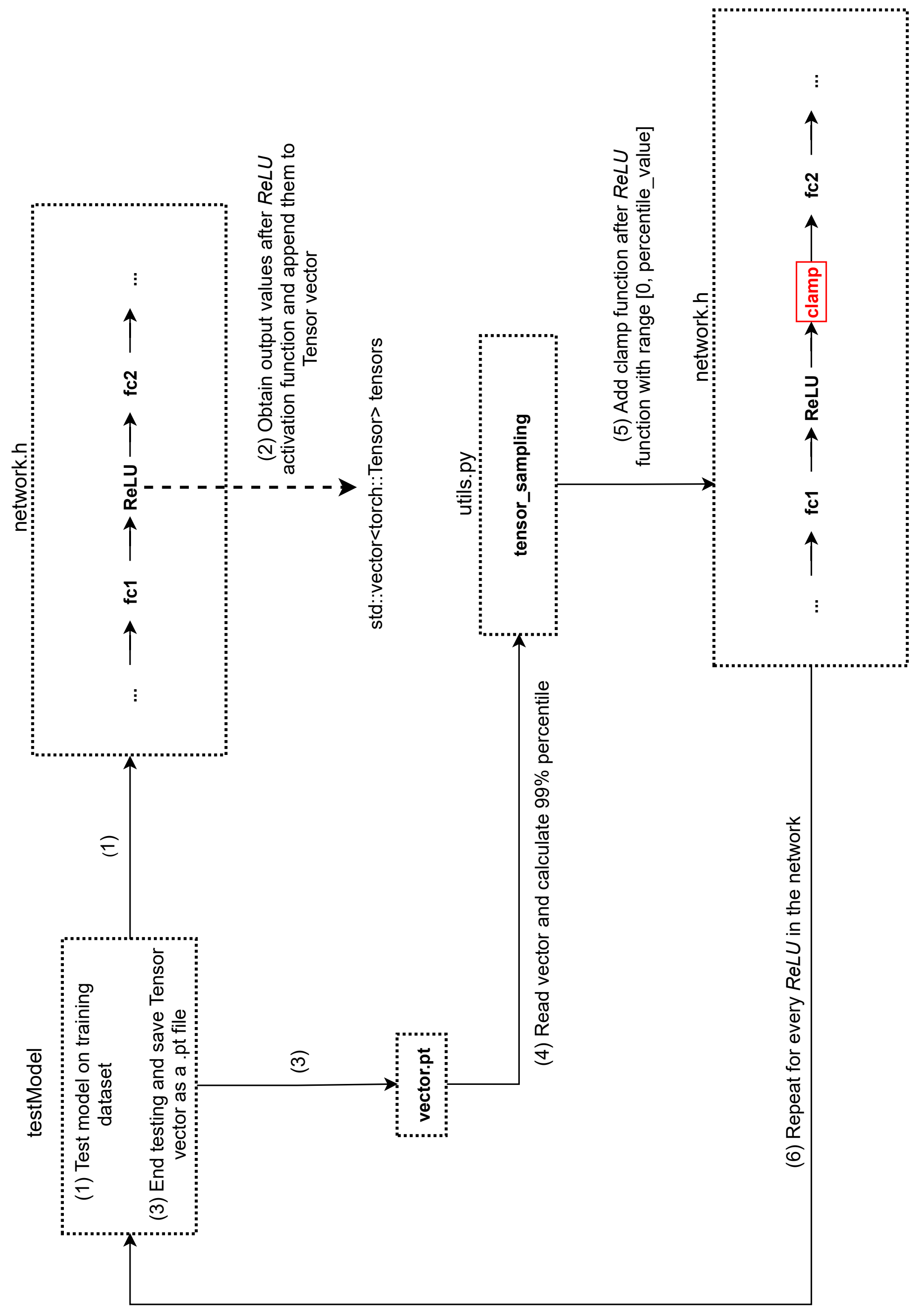

testModel
(1) Test model on training dataset
(3) End testing and save Tensor vector as a .pt file
(1)
network.h
... → fc1 → ReLU → fc2 → ...
(2) Obtain output values after ReLU activation function and append them to Tensor vector
std::vector<torch::Tensor> tensors
(3)
vector.pt
(4) Read vector and calculate 99% percentile
utils.py
tensor_sampling
(5) Add clamp function after ReLU function with range [0, percentile_value]
network.h
... → fc1 → ReLU → clamp → fc2 → ...
(6) Repeat for every ReLU in the network

Figure 4.18: Diagram with application of Ranger

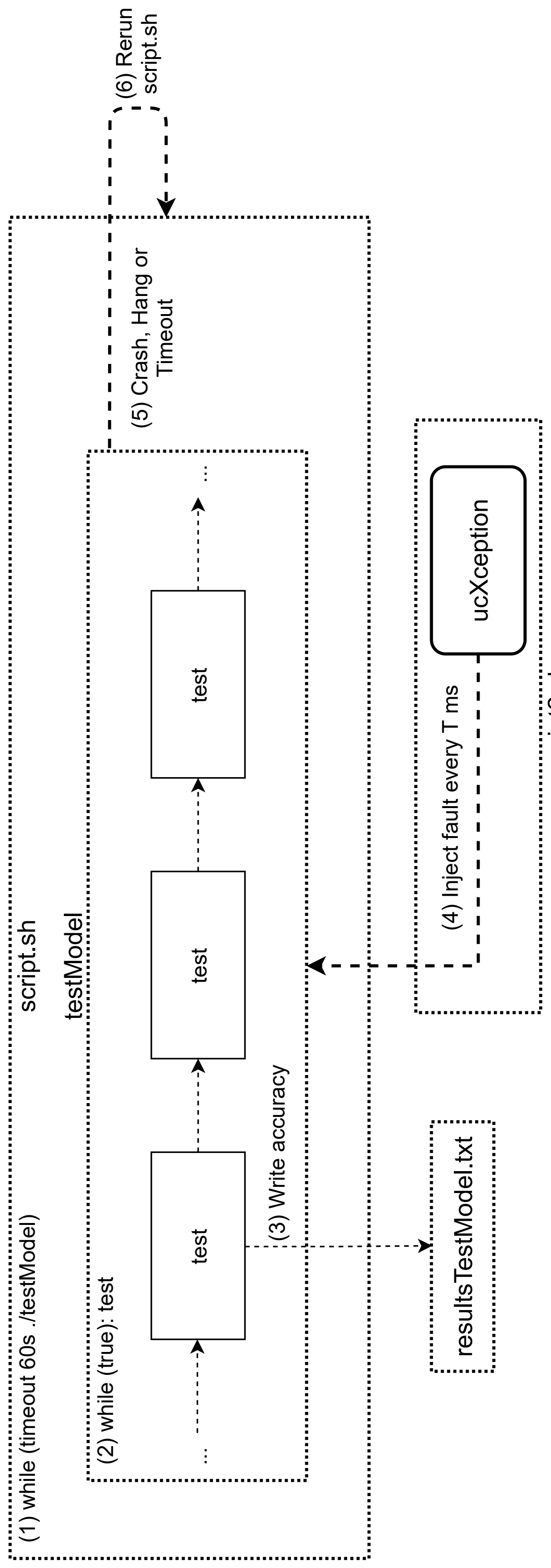


Figure 4.19: Diagram of fault injection experiment

This page is intentionally left blank.

Chapter 5

Results and Discussion

In this Chapter we will present and analyse the results of the experiments, evaluate the performance of the models and compare them according to several metrics.

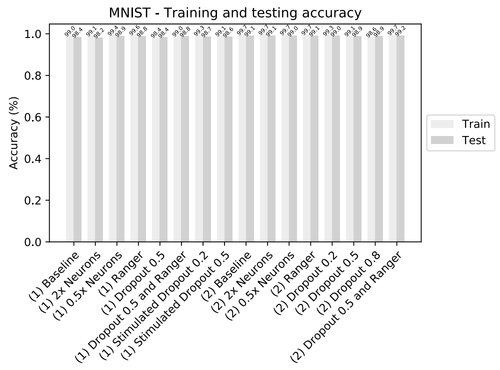
5.1. Training and testing metrics

After figuring out the architecture for the neural networks, we trained and tested them in the mentioned datasets. In order to evaluate the model performance, we calculated metrics during training and testing. These metrics are described as follows:

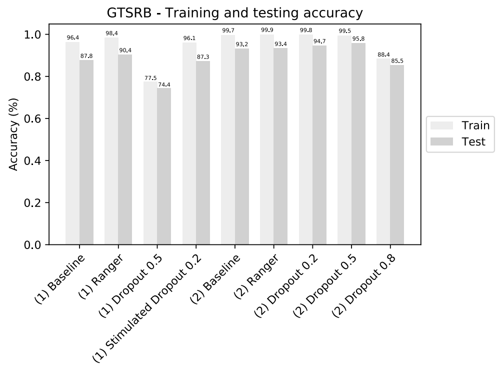
- Classification Accuracy: percentage of correct predictions for some given data;
- Loss: measure of how bad a model is. We use negative log likelihood loss;
- Loading time: time in milliseconds it takes to load a given dataset;
- Training/testing time: time in seconds/milliseconds it takes to train/test some model on a given dataset;
- Experiment time: time in hours it takes to perform an experiment.

5.1.1. Classification accuracy

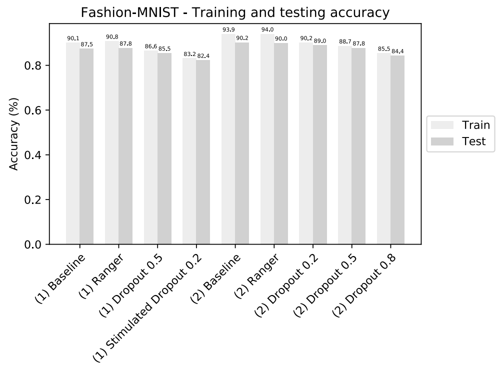
We obtained the accuracy values for the different models, subsets (training and testing) and datasets, as we can see in Figure 5.1.



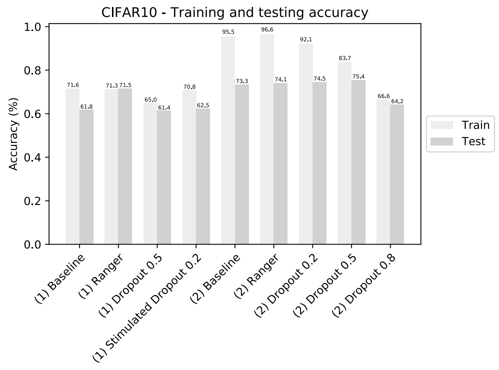
(a) Training and testing accuracy of each model for the MNIST dataset



(b) Training and testing accuracy of each model for the GTSRB dataset



(c) Training and testing accuracy of each model for the Fashion-MNIST dataset

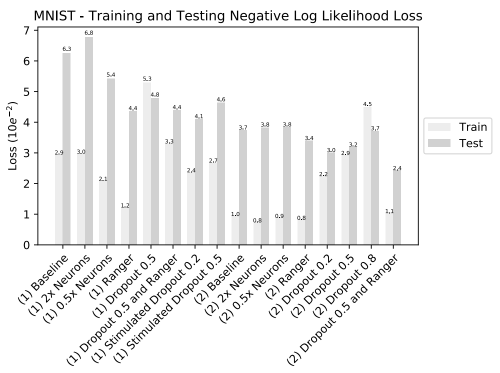


(d) Training and testing accuracy of each model for the CIFAR10 dataset

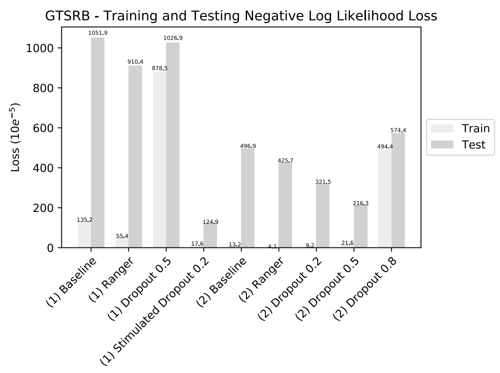
Figure 5.1: Training and testing accuracy per dataset

5.1.2. Loss

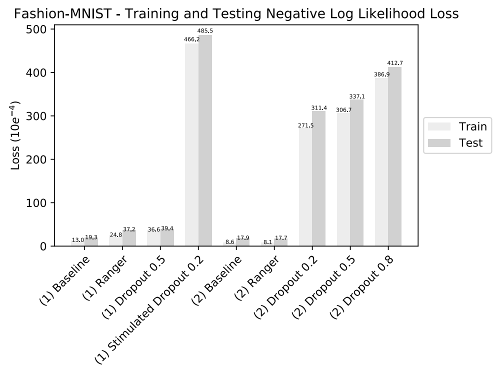
In Figure 5.2 the training and testing NLL loss of all datasets is represented.



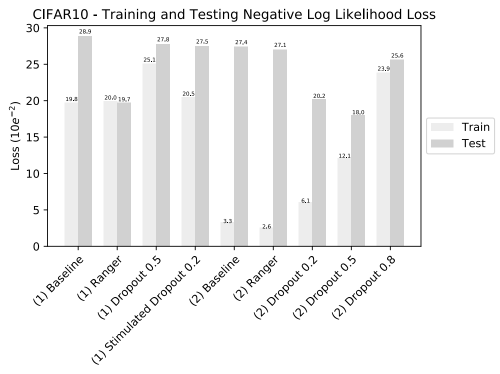
(a) Training and testing NLL loss of each model for the MNIST dataset



(b) Training and testing NLL loss of each model for the GTSRB dataset



(c) Training and testing NLL loss of each model for the Fashion-MNIST dataset



(d) Training and testing NLL loss of each model for the CIFAR10 dataset

Figure 5.2: Training and testing Negative Log Likelihood Loss per dataset

5.1.3. Loading time

A comparison of the approximate loading times for the different datasets is visible in Figure 5.3. The loading time is the time in milliseconds that it takes for the dataset to be loaded before being used for training or testing.

The loading time of a dataset is important because it impacts the experiment time, since whenever the experiment resets, the dataset has to be loaded.

5.1.4. Training time

The time it took to train each model, excluding the loading time, is presented in Figure 5.4. The training in these models is done using GPU acceleration.

Ranger is not represented in this graph, since it is applied the model after training. Applying Ranger is a one-time expense called instrumentation time [28]. In our implementation, this time depends on the testing time of the model, since we are testing the model in 20% of the training dataset for every *ReLU* function.

Figure 5.5 shows the training time per dataset for the models on which Stimulated Dropout was applied. Instead of using GPU acceleration, these models were trained with the CPU, since the PyTorch version used for Stimulated Dropout (v1.4) had incompatibilities with

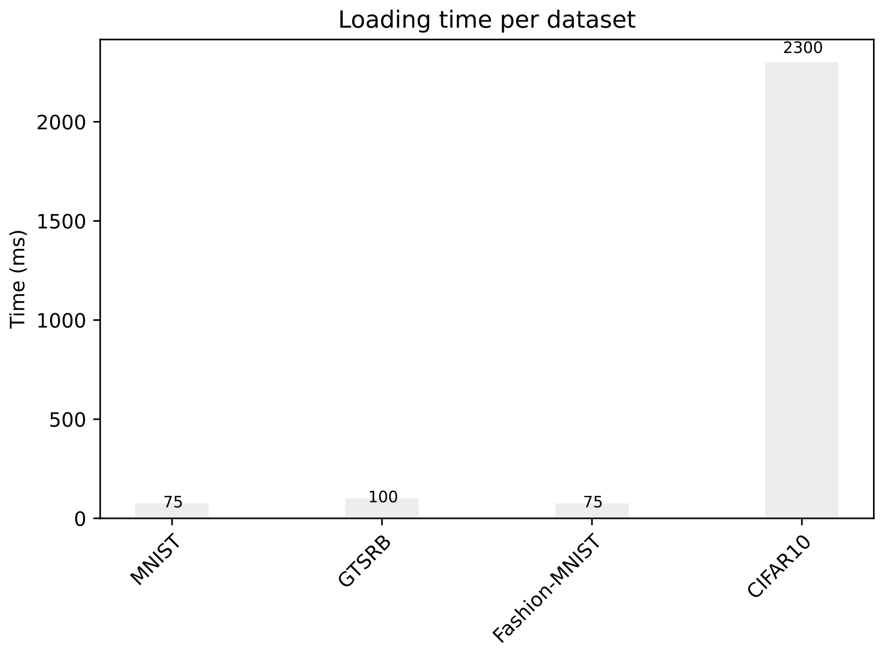
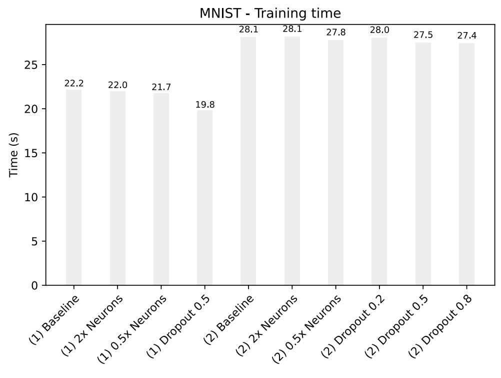
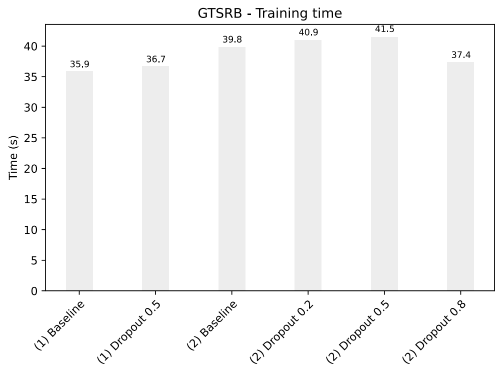


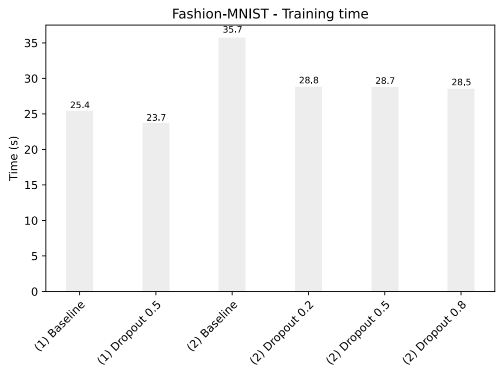
Figure 5.3: Loading time for each dataset



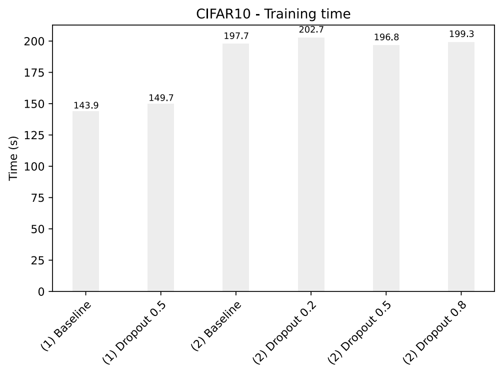
(a) Training time (seconds) for the MNIST dataset



(b) Training time (seconds) for the GTSRB dataset



(c) Training time (seconds) for the Fashion-MNIST dataset



(d) Training time (seconds) for the CIFAR10 dataset

Figure 5.4: Training time per dataset in seconds, using GPU acceleration (CUDA)

the CUDA version of the graphics card. We also trained the baseline model for the *MNIST* dataset with the CPU to verify the differences.

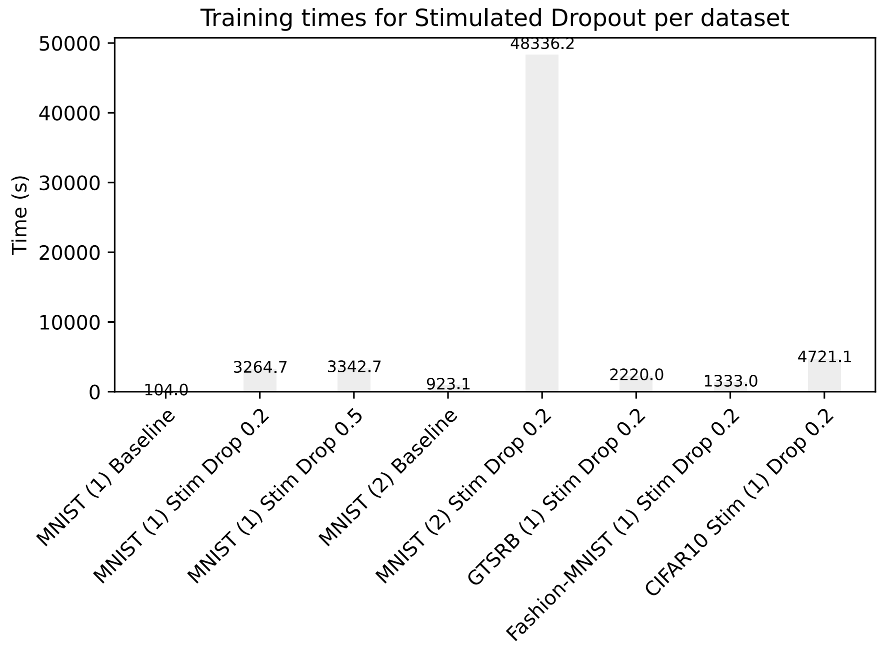


Figure 5.5: Training time (seconds) for models with Stimulated Dropout per dataset, using CPU

5.1.5. Testing time

The testing time for all models is shown in Figure 5.6. Every test was done in the CPU.

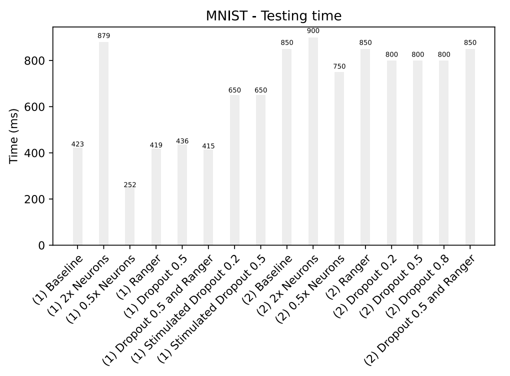
5.1.6. Experiment time

The experiment time depends on the testing and loading times, since these variables directly influence the time between the fault injections, as well as the time it takes to resume injection after the model crashes.

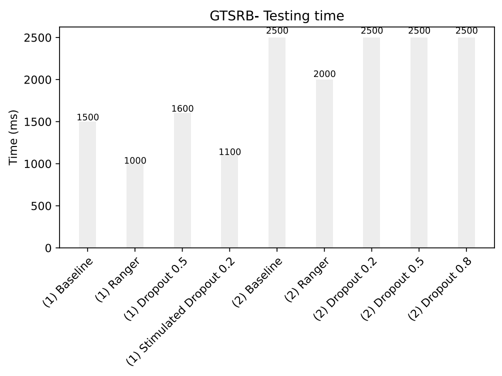
5.2. Experimental outcomes

The results of the experiments for the *MNIST*, *GTSRB*, *Fashion-MNIST* and *CIFAR10* datasets are shown in Figures 5.8 through 5.11.

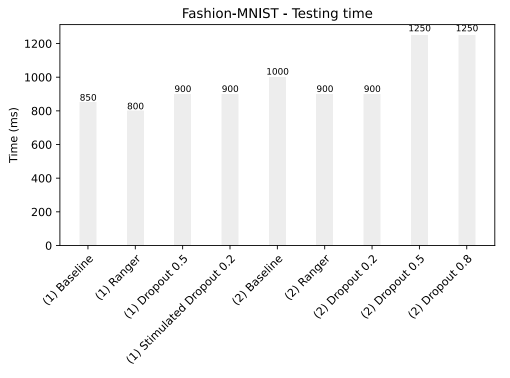
The possible outcomes for a bit-flip are Silent Data Corruption (SDC), Hang/Crash or No Effect. The metrics in this Section are the relative frequencies of each outcome.



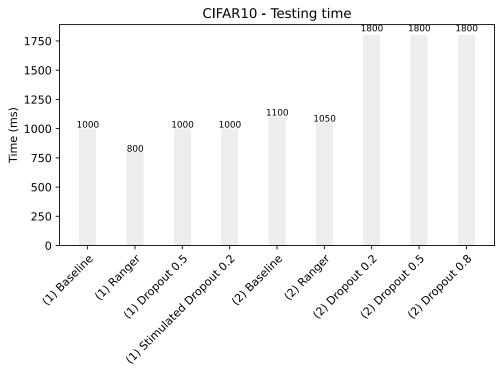
(a) Testing time (milliseconds) for the MNIST dataset



(b) Testing time (milliseconds) for the GTSRB dataset



(c) Testing time (milliseconds) for the Fashion-MNIST dataset



(d) Testing time (milliseconds) for the CIFAR10 dataset

Figure 5.6: Testing time per dataset in milliseconds

5.2.1. Redundancy

In Figure 5.8 we can see the experimental outcomes for models that utilise redundancy techniques for the *MNIST* dataset.

5.2.2. Dropout

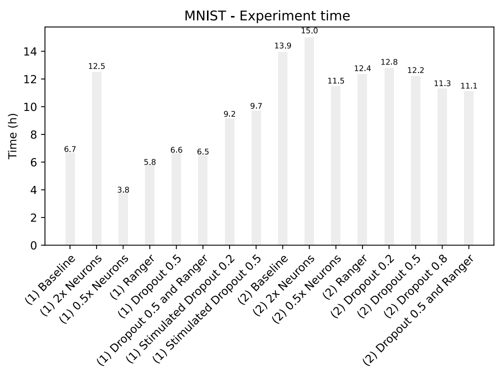
Experimental results for models with Dropout for every dataset can be seen in Figure 5.9.

5.2.3. Ranger

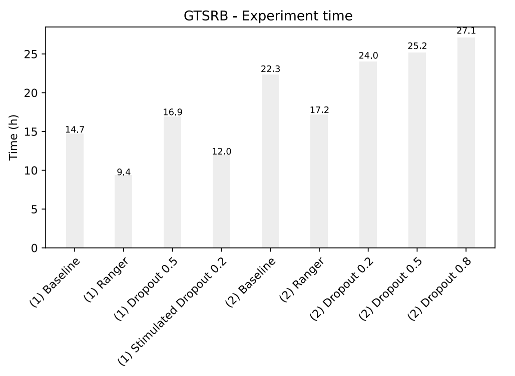
In Figure 5.10 we can see the experimental outcomes for models with Ranger for every dataset. The models were obtained by applying Ranger to baseline models of architectures (1) and (2) for every dataset. In the *MNIST* dataset, additional tests were made to evaluate the impact of Ranger on models with Dropout.

5.2.4. Stimulated Dropout

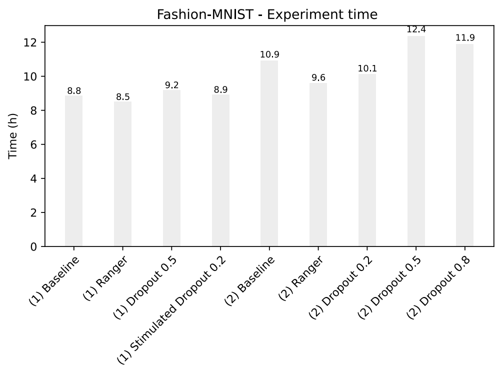
Figure 5.11 contains the experimental outcomes for models with Stimulated Dropout for every dataset. A Stimulated Dropout probability of 0.2 was used in every dataset, since



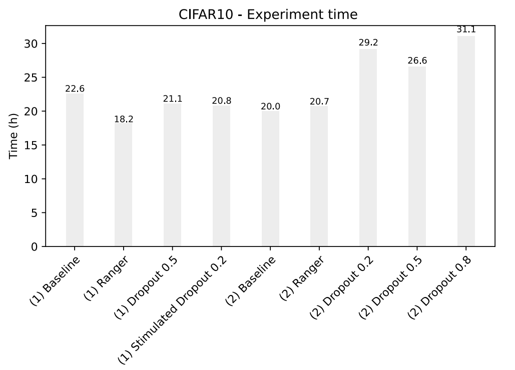
(a) Experiment time (hours) for the MNIST dataset



(b) Experiment time (hours) for the GTSRB dataset



(c) Experiment time (hours) for the Fashion-MNIST dataset



(d) Experiment time (hours) for the CIFAR10 dataset

Figure 5.7: Experiment time per dataset in milliseconds

this is the probability that provided the best results in the original thesis [15]. Additionally, a probability of 0.5 was tested in the *MNIST* dataset in order to evaluate the difference in SDC rate.

5.3. Discussion

5.4. Training and testing metrics

5.4.1. Classification accuracy

The results obtained in this category are meant to evaluate the type of impact that the fault tolerance techniques have on the accuracy of baseline models. The clearest observation is that architecture (2) has better accuracy than architecture (1), which is to be expected since the latter is a basic architecture originally intended to be used with *MNIST*.

Another remark is that Dropout has an inconsistent impact on both training and testing

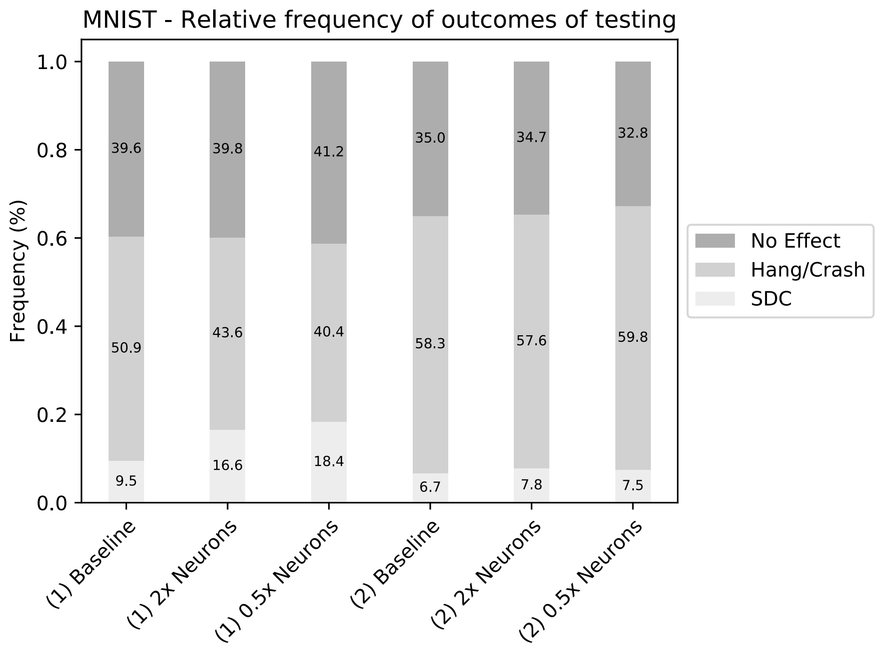


Figure 5.8: Experimental outcomes for models with redundancy for the *MNIST* dataset

accuracy. In architecture (1), this technique decreases these metrics in all datasets, whereas in architecture (2) it has a mixed influence, decreasing them in *MNIST* and *Fashion-MNIST* while increasing them in *GTSRB* and *CIFAR10*.

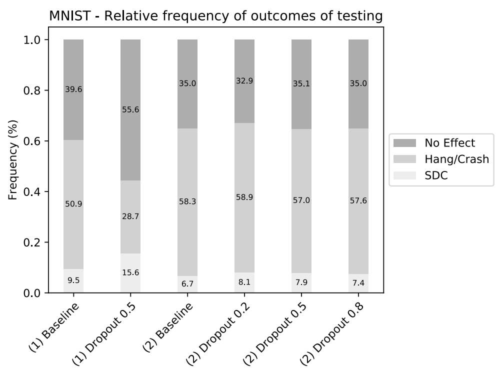
In theory, since Dropout is a regularization method, it should increase the accuracy for models that are overfit. Since we are using the same architectures for all datasets, we know that models trained on the simpler datasets (*MNIST* and *Fashion-MNIST*) should be more overfit than those trained on more complex ones (*GTSRB* and *CIFAR10*). However, in our case, we can see that Dropout benefits the models that are less overfit. Our results therefore disagree with the results in the original paper [107].

Different Dropout probabilities impact architecture (2) in mixed ways, such as decreasing the testing accuracy in the *MNIST* and *Fashion-MNIST* datasets, while increasing it in *GTSRB* and *CIFAR10*. The exception to this is Dropout with probability of 0.8, which consistently provides the lowest training and testing accuracy values. This is to be expected, due to the large proportion of connections which are dropped out.

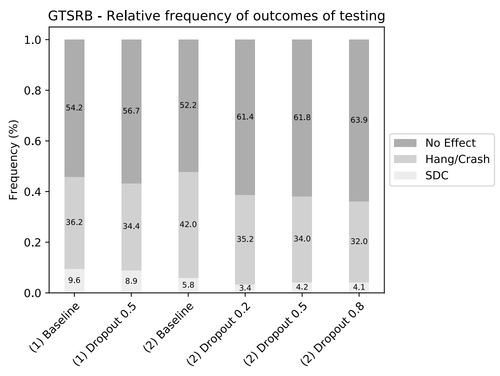
Stimulated Dropout has a negative impact on the accuracy in all datasets with the exception of *MNIST*. However, it shows an improvement in accuracy in comparison with the Dropout technique in every dataset with the exception of *Fashion-MNIST*.

Finally, applying Ranger improves the accuracy in both architectures, including in cases where it is combined with Dropout. This contrasts with the findings in the original paper [28], which state that Ranger does not affect the accuracy in LeNet.

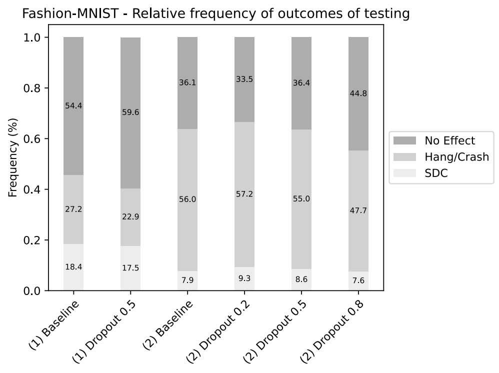
The training and testing accuracies for the *MNIST* dataset (Figure 5.1a) present little variation between models and between the training and testing sets. The redundancy techniques, particularly *0.5x Neurons*, provide the best testing accuracy out of every model for the (1) architecture, whereas in (2) the best training and testing accuracy is achieved by the combination of *Dropout 0.5 and Ranger*.



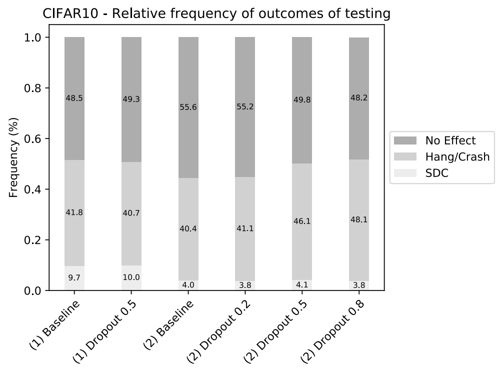
(a) Experimental outcomes for models with Dropout for the MNIST dataset



(b) Experimental outcomes for models with Dropout for the GTSRB dataset



(c) Experimental outcomes for models with Dropout for the Fashion-MNIST dataset



(d) Experimental outcomes for models with Dropout for the CIFAR10 dataset

Figure 5.9: Experimental outcomes for models with Dropout

5.4.2. Loss

This metric shows huge differences between datasets. Different scales were used to improve visibility in some of the results with lower loss values.

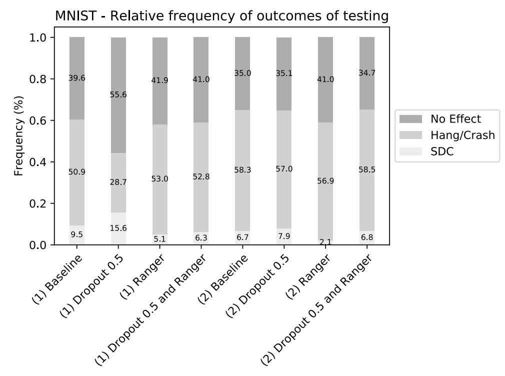
Overall, architecture (2) shows a decrease in the loss value in comparison to architecture (1). This reduction is most significant in the *GTSRB* dataset, where there is a decrease of 90% in the training loss and 53% in the testing loss.

Stimulated Dropout reduces or maintains the loss value in most datasets, with the exception of *Fashion-MNIST* where there is an exponential increase. Dropout increases the training loss while decreasing the testing loss in both architectures and in most datasets, except in *Fashion-MNIST*.

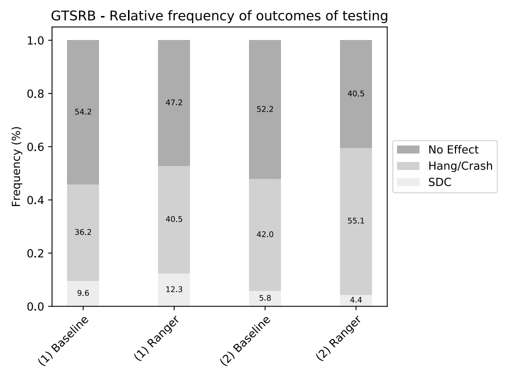
5.4.3. Loading time

Since *MNIST* and *Fashion-MNIST* have the same number of images, image dimensions, file formats and loading procedure, the loading time is equivalent. *GTSRB* has equal image dimensions to *MNIST* and a lower number of images, but the loading time is higher because it has three color channels instead of one.

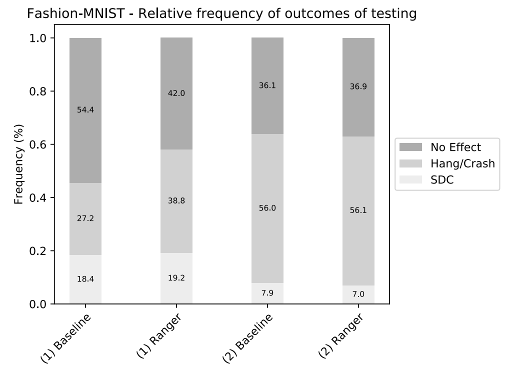
Out of the selected datasets, *CIFAR10* is the one that takes longer to load, because of increased image dimensions, three color channels and an unoptimized loading procedure.



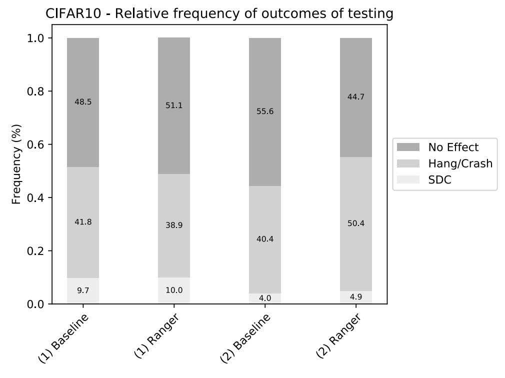
(a) Experimental outcomes for models with Ranger for the MNIST dataset



(b) Experimental outcomes for models with Ranger for the GTSRB dataset



(c) Experimental outcomes for models with Ranger for the Fashion-MNIST dataset



(d) Experimental outcomes for models with Ranger for the CIFAR10 dataset

Figure 5.10: Experimental outcomes for models with Ranger

5.4.4. Training time

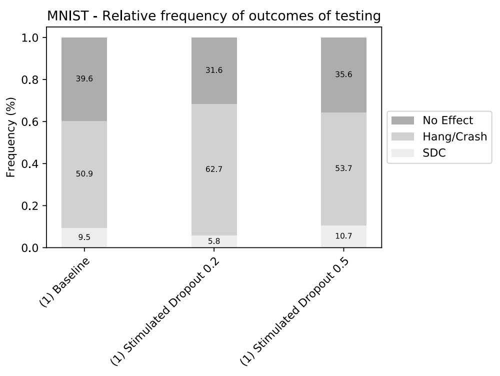
Generally we can conclude that the (2) architecture has increased training time compared to (1) throughout every dataset. As expected, *MNIST* and *Fashion-MNIST* have similar training times, whereas *GTSRB* and *CIFAR10* take longer.

The training times for Stimulated Dropout (Figure 5.5) need to be evaluated separately since the training was done using the CPU. We also trained baseline models for architectures (1) and (2) for the *MNIST* dataset for comparison purposes.

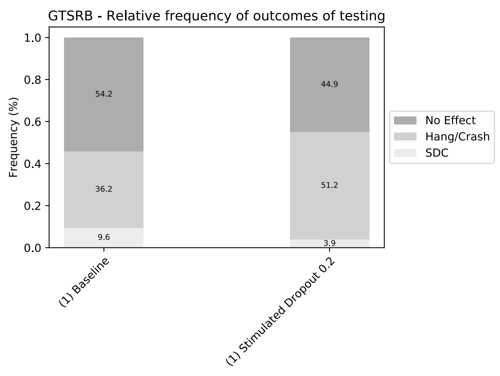
Training a Stimulated Dropout model of architecture (1) on the *MNIST* dataset results in an increase of 31x in training time when compared to the baseline model. This is a considerable difference, especially considering architecture (1) is a very simple Convolutional Neural Network (CNN).

In order to verify if this difference changes with the complexity of the network, a Stimulated Dropout model with architecture (2) was developed and trained on the *MNIST* dataset. The results show an increase of 52x, which made it unfeasible to train models with architecture (2) for every dataset.

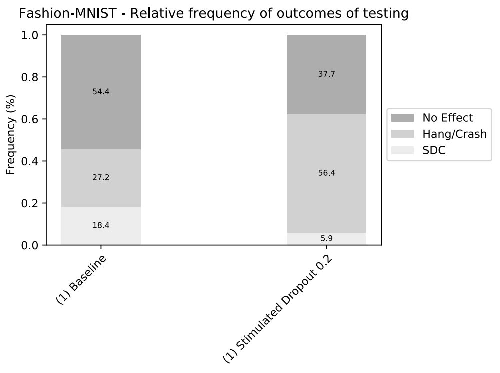
Similarly to the training times in Figure 5.4, *CIFAR10* has the highest time value out of every dataset for architecture (1). In contrast with what was expected, *Fashion-MNIST* and *GTSRB* have lower training times than *MNIST*. This is possibly due external interference of processes running on the CPU while training the *MNIST* model (e.g. running



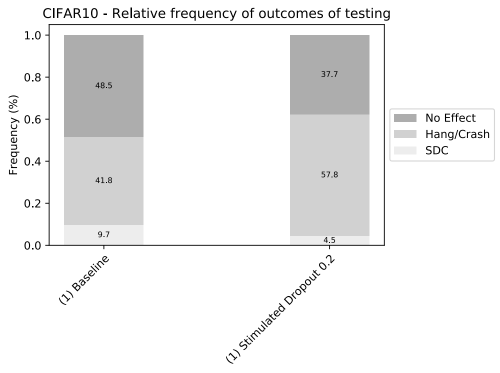
(a) Experimental outcomes for models with Stimulated Dropout for the *MNIST* dataset



(b) Experimental outcomes for models with Stimulated Dropout for the *GTSRB* dataset



(c) Experimental outcomes for models with Stimulated Dropout for the *Fashion-MNIST* dataset



(d) Experimental outcomes for models with Stimulated Dropout for the *CIFAR10* dataset

Figure 5.11: Experimental outcomes for models with Stimulated Dropout

an experiment). In the future, these results should be validated in other machines.

5.4.5. Testing time

Such as in previous timing metrics, the testing time is higher in the models that use the (2) architecture. Another key point is that applying Ranger decreases the testing time on every dataset. Stimulated Dropout has mixed results, increasing the testing time in some datasets (*MNIST* and *Fashion-MNIST*), while decreasing it in others (*GTSRB* and *CIFAR10*).

5.4.6. Experiment time

In every dataset, adding Ranger led to a decrease in the experiment time. Similarly to what occurs in previous time metrics, the impact of Stimulated Dropout depends on the dataset. This technique increases the experiment time in *MNIST* and *Fashion-MNIST*, while decreasing it for*GTSRB* and *CIFAR10*.

5.5. Experimental outcomes

5.5.1. Redundancy

In general, we can identify a decrease in SDCs when moving from architecture (1) to architecture (2). However, when we modify exclusively the number of neurons, there is no such decrease. In fact, doubling or halving the number of neurons while maintaining other factors intact always increases the SDC rate, although the degree of this increase depends on the architecture. In architecture (1), these modifications increase the SDC rate by around 70% to 90%, whereas the same methods in architecture (2) have an impact of 11% to 16%. We can conclude that altering the number of neurons of the network does not have a positive impact on the SDC rate for the dataset, and as such will not apply this technique to other datasets.

5.5.2. Dropout

The tendency across *MNIST*, *Fashion-MNIST* and *CIFAR10* is that Dropout either increases the SDC rate, or has a mixed impact. Examples of the former situation occur in Figure 5.9a, where Dropout increases SDC rates for both architectures, and in architecture (1) in Figure 5.9d, where there is an increase of around 3%. Mixed results can be seen in Figure 5.9c, where in architecture (1) the SDC rate increases by 5%, and in architecture (2) this metric fluctuates between around 17% more and 4% less; as well as in architecture (2) of Figure 5.9d, where the SDC rate varies between 5% lower and 2.5% higher than the baseline model. This indicates that Dropout has an unpredictable impact on these datasets, and as such does not consistently decrease the SDC rate.

The experimental results on the *GTSRB* dataset provide different conclusions from the previously mentioned datasets. As we can see in Figure 5.9b, applying Dropout consistently decreases the SDC rate, regardless of the architecture. Furthermore, it provides the best results out of the Dropout experiments, with a decrease in SDC rate of 41% for Dropout with probability 0.2 applied to architecture (2).

Regarding the impact of different Dropout probabilities on the experimental outcomes, one can relate *MNIST* to *Fashion-MNIST*, since in both datasets the SDC rate decreases with the increase of Dropout probability. Additionally, there is a similarity between *GTSRB* and *CIFAR10*, since Dropout with a probability of 0.5 has a higher SDC rate than the other probabilities. Further research is needed to understand why there is a difference between datasets, and to find the Dropout probability that provides the lowest SDC rate.

5.5.3. Ranger

In contrast to the results obtained in the original paper [28], our experiments do not show a consistent decrease in the SDC rate when using Ranger. *GTSRB* and *Fashion-MNIST* have mixed results, with higher SDC rates for architecture (1), and lower rates for architecture (2). In *CIFAR10*, applying Ranger actually has a negative impact, increasing the SDC rate between 3% and 22%.

The exception to these negative results is *MNIST*, where Ranger significantly decreased the SDC rate of every tested model; in fact, the lowest SDC rate for the *MNIST* dataset across all experiments was obtained by applying Ranger to architecture (2) (2%, which means a reduction of 69% compared to the baseline model).

The experiments using Dropout in the *MNIST* dataset reiterate that Ranger can be used to improve models with different techniques. Since applying Ranger to Dropout with probability 0.5 did not produce better results than applying Ranger to the baseline, this combination was not tested on other datasets.

There is a possibility that the differences between our observations and the ones in the original paper [28] can be attributed to the fact that we are using a different programming language and Machine Learning (ML) framework. This raises questions about the impact of using different tools to train, test and experiment with the same model architectures. These results should be validated in the future by performing experiments with the original implementation of this technique in Tensorflow.

5.5.4. Stimulated Dropout

After analysing the subfigures, we can see that applying Stimulated Dropout with a probability of 0.2 to architecture (1) lowers the SDC rate in every dataset. This reduction ranges from 39% in *MNIST* to 68% in *Fashion-MNIST*. After comparing these results to those obtained with previous techniques, we can conclude that applying Stimulated Dropout to *GTSRB*, *Fashion-MNIST* and *CIFAR10* results in the lowest SDC rates for these datasets.

A model with Stimulated Dropout probability of 0.5 was also tested on the *MNIST* dataset, as can be seen in Figure 5.11a. The result is an increase of the SDC rate of 84% compared to the model with a probability of 0.2. This rise is consistent with the results obtained in [15].

There is a possibility that since the model with Stimulated Dropout is using a different PyTorch version from the baseline model, the experimental outcomes for the models with Stimulated Dropout may be affected in some way. To prove this, further testing is required to compare both versions. This can be done by adapting the code of both models to a single version.

This page is intentionally left blank.

Chapter 6

Conclusion and Future Work

In recent years, an increasing number of tasks with safety considerations has been performed by computerised systems employing Machine Learning (ML) models. Failures in these systems can lead to harmful consequences from an environmental, property or human viewpoint; because of this, Safety-Critical Systems (SCSs) have strict performance requirements and the possibility of failure should be minimised.

Hardware faults are events that can occur arbitrarily in any computerised system. These faults can affect the processes that are running on the processor, causing errors. In the cases where that process is an Artificial Neural Network (ANN), these errors can change numeric values in the network, causing the output values to change. This can result in a misclassification, and therefore a safety violation.

This work focused on evaluating and improving the fault tolerance of Convolutional Neural Network (CNN) models to prevent such safety violations from occurring. To achieve this, we selected the necessary tools and structured the project in order to implement CNN-based ML models; we prepared an experimental analysis to evaluate the fault tolerance of a model; we applied a set of fault tolerance techniques to baseline models and finally we compared the performance of all models. Our goal was to assess which of the fault tolerance techniques produced the best results on a wide group of metrics, with particular emphasis on the Silent Data Corruption (SDC) rate since this metric represents misclassifications resulting from a fault injection. We also wanted to evaluate the difference in these metrics when applying the same model to different datasets.

The results show that regularization methods such as Dropout and methods that employ Redundancy do not have a consistent, positive impact on the SDC rate. The results also show that the Ranger method has better performance in virtually all training, testing and experimental metrics. In addition to increasing the accuracy and decreasing the loss in most cases, Ranger also decreases the testing and experiment time, while maintaining the same training time as the model it is applied to (although it requires extra time to calculate the bounds). These results contrast with those obtained in the original paper [28], since it anticipates an overhead of 0.53% and the same accuracy. In the fault tolerance metrics, our implementation of Ranger again differs from the expected values, since the SDC rate does not decrease as predicted, and in some architectures actually increases.

Stimulated Dropout shows a slight loss in accuracy in most datasets; however, the benefit of this technique is that it has the lowest SDC rates in most datasets out of the tested methods. Comparing this technique with Ranger, we show it has a lower SDC rate in all but the simplest dataset (*MNIST*); however, the downside is a training time that is considerably higher than the baseline, in contrast with Ranger which has a low overhead in training time. This difference in training time weakens the use of Stimulated Dropout in larger CNN architectures.

Additions to this work should start by validating the results of Stimulated Dropout by using the same PyTorch version. Afterwards, other recent fault tolerance methods should be implemented for comparison purposes.

One major focus of future work should be improving the efficiency of Stimulated Dropout to allow it to be considered in modern architectures. In addition, further experimentation is needed to validate the results of this method on larger CNN architectures. Future work should also attempt to understand the change in the SDC rate when the Stimulated Dropout probability is modified.

If the issues with Stimulated Dropout are solved, this technique can become an important addition to any CNN-based ML model that is used in an SCS. We hope that using this and other methods can make these models more tolerant to faults, thus preventing these events from causing damage in a real-life situation.

References

[1] CIFAR-10 and CIFAR-100 datasets.

[2] Dropout — PyTorch 1.9.0 documentation.

[3] Efficient Processing of Deep Neural Networks: A Tutorial and Survey | IEEE Journals & Magazine | IEEE Xplore.

[4] nohup(1) - Linux manual page.

[5] NumPy.

[6] Papers with Code - Papers With Code : Trends.

[7] ptrace(2) - Linux manual page.

[8] Releases · pytorch/pytorch.

[9] seaborn: statistical data visualization — seaborn 0.11.1 documentation.

[10] TensorFlow.

[11] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. *IEEE Std 610*, pages 1–217, January 1991. Conference Name: IEEE Std 610.

[12] Fashion-MNIST, September 2021. original-date: 2017-08-25T12:05:15Z.

[13] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: generic object-oriented fault injection tool. In *2001 International Conference on Dependable Systems and Networks*, pages 83–88, July 2001.

[14] M. Al-Kuwaiti, N. Kyriakopoulos, and S. Hussein. A comparative analysis of network dependability, fault-tolerance, reliability, security, and survivability. *IEEE Communications Surveys Tutorials*, 11(2):106–124, 2009. Conference Name: IEEE Communications Surveys Tutorials.

[15] Luís Alberto Pires Amaro. Resilient Artificial Neural Networks. In *Resilient Artificial Neural Networks*, September 2020. Accepted: 2021-01-14T23:04:50Z Journal Abbreviation: Redes Neurais Artificiais Resilientes University: Universidade de Coimbra.

[16] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990. Conference Name: IEEE Transactions on Software Engineering.

[17] Amanullah Asraf, Md. Zabirul Islam, Md. Rezwanul Haque, and Md. Milon Islam. Deep Learning Applications to Combat Novel Coronavirus (COVID-19) Pandemic. *SN COMPUT. SCI.*, 1(6):363, November 2020.

[18] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[19] Raul Barbosa, Johan Karlsson, Henrique Madeira, and Marco Vieira. Fault Injection. In Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad van Moorsel, editors, *Resilience Assessment and Evaluation of Computing Systems*, pages 263–281. Springer, Berlin, Heidelberg, 2012.

[20] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, April 1990. Conference Name: IEEE Transactions on Computers.

[21] G. Bolt. Fault models for artificial neural networks. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, pages 1373–1378 vol.2, 1991.

[22] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005. Conference Name: IEEE Micro.

[23] Christopher Bowles, Liang Chen, Ricardo Guerrero, Paul Bentley, Roger Gunn, Alexander Hammers, David Alexander Dickie, Maria Valdés Hernández, Joanna Wardlaw, and Daniel Rueckert. GAN Augmentation: Augmenting Training Data using Generative Adversarial Networks. *arXiv:1810.10863 [cs]*, October 2018. arXiv: 1810.10863.

[24] G. Bresson, Z. Alsayed, L. Yu, and S. Glaser. Simultaneous localization and mapping: A survey of current trends in autonomous driving. *IEEE Transactions on Intelligent Vehicles*, 2(3):194–220, 2017.

[25] C++. Standard c++ foundation. `https://isocpp.org/`.

[26] J. Carreira, H. Madeira, and J.G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998. Conference Name: IEEE Transactions on Software Engineering.

[27] F. Cerveira, A. Fonseca, R. Barbosa, and H. Madeira. Evaluating the Inherent Sensitivity of Programming Languages to Soft Errors. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 65–72, 2018.

[28] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. A Low-cost Fault Corrector for Deep Neural Networks through Range Restriction. *arXiv:2003.13874 [cs, stat]*, March 2021. arXiv: 2003.13874.

[29] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. *BinFI*: an efficient fault injector for safety-critical machine learning systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pages 1–23, New York, NY, USA, November 2019. Association for Computing Machinery.

[30] C. Cheng, F. Diehl, G. Hinz, Y. Hamza, G. Nuehrenberg, M. Rickert, H. Ruess, and M. Truong-Le. Neural networks for safety-critical applications — challenges, experiments and perspectives. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1005–1006, 2018.

[31] CMake. Open-source software for build automation, testing and packaging. `https://cmake.org/`.

[32] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. *SIGSOFT Softw. Eng. Notes*, 21(3):158–171, May 1996.

[33] D. de Andres, J. C. Ruiz, D. Gil, and P. Gil. Fault emulation for dependability evaluation of vlsi systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(4):422–431, 2008.

[34] D. Deodhare, M. Vidyasagar, and S. Sathiya Keethi. Synthesis of fault-tolerant feedforward neural networks using minimax optimization. *IEEE Transactions on Neural Networks*, 9(5):891–900, September 1998. Conference Name: IEEE Transactions on Neural Networks.

[35] P.E. Dodd and L.W. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science*, 50(3):583–602, June 2003. Conference Name: IEEE Transactions on Nuclear Science.

[36] dragon119. Retry general guidance - Best practices for cloud applications.

[37] João Durães, Marco Vieira, and Henrique Madeira. Dependability Benchmarking of Web-Servers. In Maritta Heisel, Peter Liggesmeyer, and Stefan Wittmann, editors, *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 297–310, Berlin, Heidelberg, 2004. Springer.

[38] Andre Esteva, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115–118, Feb 2017.

[39] Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. A Survey of Data Augmentation Approaches for NLP. *arXiv:2105.03075 [cs]*, July 2021. arXiv: 2105.03075.

[40] Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. The State of Fault Injection Vulnerability Detection. In Mohamed Faouzi Atig, Saddek Bensalem, Simon Bliudze, and Bruno Monsuez, editors, *Verification and Evaluation of Computer and Communication Systems*, volume 11181, pages 3–21. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.

[41] Chengyue Gong, Tongzheng Ren, Mao Ye, and Qiang Liu. MaxUp: A Simple Way to Improve Generalization of Neural Network Training. *arXiv:2002.09024 [cs, stat]*, February 2020. arXiv: 2002.09024.

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* The MIT Press, 2016.

[43] GTSRB. Traffic Signals dataset. Published: https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign.

[44] GTSRB. Traffic signals dataset. `https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign`.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. arXiv: 1512.03385.

[46] Bruce Hemingway and Ian Summerville. CSE466: Software for Embedded Systems, Slide 13 - Critical Systems.

[47] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[48] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997. Conference Name: Computer.

[49] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. *arXiv:1608.06993 [cs]*, January 2018. arXiv: 1608.06993.

[50] Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Normalization Techniques in Training DNNs: Methodology, Analysis and Application. *arXiv:2009.12836 [cs, stat]*, September 2020. arXiv: 2009.12836.

[51] Sharon Hudson, R. S. Shyama Sundar, and Srinivas Koppu. Fault Control Using Triple Modular Redundancy (TMR). In Prasant Kumar Pattnaik, Siddharth Swarup Rautaray, Himansu Das, and Janmenjoy Nayak, editors, *Progress in Computing, Analytics and Networking*, Advances in Intelligent Systems and Computing, pages 471–480, Singapore, 2018. Springer.

[52] IEEE. 754-2019 - ieee standard for floating-point arithmetic. `https://ieeexplore.ieee.org/document/8766229`.

[53] ImageNet. Large scale visual recognition challenge 2012 (ilsvrc2012). `http://image-net.org/challenges/LSVRC/2012/index`.

[54] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs]*, March 2015. arXiv: 1502.03167.

[55] ISO. Iec 62304:2006 medical device software — software life cycle processes. `https://www.iso.org/standard/38421.html`.

[56] ISO. Iso 26262-1:2018 road vehicles — functional safety. `https://www.iso.org/standard/68383.html`.

[57] Michael Jordan and T.M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science (New York, N.Y.)*, 349:255–60, 07 2015.

[58] W.-I. Kao, R.K. Iyer, and D. Tang. FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, November 1993. Conference Name: IEEE Transactions on Software Engineering.

[59] Wei-Lun Kao and R.K. Iyer. DEFINE: a distributed fault injection and monitoring environment. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259, June 1994.

[60] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin. TWO FAULT INJECTION TECHNIQUES FOR TEST OF FAULT HANDLING MECHANISMS. In *1991, Proceedings. International Test Conference*, pages 140–, October 1991. ISSN: 1089-3539.

[61] Cherry Khosla and Baljit Singh Saini. Enhancing Performance of Deep Learning Models with different Data Augmentation Techniques: A Survey. In *2020 International Conference on Intelligent Engineering and Management (ICIEM)*, pages 79–85, June 2020.

[62] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, 2002.

[63] John C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, May 2002. Association for Computing Machinery.

[64] Israel Koren and C. Mani Krishna. Chapter 9 - Simulation Techniques. In Israel Koren and C. Mani Krishna, editors, *Fault-Tolerant Systems (Second Edition)*, pages 291–340. Morgan Kaufmann, San Francisco (CA), January 2021.

[65] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[66] Alex Labach, Hojjat Salehinejad, and Shahrokh Valaee. Survey of dropout methods for deep neural networks, 2019.

[67] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. Conference Name: Proceedings of the IEEE.

[68] 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. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 1–12, New York, NY, USA, November 2017. Association for Computing Machinery.

[69] Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. TensorFI: A Configurable Fault Injector for TensorFlow Applications. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 313–320, October 2018.

[70] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, page 6, USA, June 2010. USENIX Association.

[71] Yu Li, Yannan Liu, Min Li, Ye Tian, Bo Luo, and Qiang Xu. D2NN: a fine-grained dual modular redundancy framework for deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, pages 138–147, New York, NY, USA, December 2019. Association for Computing Machinery.

[72] Zewen Li, Wenjie Yang, Shouheng Peng, and Fan Liu. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *arXiv:2004.02806 [cs, eess]*, April 2020. arXiv: 2004.02806.

[73] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11 – 26, 2017.

[74] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11 – 26, 2017.

[75] N. Looker, M. Munro, and Jie Xu. A comparison of network level fault injection with code insertion. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 1, pages 479–484 Vol. 2, July 2005. ISSN: 0730-3157.

[76] R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962. Conference Name: IBM Journal of Research and Development.

[77] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. RIFLE: A general purpose pin-level fault injector. In Klaus Echtle, Dieter Hammer, and David Powell, editors, *Dependable Computing — EDCC-1*, Lecture Notes in Computer Science, pages 197–216, Berlin, Heidelberg, 1994. Springer.

[78] Matplotlib. Visualization with python. `https://matplotlib.org/`.

[79] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[80] B. Melhart and S. White. Issues in defining, analyzing, refining, and specifying system dependability requirements. In *Proceedings Seventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2000)*, pages 334–340, April 2000.

[81] El Mahdi El Mhamdi and Rachid Guerraoui. When Neurons Fail. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1028–1037, May 2017. arXiv: 1706.08884.

[82] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of MacOS applications using random testing. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 46–54, New York, NY, USA, July 2006. Association for Computing Machinery.

[83] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[84] Barton P. Miller, Mengxiao Zhang, and Elisa R. Heymann. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IIEEE Trans. Software Eng.*, pages 1–1, 2020. arXiv: 2008.06537.

[85] Tom Mitchell. *Machine Learning.* McGraw Hill, 1997.

[86] MNIST. Handwritten digits database. `http://yann.lecun.com/exdb/mnist/`.

[87] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing Dependability with Software Fault Injection: A Survey. *ACM Comput. Surv.*, 48(3):1–55, February 2016.

[88] Michael A. Nielsen. Neural networks and deep learning, 2018.

[89] OpenCV. Programming Library aimed at real-time Computer Vision. Published: https://opencv.org/.

[90] K. Pattabiraman, G. Li, and Z. Chen. Error resilient machine learning for safety-critical systems: Position paper. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–4, 2020.

[91] K. Pattabiraman, G. Li, and Z. Chen. Error Resilient Machine Learning for Safety-Critical Systems: Position Paper. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–4, 2020.

[92] E. Phaisangittisagul. An analysis of the regularization between l2 and dropout in single hidden layer neural network. In *2016 7th International Conference on Intelligent Systems, Modelling and Simulation (ISMS)*, pages 174–179, 2016.

[93] Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. Fault attacks, injection techniques and tools for simulation. In *2015 10th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, April 2015.

[94] Paul Pop, Viacheslav Izosimov, Petru Eles, and Zebo Peng. Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):389–402, March 2009. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[95] Lutz Prechelt. Early Stopping - But When? In Genevieve B. Orr and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, pages 55–69. Springer, Berlin, Heidelberg, 1998.

[96] PyTorch. C++ frontend. `https://pytorch.org/cppdocs/frontend.html`.

[97] PyTorch. Mnist example. `https://github.com/pytorch/examples/tree/master/cpp/mnist`.

[98] Pranav Rajpurkar, Awni Y. Hannun, Masoumeh Haghpanahi, Codie Bourn, and Andrew Y. Ng. Cardiologist-level arrhythmia detection with convolutional neural networks, 2017.

[99] S. Raschka. *Python Machine Learning*. Packt Publishing, 2015.

[100] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957.

[101] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[102] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 97–108, June 2017. ISSN: 2158-3927.

[103] Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. *J Big Data*, 6(1):60, December 2019.

[104] Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6(1):60, July 2019.

[105] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*, April 2015. arXiv: 1409.1556.

[106] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[107] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[108] Jiacheng Sun, Xiangyong Cao, Hanwen Liang, Weiran Huang, Zewei Chen, and Zhenguo Li. New Interpretations of Normalization Methods in Deep Learning. *arXiv:2006.09104 [cs, stat]*, June 2020. arXiv: 2006.09104.

[109] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv:1512.00567 [cs]*, December 2015. arXiv: 1512.00567.

[110] C. Torres-Huitzil and B. Girau. Fault and Error Tolerance in Neural Networks: A Review. *IEEE Access*, 5:17322–17341, 2017.

[111] ucXception. Fault Injection Framework. Published: https://github.com/ucx-code/ucXception.

[112] ucXception. Fault injection framework. `https://github.com/ucx-code/ucXception`.

[113] Ramakrishna Vadlamani, Jia Zhao, Wayne Burleson, and Russell Tessier. Multicore soft error rate stabilization using adaptive dual modular redundancy. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 27–32, March 2010. ISSN: 1558-1101.

[114] Naihong Wei, Shiyuan Yang, and Shibai Tong. A modified learning algorithm for improving the fault tolerance of BP networks. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 1, pages 247–252 vol.1, June 1996.

[115] Yuxin Wu and Kaiming He. Group Normalization. *arXiv:1803.08494 [cs]*, June 2018. arXiv: 1803.08494.

[116] Zheyu Yan, Yiyu Shi, Wang Liao, Masanori Hashimoto, Xichuan Zhou, and Cheng Zhuo. When Single Event Upset Meets Deep Neural Networks: Observations, Explorations, and Remedies. *arXiv:1909.04697 [cs, stat]*, September 2019. arXiv: 1909.04697.

[117] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *CoRR*, abs/1906.05113, 2019.

[118] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into Deep Learning. *arXiv:2106.11342 [cs]*, June 2021. arXiv: 2106.11342.

[119] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random Erasing Data Augmentation. *arXiv:1708.04896 [cs]*, November 2017. arXiv: 1708.04896.

[120] Zhi-Hua Zhou and Shi-Fu Chen. Evolving Fault-Tolerant Neural Networks. *Neur. Comp. App.*, 11(3):156–160, June 2003.