POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Fault Injection in Normalizing Flow Models for Space Applications

Supervisors

Prof. Marcello CHIABERGE

Doct. Carlo CENA

Candidate

Gabriele GRECO

July 2025

Abstract

There are thousands of satellites in orbit around the Earth and in deep space, each performing different tasks crucial to sectors such as research, communication, Earth observation, and space science. Since the space environment differs significantly from Earth's, the robustness and resilience of these spacecraft must be guaranteed. One of the main issues they face is radiation, which can compromise their lifespan or effectiveness. To prevent these problems, space applications use different systems designed to detect anomalies. Machine learning and artificial intelligence systems have been adopted to exploit their efficiency and decision-making capabilities. More specifically, modern anomaly detection techniques leverage neural networks to capture temporal dependencies and handle high-dimensional data effectively. However, to ensure that these solutions are robust and resilient in space, numerous tests must be conducted. This is where fault injection comes into play. In our study, we are applying a physics-informed (PI) real-valued non-volume preserving (real NVP) model, a type of normalizing flow model, for fault detection in space systems. This study begins with an analysis of the different hazards posed by the space environment, focusing on single-event upsets (SEUs) that cause malfunctions in systems. It then examines the normalizing flow network and the various proposed solutions for injection testing, such as PytorchFI and TensorFI, two injection frameworks. Following this, the study proposes a framework implemented in TensorFlow to test the network and evaluate the resilience of its components. Injections can be performed using two main fault functions, each presenting different network targets. Layer States injection involves introducing faults into the internal components of the network, such as weights and biases while leaving the input and output untouched. Layer Outputs Injection is where the injection is applied to the output of different layers, targeting different layers and activations functions. All those faults are generated through different operations like zeros, random, and bitflips modifying the value of a tensor, and they are done at different levels of ablation by selecting the percentage of values to inject and the depth of the layer. Results show that the most critical part of the network consists of the layers preceding the final output. As expected, bit flips in the most significant bits can lead to significantly worse performance and even system failure.

Table of Contents

List of Tables

List of Figures

Acronyms

AD

Anomaly Detection

AI

Artificial Intelligence

ANN

Artificial Neural Network

AUC-PR

Area Under the Precision-Recall Curve

AUC-ROC

Area Under the Curve

CNN

Convolutional Neural Network

DDD

Displacement Damage Defects

DL

Deep Learning

DNN

Deep Neural Network

DUE

Detected Unrecoverable Error

ELU	Exponential Linear Unit
EPS	Electric Power System
FIT	Failure in Time
FPGA	Field-Programmable Gate Array
GAN	Generative Adversarial Network
GRU	Gated Recurrent Unit
LSTM	Long Short Term Memory
ML	Machine Learning
NN	Neural Network
Real NVP	Real-valued non-volume preserving
RELU	Rectified Linear Unit
RNN	Recurrent Neural Network
SAA	South Atlantic Anomaly

SDC	Silent Data Corruption
SDE	Silent Data Error
SEB	Single Event Burnout
SEE	Single Event Effect
SEGR	Single Event Gate Rupture
SEL	Single Event Latchups
SET	Single Event Transient
SEU	Single Event Upset
SRU	Stellar Reference Unit
TID	Total Ionizing Dose
TSAD	Time Series Anomaly Detection

Chapter 1

Introduction

Satellites play a vital role in various fields, including communication, navigation, and Earth observation missions like ERS-1 (European Remote-Sensing Satellite) [1], designed to monitor environmental factors such as oceans and polar ice caps. As the use of space devices continues to grow rapidly, protecting them from the harsh conditions of space, such as radiation, becomes increasingly important. This underscores the need for developing robust hardware and software models. Simultaneously, the use of Neural Networks (NN) has expanded significantly due to their ability to process large amounts of data and extract relevant insights. In the space sector, neural networks are employed to ensure mission success and safeguard valuable assets. For example, ESA's BepiColombo [2] uses neural networks in its fault detection and control system to monitor the spacecraft's operational health as it travels to Mercury. However, evaluating the reliability of NNs is highly challenging due to their complex structure and software, often consisting of hundreds of layers.

This study aims to test a Normalizing Flow network using fault injection to simulate the radiation present in space and evaluate the model's robustness and resilience to faults after the injections. Therefore, the primary goal is to create a framework capable of testing multiple models trained with various hyperparameters simultaneously, under different injection configurations.

This introduction provides an overview of the thesis. Section 1.1 addresses the effects and risks of radiation on space devices and Neural Networks. Section 1.2 begins by explaining the fundamental concepts of Neural Networks, the structure of the Normalizing Flow models, and details regarding the Anomaly Detection task. Section 1.3 introduces potential solutions to the radiation issue. Finally, Section 1.4 outlines the thesis organization to give a clearer understanding of the research.

1.1 The impact of space radiation on electronic devices

Primary cosmic rays interact with gaseous and other matter at high altitudes, producing secondary radiation, both of which contribute to the space radiation environment. The solar wind provides the transportation of electrons and protons produced by the Sun's fusion process, Figure 1.1. In addition to the particles originating from the sun are particles from other stars and heavy-ion sources such as novas and supernovas in our galaxy and beyond. These particles are influenced by planetary or Earth's magnetic field to form radiation belts, which in Earth's case are known as Van Allen Radiation belts, containing trapped electrons in the outer belt and protons in the inner belt. The composition and intensity of the radiation vary significantly with the trajectory of a space vehicle and its position on Earth, for example, the South Atlantic Anomaly (SAA) has relatively high concentrations of electrons causing problems such as single event upsets (SEU). In addition to these particles, the spacecraft experience radiation threats from high-energy ions in space called Galactic Cosmic Rays. Solar flares also contribute varying quantities of electrons, protons, and lower-energy heavy ions that cause single-event effects (SEE) [3]. As a result, space vehicles and satellites are constantly exposed to a radiation environment, which can degrade component performance and cause various malfunctions in electronic and electrical systems.

In 1998, Deep Space 1 (DS-1) [4] experienced a failure of its Stellar Reference Unit (SRU) and an upset in a Field-Programmable Gate Array (FPGA) register. The SRU had previously exhibited intermittent problems starting shortly after launch. While these issues were thoroughly investigated, no explanation was deemed satisfactory. The final failure of the SRU may have been due to a latchup, with earlier transient issues potentially caused by Single Event Upsets (SEUs). Although SEUs can, and occasionally have, caused failures requiring lengthy system recovery times, the specific failure observed here is unusual. A recovery time of 28 minutes lacks a clear explanation unless a thermal event was triggered, which could have influenced the behavior of the system. This remains speculative because of the absence of corroborating evidence.

Radiation-induced impacts on electronics can be categorized into two types: cumulative effects and single-event errors, each based on the way radiation causes errors.

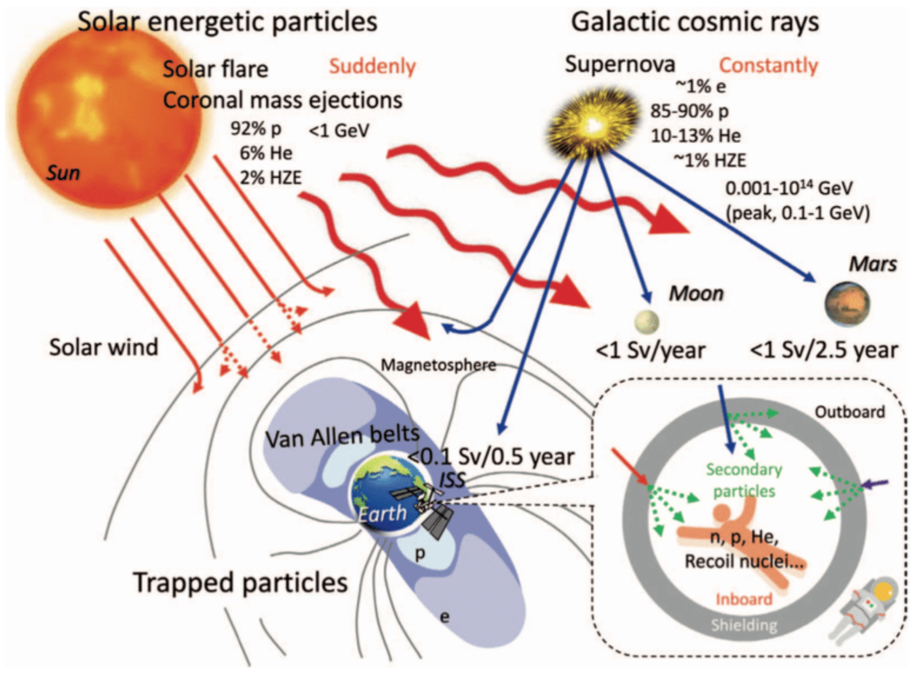


Figure 1.1: Components of the space radiation environment

1.1.1 Cumulative Effect

Cumulative Effects Cumulative effects refer to the gradual degradation of electronic devices due to continuous exposure to ionizing radiation over time. These effects primarily impact hardware at the physical level. As such, we will not cover them in detail here.

The **Total Ionizing Dose (TID)** is the accumulated dose of ionizing radiation that a device absorbs over time. TID effects are generally caused by prolonged exposure to radiation, particularly from particles trapped in the Earth's Van Allen belts.

Displacement Damage Defects (DDD) arise when high-energy particles such as protons or heavy ions displace atoms within a device's semiconductor material. This displacement creates defects in the crystal lattice, potentially leading to increased leakage currents and shifts in the device's threshold voltage.

1.1.2 Single Event Effects

On the other hand, Single Event Effects (SEEs) result from the interaction of a single high-energy particle with a semiconductor device. These effects are typically immediate or occur shortly after the particle strike. SEEs can be categorized into soft and hard errors: **Soft errors** are temporary and non-destructive, causing data corruption or transient malfunctions without permanent damage to the component, meanwhile **Hard errors** are permanent and can cause physical damage to the electronic device.

In the category of soft errors belong:

- **Single Event Upsets (SEUs)** occur when a single ionizing particle hits a sensitive node in an active microelectronic device, such as a microprocessor, semiconductor memory, or power transistor. This impact alters the state of a logic element, such as a memory 'bit', due to the free charge generated by ionization.
- **Single Event Transients (SETs)** are brief pulses of current or voltage induced by a high-energy particle passing through a circuit, which can lead to temporary glitches or erroneous outputs.

Hard errors instead can be differentiated as:

- **Single Event Latchups (SELs)** occur when a high-energy particle triggers a short circuit within the device, causing excessive current flow that can damage or destroy the component.
- **Single Event Burnout (SEB)** happens when a high-energy particle causes a power semiconductor to conduct excessively, which can result in thermal runaway and permanent damage.
- **Single Event Gate Rupture (SEGR)** is an event in which a single energetic-particle strike results in a breakdown and subsequent conducting path through the gate oxide of a MOSFET.

1.1.3 Effect of Radiation

As mentioned previously, when a particle hits a transistor, it can cause bit flips in memory. A radiation-induced transient error in a computing device executing code may result in one of three outcomes: no effect on the program output, silent data corruption (**SDC**) which produces incorrect program output, or a detected unrecoverable error (**DUE**) which leads to a program crash or device reboot. When a particle deposits enough charge to alter the state of a transistor, it generates a fault. This fault may either be masked or propagated to a visible state, becoming an *error*. If this corrupted visible state is used in computation and the error propagates to the software output, it results in a *failure*. SDCs or DUEs are considered failures when they affect the software output.

In the case of Deep Neural Networks (DNNs), tracking the fault propagation from a corrupted transistor to the detection or classification output is particularly challenging. Unlike deterministic computing, DNN outputs are probabilistic. A fault may alter low-probability outcomes, which might not be selected in the final detection or classification, or it may entirely change the output vector.

Fortunately, not all SDCs are critical in object detection frameworks. If an SDC does not affect detection and classification, it can be considered tolerable. SDCs that alter the object probability without impacting the object's rank are generally not critical. However, SDCs that lead to misdetection or misclassification are deemed critical. We will discuss the SDC metric in Chapter 3.

Although cumulative, permanent, or destructive effects such as Total Ionizing Dose (TID) or Single Event Latch-up (SEL) are crucial for space mission deployment, this study does not focus on reproducing these effects in detail. This is because these radiation responses are more related to technology implementation rather than the specific architecture or Deep Neural Networks (DNNs).

Therefore, our goal is to emulate effects with an immediate impact on software, such as Single Event Upsets (SEUs). However, it's important to remember that hard errors can also affect software through hardware propagation, as explained in Section 1.3.

1.2 Normalizing Flow Network and Anomaly Detection

There has been a fast-forward evolution of Artificial Intelligence in the last years, especially for deep learning (DL), the most interesting type of Machine learning, in which several processing layers are used to extract higher-level features from the input data to find patterns. It is crucial to understand its components to further access a reliability evaluation and improvement of neural networks.

1.2.1 Basic Concepts of Neural Networks Effect

Artificial Neural Networks (ANN) are capable of approximating a wide range of functions thanks to backpropagation training, enabling the solutions of a variety of tasks, ranging from classification, detection, and regression.

Designing deep neural networks (DNNs) involves selecting the number, types, and structure of layers that, when connected, can be adapted to solve a specific task, as well as tuning hyperparameters, such as the number of neurons and kernel size. This adaptation occurs during the **training phase**, where the model learns from the data. The data includes input data and corresponding labels if the training is **supervised**, lacks labels if the training is **unsupervised**, or contains a mix of labeled and unlabeled data in the case of **semi-supervised** training. Through this phase, the parameters of the network are modified in such a way that for each training input-output pair, the network response is as close as possible to the ground truth. The distance measured between true and predicted values is called the **loss function**. Backpropagation of the output error from the last layer back to the input one helps the parameters in the process of tuning to fit the dataset.

Each network design has a specific set and organization of layers:

- **Fully Connected Layer:** In this layer, every neuron from the previous layer is connected to every neuron in the current layer. This dense connectivity allows the network to integrate features learned by previous layers and make final predictions.
- **Convolutional Layers** are used to automatically and adaptively learn spatial hierarchies of features from input images or other grid-like data.
- **Pooling layers** reduce the spatial dimensions of the input, filtering and decreasing the number of parameters and computational load in the network. They also help make the model invariant to small translations in the input data.
- **Normalization layers** stabilize and speed up the training process by normalizing the input to each layer, leading to faster network convergence and improved generalization. The most widely used normalization technique is Batch Normalization, which learns an approximation of the first and second statistical moments of each feature map during training to normalize input tensors. After normalization, it is common practice to apply an affine transformation using an additive bias β and a scaling factor γ. The normalization operation is defined as follows:

$$BatchNorm(x) = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta \tag{1.1}$$

- **Dropout Layer:** This layer helps prevent overfitting by randomly setting a fraction of the input units to zero during training, encouraging the network to learn more robust features.

Activation functions introduce non-linearity into the network, enabling it to learn complex patterns and represent non-linear decision boundaries. Common activation functions include:

- **Rectified Linear Units (ReLUs):** defined as $f(x) = max(0, x)$, ReLUs are fundamental for avoiding the vanishing gradient problem and are widely used due to their simplicity and effectiveness. Variants include: Leaky ReLU ELU (Exponential Linear Unit) SELU (Scaled ELU)

- **Sigmoid:** defined as $f(x) = \frac{1}{1+e^{-x}}$, the sigmoid function maps input values to the range [0, 1] and is often used in binary classification tasks.
- **Softmax:** defined as $f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$, the softmax function converts logits (raw model outputs) into a probability distribution over multiple classes, ensuring that the sum of all probabilities equals 1.
- **Tanh (Hyperbolic Tangent):** defined as $f(x) = tanh(x)$, this function maps input values to the range [-1, 1], making it zero-centered, which can lead to faster convergence and enhanced performance in hidden layers.
- **Linear Activation:** defined as $f(x) = ax + b$, where a and b are constants, this simple function is used in situations where the model is expected to perform linear regression or when a direct proportional relationship between input and output is required.

In every layer, we can find **weights**, which are typically real numbers that represent the strength or importance of a connection between neurons. These values are initialized randomly and are adjusted during the training process. Weights control how much influence a particular input has on the neuron's output. A higher weight amplifies the input signal, while a lower weight attenuates it.

Bias is an additional parameter in NN associated with each neuron. It is a constant added to the weighted sum of inputs before the activation function is applied. It's crucial because it helps the network to approximate the output even when all inputs are zero, allowing the network to model patterns that are not centered around the origin and enabling a better data fit, adding flexibility in modeling.

There are different types of NN, each of them with a different architecture formed by the combination of the proposed components. The most common ones are CNN (Convolution Neural Network), RNN (Recurrent Neural Network), LSTM (Long Short Term Memory), GANs (Generative Adversarial Networks), Autoencoders, GRUs (Gated Recurrent Units) and many more, each of them tailored for specific kinds of data and tasks, making them suitable for a wide range of applications in ML and AI.

In our study we are going to deal with Normalizing Flow Network and more specifically with a REAL NVP (Real-valued non-volume preserving) Network.

1.2.2 Normalizing Flow Models

Real NVP is a neural network architecture used in the field of normalizing flows, a class of generative models. These models are especially useful for tasks like density estimation, where the challenge lies in constructing models that are both powerful enough to capture complex patterns and scalable enough to be trained on high-dimensional, highly structured data.

Normalizing flows are a family of models that transform a simple, known probability distribution like a Gaussian into a more complex distribution that matches the data distribution. This transformation is done using a sequence of invertible (bijective) functions. **Invertible** means that the input can be exactly recovered from the output and this is crucial in tasks like generative modeling, where you need to map from latent space back to data space. In traditional flow-based models, transformations are often designed to preserve volume, which simplifies the computation of probability density. However, Real NVP introduces non-volume-preserving transformations, which offer more flexibility in modeling complex distributions while still allowing efficient computation of the determinant of the Jacobian, which is necessary for calculating densities.

Coupling Layers are the fundamental component of these networks built as a flexible and tractable bijective function by stacking a sequence of simple bijections. In each simple bijection, part of the input vector is updated using a function that is simple to invert but depends on the remainder of the input vector in a complex way. These are the formula

$$y_{1:d} = x_{1:d} \tag{1.2}$$

$$y_{d+1:D} = x_{d+1:D} \odot exp(s(x_{1:d})) + t(x_{1:d}) \tag{1.3}$$

where s and t stand for scale and translation

- **Scale Layers** apply a multiplicative transformation to the modified portion of the data, allowing the network to learn how to stretch or compress it across different dimensions. This helps capture the variability in the data distribution.

- **Translation layers** add an offset to the transformed data, shifting them and adjusting the mean of the distribution in the transformed space.

The RealNVP architecture is constructed by stacking multiple coupling layers in an alternating pattern, ensuring that components left unchanged in one layer are updated in the next, addressing issues with the forward transformation. [5] utilizes the network's capability to model natural images across four datasets - CIFAR-10, ImageNet, Large-scale Scene Understanding (LSUN), and CelebFaces Attributes (CelebA) - through techniques such as sampling, log-likelihood evaluation, and latent variable manipulations. [6] uses a LSTM Encoder-Decoder network with a

RealNVP model for density-based time anomaly diagnostics to locate anomalous road segments in the anomalous clusters.

1.2.3 Anomaly Detection Task

The study of time series anomaly detection (TSAD) has become increasingly popular in recent years due to its widespread applications, Figure 1.2. A time series is a sequence of numbers or vectors indexed by time. In the context of space devices, the time series at hand has a high number of features representing all the data collected by instruments. Due to the harsh environment of space and its conditions, which are very different from those on Earth, this data often contains many anomalous points caused by space radiation impacting various components on board. Since space radiation causes a high number of anomalies, it is crucial to analyze their different types, their sources, and the problems they cause.

The first factor to consider is how balanced the classes are in our dataset during testing. Metrics like accuracy, precision, and recall must account for class distribution. Since most data points are normal, predicting only normal points could result in high accuracy, even though the prediction wouldn't be useful. Additionally, precision and recall fail to adequately penalize false negatives and false positives, respectively.

The Area Under the Curve (AUC-ROC) can also give an overly optimistic view of classifier performance in highly imbalanced datasets. This is because a high number of true negatives keeps the false positive rate (FPR) low, even if many minority class instances (anomalies) are misclassified. For this reason, AUC-PR (Area Under the Precision-Recall Curve) is a more reliable metric in such cases.

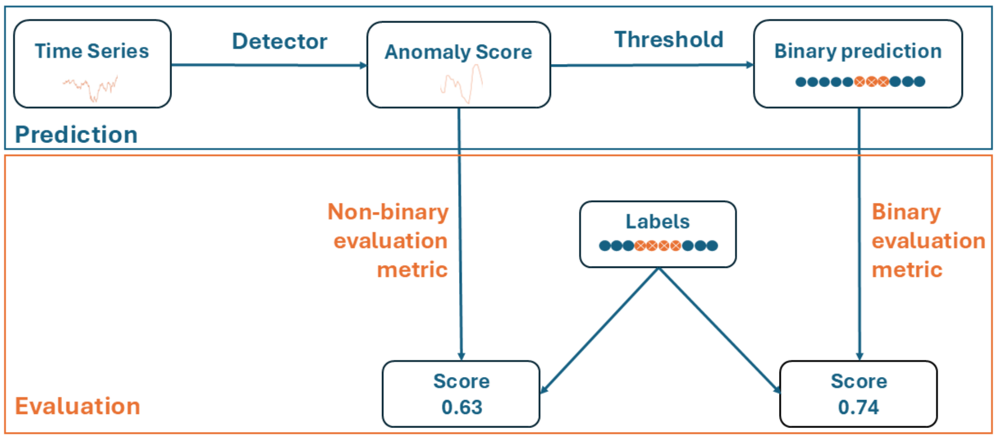


Figure 1.2: A time series anomaly detection pipeline.

In our scenario, we are working with a slightly imbalanced dataset, as we will discuss in Chapter 3, where the number of anomalies is lower than the number of normal cases, as observed in the ADAPT Dataset. This imbalance must be addressed to effectively use metrics like precision and recall without encountering the challenges typically associated with imbalanced data. The characteristics of the dataset and environmental conditions guide the selection of time series anomaly detection (TSAD) algorithms and the evaluation metrics used for the anomaly detection task. Selecting the appropriate metric for anomaly detection remains a complex and critical challenge. Papers such as [7] and [8] tackle this issue by analyzing various environments and offering potential solutions. Using inappropriate metrics can lead to inaccurate conclusions about an algorithm's performance, potentially resulting in misguided decisions.

1.3 Potential Solutions to Reliability Issues

When evaluating the reliability of complex computing devices executing DNNs, we must consider that the radiation-induced fault occurs in the physical transistor and then propagates through the architectures, Figure 1.3, reaching the software and eventually modifying the output. Evaluations closer to the physical layer are more realistic, while assessments close to the software layer are more efficient [9]. Several methodologies exist for evaluating the reliability of computing devices, covering various abstraction levels, from the gate level to the architectural and system levels.

- **Field Test** involves exposing the device to natural particle flux and counting the number of observed errors [10].

- **Beam Experiment** induces faults by interacting accelerated particles with the silicon lattice at the transistor level, providing highly realistic error rates. [11] presents a neutron beam fault injection for FPGA implementation of QNNs accelerators.

- **Microarchitecture-level** fault injection offers broader fault coverage compared to software-level injection, as faults can, in principle, be injected across most system modules [12].

- **Software Fault injection**, performed at the highest level of abstraction, has proven effective in identifying code sections, such as NN components in our case, that are more likely to affect computation when corrupted [13].

- **Circuit- or gate-level** simulations operate at the lowest level of abstraction, inducing either analog current spikes or digital faults while still tracking fault propagation [14].

- To improve efficiency and maintain accuracy, **hybrid approaches** combining different abstraction levels are often employed. This combination of methods is typically the best approach for a comprehensive reliability assessment of the system [15].

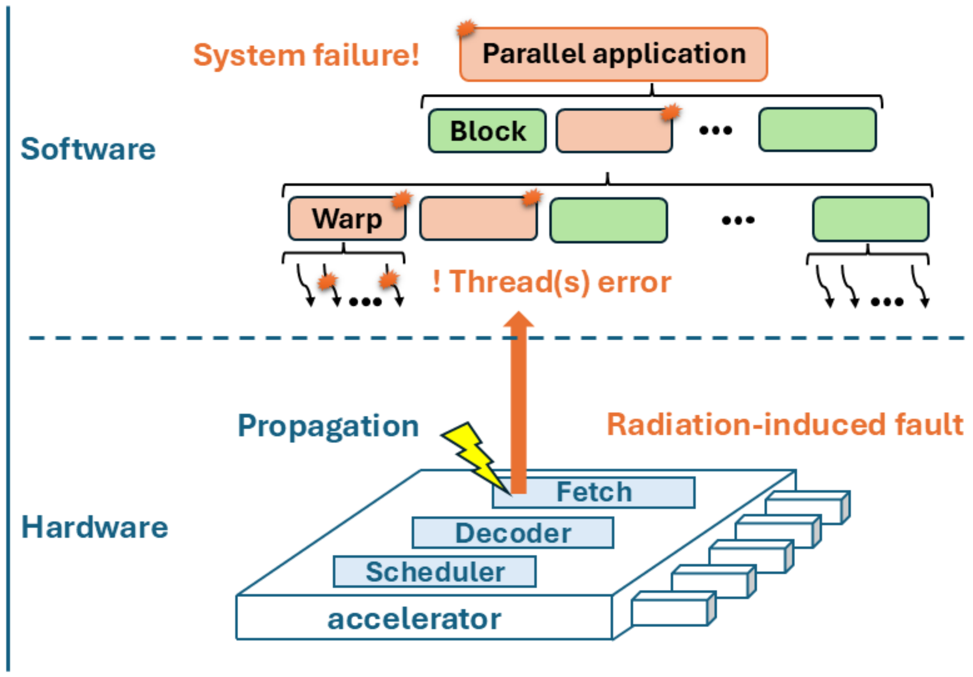


Figure 1.3: Potential impact of a single-event effect corrupting essential or shared resources in a parallel device.

In this study, we focus on the software fault injection that needs to be carefully engineered so as not to have unrealistic results. It has good pros like lower costs, is more controllable, and easier for developers to deploy, however, the adopted fault model (typically bit flip [16], [17], [18], [19], [20], [21]) might be accurate for the main memory structures but risks being unrealistic when considering faults in the computing cores or control logic where the programmer has no power.

For this reason, software fault injection is typically used ([22], [23], [24]) to analyze critical software operations, algorithmic procedures, or components, rather than to estimate the device's error rate. Our approach involves emulating the effects of radiation by modifying memory values and altering key elements of the network, temporarily disabling them to assess their impact on performance and the overall network.

1.4 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 discusses state-of-the-art and the most common framework used in the field of software fault injection, focusing on their key differences, and weak and strong points. Chapter 3 starts by describing the environment we are dealing with, the dataset, the structure of the framework, the pre-trained model, and the evaluation technique. Chapter 4 shows the results obtained from the injection of faults comparing the model in their main type layers state and layer outputs. Chapter 5 discusses the results achieved in this work and explores potential avenues for future research.

Chapter 2

Techniques for Fault Injection

This chapter provides an overview of the currently available tools, frameworks, and techniques for fault injection. Some methods are applied during the training phase to enhance model robustness, while others are used post-training during inference runs. In Section 2.1, we begin by introducing the resilience features of deep neural networks (DNNs) and various techniques used to perturb them. Section 2.2 explores the fault injection frameworks developed using the TensorFlow library, with a focus on injectors such as TensorFI, BinFi, and Ares. Finally, Section 2.3 examines the PyTorch counterparts, specifically the PyTorchFI and PyTorchALFI frameworks.

2.1 Resilience Characteristics of DNN

The resilience and robustness of deep neural networks (DNNs) have been widely studied by researchers. [19], [25] analyze these aspects specifically in DNN accelerators and applications. Before exploring various fault injection frameworks, it is crucial to understand the consequences of soft errors and the impact of perturbations introduced at different components of the network.

In Section 1 we introduced the key components of neural networks. Now, we focus on four main parameters that influence the impact of soft errors in DNNs, as indicated in [19]:

- **Topology and Data Type:** Each DNN has a unique architecture that affects error propagation. Moreover, different data types are used in the implementation process, further influencing how errors spread.

- **Bit Position and Value:** The sensitivity of each bit position varies, depending on the data type. The interpretation of a bit depends on its position, as different data types handle bit values differently.
- **Layers:** DNNs differ in the number, types, and positions of layers. Errors propagate in distinct ways across these layers.
- **Data Reuse:** The study examines how various data reuse strategies in DNN accelerators' dataflows impact the Silent Data Corruption (SDC) probability.

The study tested networks such as AlexNet, CaffeNet, NiN, and ConvNet using a DNN simulator on various datasets to inject faults and evaluate performance by calculating:

- **SDC Probability:** The likelihood of silent data corruption when faults affect an architecturally visible state.
- **FIT (Failure-in-Time) Rate:** The rate of failure in the hardware structure, calculated based on SDC probabilities.

$$FIT = \sum_{component} R_{raw} * S_{component} * SDC_{component} \tag{2.1}$$

One of the first results found is that SDC probabilities vary between networks, even for the same data type. This means certain networks may be more robust with specific data types. Regarding bit positions, they observed that high-order exponent bits are more likely to cause SDCs when corrupted, while mantissa and sign bits are less critical. Bitflips from 0 to 1 in higher-order bits are more likely to cause errors than those from 1 to 0, as correct values in DNNs tend to cluster around zero. Thus, small changes in magnitude or sign bits often do not significantly affect results, as explained in [26].

When considering layer position and type, [19] observed a decreasing probability of fault propagation across network layers. Faults occurring in earlier layers have a higher likelihood of propagating through the network, but even though many faults spread into multiple locations and reach the final layer, only a small percentage of them impact the final ranking of output candidates. This is because the numerical value of activation functions affected by faults is a more significant factor influencing Silent Data Corruption (SDC) probability than the mere number of erroneous activations. Another key observation is that many faults are masked by layers such as POOL or ReLU, which prevent them from propagating to the output layer. This result will be further demonstrated through fault injection into layer outputs, as discussed in Chapter 4.2.

After the identification of critical components of DNNs, they suggest strategies to improve resilience. For data type selection, DNNs should use a type that offers sufficient numerical range and precision to operate with success. Managing sensitive bits is challenging, but restricting data types and suppressing dynamic value ranges can help mitigate the effects of bitflips. Additionally, normalization layers can improve accuracy by mitigating SDCs, as faulty values are normalized alongside fault-free values. Therefore, their use should be considered to enhance network resilience.

This analysis provides initial insights into DNN resilience. Our study, however, will focus on time series data with high dimensionality, using 32-bit floating-point precision as explained in Section 3.3.1.

2.1.1 Mutation

Mutation testing is a well-established technique that introduces slight modifications to program code to simulate faults. While it is generally more suitable for analyzing test set coverage than for robustness testing, we will still examine how it works. The framework proposed in [27] applies mutation testing to deep learning (DL) systems. A key aspect of mutation testing involves designing and selecting mutation operators that introduce potential faults into the software under test, creating modified versions of the program. Mutation operators can be categorized into two groups: data mutation operators and program mutation operators, both of which modify the software at the source level to introduce faults.

- **Data Mutation Operators:** These operators can be applied globally to all data types or locally to specific data types within the training dataset. They are summarized in the Table 2.1.

- **Program Mutation Operators:** Focused on different aspects of DL training programs, three operators are designed to inject faults: Layer Removal (randomly deletes a layer from the neural network), Layer Addition (adds an extra layer to the network structure), and Activation Function Removal (removes an activation function, which plays a crucial role in the non-linearity of deep networks).

Once these operators mutate the training data and program, a set of mutant models is generated, and the test set is evaluated on these mutants. This source-level mutation process provides insights into potential fault injection techniques. Another type of mutation testing operates at the model level. Unlike source-level mutation, which alters the original training data (D) and training program (P), model-level mutation directly modifies the DL model. Specific model-level mutation operators have been created and summarized in the Table 2.2 to efficiently introduce faults.

Fault Type	**Level**	**Target**	**Operation Description**
Data Repetition (DR)	Global	Data	Duplicates training data
	Local		Duplicates specific type of data
Label Error (LE)	Global	Data	Falsify results (e.g. labels) of data
	Local		Falsify specific results of data
Data Missing (DM)	Global	Data	Remove selected data
	Local		Remove specific types of data
Data Shuffle (DF)	Global	Data	Add noise to training data
	Local		Add noise to specific type of data
Layer Removal (LR)	Global	Prog.	Remove a layer
Layer Addition (LA_s)	Global	Prog.	Add a layer
Act. Fun. Remov. (AFR_s)	Global	Prog.	Remove activation functions

Table 2.1: Source-level mutation testing operators for DL systems.

Mutation Operator	**Level**	**Description**
Gaussian Fuzzing (GF)	Weight	Fuzz weight by Gaussian distribution
Weight shuffling (WS)	Neuron	Shuffle selected weights
Neuron Effect Block (NEB)	Neuron	Block a neuron effect on following layers
Neuron Activation Inverse (NAI)	Neuron	Invert the activation status of a neuron
Neuron Switch (NS)	Neuron	Switch two neurons of the same layer
Layer Deactivation (LD)	Layer	Deactivate the effects of a layer
Layer Addition (LA_m)	Layer	Add a layer in a neuron network
Act. Fun. Remov. (AFR_m)	Layer	Remove activation functions

Table 2.2: Model-level mutation testing operators for DL systems.

2.2 Tensor Fault Injection

TensorFlow [28] is an open-source machine learning and deep learning framework developed by Google Brain and released in 2015. It was designed to simplify the creation and deployment of machine learning models. Over the years it has received many updates, improving its usability and performance, with the changes reflected in two major versions. The differences between these versions, as described below, have influenced the development of fault injection frameworks.

Originally, TensorFlow used a static computational graph 2.1 where the entire graph was defined and compiled before running a model. This allowed for certain optimizations, especially for large models, but at the cost of flexibility, meaning that the optimization had to be done before running the sessions. However, in

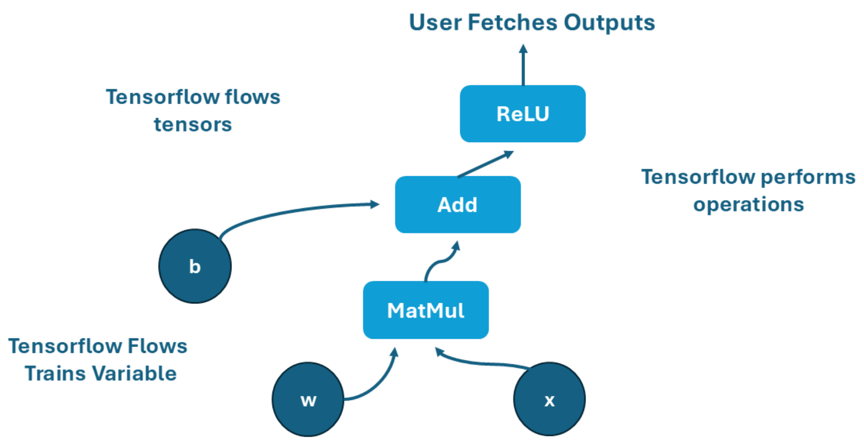

User Fetches Outputs
Tensorflow flows tensors
ReLU
Add
Tensorflow performs operations
b
MatMul
Tensorflow Flows Trains Variable
w
x

Figure 2.1: A TensorFlow dataflow graph.

TensorFlow 2.x, eager execution (dynamic execution) was introduced, similar to PyTorch. This feature allows users to run code immediately without building a static graph and when it's needed it could explicitly convert parts of the code into graph execution mode for performance improvements.

In TensorFlow 1.x, model building was low-level and session-based, requiring more manual work. After the update to the second version, the Keras API became the primary interface, promoting high-level model-building methods that enable users to create, train, and evaluate models more easily. Keras allows developers to build and train DL models using an intuitive, user-friendly syntax, removing much of the complexity involved in lower-level operations such as tensor manipulation, gradient computation, and device management.

Overall, the introduction of eager execution marked a significant step toward making TensorFlow more accessible and competitive with PyTorch. TensorFlow 1.x had a steeper learning curve due to its static computational graph, while TensorFlow 2.x simplified many processes, enabling rapid prototyping, interactive debugging, and a more Pythonic coding style.

2.2.1 TensorFI Version 1

TensorFI [29] is a high-level fault injection framework for TensorFlow-based applications, designed to inject both hardware and software faults into a program. During its development, researchers focused on three key design constraints:

- **Ease of Use and Compatibility:** The framework was designed to be compatible with third-party libraries, with fault injection requiring minimal changes to the application code.

- **Portability:** TensorFI was developed to run on any system where TensorFlow is installed, ensuring wide usability across different platforms.
- **Minimal Interference:** The injection process should not interfere with the normal execution of the TensorFlow graph when no faults are being injected.

To meet these requirements, TensorFI creates a parallel replica of the original TensorFlow graph, equipped with new operators capable of injecting faults during execution. This is necessary because TensorFlow does not allow modification of the dataflow graph after its creation. At runtime, the system decides which graph - the original or the fault-injected one - should be invoked for each execution of the machine learning algorithm. The process begins by initializing the original graph and creating a duplicate of each node for fault injection purposes. In the second phase, faults are injected at runtime, and the system returns the corresponding faulty output. TensorFI uses a YAML file to configure the fault injection parameters, which include:

- **Seed:** The random seed used for reproducibility.
- **TensorFaultType:** The type of fault to inject into tensor values.
- **InjectMode:** The mode of fault injection, such as errorRate (specifying the error rate for components), dynamicInstance (randomly injecting faults into random instances of operators), or oneFaultPerRun (injecting a single fault at random during execution).
- **Ops:** A list of TensorFlow operators to target for fault injection, along with the probability of injecting faults into each.
- **SkipCount:** An optional parameter for skipping the first 'n' invocations before starting fault injections.

For evaluation, the framework uses the Silent Data Corruption (SDC) Rate, which measures how much an injected incorrect output deviates from the expected output of the program. TensorFI has also received an update, with further details outlined in Section 2.2.4.

2.2.2 BinFI

BinFi [30] is a popular fault injection framework designed to identify safety-critical bits in machine learning applications that lead to safety condition violations when impacted by hardware transient faults (soft errors). Unlike traditional fault injection methods that rely on random sampling of fault locations to provide a statistically significant estimate of resilience, BinFi aims to identify all critical bits in the application, rather than focusing on the behavior of individual neural network components. BinFI defines Silent Data Corruption (SDC) as a discrepancy between the outputs of faulty and fault-free program execution, with critical bits being those where faults would result in SDC. BinFI is based on an extended version of TensorFI, on which a monotonicity property of neural networks functions and binary search is added, Figure 2.2.

Recognizing that the time required to run an exhaustive fault injection analysis is directly proportional to the number of bits in the program, they aim to reduce this time by leveraging monotonicity properties.

- **Non-strictly monotonic function:** A non-strictly monotonic function is one that is either monotonically increasing, where $f(x_i) \geq f(x_j), \forall x_i > x_j$, or monotonically decreasing, where $x_i \leq f(x_j), \forall x_i > x_j$. A function is considered monotonic if it is non-strictly monotonic.

- **Approximately monotonic:** A function is approximately monotonic if it behaves as a non-strictly monotonic over a non-trivial interval (e.g. the sine function). A function has approximate monotonicity when it meets this condition. For instance, the function $f(x) = 100 * max(x-1, 0) - max(x, 0)$ is monotonically increasing when $x > 1$, but not when $x \in (0, 1)$, so it can be considered as approximately monotonic.

- **Error Propagation (EP) function:** An EP function is a composite of functions that propagate a fault from its origin to the model's output. For example, if a fault occurs in a system with a *MatMul* operation followed by a *ReLu* function, the function EP would be the composite of both. They are expressed as $EP(x) = ReLu(MatMul(x_{org} - x_{err}, w))$, where w is the weight used in MatMul, and x_{org} and x_{err} are the values before and after the presence of a fault.

Since the majority of functions in an ML model are monotonic, the EP function is likely to be monotonic or approximately monotonic as well. This characteristic helps reduce the fault injection space. By modeling the EP function as: $|EP(x)| \geq |EP(y)|, (x > y >> 0) \cup (x < y << 0))$, x and y represent faults in the same data bit (0 or 1), it can be expected that larger faults (in absolute value) will cause greater deviations than smaller ones. Furthermore, larger deviations at the output are more likely to trigger an SDC. Based on this, they can first inject x, and if it doesn't cause an SDC, it can infer that faults in lower-order bits y are unlikely to result in SDCs, avoiding the need for additional simulations. However, this approach may introduce some inaccuracy due to the approximation of monotonicity.

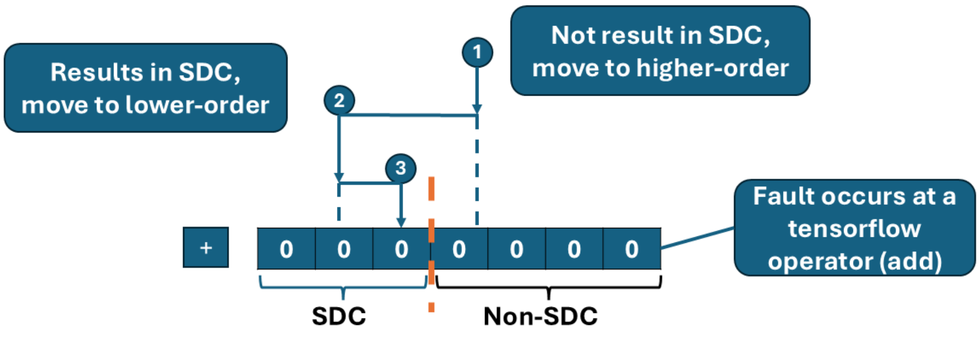

Results in SDC, move to lower-order
Not result in SDC, move to higher-order
1
2
3
+
0 0 0 0 0 0 0
Fault occurs at a tensorflow operator (add)
SDC
Non-SDC

Figure 2.2: Illustration of binary fault injection.

Regarding the binary search approach, they want to find the SDC boundary bit, where faults in high-order bits lead to SDCs, while faults in lower-order bits are masked. Identifying this boundary is equivalent to searching for a specific target in a sorted array of bits, which is why they choose a binary search-like algorithm as their fault injection strategy. Once this boundary is found, the binary search can determine the number of critical bits within a range, such as a 32-bit segment.

BinFI can be used to assess the overall behavior of a model and identify its total critical bits, although its main limitation is the inability to inject dynamically during inference runs because it is based on the first version of TensorFI. Additionally, some components may lack the monotonicity property, making one of its key features less effective.

2.2.3 ARES

ARES [31] is a lightweight DNN-specific fault injection framework designed to quantify the fault tolerance and accuracy of three DNN models: fully connected, CNNs, and GRUs, Figure 2.3. It supports two modes of fault injection: static and dynamic. Static injection introduces faults offline, before inference execution, and is preferred as it adds no performance overhead. Dynamic injection, on the other hand, introduces faults during inference, with minimized overhead by leveraging native tensor operations to emulate fault behavior. This is similar to layers state and layers output modes in TensorFI2 as explained in Section 2.2.4, although this was done before the introduction of the eager-execution in TensorFlow 2.x.

ARES defines three fault points during DNN inference time: weights, activation functions, and hidden states. It injects into these components based on the DNN's topology and the specific experiment setup, requiring only a bit error rate, and the targeted structure. Faults are modeled in memory structures of DNNs using bit-level fault injection to alter values. By sweeping fault rates, the user can then analyze the fault tolerance of the DNN.

Fault injection is performed at two different phases: construction time (static injection) and evaluation time (dynamic injection). After training, the weights are known, and ARES performs static fault injection by altering saved weights outside of Keras. In contrast, activations and hidden states are dynamic, input-dependent, dynamic values, requiring modifications to the Keras inference computation. Fault injection for activations and states is implemented using GPU-compatible element-wise tensor operations.

To further validate their framework, they performed silicon validation to demonstrate that it accurately captures the bit error behavior exhibited by real hardware. This was achieved by comparing simulation results with measurements from a fabricated DNN accelerator designed to induce and measure SRAM faults.

Ares, together with TensorFI, is designed to operate within the Keras and TensorFlow frameworks, respectively. Both of them allow modifying the state of the layers in DNNs as they are executing. However, Ares requires modifications to the Keras inference process to enable dynamic fault injection, as this feature was not yet available in TensorFlow, which is essential for most reliability research studies.

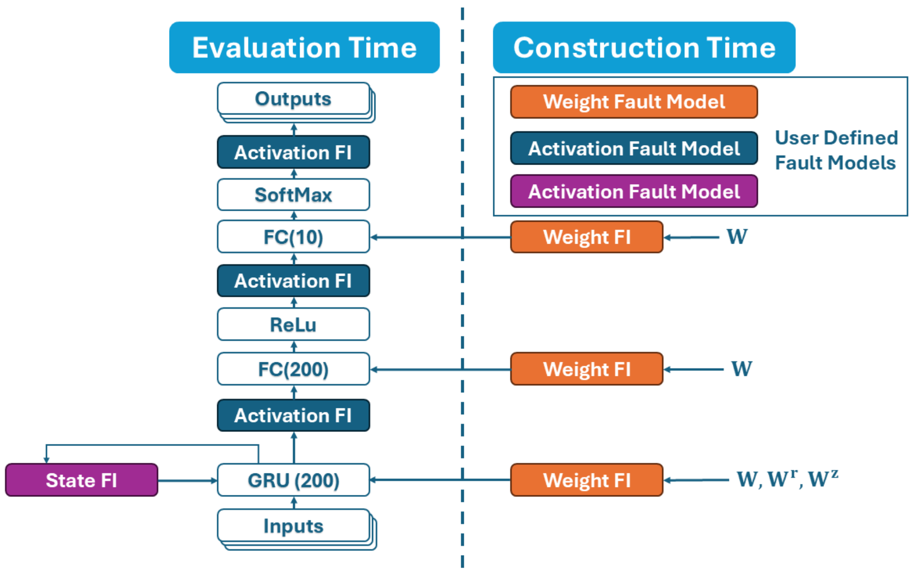


Figure 2.3: An illustration of the Ares framework applied to a DNN.

2.2.4 TensorFI Version 2

TensorFI2 [21] is the updated version of TensorFI based on Tensorflow 2.x. At the same time, it makes use of Keras Model APIs to inject static faults in the layer states and dynamic faults in the layer outputs. ML models are made up of input data, weight matrices that are learned during training, and activation matrices that are computed from the weights and data. TensorFI targeted only the activation matrices for fault injection, meanwhile TensorFI2 is capable of injecting faults into both weight and activation matrices. The two types of injection defined above support single and multiple faults along with three types of faults - zeros, random value replacements, and bit-flips.

Layer States: this mode is static, it can be done before inference runs and it injects into the layer states that hold the learned weights and biases. This is illustrated in the Figure 2.4. The coupling layers are part of a RealNVP network. Assume that the first coupling layer is selected for fault injection. TensorFI 2 injects faults into the weights or biases of this layer, and the faulty parameters are stored back into the model. During inference, as the test input moves through the various layer computations, the fault is activated when the execution reaches the output of the final layer. This fault can then propagate through subsequent layers, potentially leading to a faulty prediction.

Layer Outputs: this other mode is dynamic, it is executed during inference runs and it injects into the layer outputs that hold the activation functions. Modifying the layer outputs is a dynamic process that occurs during inference runs. This is illustrated in the Figure 2.4. The coupling layers are part of a RealNVP network. Suppose the outputs of the first coupling layer are selected for fault injection. TensorFI 2 creates two Keras backend functions, K_{func1} and K_{func2}, which operate on the original model without duplicating it but with specified inputs and outputs. During inference, TensorFI 2 passes the inputs through K_{func1}, which intercepts the computation at the coupling layer 1. It injects faults into the layer's outputs or activation states and then passes the modified outputs to K_{func2}. This function feeds the faulty outputs into the next layer, continuing the execution through the rest of the original model. Since K_{func2} operates on the faulty computation, the faults can propagate through the model and result in a faulty prediction.

As in the first version, both modes are done considering SDC Rate as the standard metric rather than accuracy. This choice was made to focus specifically on the model's resilience to faults. Only data points that were correctly predicted in fault-free runs are considered during fault injection testing. Accuracy alone does not account for this, as it includes cases where the model makes incorrect predictions for reasons unrelated to the injected faults.

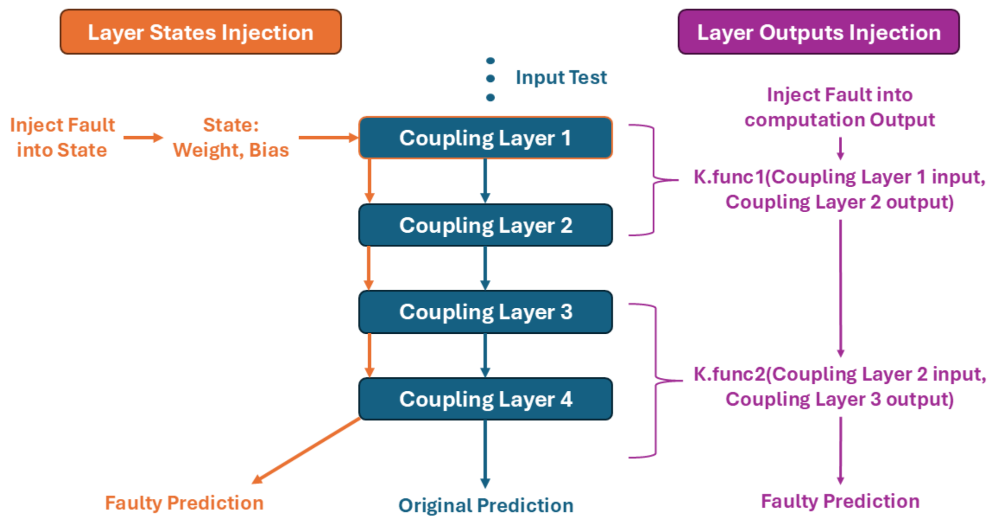


Figure 2.4: Representation of Layer States and Outputs injections inside a ReaNVP Network.

2.3 Pytorch Fault Injection

Pytorch [32] is an open-source machine learning library primarily used for DL applications developed by Facebook's AI Research Lab (FAIR) and released in 2016. One of his key characteristics is the dynamic computational graph which, unlike TensorFlow's 1.x static graphs, is built on-the-fly during execution. This makes it easier to debug, modify, and experiment with models.

Another interesting feature is an Autograd module that automatically computes gradients for backpropagation by tracking operations on tensors. This enables easier implementation of custom backpropagation logic without manually calculating derivatives. DNN implemented with pytorch requires their injection framework.

2.3.1 PyTorchFI

PytorchFI [33] is a runtime perturbation tool for DNNs, designed for the widely used PyTorch DL platform. The tool demonstrates several applications, including reliability analysis of CNNs, reliability analysis on object detection networks, resilience analysis of models built to withstand adversarial attacks, and training models that are resilient to errors. The primary way for dynamic neuron injection is achieved through Pytorch's hook functionality to perturb neuron values during the forward pass of a computational model. By leveraging the hook API for error injection, PytorchFI avoids modifying Pytorch source code, allowing perturbations to run at native Pytorch speed with minimal overhead.

For weight perturbations, PytorchFI introduces wrapper functions that modify the weight tensors before inference, effectively perturbing weights offline and away from the critical path. PytorchFI is user-friendly, requiring only three steps for injection. After installing the framework, the user provides the model to be perturbed and runs a dummy inference to profile the model and collect network hyperparameters. The final step involves selecting a model and specifying a perturbation location. Similar to TensorFI, PytorchFI offers default injection types, such as random value injection, single bit-flip, or zero-value injection.

Users can specify a single neuron or weight location for fault injection (by layer, feature map, or neuro coordinates), or multiple locations for repeated injections throughout the network. After defining these parameters, the actual injection is performed at runtime by appending the targeted locations to a list, and during each forward pass, the relevant layers iterate through the list to corrupt the specified values according to the chosen perturbation model.

While PyTorchFI is effective for injecting faults at the application level of DNNs and analyzing their impact on the system level, a limitation is that it cannot inject faults at lower levels, such as register-level faults.

2.3.2 PyTorchALFI

PyTorchALFI [18] introduces a novel fault injection framework called PyTorchALFI (Application Level Fault Injection for PyTorch) based on PytorchFI. On top of that it offers an efficient method for defining randomly generated, reusable fault sets for injection into Pytorch Models, and at the same time, it allows the creation of complex test scenarios, dataset augmentation, and the generation of test KPIs while integrating fault free, faulty, and modified NNs.

From Figure 2.5 we can see the software architecture of the framework with every component. The principal unit is the **alficore**, which provides a test class that integrates all essential functionalities. The scenario is responsible for initializing the fault injection run, containing information like the fault model and the number of injected faults. With the enhanced PyTorchFI component, it can take care of the fault injection execution, modifying the neuron values in place with the help of PyTorch hooks.

In addition to its core functionalities, alficore introduces several user-friendly features such as data loader wrappers, monitoring tools to detect NaN or Inf values, and evaluation functions for assessing Silent Data Errors (SDE) or Detectable Uncorrected Errors (DUE) rates. It also includes visualization tools that enable users to plot key results.

Weights, neurons, or numerical values can be modified and at the same time is possible to define which layer the fault injection occurs in. All faults are pre-generated as matrices before the inference run to improve the explainability of

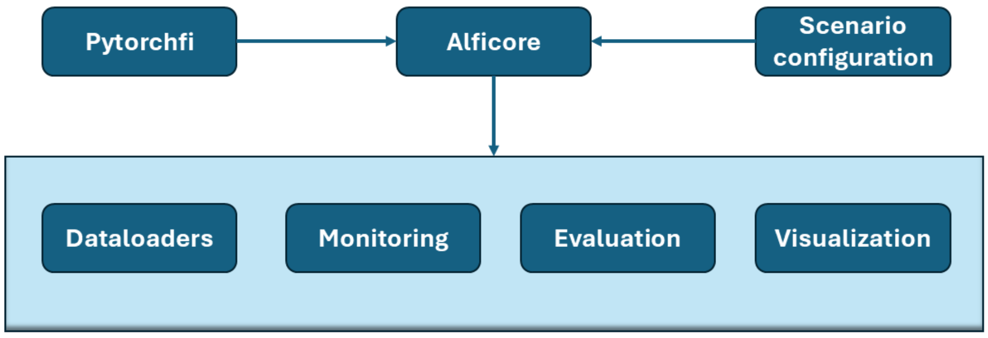


Figure 2.5: Software architecture of PyTorchALFI

faults. Table 2.3 presents the matrix rows for neuron fault injection, each targeting different components. Weight fault injection follows similar parameters as those in the table but in the first row it denotes the layer index, and in the second and third rows it specifies the weight's output and input channel, respectively. Once the faults are generated, they are stored in a binary file, along with another binary file containing details about fault locations and the original and altered neuron or weight values before and after the injection.

Overall, PyTorchALFI is a notable fault injection framework developed in PyTorch and widely used in research environments. It supports various tasks, such as identifying the most vulnerable components in neural networks, comparing model robustness, and evaluating vulnerability.

line number	ID	Description
1	Batch	number of images within a batch
2	Layer	n layer out of all available layers
3	Channel	n channel out of all available channels
4	Depth	additional index for conv3d layers
5	Height	y position in input
6	Width	x position in input
7	Value	either a number or the index of bit position

Table 2.3: Fault definition parameters for neuron fault injection.

Chapter 3

Injection Framework Implementation

This chapter offers a comprehensive overview of the design of the Injection Framework, detailing all preprocessing steps applied to the dataset and models. Section 3.1 introduces the RealNVP model being utilized, providing a detailed description of each hyperparameter. Section 3.2 outlines the dataset used in this study. Finally, Section 3.3 discusses the differences between the two implementations, with a focus on layer states and layer outputs.

3.1 Neural Network Structure

In Chapter 1, we explained the structure of RealNVP, Figure 3.1, along with its most relevant components, such as Coupling Layers. The Real-NVP network, as described in [34], incorporates physical information into its loss function, thereby enforcing the physical consistency and plausibility of the generated samples. The training of this model follows a **semi-supervised** approach: initially, it adheres to the unsupervised paradigm of RealNVP to learn the nominal data distribution, while faulty data is filtered out with human supervision to identify the training set of nominal data. Faulty data is then used to evaluate the models during testing and validation. The score samples process is illustrated in the algorithm 2. Given an input sample x, it undergoes a transformation using a flow model, resulting in y and the log determinant of the Jacobian of the transformation. This log determinant is essential for adjusting the probability density due to changes in volume caused by the transformation. After obtaining the transformed y, the probability of y is computed under a predefined probability distribution. The final score is determined by combining the log probability of y (from the distribution) and the log determinant from the transformation.

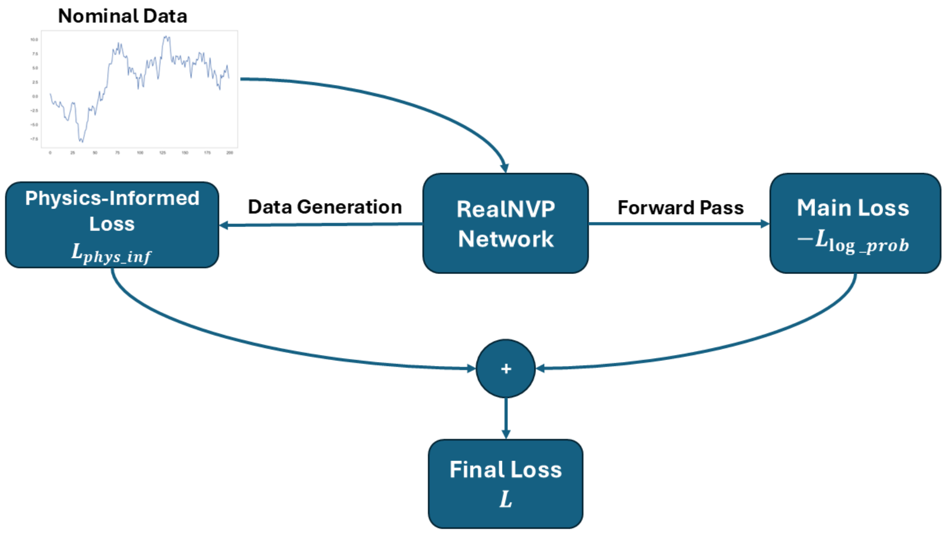


Figure 3.1: Training step for the proposed fault detection pipeline with Real NVP.

The transformation of input samples is executed through the call method of the network. This process begins by iterating through each network coupling layer in a specified direction: -1 for training and 1 for inference, as shown in Figure 3.2. A masking operation is applied to the input, where half of the input is transformed while the other half remains unchanged. The masked input is passed through two parallel layers: **scale** and **translation**.

The **scale layer** modifies the magnitude of the input data during the transformation. This serves multiple purposes:

- By scaling the input, it controls the spread of the output.
- It alters the volume of the transformed space, which must be considered when calculating probabilities.
- Scaling helps stretch or compress the input space according to the learned distribution.

Meanwhile, the **translation** layer adds a constant shift to the input data during the transformation.

- This allows the model to center the data around a new mean, which is essential for learning complex distributions where different regions of the space may require different adjustments.

Algorithm 2 Algorithmic steps for score computation within the network.

```
Require: Input $x$, Training flag $training$
Ensure: Returns log probabilities
  function ScoreSamples($x$)
      $y, \log\det \leftarrow$ Call($x$, True)
      $y \leftarrow$ distribution.log_prob($y$)
      $\log \text{probs} \leftarrow y + \log\det$
      return $\log \text{probs}$
  end function
  function Call($x$, $training$)
      $\log\det_{\text{inv}} \leftarrow 0$
      $\text{direction} \leftarrow 1$
      if $training = \text{True}$ then
          $\text{direction} \leftarrow -1$
      end if
      for $i \in \{0, \ldots, \text{num\_coupling} - 1\}$ in step direction do
          $x_{\text{masked}} \leftarrow x \times \text{masks}[i]$
          $\text{reversed\_mask} \leftarrow 1 - \text{masks}[i]$
          $(s, t) \leftarrow \text{layers\_list}[i](x_{\text{masked}})$
          $s \leftarrow s \times \text{reversed\_mask}$
          $t \leftarrow t \times \text{reversed\_mask}$
          $\text{gate} \leftarrow \frac{\text{direction}-1}{2}$
          $x \leftarrow \text{reversed\_mask} \times (x \times e^{\text{direction} \times s} + \text{direction} \times t \times e^{\text{gate} \times s}) + x_{\text{masked}}$
          $\log\det_{\text{inv}} \leftarrow \log\det_{\text{inv}} + \text{gate} \times \sum s$
      end for
      return $(x, \log\det_{\text{inv}})$
  end function
```

- It acts as a bias that helps the model adjust outputs based on learned features, enabling the learning of more complex relationships.

After applying the transformations s (scale) and t (translation), the reverse mask is applied to both s and t, ensuring that these transformations only affect the unmasked portion of the input. Then a new value for the input is computed and passed to the next layer.

We will generate and test 18 models, each with different hyperparameters. These models are created by combining three dynamic parameters: the number of coupling layers, the number of units per coupling layer, and the depth of each coupling layer.

- **Number of Coupling Layers:** Defines the number of coupling layers within the RealNVP flow network. Increasing the number of layers enhances the

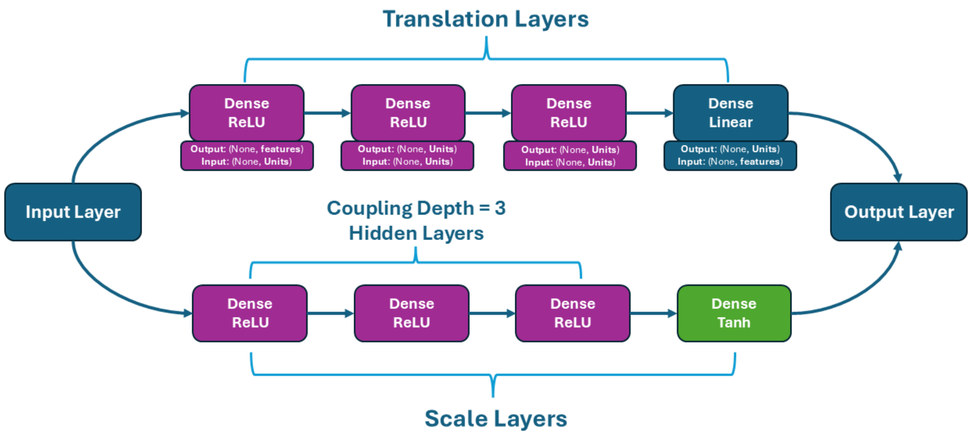

Translation Layers
Dense ReLU
Output: (None, features) Input: (None, Units)
Dense ReLU
Output: (None, Units) Input: (None, Units)
Dense ReLU
Output: (None, Units) Input: (None, Units)
Dense Linear
Output: (None, Units) Input: (None, features)
Input Layer
Coupling Depth = 3
Hidden Layers
Output Layer
Dense ReLU
Dense ReLU
Dense ReLU
Dense Tanh
Scale Layers

Figure 3.2: Coupling Layer architecture.

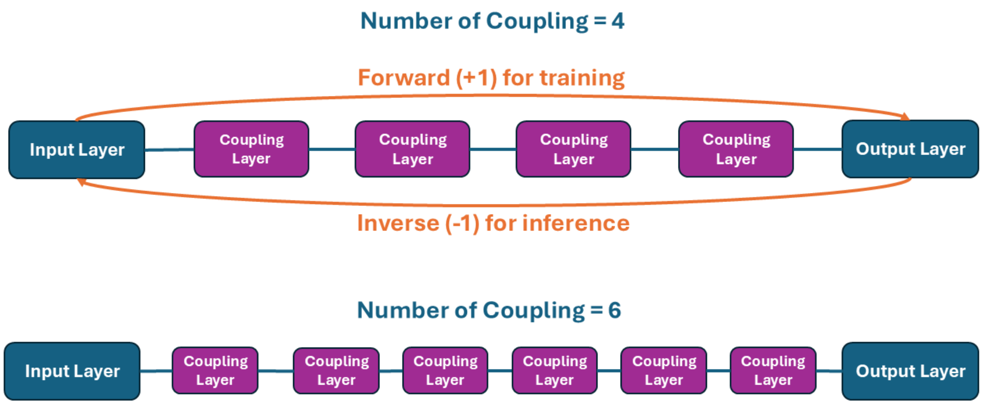


Figure 3.3: RealNVP DNN Architecture varying the number of Coupling Layers.

model's expressiveness and flexibility but also makes it more complex and computationally expensive, Figure 3.3.

- **Units per Coupling Layer:** Refers to the number of neurons in each coupling layer. Adding more neurons allows the model to capture more complex patterns, but at the cost of increased computational load.
- **Depth of Coupling Layers:** Refers to the number of hidden layers within each coupling layer. Deeper coupling layers enable the model to capture more intricate transformations.

In addition to these variable parameters, there are several fixed hyperparameters that remain constant across all models. These will not be analyzed during fault injection and are as follows:

Model	Dimension (KB)	Parameters	Weights
c4-u32-d3	5794	480472	64
c4-u32-d4	5925	488920	80
c4-u32-d5	6058	497368	96
c4-u48-d3	8711	729432	64
c4-u48-d4	8964	748248	80
c4-u48-d5	9218	767064	96
c4-u64-d3	11725	986584	64
c4-u64-d4	12147	1019865	80
c4-u64-d5	12571	1053144	96
c6-u32-d3	8865	720708	64
c6-u32-d4	8880	733380	80
c6-u32-d5	9076	746052	96
c6-u48-d3	13061	1094148	64
c6-u48-d4	13439	1122372	80
c6-u48-d5	13817	1150596	96
c6-u64-d3	17581	1479876	64
c6-u64-d4	18213	1529796	80
c6-u64-d5	18846	1579716	96

Table 3.1: Dimension of each models analyzed.

- **Epochs:** Specifies the number of complete passes through the entire training dataset. Increasing the number of epochs allows the model to train longer, potentially improving performance.
- **Batch Size:** Determines the number of training examples processed before updating the model's weights. A larger batch size can stabilize training by averaging over more samples but may slow down convergence.
- **Input Window Length:** Defines the length of the time windows used in the model to make predictions, related to the temporal dimension of our time series data.
- **Beta:** Controls the balance between different loss components. In this case, since we are not using log loss, this value is set to 0.
- **Reg Coupling:** A regularization term applied to the coupling layers to prevent overfitting by penalizing large weights.
- **Physics Samples:** Specifies the number of physics-related samples used in the loss function, ensuring that the model adheres to known physical constraints

or laws.

For training, we have selected the following values: epochs = 100, batch_size = 32, input_window_length = 10, beta = 0.0, reg_coupling = 0.001, physics_samples = 16. Number, depth and units of coupling layers are the three main parameters that influence our network size. One key consideration in selecting these parameters is that the networks need to be small and compact due to the limited space available in our application. The dimensions of each model, including the number of weights and layers, are shown in Table 3.1, which details the number of parameters in each model. Our injection process is applied to these networks, with the first step focusing on generating reliable results and reproducibility experiments. Since the training is done on one subset, we generate three different subsets for training and testing by using different split seeds. The ADAPT dataset, explained in Section 3.2, is divided into an 80% split for training and 20% for testing.

The results for a single model are obtained by averaging the outcomes from all three subsets. The Table 3.2 presents the evaluation metrics for each model, including F1 score, accuracy, AUC-ROC, AUC-PR, precision, recall, false positive rate (FPR), and true positive rate (TPR) at 95%. The results are similar across models, although models with more coupling layers tend to perform slightly better compared to those with fewer layers.

Model	$F1_{score}$	**Accuracy**	AUC_{roc}	AUC_{pr}	**fpr95**	**tpr95**	**precision**	**recall**
c4-u32-d3	0.743	0.682	0.926	0.945	0.569	0.951	0.610	0.951
c4-u32-d4	0.750	0.692	0.931	0.948	0.552	0.954	0.618	0.954
c4-u32-d5	0.742	0.679	0.927	0.946	**0.577**	0.953	0.607	0.953
c4-u48-d3	0.757	0.703	0.936	0.953	0.533	0.955	0.626	0.955
c4-u48-d4	0.746	0.690	0.942	0.958	0.546	0.943	0.618	0.944
c4-u48-d5	0.746	0.688	0.936	0.953	0.555	0.949	0.615	0.949
c4-u64-d3	0.755	0.699	0.936	0.954	0.543	0.959	0.623	**0.959**
c4-u64-d4	0.754	0.702	0.938	0.955	0.525	0.944	0.627	0.944
c4-u64-d5	0.774	0.730	0.950	0.963	**0.579**	0.954	0.653	0.944
c6-u32-d3	0.795	0.762	0.956	0.966	0.418	0.954	0.682	0.954
c6-u32-d4	0.790	0.755	**0.958**	**0.968**	0.427	0.951	0.675	0.951
c6-u32-d5	0.794	0.760	0.957	0.967	0.421	0.955	**0.680**	0.955
c6-u48-d3	**0.800**	**0.768**	0.956	0.966	0.409	**0.958**	**0.688**	**0.958**
c6-u48-d4	0.773	0.730	0.952	0.964	0.478	0.953	0.651	0.953
c6-u48-d5	0.780	0.740	0.953	0.965	0.456	0.950	0.661	0.950
c6-u64-d3	0.781	0.741	0.956	**0.968**	0.459	0.955	0.660	0.955
c6-u64-d4	0.785	0.746	0.953	0.965	0.448	0.955	0.666	0.955
c6-u64-d5	**0.795**	0.763	0.957	0.967	0.408	0.946	0.687	0.946

Table 3.2: Results for the main metrics utilized.

3.2 Dataset

The training and testing of models are applied to the ADAPT [35] (Advanced Diagnostic and Prognostic Testbed) dataset, an Electrical Power System (EPS) dataset created by NASA in a laboratory setting to simulate various types of faults. EPS loads in an aerospace vehicle may include crucial subsystems such as avionics, propulsion, life support, and thermal management systems.

Given a dataset D, we aim to create N splits as shown in Figure 3.4; in this case, $N = 3$. This results in three different sets for training and testing, allowing us to train three models with the same hyperparameters on three distinct training sets, which are then evaluated on their corresponding test sets. The final score for a specific metric is calculated as the average of the scores obtained from each model. Therefore, when we refer to the results from a model, we mean the average across these different splits. This approach can be defined as Ensemble Training, specifically using Bagging (Bootstrap Aggregating), where multiple instances of the same model type are trained on different random subsets of the data. Their predictions are then averaged or combined through majority voting.

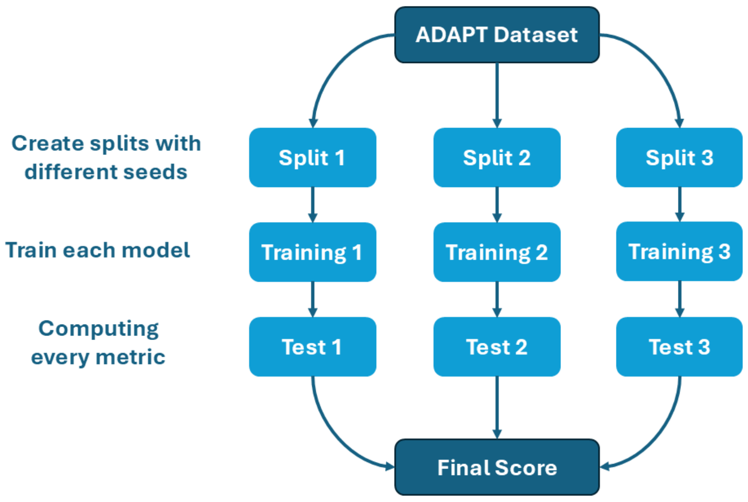


Figure 3.4: Process of splitting, training and testing to obtain the final score.

All of these operations are conducted for various reasons:

- **Improved Generalization:** Splitting the dataset into multiple training and validation sets helps the model generalize better to unseen data.
- **Model Performance Estimation:** A single train-test split may provide a misleading estimate of the model's performance on unseen data, increasing

the risk of overfitting. Using multiple splits and averaging results allows for a more accurate estimate of the model's real-world performance, as it has been validated across multiple data subsets.

- **Bias-Variance Trade-off:** Utilizing multiple dataset splits balances the trade-off between bias and variance. Training on various splits and averaging the results helps reduce both high bias (**underfitting**) and high variance (**overfitting**), as it minimizes dependence on any single data subset.
- **Improved Accuracy:** Ensemble learning can significantly enhance predictive performance by combining multiple models. Individual models may have weaknesses, but when aggregated, they often complement each other, resulting in more accurate predictions.

As explained in the previous Section 3.1, we will test 18 models using the SDC metric, where only correctly predicted samples undergo the injection process. All these models use different subsets for this specific task, since they have very different prediction power and so different accuracy value and so a different correct sample to start from. To facilitate the comparison and to have more reliable evaluation results we compute from the different splits, the absolute test subset in which all the elements inside them are correctly predicted by all the models. This is done for several reasons. We don't want to have variations in the dataset that could unfairly favor one model over another and that the differences in performance are due to the models themselves and not due to differences in the data they were tested on. The dimesion of each subsets split are reported in this Table 3.3.

Split Type	**Split**	**Elements**	**Features**	**Classes**	**% class 0**	**% class 1 (anomalies)**
Training	0	2179	894	2	51.629	48.370
	1	2179	894	2	51.858	48.141
	2	2179	894	2	51.629	48.370
	Average	2179	894	2	51.710	48.290
Test	0	2179	894	2	51.629	48.370
	1	2179	894	2	51.858	48.141
	2	2179	894	2	51.629	48.370
	Average	2179	894	2	51.710	48.290
Subsets	0	167	894	2	**13.365**	**82.634**
	1	360	894	2	**31.666**	**68.333**
	2	93	894	2	**7.526**	**92.473**
	Average	/	894	2	**51.620**	**81.150**

Table 3.3: Dimension of train, test and subsets dataset for each split and average.

3.3 Framework

In this Section, we will explore the implementation and operation of the framework, with a focus on the two enhanced injection methods introduced in Section 2.2.4: Layer States and Layer Outputs. Section 3.3.1 provides background on single-precision floating-point format, while Section 3.3.2 covers the metric used in our experiments. The parameters for Layer States and Layer Outputs are detailed in Sections 3.3.3 and 3.3.4, respectively. This introduction explains the common processes shared between both injection modes. First, given a pre-trained model M, we aim to run multiple fault injection experiments, determined by the $number\_of\_faults$ parameter (which will be analyzed in chapter 4), and then average the scores for each metric across all experiments. This approach helps mitigate random variations in the model, as the manifestation and impact of faults can vary. By running several experiments and averaging the results, we minimize the influence of any single outlier or extreme case, allowing us to assess the model's robustness across a range of fault scenarios.

In each experiment, we reset the network state to its original form after every fault injection iteration. The **Target** parameter allows us to specify whether we are injecting faults into the layer states or layer outputs. Once the model has been injected, we compute the score and check for any **NaN** (Not a Number) or **Inf** (Infinite) values. This helps us detect any system failures. For simplicity, we assign an SDC (Silent Data Corruption) rate of 1, meaning the system is considered to have completely failed if NaN or Inf values are found.

Layer states and layer outputs parameters are discussed in their relative Section. Here we show the common procedure for both targets for modifying the values of the tensor after deciding the parameters of each configuration. For a fault experiment the algorithm follow these steps:

1. For each weight layer or output to be modified, we select the "**Type**" of fault to inject-either zeros, random, or bitflips.
2. After selecting the percentage of values to inject using the "**Amount**" parameter, the algorithm checks the injection type.
3. For zeros and random, the process is the same. It selects random indices from the tensor and applies a *tensor_scatter_nd_update* operation, Figure 3.5, which creates a new tensor by applying sparse updates to the input tensor, similar to assigning values at specific indices.
4. For bitflips, the procedure is similar. After selecting the indices, the function converts 32-bit floats into their binary integer representation, flips the specified bits at the provided positions, and returns both the updated values and the original bit values before the flip.

5. The original values are used to calculate the percentage of negative-to-positive flips, positive-to-negative flips, as well as flips from 0 to 1 and 1 to 0.

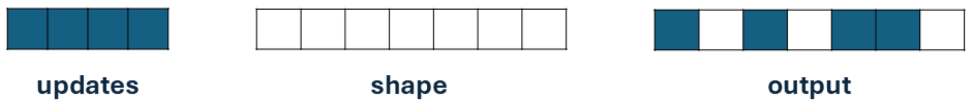


Figure 3.5: Graphic representation of tensor_scatter_nd_update operation.

3.3.1 The Bitflip Operation in Single-Precision Floating Point Format

The Bitflip operation applies to values in single-precision floating-point format [36], a computer number format that typically occupies 32 bits in memory. This format, part of the IEEE 754 standard and officially called binary32, represents a wide dynamic range of numeric values by using a floating radix point. The binary32 format specifies:

- **Sign** bit: 1 bit (S)
- **Exponent** width: 8 bits (E)
- **Significand** precision: 24 bits (with 23 explicitly stored) (F)
- The **Value** of a floating-point number is generally defined by the formula: $Value = (-1)^S * 2(E - 127) * (1.F)$

This formula allows for a precision of approximately 6 to 9 significant decimal digits. For instance, a decimal value with up to 6 significant digits, when converted to IEEE 754 single-precision format as a normal number, will match the original value if converted back to a decimal with the same precision.

In this format:

- The sign bit (S) indicates the sign of the number.
- The exponent (E) is an 8-bit unsigned integer ranging from 0 to 255, stored in a "biased" form. This approach simplifies comparisons by storing the exponent as an unsigned value, making it straightforward to represent both very large and very small numbers. When interpreted, the exponent is adjusted by subtracting the bias (127) to yield a signed range.
- The true significand (or mantissa) of normal numbers comprises 23 fractional bits to the right of the binary point, with an implicit leading bit set to 1.

An accompanying figure 3.6 illustrates the bit representation of a floating-point number in binary32 format.

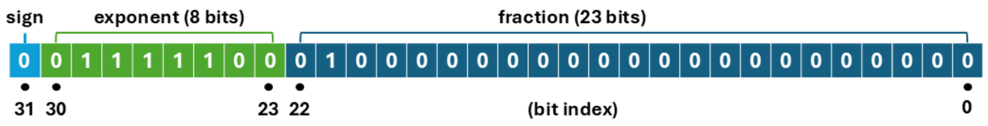


Figure 3.6: Bit representation of a floating-point number in binary32 format.

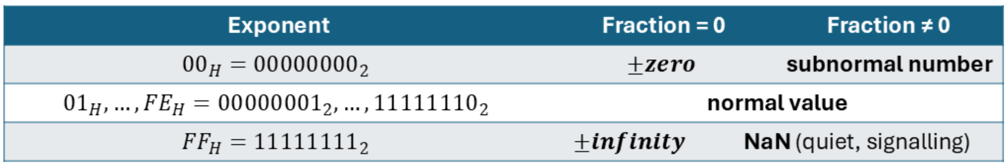
The behavior of floating-point numbers changes when specific bit patterns are applied to the exponent, Figure 3.7:

1. Exponent is all zeros (E = 00000000): This pattern represents a denormalized number or zero.

2. Exponent is all ones (E = 11111111): When this pattern is used if the fraction (F) is zero, the value represents infinity (positive or negative, depending on the sign bit) or if the fraction (F) is non-zero, the value is NaN (Not a Number), indicating an undefined or unrepresentable result.

Understanding these special cases is critical for our study, as bit flips in a network may cause values to switch to these states. Identifying and addressing such transitions is essential for maintaining numerical accuracy and preventing errors in computation.

The Figure 3.8 illustrates the percentage of bits set to 1 across all network weights. We observe a steady distribution in the mantissa bits. However, the exponent bits, particularly from bits 24 to 30, display different behaviors. Bits 29, 28, and 27 are consistently set to 1, while bit 30 is always set to 0. Flipping bit 30 from 0 to 1 can lead to severe consequences, as it can result in the value becoming either infinity or NaN (Not a Number). As we will demonstrate in the experiment Section 4, most fatal faults arise from setting this particular bit to 1.



Exponent	Fraction = 0	Fraction ≠ 0
$00_H = 00000000_2$	$\pm zero$	**subnormal number**
$01_H, \ldots, FE_H = 00000001_2, \ldots, 11111110_2$	**normal value**	
$FF_H = 11111111_2$	$\pm \boldsymbol{infinity}$	**NaN** (quiet, signalling)

Figure 3.7: The behavior of floating-point numbers.

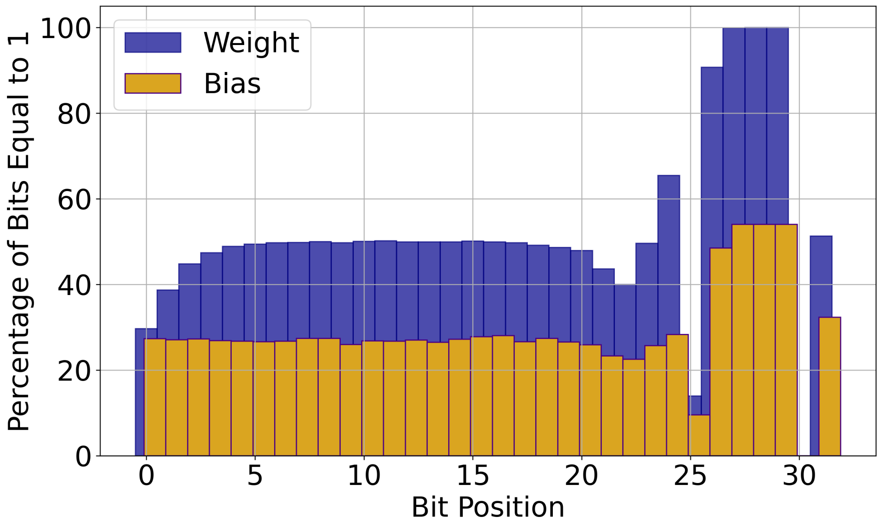


Figure 3.8: Distribution of bits equal to 1 for each bit position in Physics-Informed Real NVP Model variables.

3.3.2 Metric

As outlined in Section 1.2.3, various metrics are used for evaluating anomaly detection tasks. In our evaluation process (detailed in Section 3.1), we rely on the following metrics:

- **Precision:** The ratio of correctly identified positive instances to the total instances predicted as positive. Defined as $P = \frac{TP}{TP+FP}$. where TP is True Positives and FP is False Positives.

- **Recall** (also known as Sensitivity or True Positive Rate): The ratio of correctly identified positive instances to the actual positive instances. Calculated as $R = \frac{TP}{FN+TP}$, where FN are False Negatives.

- **AUC-ROC (Area Under the Receiver Operating Characteristic Curve):** Measures the area under the ROC curve, which plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold levels.

- **AUC-PR (Area Under the Precision-Recall Curve):** Measures the area under the Precision-Recall curve, which is particularly useful for evaluating performance on imbalanced datasets.

- **F1 Score:** The harmonic mean of precision and recall, calculated as $F1_{score} = 2\frac{P+R}{PR}$.
- **Accuracy:** The overall ratio of correctly classified instances to total instances.
- Additionally, **FPR** and **TPR** at 95% are used as threshold-specific metrics to assess model behavior.

To assess resilience and robustness of machine learning models under fault conditions, various papers adopt specific metrics: **SDC Rate (Silent Data Corruption)**: used in [29] and [19], the SDC rate evaluates resilience by measuring the fraction of injected faults that lead to incorrect outputs (i.e., silent data corruptions). In classification tasks, an SDC is any misclassification. This metric re-evaluates accuracy after fault injection, focusing on cases where previously correct predictions become incorrect.

SDE Rate (Silent Data Error): [18] uses the SDE rate to measure the proportion of faults that cause errors in the Deep Neural Network (DNN). While related to SDC rate, the precise calculation method for SDE rate may differ.

PVF (Parameter Vulnerability Factor): Introduced in [37], PVF quantifies parameter-level vulnerability by calculating the probability that corruption in a specific model parameter will lead to an incorrect model output, Figure3.9, in contrast to SDC Rate that works at model-l evel. This metric answers the question, "How likely is parameter corruption to result in an incorrect model output?" For instance, PVF might measure the probability of errors in specific tasks such us click prediction or image classification. We believe that standardizing these diverse metrics would facilitate comparisons the reliability of AI systems.

In our study, we primarily use the **SDC rate** [19] to evaluate overall model performance following fault injection. However, our framework allows for selective fault injection at specific network layers or on individual activation functions. By isolating these components, we can conduct a parameter-level evaluation of our network elements, providing a more granular understanding of model robustness similar to PVF.

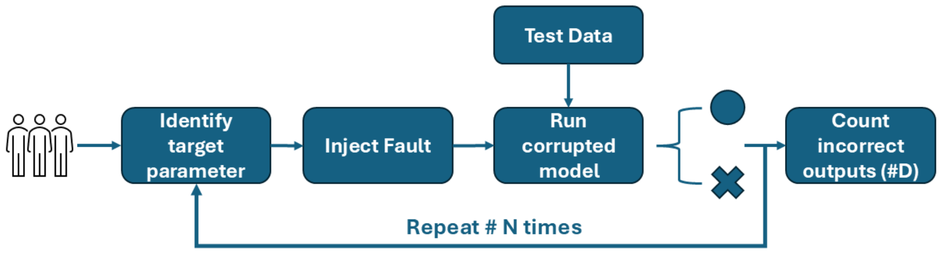


Figure 3.9: Fault injection experiments flow.

3.3.3 Layer States

The layer states injection process is controlled by several parameters:

- **Mode:** This parameter determines the number of weight layers used based on the depth of the coupling layers in each network, as seen in the Table 3.4. The percentage of layers to be injected can be adjusted using the $Mode$ parameter. We selected five different percentage ranges: 20%, 40%, 60%, 80%, and 100% (where all layers are selected)
- **Variable:** This defines the type of injection based on the network layers. It can target *bias* layers, which help the activation function fit the data more effectively, or *weight* layers, which influence neuron connections between layers. Alternatively, both can be selected by choosing *all*.
- **Type:** The type of injection can be selected from three options: *zeros* (sets the weights to zero), *random* (applies Gaussian noise to the weights), or $bitflips$ (flips the bits of the weight values).
- **Amount:** Similar to the $Mode$ parameter, this allows you to specify the injection rate for each layer with an input percentage.
- **Bit:** This parameter specifies which bit position will be flipped. If set to N, a random bit position will be selected.
- **Direction:** Available only when $bitflips$ is selected as the injection type. If set to 0, the flip changes bits from 0 to 1; if set to 1, it flips from 1 to 0. Selecting *all* will apply flips in both directions.
- **Sign:** Also only applicable to $bitflips$. If set to 0, only positive values are flipped; if set to 1, only negative values are flipped. Choosing *all* allows for both positive and negative flips.

All these parameters can be combined to create different injection configurations. Some configurations can be more generalized to evaluate the overall behavior of the model, while others can be more specific to target a particular component. As mentioned, we aim to modify the network's state values before inference to effectively assess the network's ability to generate representative weights and connections between layers.

Parameter	**Value**	**Description**
Mode	*percentage*	Number of layers to inject
Variable	*bias weight all*	Type of layers to target
Type	*zeros random bitflips*	Type of injection
Amount	*percentage*	Injection rate
Bit	*bit_position N*	Bit position from 0 to 31
Direction	0 1 *all*	Direction of flips
Sign	0 1 *all*	Sign of bitflips

Table 3.4: Parameters for layer states configurations.

3.3.4 Layer Outputs

The layer outputs injection shares several common parameters with layer states, such as **Type**, **Amount**, **Bit**, **Direction**, and **Sign**. The differences are explained below:

- **Mode:** In this case, the mode allows specifying the exact position of the coupling layer using a real number (e.g., 1, 2, 3...). This number must be equal to or less than the total number of coupling layers. You can also select a random layer using *N* or apply the injection to all layers by choosing *all*.
- **Variable:** This parameter allows you to choose between two types of nodes within the coupling layer. Selecting *scale* targets the scale layers, *translation* targets the translation layers, and choosing *all* applies the injection to both.
- **Activation:** Both the scale and translation layers use different activation functions. The hidden layers of both have *ReLU* activation. The scale layer uses *tanh* activation in its final layer, while the translation layer uses a *linear* activation. You can choose to inject into all of these by selecting *all*.
- **Method:** This parameter specifies where the injection occurs. You can inject at the final output of the coupling layer with the *partial* option, or you can inject into all the sublayers of the coupling layer with the *complete* option. A graphical representation showing these differences is provided in Figure 3.10.

Once again, all these parameters can be combined to create different levels of ablation, either by focusing on specific activation functions or network depth, or by applying a more general injection. The Table 3.5 below lists the major injection configurations for layer outputs, highlighting changes in the **Variable**, **Activation**, **Method**, and **Type** of injection.

This choice of picking between partial or complete method of injection was made to analyze the behavior of scale and translation layer propagation. As we will see in

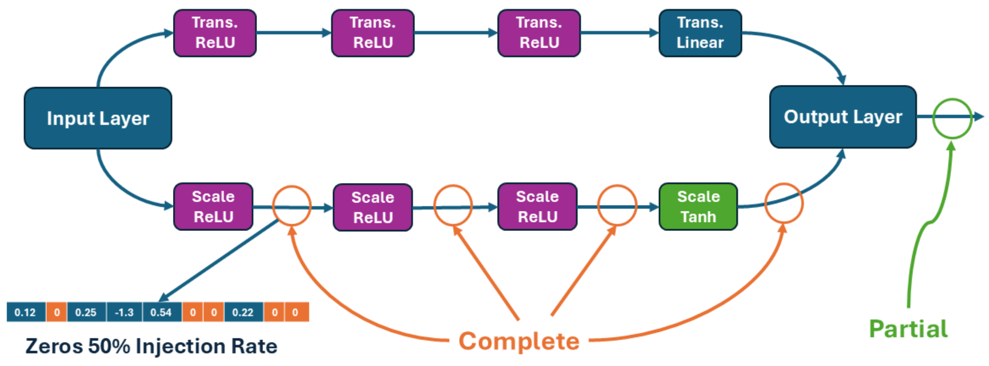


Figure 3.10: Difference between partial and complete method parameters.

Parameter	Value	Description
Mode	*layer_position random all*	Position of the coupling layer
Variable	*scale translation all*	Type of nodes within coupling layer
Type	*zeros random bitflips*	Type of injection
Amount	*percentage*	Injection rate
Bit	*position N*	Bit position
Direction	0 1 *all*	Direction of flips
Sign	0 1 *all*	Sign of bitflips
Activation	*ReLU tanh linear all*	Target a different activation function
Method	*partial complete*	Inject at the output of the layer or inside

Table 3.5: Parameters for layer states configurations.

the experimental Section, scale operations are more resistant to complete injection but more vulnerable to intermediate injections, while translation layers show the opposite behavior, being more robust to partial injections.

Since computing the output from an input involves multiple steps, we focus only on the outputs of the scale and translation layers, ignoring the masked operations and the computation of the next layer's input.

Considering that layer outputs injection is highly dynamic, we have utilized Keras version 3 [38] to create a custom model from our networks. The Model class allows us to group layers into an object with both training and inference features, inheriting all the characteristics of the original models while enabling us to add new functionalities. This is essential for modifying the call method, which handles all the operations and steps an input goes through. The process of modifying the values specified by the *Type* parameter - whether zeros, random, or bitflips - remains the same as explained earlier in Section 3.2.

Chapter 4

Experimental Studies

In this chapter, we present the experimental results from different fault injection configurations applied to our models. For layer states injections, we performed approximately 2,000 experiments per model, totaling 38,000 experiments across all models. In Section 4.1, we begin by analyzing the behavior of layer states within a single model, followed by a cross-model analysis. For layer outputs injections, we ran around 7,000 experiments per model, resulting in a total of 126,000 configurations. In Section 4.2, we apply the same methodology to examine the behavior of layer outputs targets. Finally, in Section 5.1, we summarize the findings of this study.

4.1 Layer States

4.1.1 Single Model

Our initial experiment aimed to determine the number of faults needed per injection run to obtain a reliable SDC Rate. We started by testing with 10 and 20 faults per injection. The final metric was calculated by averaging results across all injections. In particular, we observed a minimal difference in performance reduction between 10 and 20 faults, Figure 4.1. Based on this finding, we proceeded with 10 faults per injection to optimize for time efficiency.

As explained in Section 3.2, we will calculate our SDC metric in two ways: an absolute SDC metric, which uses the same set of correctly classified test samples across all models, and a relative SDC rate, calculated individually for each model. Since each model has a unique accuracy and thus a distinct set of correctly classified samples, the relative SDC rate will vary by model. The resulting SDC rate will be determined as the average of the relative SDC rates across each model's test subset.

The Figure 4.2 illustrates the differences between using two types of SDC rates for

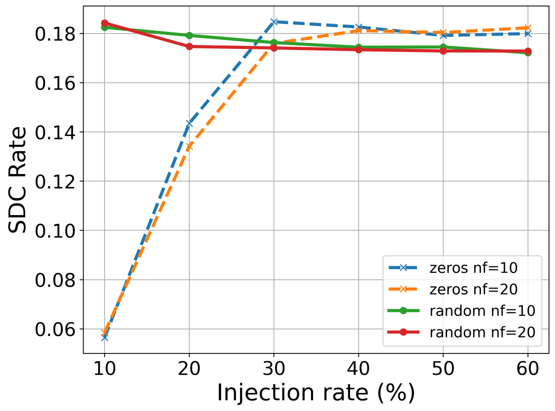
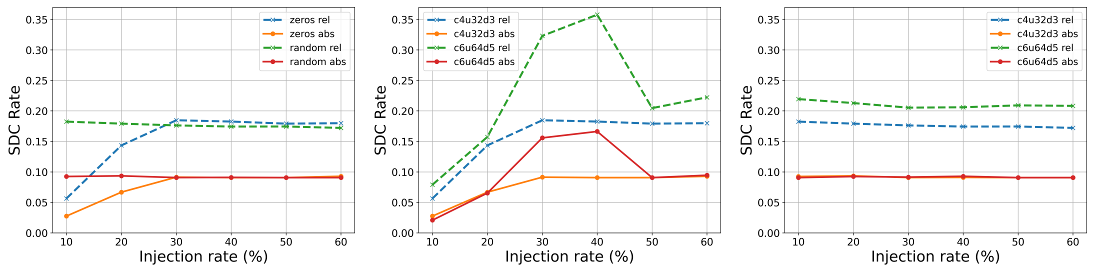


Figure 4.1: Difference between 10 and 20 faults on zeros and random Injection.



(a) Zeros/Random injection on **one model**.

(b) **Zeros** injection on two models.

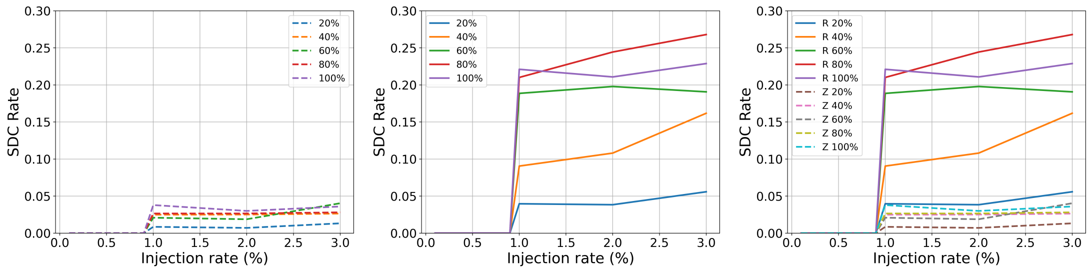
(c) **Random** injection on two models.

Figure 4.2: Difference between relative and absolute SDC Rate.

both random and zero injections. The results for each are quite distinct. Models that use their respective subsets for evaluation tend to produce different results compared to models evaluated using the same subset. However, using the same subset often yields similar results, which can help highlight any differences. After deciding to use a number of faults equal to 10 and SDC relative ($SDCrel$) for single models, and SDC absolute ($SDCabs$) for comparing multiple models, we will now focus on the performance of a single network with the smallest parameters: $num\_coupling = 4$, $depth\_coupling = 3$, and $units\_coupling = 32$. After analyzing the performance of a single model, we will then compare multiple models.

The first key observation is when the model begins to show signs of performance degradation. The Figure 4.3 demonstrates that the SDC rate starts to change at approximately a **0.8%** injection rate, with varying target layer percentages (20%, 40%, 60%, 80%, and 100%). It displays both random and zero injections, with the random injections showing higher values for the SDC rate.

After identifying when the model begins to experience performance degradation,



(a) Zeros Injections. **(b)** Random Injections. **(c)** Zeros/Random Injections.

Figure 4.3: Start of performance degradation at States Injections.

we continue increasing the injection rate up to 100%. We observe that zero injections show a saturation in performance scores around a 30% injection rate, whereas random injections reach saturation even earlier, around 10%, as shown in Figure 4.4. It is crucial to monitor these behaviors to ensure reliable results. To understand why this behavior occurs, we must investigate what happens within the network after the model's input passes through the injected weights.

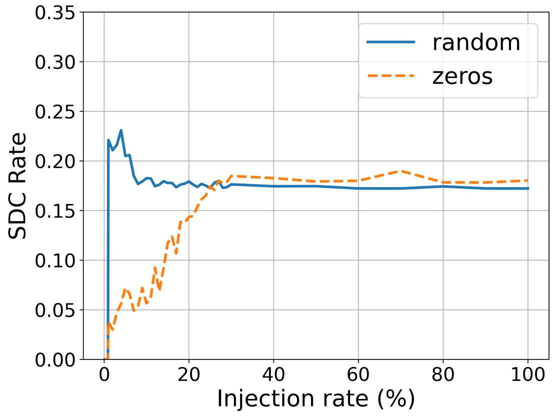
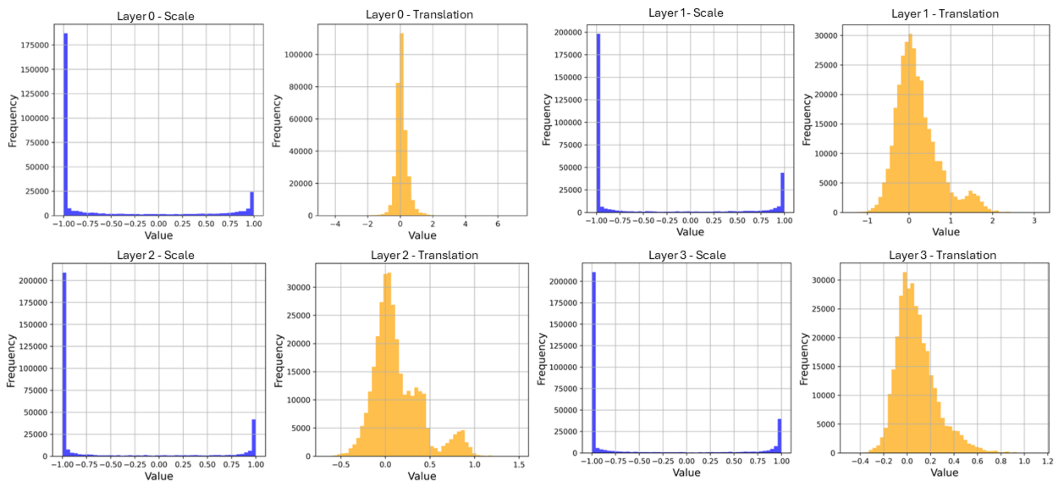
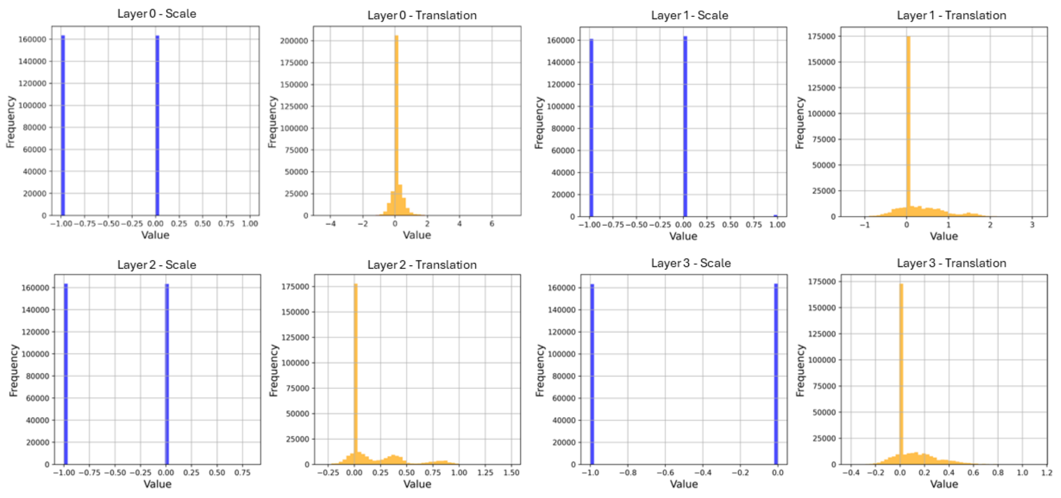


Figure 4.4: Random and Zeros States Injection from 0 to 100.

Some things to consider are that even if all the network's weights are set to zero or assigned random values, the network will still produce an output able to compute scores due to its internal architecture, as described in Section 3.1. In the provided plots, the input passes through each layer, from coupling layer 3 to coupling layer 0 and it is important to note that the final scale layer uses a tanh activation function, which maps values to the range [-1, 1], while the final translation layer employs a linear activation function. When discussing *zeros injection* and *random injection*, we refer specifically to the injection in the layer states configurations - impacting the weights and biases of the network - rather than directly altering the network's



(a) Normal Model



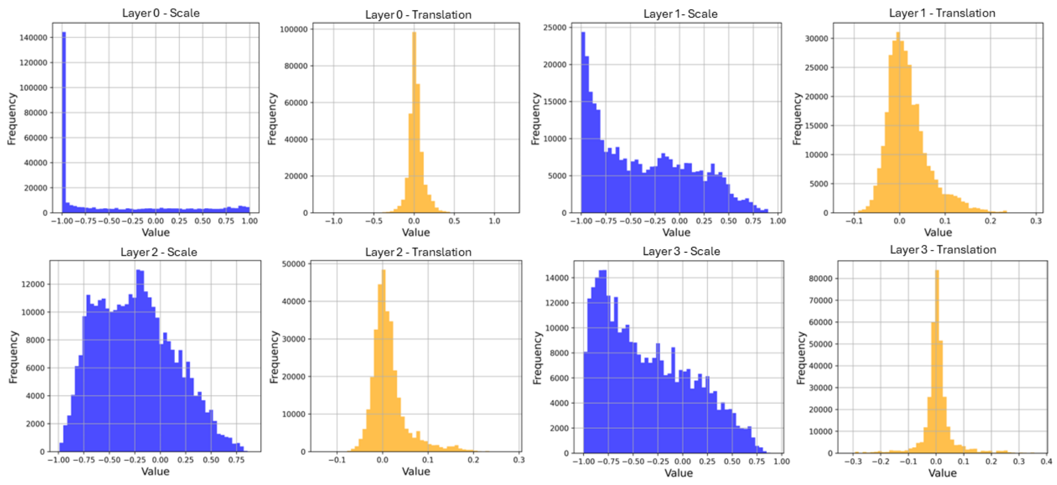
(b) Normal Model Masked

Figure 4.5: Distribution of scale and translation layers in normal condition on masked and unmasked outputs.

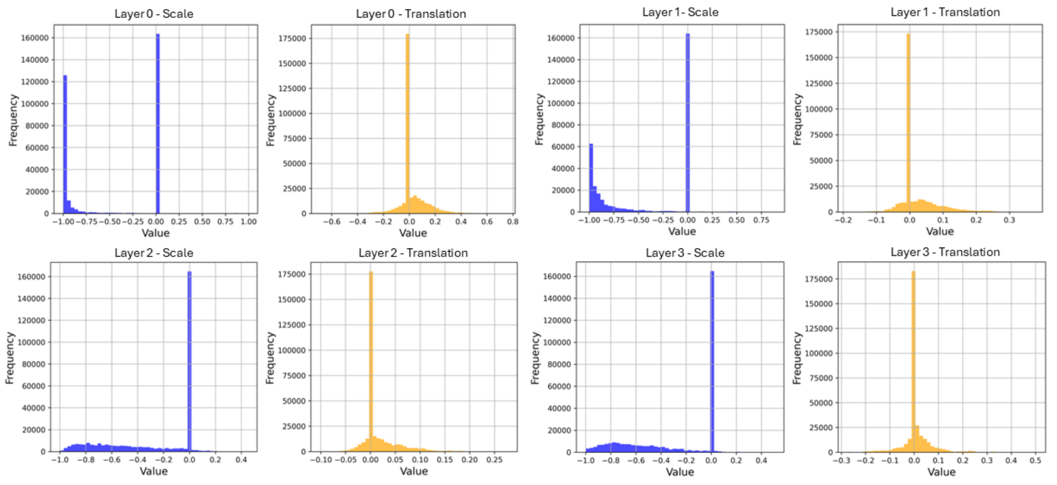
output.

To analyze this behavior, we first evaluated the network output without injection. We then selected configurations where both zero and random injections caused the network to saturate, for example, around 50%.

Without injection, Figure 4.5, the input propagates through the scale and translation layers, followed by a masked operation that modifies half of the input while leaving the rest unchanged. Section 4.2 presents experimental results demonstrating that the scale and translation layers exhibit different behaviors under zero and



(a) Zeros



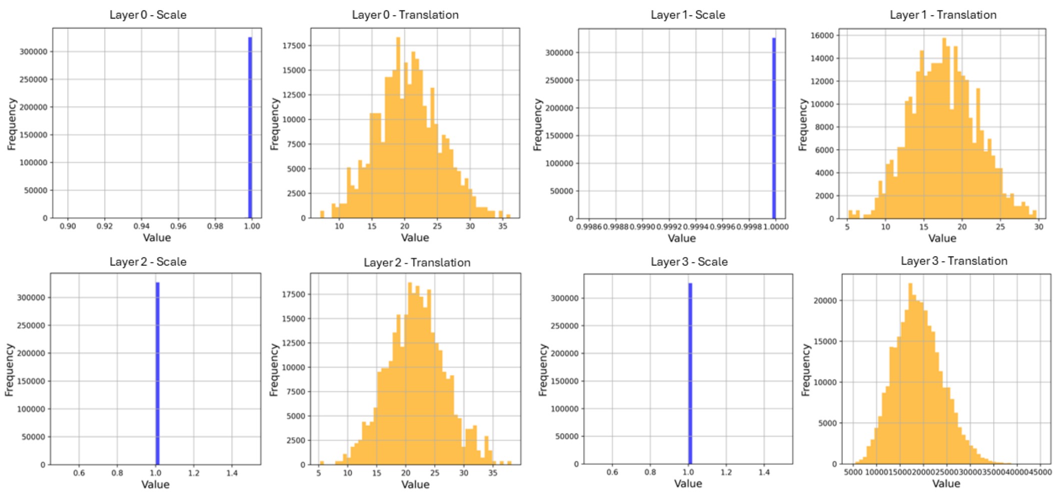
(b) Zeros Masked

Figure 4.6: Distribution of scale and translation layers with 50% Zeros Injections on masked and unmasked outputs.

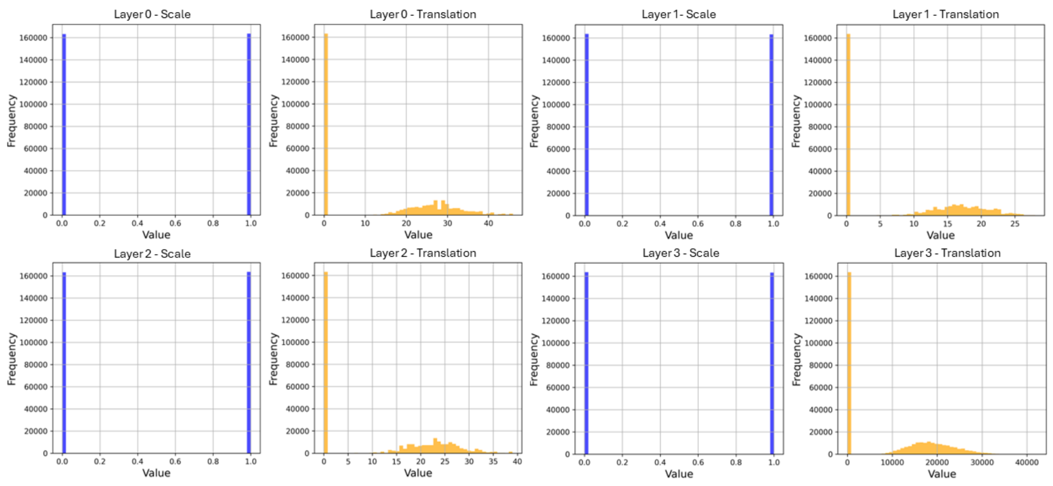
random injections.

When zeros are injected into the layer states:

- The scale layer produces mixed values, primarily oscillating between zero and one.
- As shown in Figure 4.6, the tanh activation saturates the scale values near -1.
- After the masking operation, these values are predominantly distributed between -1 and 0. Ideally, the scale operation should leave part of the input



(a) Random



(b) Random Masked

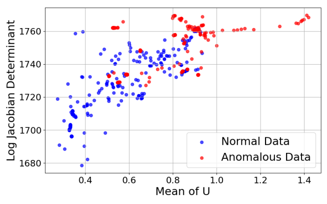
Figure 4.7: Distribution of scale and translation layers with 50% Random Injections on masked and unmasked outputs.

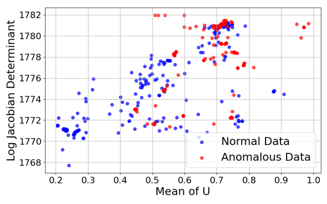
unchanged. Additionally, due to the saturation effect, the translation layer shifts the output to higher values, as ReLU only modifies negative inputs.

When random values are injected into the layer states:

- The final masked scale output is confined to values near 0 or 1. As observed in Figure 4.7b, the translation layer outputs higher parameters, deviating significantly from the peak around zero.

The observed behavior could explain the saturation issue associated with the


(a) Seed = 0.


(b) Seed = 1.

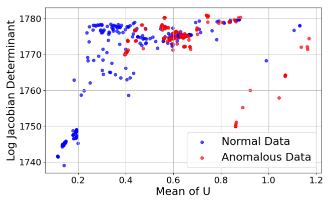

(c) Seed = 2.

Figure 4.8: Normal vs Anomalous Data in different datasets.

use of the tanh activation function, which saturates values near -1 and 1. Another potential explanation for these behaviors is the type of seed used during dataset splitting. In the Figure 4.8, **U** represents the latent space output, which corresponds to the transformed input data. The mean of U provides an aggregated measure of the latent representation for each sample and can be utilized for anomaly detection. The **Log Jacobian Determinant** quantifies the volume change induced by the transformation of data from the input space to the latent space. It plays a critical role in evaluating the probability density of the input data. The determinant provides insight into how the transformation affects the density of points in the latent space, offering a measure of the anomalousness of the transformation for a given input. While the mean of U may not always be a reliable predictor, it can still aid in distinguishing between normal and anomalous classes under certain conditions. This challenge is illustrated again in 4.8, where overlapping classes impede clear separability.

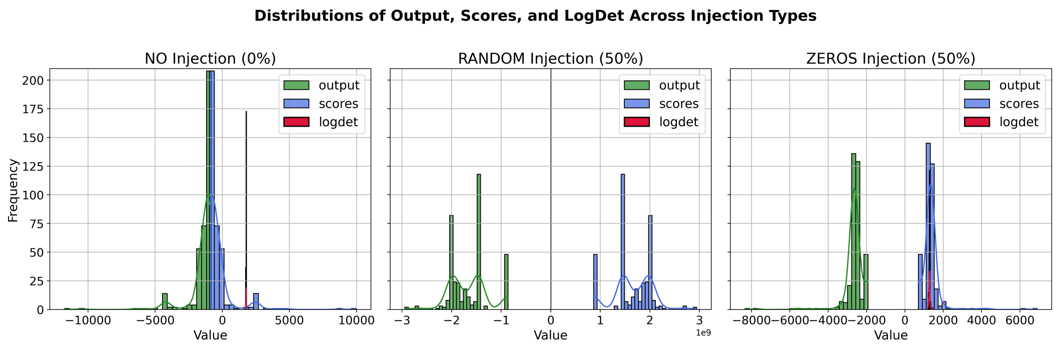
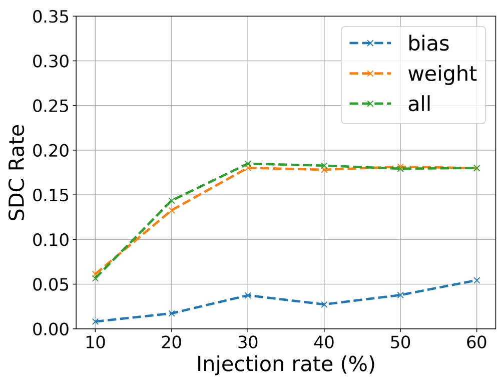
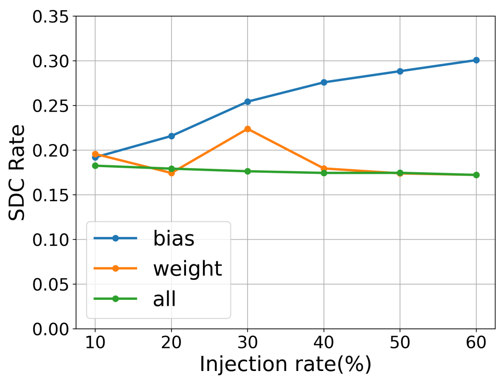


Figure 4.9: Distributions of scores, outputs, and log-determinants for three models: a baseline model, one with 50% random value injection, and one with 50% zero-value injection.

In Figure 4.9, we compare the distributions of the network's output, scores, and log-determinant values for the following configurations: No injection, zeros injection at 50% and random state injection at 50%. This shows the ability of input alone to capture the relevant information about anomalies and perform in the way presented.



(a) Zeros



(b) Random.

Figure 4.10: Layer States Injection with mode = 100%.

Variables (Bias, Weight, All): Let's begin by analyzing the results of our injections. Our goal is to examine how different variables - *Bias* and *Weight* - affect the outcome. Both variables are collectively referred to as *all*.

The Figure 4.10 illustrates the SDC rate (Silent Data Corruption) in various injection rates (expressed as a percentage). In this scenario, the injection set targets 100% of the layers (mode = 100%). We observe that injecting zero values into the bias causes a minor performance drop compared to weights. In contrast, introducing random noise into the bias results in significantly greater degradation. These behaviors act in an opposite way. This phenomenon can be explained by the role bias plays in neural networks. Bias is typically added after the weighted sum of inputs. Changing bias to random values disrupts the controlled shift in the activation function, introducing global perturbations to the output and leading to errors. However, injecting zeros into the bias removes the offset but does not affect the relative contributions of the inputs from the weights. Many activation functions, such as ReLU (used in this case), can still function effectively under these conditions. This makes zeroing the bias less harmful, especially when input alone can determine the output. In contrast, weights exist in larger quantities and are distributed across numerous connections. Random injections into weights typically affect only a subset of the overall weight matrix, which may not drastically disrupt the model's output due to redundancy and averaging effects. However, injecting zeros into weights is equivalent to separate specific connections entirely. This can significantly degrade the model's ability to learn or infer patterns, effectively pruning parts of the network and potentially collapsing complex learned representations.

Fixed Amount: We will continue analyzing the bias and weight variables, but from a different perspective. In this analysis, the x-axis represents the percentage of layers selected for the injection ranging from [20%, 40%, 60%, 80%, 100%], while the injection rate is fixed.

In the Figure 4.11, we present both zero-value and random-value injections simultaneously, highlighting which type of injection is most critical. It is also important to note the model's saturation at higher injection rates, which can further impact performance shown in the previous experiment.

With a small injection rate during random-value injections, the performance of both variables remains nearly identical. However, as the injection rate increases, bias begins to outperform weights, as previously demonstrated.

In contrast, with zeros-value injections, with an injection rate of 10% the performance of weights remains the same for all the layers target, meanwhile after 20% injection rate and around 60% and 80% the weights variable presents a steep jump in decreasing performance, while the performance of bias remains steady across all percentages of selected layers.

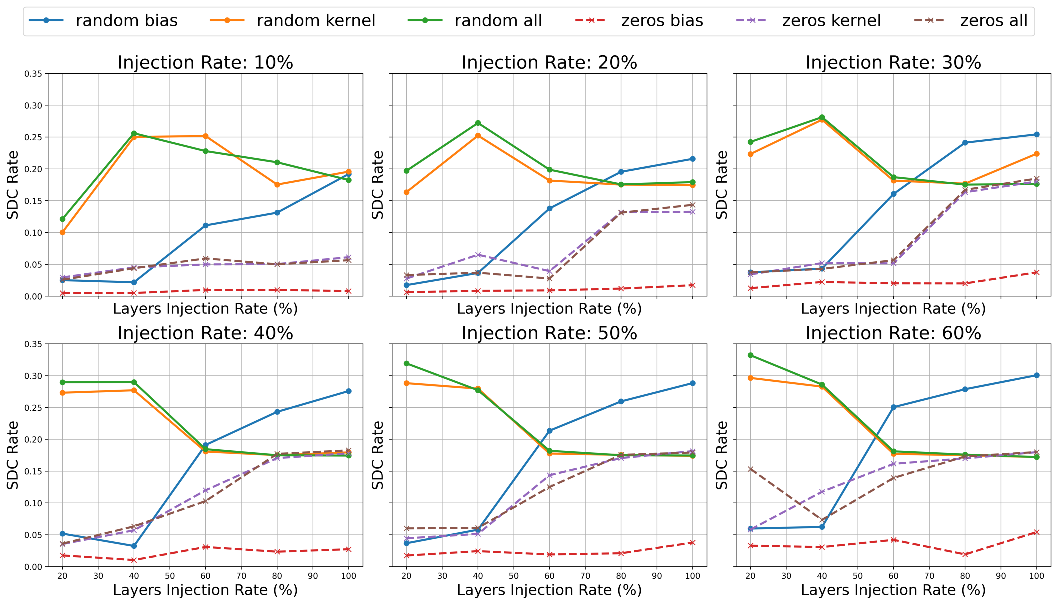
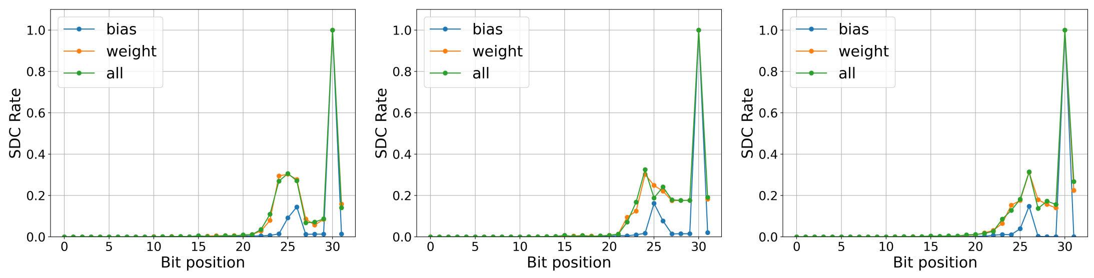


Figure 4.11: SDC Rate at different layers percentage for zeros and random injections.



(a) Sign all (all flips). **(b)** Sign 0 (positive flips). **(c)** Sign 1 (negative flips).

Figure 4.12: Bitflips State Injections for **'all' direction** on different variables.

Bitflips: Let's now explore a different type of injection: bit flips. In Section 3.3.1, we previously demonstrated the bit-flip operation in single-precision floating-point format. Here, we will first analyze the Silent Data Corruption (SDC) rate, as done earlier, followed by examining the rate of negative and positive flips, as well as flips from 0 to 1 and 1 to 0. All bitflip injections were performed with a fixed injection rate of 10%, targeting all layers (mode = 100%). In this scenario, the steady saturation state is not observed. The direction of the flips is defined as follows:

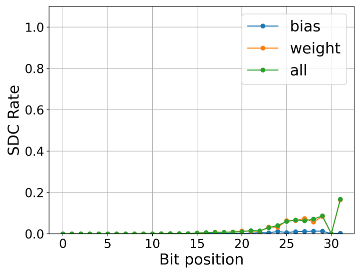
- 0: The bit transitions from 1 to 0
- 1: The bit transitions from 0 to 1
- All: Both directions are randomly chosen

Similarly, the sign configuration is specified as:

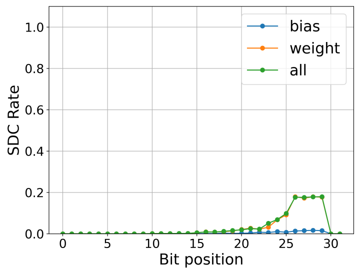
- 0: Only positive values can be flipped
- 1: Only negative values can be flipped
- All: Both positive and negative values can be flipped

The previously presented weights distribution highlighted the number of 1s and 0s across all weights and biases. We expect a system value of SDC Rate = 1 at the 30th bit flip since the value of this bit is 0 in all scenarios. The results indicate no performance degradation for flips affecting the mantissa bits. However, performance degradation begins at the 20th bit and follows a Gaussian distribution from bit 20 to 31, peaking at the 25th bit in the exponents as seen in Figure 4.12.

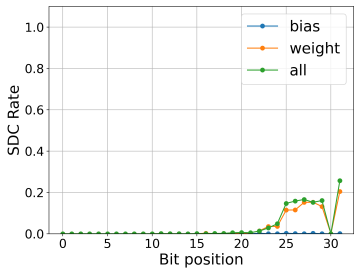
From a variable perspective, bias has a minimal impact on performance, suggesting that the network behavior is more influenced by the weights. Fixing the direction to 1, flips from 0 to 1 Figure 4.13, shows no significant difference compared to allowing both directions. In contrast, flips with direction set to 0, flips from 1 to



(a) Sign all (all flips).

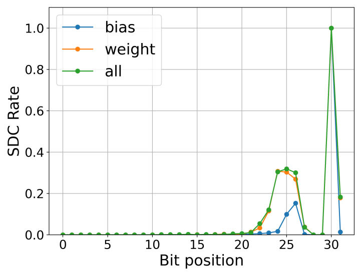


(b) Sign 0 (positive flips).

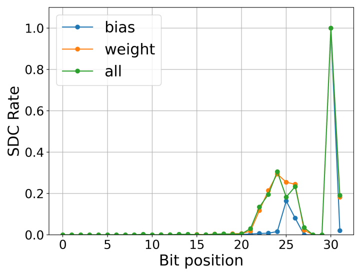


(c) Sign 1 (negative flips).

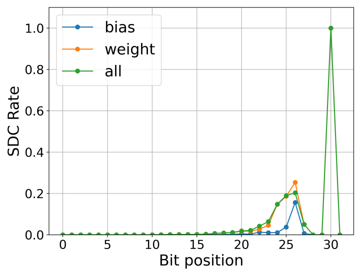
Figure 4.13: Bitflips State Injections for **'0 to 1' direction** on different variables.



(a) Sign all (all flips).



(b) Sign 0 (positive flips).

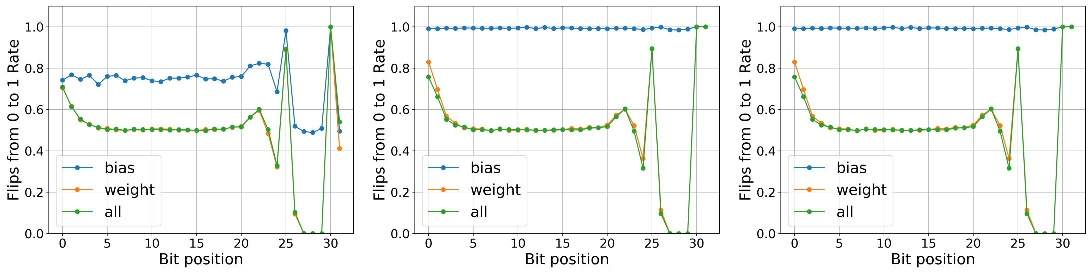


(c) Sign 1 (negative flips).

Figure 4.14: Bitflips State Injections for **'1 to 0' direction** on different variables.

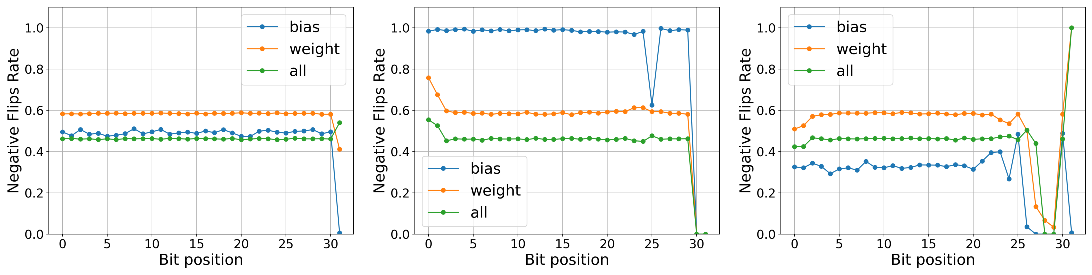
0 Figure 4.14, result in no system failures and exhibit less performance degradation than flips in the opposite direction.

The rate of flips from 0 to 1 and 1 to 0, Figure 4.15, follows the distribution pattern of weights and biases discussed in Section 3.2. Additionally, the analysis of negative and positive flips in Figure 4.16 reveals that the weights have a slightly higher proportion of negative values compared to positive ones.



(a) Sign all (all flips). **(b)** Sign 0 (positive flips). **(c)** Sign 1 (negative flips).

Figure 4.15: Flips from 0 to 1 Rate on different variables.



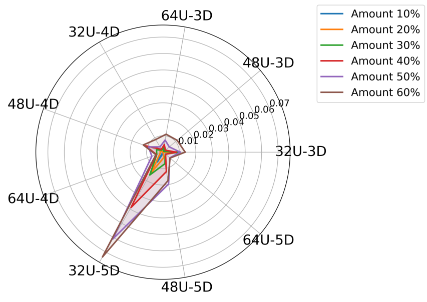
(a) Direction all (all flips). **(b)** Direction 0 (0 to 1 flips). **(c)** Direction 1 (1 to 0 flips).

Figure 4.16: Negative Flips Rate on different variables.

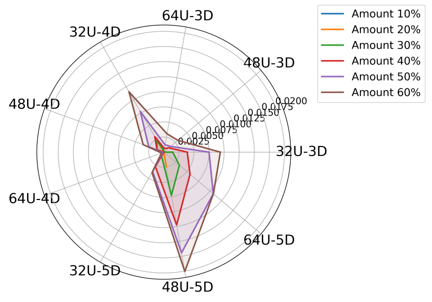
4.1.2 Multiple Models

In the previous Section, we analyzed the behavior of injections on a single model, providing insights into the network's response to various variables and types of injections. We will now examine the effects of injections on the 18 models mentioned in Section 3.1. Due to the extensive number of experiments, we will separate the models with four and six coupling layers and first analyze the combination of units and coupling layer depths. We used radial charts to simultaneously plot nine models for each coupling layer configuration. If the plot resembles a circle, it indicates that the models exhibit similar performance. We aim to identify significant deviations where the shape of the plot differs drastically between models.

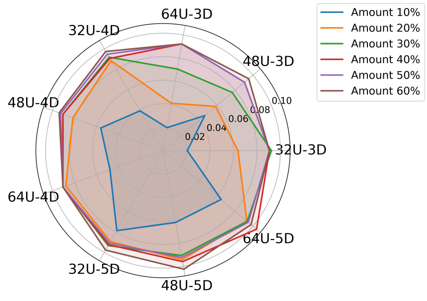
Zero Injections: By varying the layer injection rate, we observe that models with a greater depth of hidden layers exhibit poorer performance. This degradation can be mitigated by increasing the number of coupling units in these layers. Models with deep layers and a small number of units tend to perform worse as the layer injection percentage increases. When we fix the layer injection percentage at 100% and vary only the injection rate, we notice that at high injection rates, models tend to saturate, averaging similar results. As shown in Figure 4.17, when the injection rate is high, the radial plot forms an almost perfect circle.



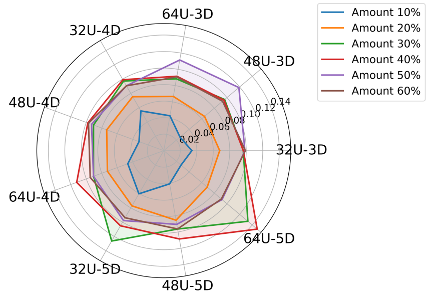
(a) Four Coupling Layers - Bias Variable.



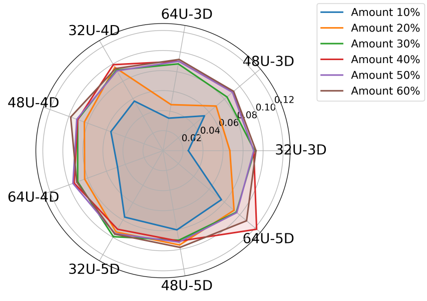
(b) Six Coupling Layers - Bias Variable.



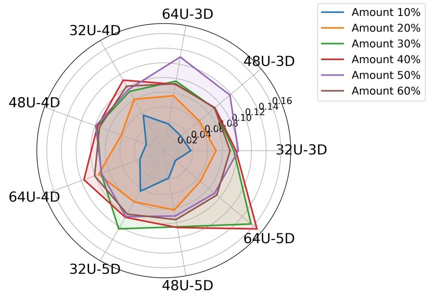
(c) Four Coupling Layers - Weight Variable.



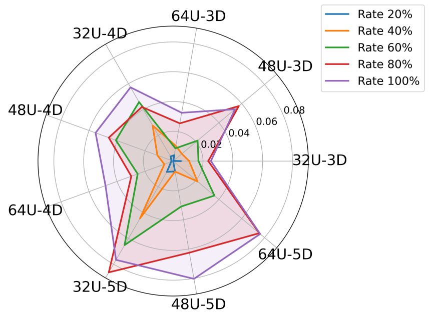
(d) Six Coupling Layers - Weight Variable.



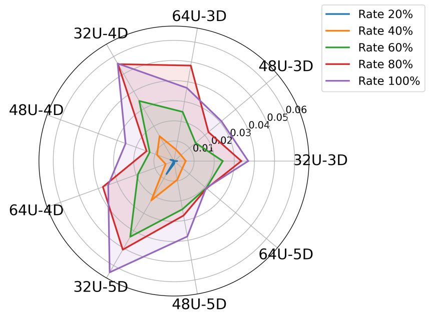
(e) Four Coupling Layers - 'all' Variable.



(f) Six Coupling Layers - 'all' Variable.



(g) Four Coupling Layers - Amount = 10%.



(h) Six Coupling Layers - Amount = 10%.

Figure 4.17: Zeros injections on multiple models on different variables.

Variable-Specific Observations

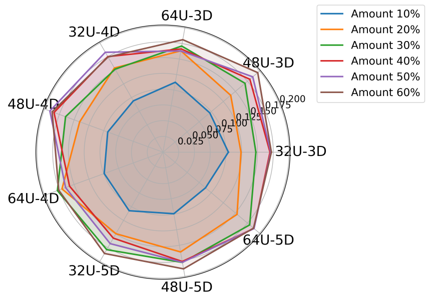
- **Bias:** there is no significant change in performance, except for models with greater depth and fewer units, which perform worse. It is important to note that bias is more susceptible to random errors.

- **Weights:** models with a larger number of parameters tend to perform worse under injection.

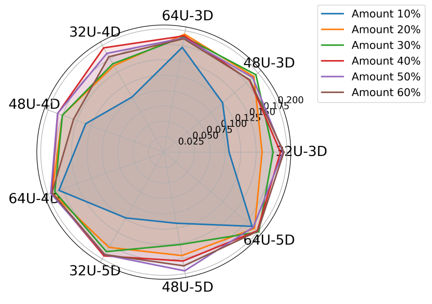
  Combined (Bias and Weights): for configurations where both bias and weights are targeted (e.g., coupling layers = 6, units = 64, depth = 3, and units = 32, depth = 3), a 10% injection rate causes small performance changes in models with higher depth and significant parameter mismatches.

Our findings indicate that not only is the total number of parameters important, but the combination of depth and the number of units must be appropriately balanced to allow the network to effectively capture complex patterns and information.

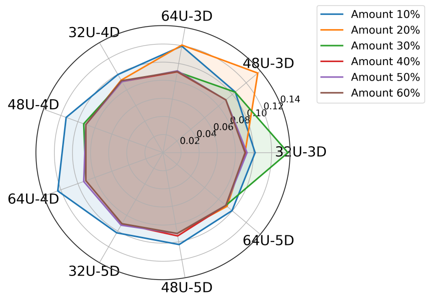
Random Injections: the observations from zero injections largely apply to random injections as well. Starting with a fixed injection rate, models with four coupling layers show chaotic results, especially at a 40% injection rate, where performance declines unexpectedly. This suggests that even small perturbations can cause performance degradation. Models with six coupling layers demonstrate consistent performance across various injection rates, except at 20%, where models with greater depth perform the worst. All models behave similarly for bias injection, except for those with a larger number of units, where injections have a more pronounced effect. Overall, as seen in Figure 4.18, there are no significant perturbations worth noting.



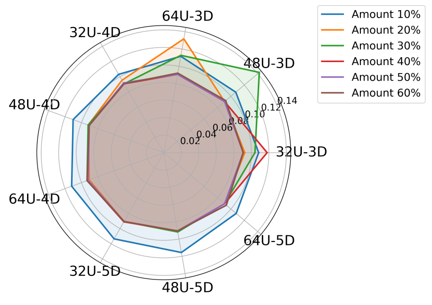
(a) Four Coupling Layers - Bias Variable.



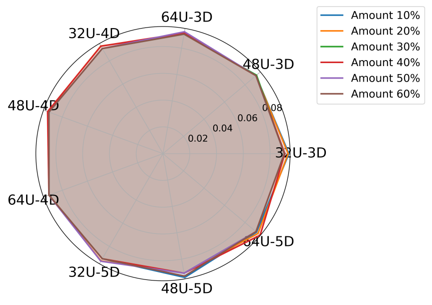
(b) Six Coupling Layers - Bias Variable.



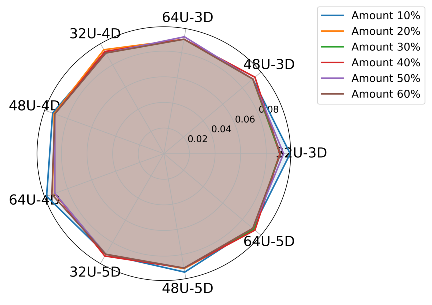
(c) Four Coupling Layers - Weight Variable.



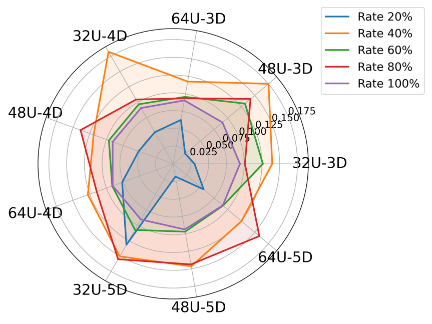
(d) Six Coupling Layers - Weight Variable.



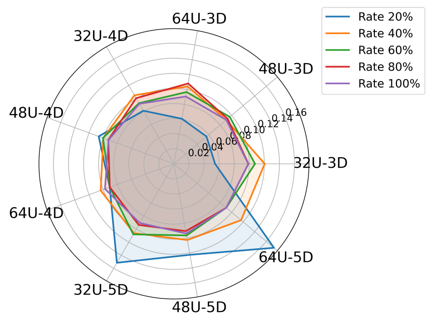
(e) Four Coupling Layers - 'all' Variable.



(f) Six Coupling Layers - 'all' Variable.



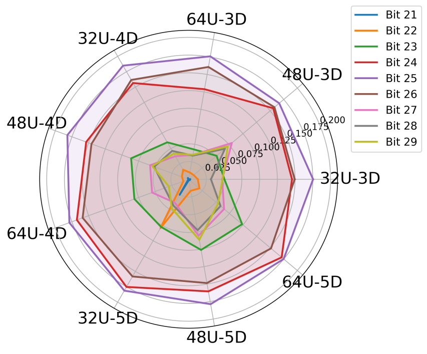
(g) Four Coupling Layers - Amount = 10%.



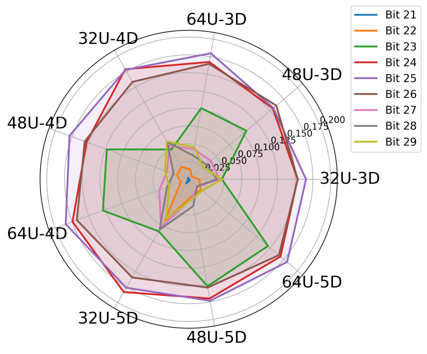
(h) Six Coupling Layers - Amount = 10%.

Figure 4.18: Random injections on multiple models on different variables.

BitFlips Injections: we plotted the results for bit positions 21 through 29, excluding the 30th bit due to system faults. These bits were identified as the most critical in the single-model study. For bits 24, 25, and 26 - representing the peak of our Gaussian distribution - the results were consistent across all models. For the remaining bits, as shown in Figure 4.19, models with a high number of units in configurations with six coupling layers experienced more pronounced performance degradation.



(a) Four Coupling Layers.



(b) Six Coupling Layers.

Figure 4.19: Bitflips injection on multiple models with injection rate = 10% and mode = 100% from 21th to 29th bit position.

4.2 Layer Outputs

After analyzing the results of the layer states, we now examine the layer outputs in greater depth. This time, the model's weights remain unchanged as we explore the internal network behavior by directly modifying the outputs. We use the same configuration as before: the number of faults is set to 10, with the SDC Rate relative for single-model evaluation and the SDC Rate absolute for multi-model evaluation. In the single-model section, we will compare how the smallest model - configured with four coupling layers, a hidden layer depth of three, and 32 units - behaves relative to the largest model, which is configured with six coupling layers, a hidden layer depth of five, and 64 units.

4.2.1 Single Model

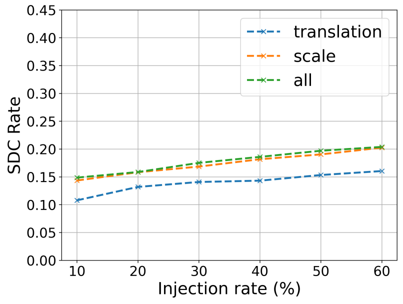
We begin by analyzing a model initialized with zeros and subjected to random injection, using a configuration of four coupling layers. The first observation pertains to how the translation and scale layers respond to these injection strategies. As outlined in Section 3.1, scale layers control the spread of the output by stretching or compressing the input space according to the learned distribution, while translation layers apply a constant shift during the transformation, adjusting the output based on the learned features.

Configuration settings are divided into two main injection strategies:

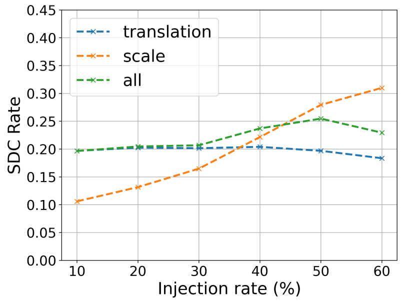
- **Partial Injection:** Injecting only the final output of a coupling layer.
- **Complete Injection:** Injecting outputs directly from within the hidden layers inside a coupling layer.

Having a look at Figure 4.20 we can see that for:

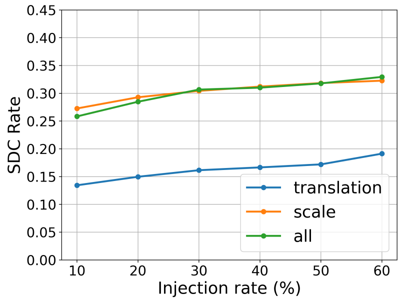
- **Partial Injection:** For both random and zero-based injections, partial injection exhibits a linear progression in performance. Translation layers demonstrate greater resilience, effectively maintaining their shifts when injection occurs only at the final layer. Figures 4.20a, 4.20c, 4.20e, 4.20g.
- **Complete Injection:** In this configuration, scale layers are more robust against random noise. However, they are more sensitive to high injection rates of zeros. Increasing the number of coupling layers beyond four does not produce significant differences. Figures 4.20b, 4.20d, 4.20f, 4.20h.



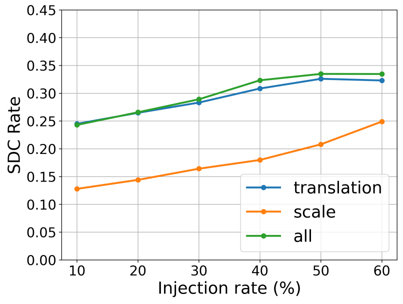
(a) Zeros Partial Four Coupling Layers.



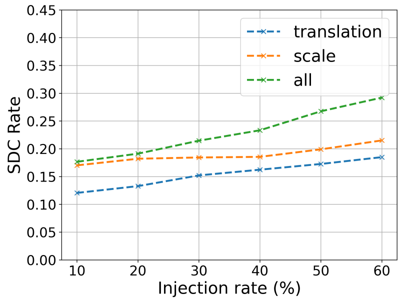
(b) Zeros Complete Four Coupling Layers.



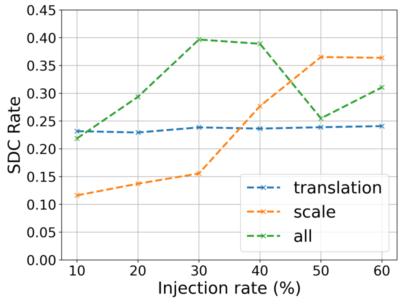
(c) Random Partial Four Coupling Layers.



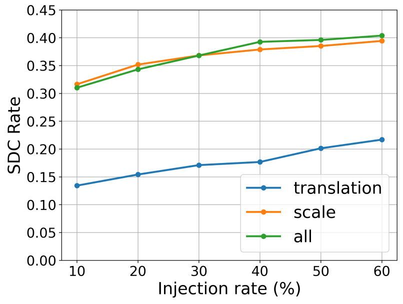
(d) Random Complete Four Coupling Layers.



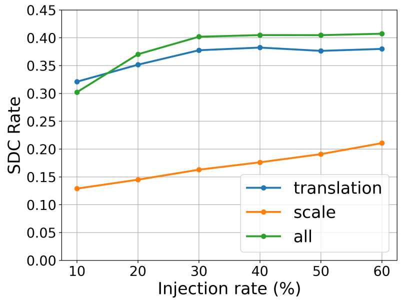
(e) Zeros Partial Six Coupling Layers.



(f) Zeros Complete Six Coupling Layers.



(g) Random Partial Six Coupling Layers.



(h) Random Complete Six Coupling Layers.

Figure 4.20: Layer Outputs injections for different models on different variables.

A summary of the combined effect of zeros and random errors, illustrated in the Figure 4.21, highlights the following: random injections strongly affect scale layers in partial configurations and translation layers in complete configurations. Meanwhile, zeros injection shows distinct behaviors, particularly in complete configurations, where scale layers encounter performance issues at higher injection rates.

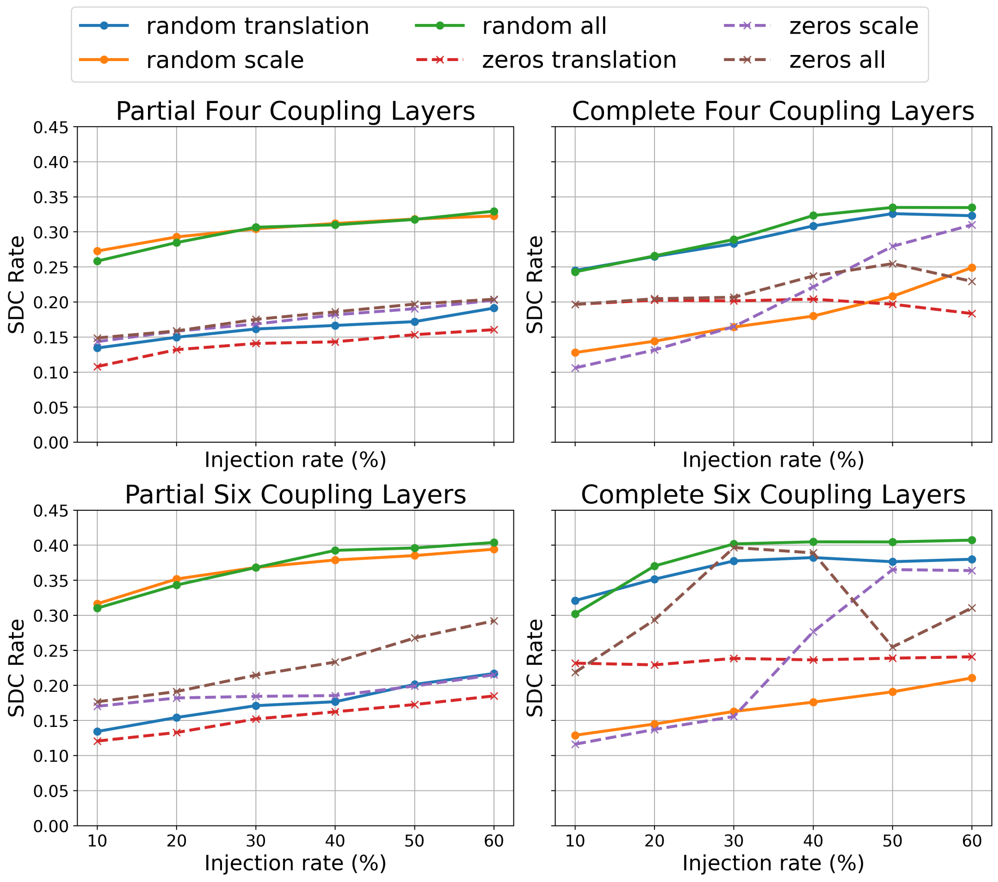


Figure 4.21: Layer Outputs Zeros and Random Injection on different variables in Partial and Complete configurations.

In Figure 4.22, the performance of both configurations is compared. Key findings include:

- For random injections, scale layers in complete configurations and translation layers in partial configurations exhibit robustness.
- Other injection types show weaker performance, particularly under random injection.

- Zeros injection reveals that scale layers in complete configurations struggle at higher injection rates.

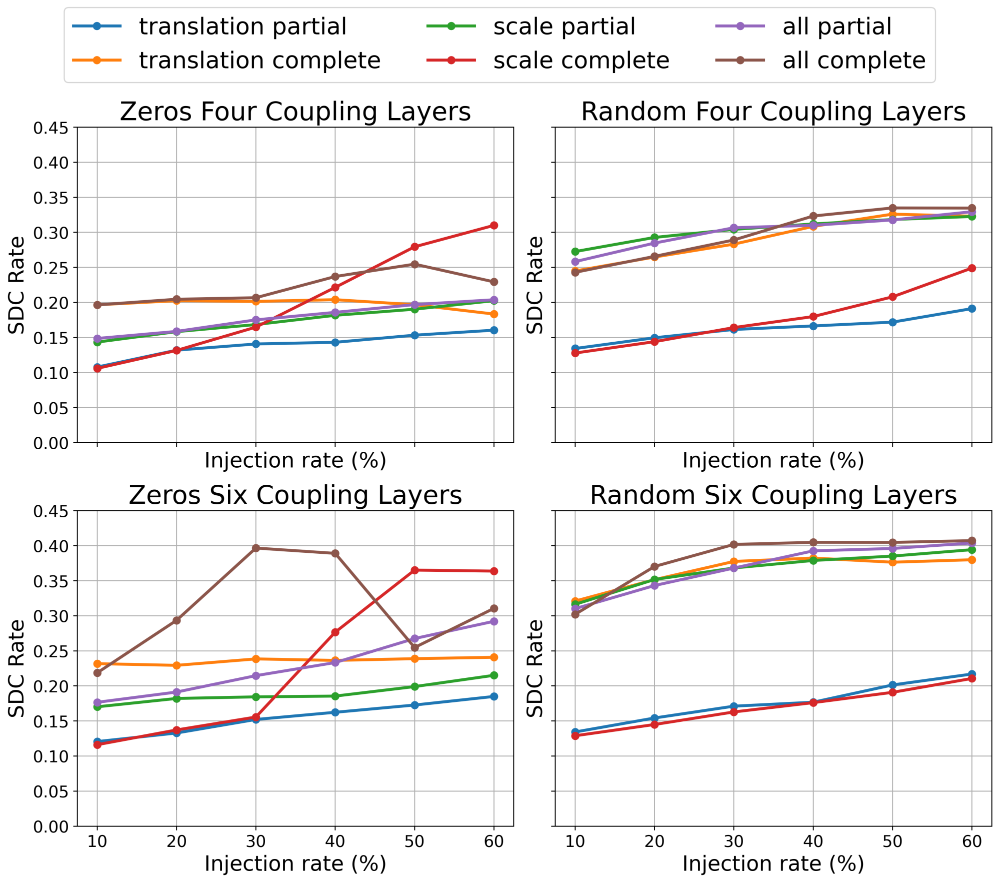


Figure 4.22: Layer Outputs Zeros and Random Injection on different variables by number of Coupling Layers.

In the Figure 4.23 we can examine how activation functions in hidden layers influence performance, focusing on the complete configuration to evaluate components deep within the coupling layers.

- **Random Injection:** Translation layers are the most susceptible, with the worst performance observed in the final linear output layer and scale layers are less affected, as depicted in the analysis.
- **Zeros Injection:** The ReLU activation function in scale layers performs poorly under zero injection, with exponential degradation as the injection rate increases. Activation functions in translation layers remain steady across all injection rates. Finally, results remain consistent even with six coupling layers,

indicating that zeros injection minimally impacts deeper networks compared to random noise.

This preliminary analysis highlights the fundamental differences between scale and translation layers under various injection strategies. Furthermore, the performance implications of partial versus complete injection, particularly in translation and scale layers and the influence of activation functions on layer outputs, with specific vulnerabilities identified for certain configurations and injection types.

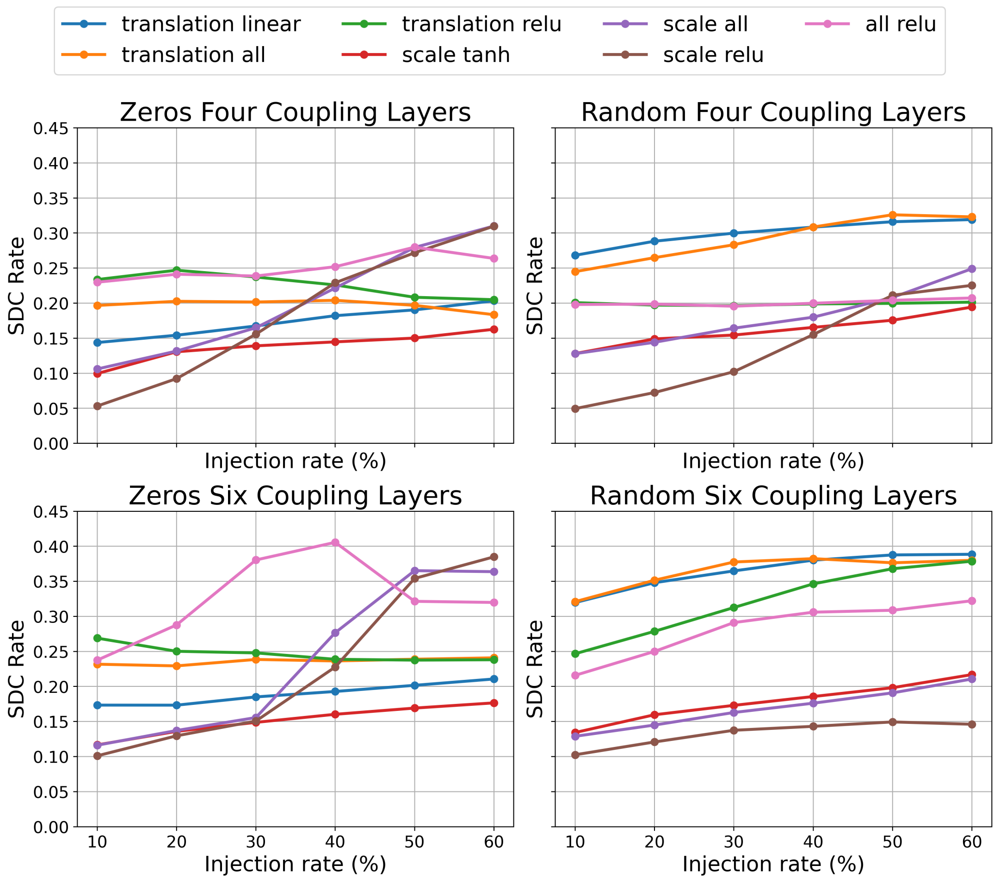


Figure 4.23: Layer Outputs Zeros and Random Injection on different activation functions.

Single Layers: To gain additional insight, we analyze the impact of errors when injections are applied to individual layers. This approach allows us to examine how the model behaves at varying depths. The experiments were conducted on models with coupling layers set to four and six. In this configuration, the layers are numbered from 0 to 3 (or 5 for the six-coupling model), where the layer 0 represents the input layer and higher numbers correspond to layers deeper in the network.

The analysis begins by examining the general behavior of scale and translation layers. Subsequently, we evaluate the configuration where both components are targeted simultaneously (*all*) to determine which is most significantly affected.

The Figures 4.24, 4.25 illustrate the effects of zeros injection on scale and translation layers: **Scale Layers:**

- **Partial Injection:** With four coupling layers at Layer 0, the injection has minimal impact, as the network effectively recovers in subsequent layers. Furthermore, in the model with six coupling layers, there is less separation between each layer, but it follows the same behavior as the model with four coupling layers.

- **Complete Injection:** For the model with four coupling layers the performance degradation increases linearly with depth and injection rate. Meanwhile for models with six coupling layers, layer 0, 1, 2 does not increase with the injection rate, as the layers 3, 4 and 5 does slightly.

Translation Layers:

- **Partial Injection:** It has a different trend from the one observed in scale layers. Lower layers are more resilient, while only the last layer exhibits significant issues. This pattern holds true for models with both four and six coupling layers.

- **Complete Injection:** Performance steadily worsens with depth, reflected in an increasing SDC rate after each layer.

In Figures 4.24, 4.25 the partial configuration the graph resembles the behavior of scale layers, highlighting their vulnerability to zeros injection. Meanwhile in the complete configuration, the adverse impact on scale layers is slightly mitigated by the translation layers, particularly at Layer 0, which behaves similarly to the previously observed complete-scale injection trend.

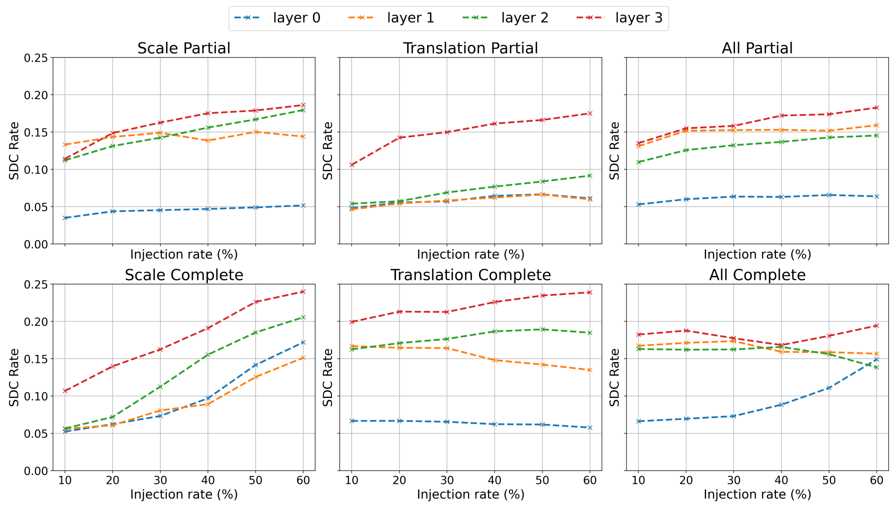


Figure 4.24: Layer Outputs Zeros Injection with four Coupling Layer on different variables presented by layer depth.

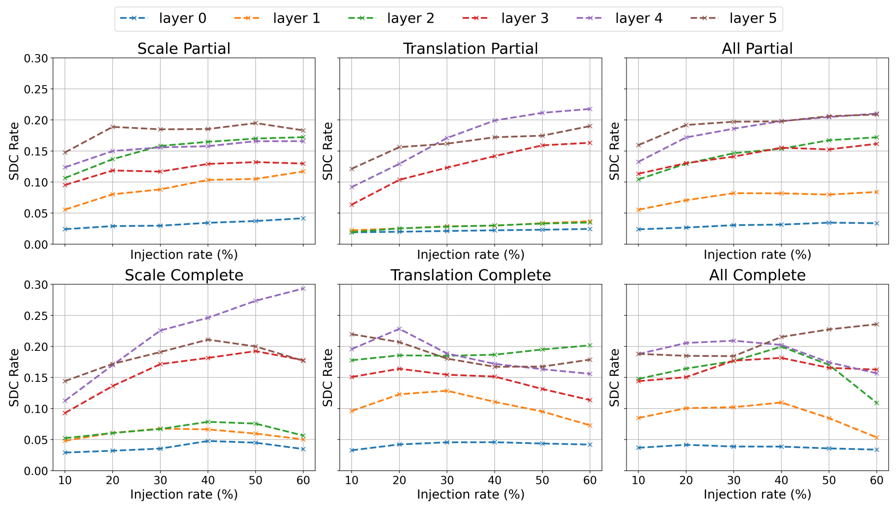


Figure 4.25: Layer Outputs Zeros Injection with six Coupling Layer on different variables presented by layer depth.

Figures 4.26, 4.27 reveal the impact of random injections, showcasing more pronounced peculiarity between scale and translation behaviors:

- **Scale Layers in Complete Injection:** The performance decreases linearly with depth for both four and six coupling layers. Lower layers experience only minor degradation, while deeper layers exhibit significant declines in performance.

- **Translation Layers in Partial Injection:** The graph for translation layers in a partial configuration closely mirrors that of scale layers in complete configurations, with lower layers maintaining resilience and deeper layers showing sharp declines.

Other configurations maintain a steady increase in SDC rate as depth increases.

Analyzing the *all* configuration in Figures 4.26, 4.27 the performance consistently worsens as injections target deeper layers. The graph closely aligns with the scale-layer behavior, demonstrating that scale layers predominantly influence network performance. Translation layers fail to counteract this impact effectively when subjected to injection.

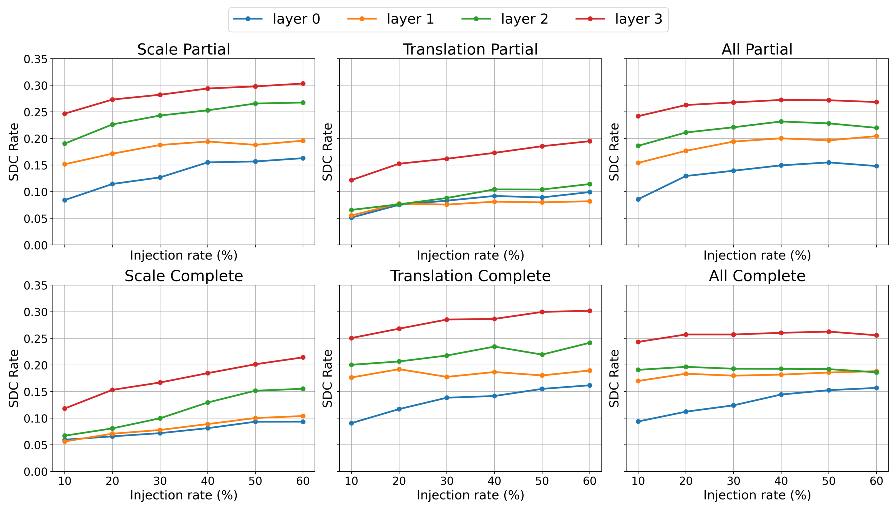


Figure 4.26: Layer Outputs Random Injection with four Coupling Layer on different variables presented by layer depth..

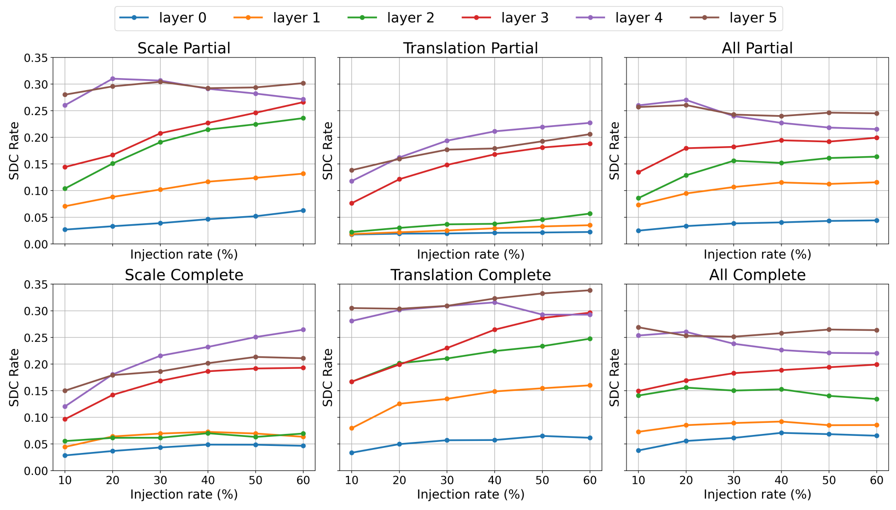


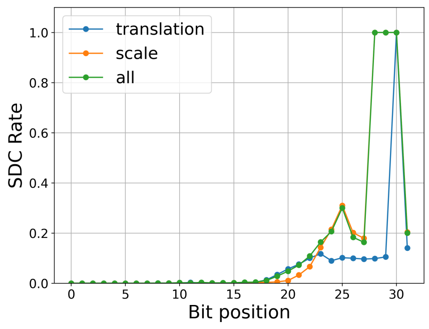
Figure 4.27: Layer Outputs Random Injection with six Coupling Layer on different variables presented by layer depth.

Bitflips: This study continues the study of bitflip injections, focusing on their impact on layer outputs. Similar to the analysis of layer states in Section 4.1.1, bitflip injections were performed with a fixed injection rate of 10% under standard configurations. These configurations included both positive and negative signs, as well as flips in all directions. The analysis was conducted on two models with coupling values of 4 and 6. If the system experienced failure due to bitflips, we assigned a SDC rate of 1.

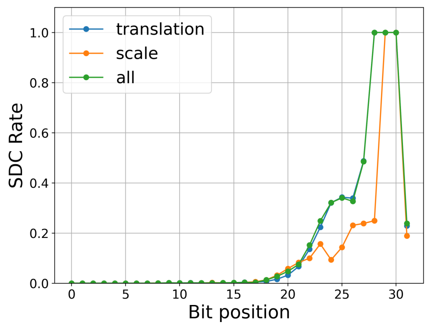
For our figures, we primarily concentrated on the exponent bits (23rd to 30th bits) and the sign bit (31st). Lower-order mantissa bits were largely overlooked, as their results were near zero and negligible. Our analysis examined the impact on both scale and translation, as well as their combined effect.

As shown in the corresponding Figure 4.28, we examined scale and translation effects individually and in combination (*all*). For partial injections, the model's performance aligned with scale-related behavior. For complete injections, it followed translation-related behavior. Consistent with prior studies on zero and random injections, translation showed resilience when injected in the final layer post-linear activation. Contrarily, scale was more robust against complete injections within hidden layers. The larger model (coupling = 6) exhibited higher failure rates in bits 27, 28, 29, and 30. It also demonstrated increased SDC rates compared to the smaller model (coupling = 4). The propagation of bitflips was more pronounced in the larger model, leading to additional failures in these critical bits.

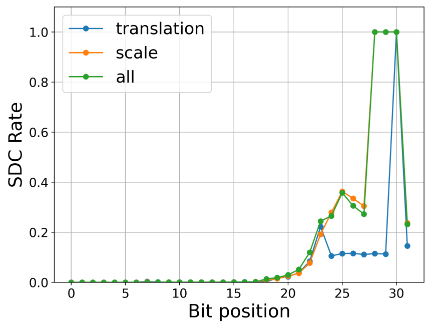
In Figures 4.29, 4.30, we studied the resilience of individual layers to injections. All layers exhibited similar failure points. As observed earlier, the model followed



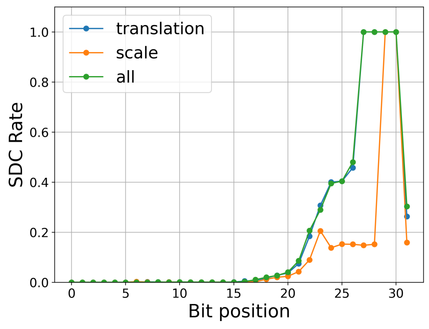
(a) Partial Four Coupling Layers.



(b) Complete Four Coupling Layers.



(c) Complete Four Coupling Layers.



(d) Complete Six Coupling Layers.

Figure 4.28: Bitflips Layer Outputs injection on different variables from 0 to 31th bit position.

scale behavior for partial injections and translation behavior for complete injections. The increased size and depth of the larger model intensified bitflip propagation, resulting in an additional number of failures in different bits.

In Figure 4.31, we analyzed the failure rates for different activation functions. The most robust activation function was scale tanh, which constrained values within the range of -1 to 1. Performance ranking from the worst to the best was as follows: translation ReLU, Translation Linear, Scale ReLU, Scale Tanh. This study was conducted exclusively for complete injections, as it would not be meaningful to inject different activations in a partial target.

The Table 4.1 shows a summary of each bitflip configuration applied, enabling us to gain an overview of where the detection power fails and which bits are affected.

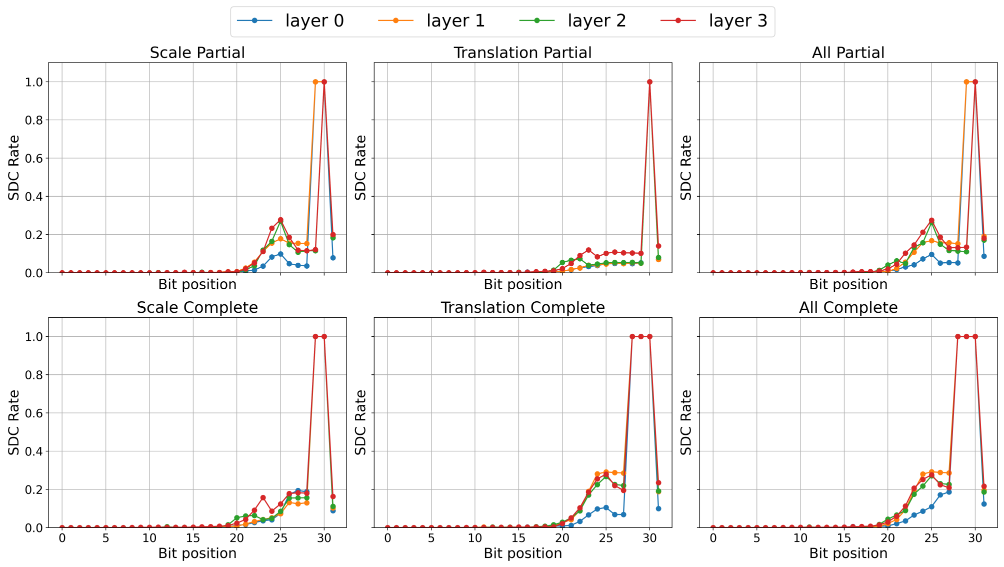


Figure 4.29: Layer Outputs Bitflips Injection with four Coupling Layer on different variables presented by layer depth.

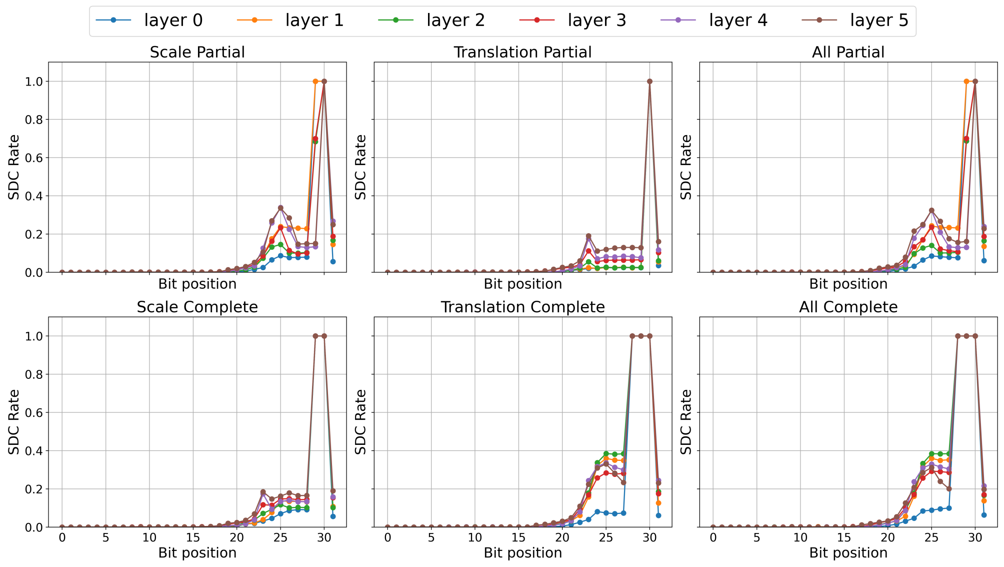
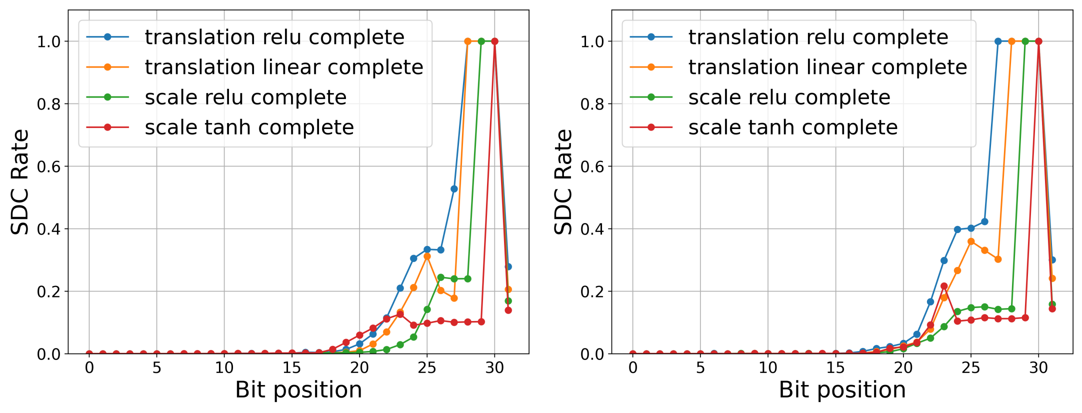


Figure 4.30: Layer Outputs Bitflips Injection with six Coupling Layer on different variables presented by layer depth.



(a) Four Coupling Layers. **(b)** Six Coupling Layers.

Figure 4.31: Layer Outputs Bitflips Injection on different activation functions.

Coupling Layers	**Partial/Complete**	**Configuration**	**Bit Position** 25	26	27	28	29	30	31
4	Partial	Scale				x	x	x	
		Translation						x	
		All				x	x	x	
	Complete	Scale					x	x	
		Translation					x	x	
		All				x	x	x	
		Translation Relu				x	x	x	
		Translation Tanh				x	x	x	
		Scale Relu					x	x	
		Scale Tanh						x	
6	Partial	Scale					x	x	
		Translation			x	x	x	x	
		All			x	x	x	x	
	Complete	Scale					x	x	
		Translation				x	x	x	
		All				x	x	x	
		Translation Relu			x	x	x	x	
		Translation Tanh				x	x	x	
		Scale Relu					x	x	
		Scale Tanh						x	

Table 4.1: Failures of the system (marked with an 'x') on different configurations and bit postion.

4.2.2 Multiple Models

After completing the analysis of the single model in Section 4.1.1, we now shift our focus to the behavior of the 18 models described in Section 3.1. Using the same guidelines outlined in Section 4.1.2, we examine the graphs of models with four and six coupling layers to identify significant differences in the areas defined by the intersection of the models' performance.

Zeros Injection: at first glance, we observe that models with a smaller number of units perform worse on both sides of the Figures 4.38a, 4.38b. This suggests that smaller hidden layers lead to information loss, making the models unable to capture relevant characteristics. Adding greater depth to the coupling layers intensifies this issue, as errors propagate more extensively. Models with a larger number of units, however, can mitigate this problem by retaining more information and reducing the impact of propagated errors.

Examining the Figures 4.35, 4.36, 4.37, from a layer perspective and considering the scale, translation, and *all* configurations, we notice that models with six coupling layers tend to exhibit more chaotic behavior. These models alternate their worst performance across varying layer depths. In particular, models with few units and a significant propagation distance (e.g., units = 32 and depth = 5) struggle with zeros propagation through layers, leading to output degradation.

The activation Figures 4.39a, 4.39b highlight certain models that perform the worst. These underperforming models are characterized by having few units in their hidden layers (e.g., units = 32). This reinforces the importance of having sufficient units to process and retain feature information, as previously discussed in the single-layer analysis. In contrast, models with substantially more units perform well, demonstrating a stronger capacity to inhibit the negative effects of zeros injection.

Random Injection: let's now examine the findings related to the injection of random values and compare them to the previously observed effects of zero-value injections. From the Figures 4.38c, 4.38d, we can immediately observe that, compared to zero-value injections, the scale-partial and all-partial configurations lie on the same plane as the all-complete and translation-complete configurations. However, the scale-complete and translation-partial configurations stand apart, residing inside this plane. Overall, the shapes in this case appear more rounded, suggesting that noise indeed causes a reduction in performance. However, the degree of separation between models remains limited. A notable exception is observed in the model with 32 units and a depth of 5, which performs distinctively for both coupling types-four and six.

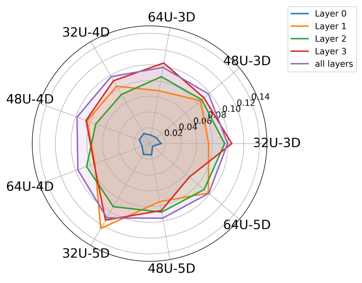
In the single-layer analysis, Figures 4.35, 4.36, 4.37, the chaotic behavior seen with zero-value injections decreases slightly. The shapes are now more rounded across the *scale*, *translation*, and *all* configurations. Notable anomalies appear in the translation layers, particularly in models with smaller numbers of units and greater depths. These anomalies tend to result in suboptimal combinations of units and depth. Interestingly, this effect does not occur in models with larger numbers of units and smaller depths, as previously noted for the scale layers. This underscores the importance of incorporating a substantial number of units within each layer. Moreover, the disruptive behavior associated with translation layers appears to be mitigated by scale layers when we examine the *all* configuration, where the dominance of scale layers becomes evident.

This Figures 4.39c, 4.39d, introduces a shift in behavior. Linear activation functions emerge as the weakest against noise, demonstrating the worst performance. Their behavior is reminiscent of ReLU activations under zero-value injections. However, there is a reduction in erratic behavior among models, which now exhibit greater robustness to random noise. Only extreme combinations of depth and units, such as 32 units and a depth of 5, show significant impacts from the noise.

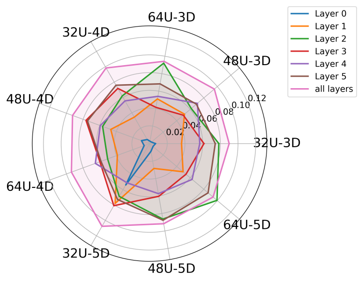
Bitflips injections: in this simplified bitflips analysis, we have removed the bits who have generated a failure, Figure 4.40, as was done in the layer states configuration. We focus on plotting the bits from 20 to 27, highlighting the following key findings:

- **Impact of Coupling Layers:** The number of coupling layers negatively affects performance. Models with more coupling layers exhibit a higher SDC rate. This is because bitflips are more likely to propagate through additional layers, compounding errors.

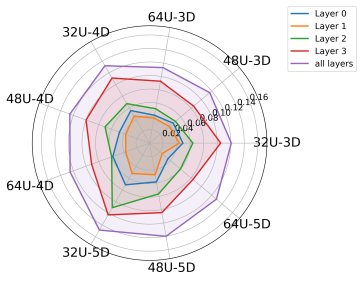
- **Hidden Layer Units and Coupling Depth:** A mismatch between the number of units in the hidden layers and the depth of coupling continues to result in information loss. This highlights the importance of aligning these parameters to maintain model performance.



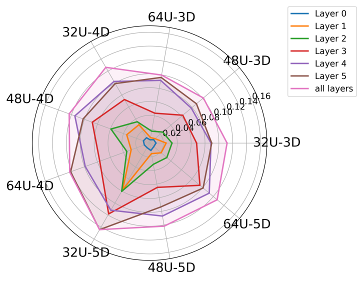
(a) Four Coupling Layers - Partial.



(b) Six Coupling Layers - Partial.

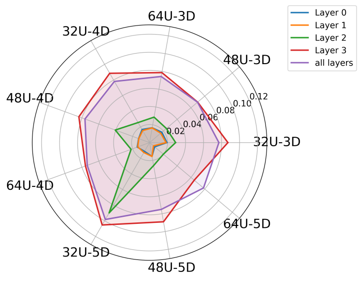


(c) Four Coupling Layers - Complete.

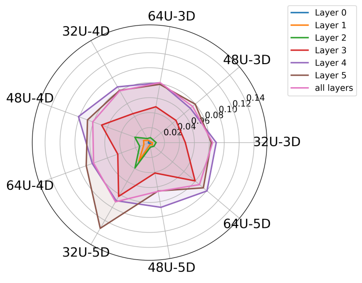


(d) Six Coupling Layers - Complete.

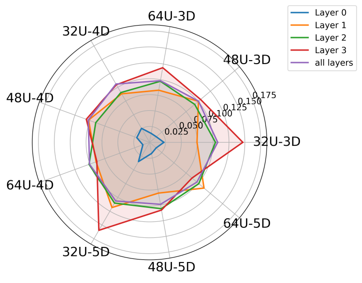
Figure 4.32: Layer Outputs Zeros Injections on Scale Variable.



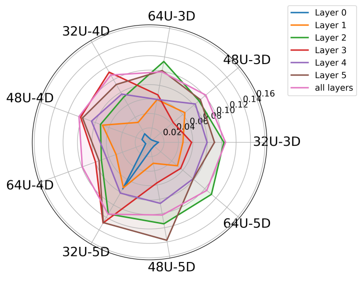
(a) Four Coupling Layers - Partial.



(b) Six Coupling Layers - Partial.

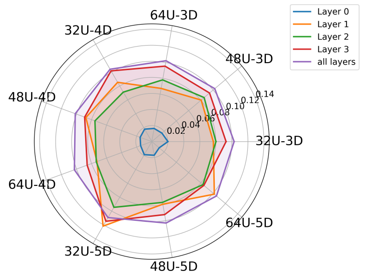


(c) Four Coupling Layers - Complete.

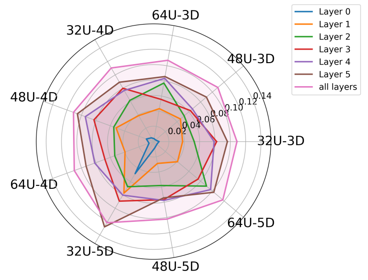


(d) Six Coupling Layers - Complete.

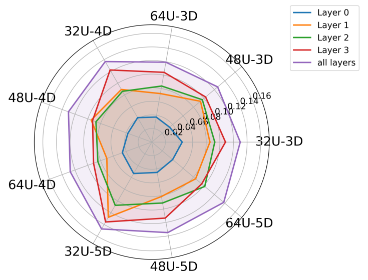
Figure 4.33: Layer Outputs Zeros Injections on Translation Variable.



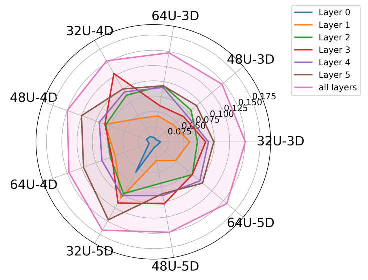
(a) Four Coupling Layers - Partial.



(b) Six Coupling Layers - Partial.

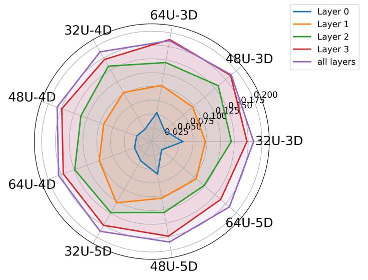


(c) Four Coupling Layers - Complete.

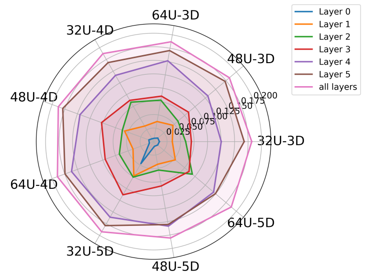


(d) Six Coupling Layers - Complete.

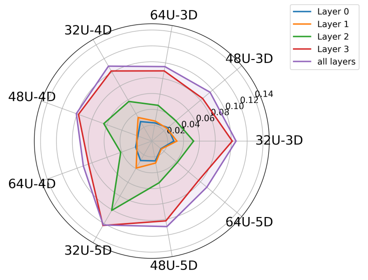
Figure 4.34: Layer Outputs Zeros Injections on All Variable.



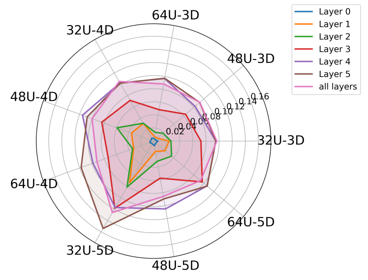
(a) Four Coupling Layers - Partial.



(b) Six Coupling Layers - Partial.

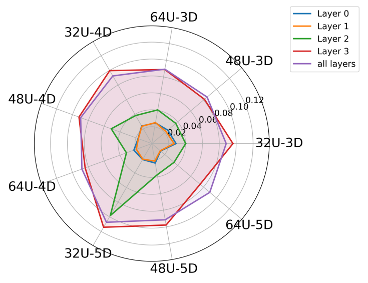


(c) Four Coupling Layers - Complete.

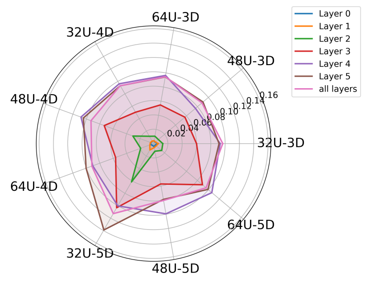


(d) Six Coupling Layers - Complete.

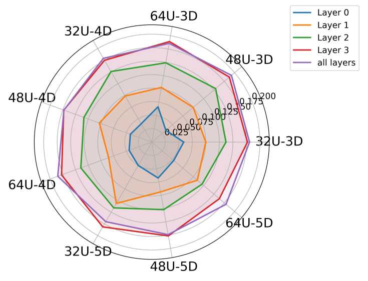
Figure 4.35: Layer Outputs Random Injections on Scale Variable.



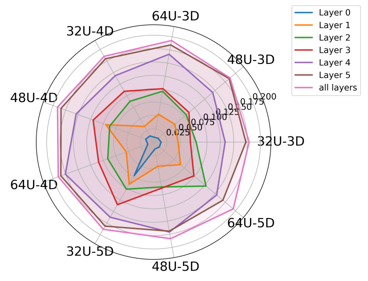
(a) Four Coupling Layers - Partial.



(b) Six Coupling Layers - Partial.

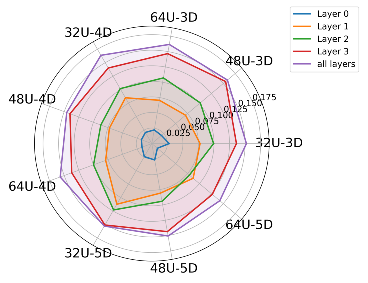


(c) Four Coupling Layers - Complete.

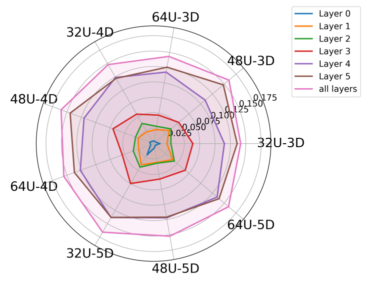


(d) Six Coupling Layers - Complete.

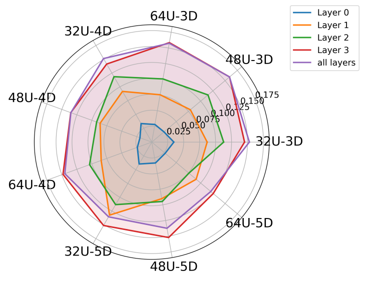
Figure 4.36: Layer Outputs Random Injections on Translation Variable.



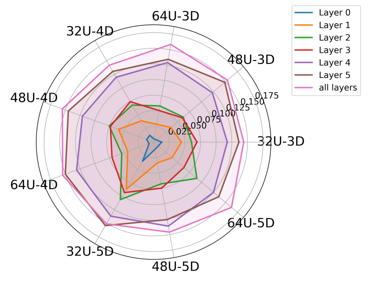
(a) Four Coupling Layers - Partial.



(b) Six Coupling Layers - Partial.

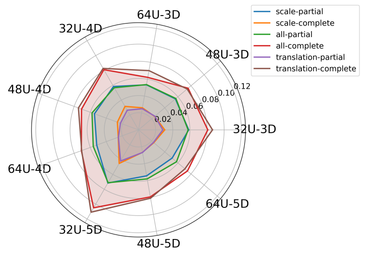


(c) Four Coupling Layers - Complete.

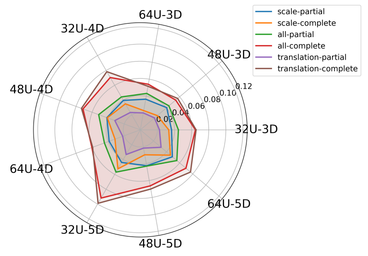


(d) Six Coupling Layers - Complete.

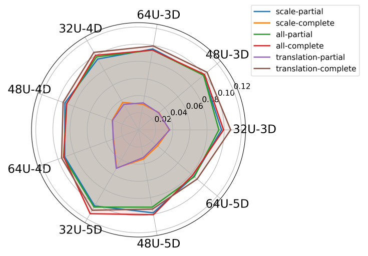
Figure 4.37: Layer Outputs Random Injections on All Variable.



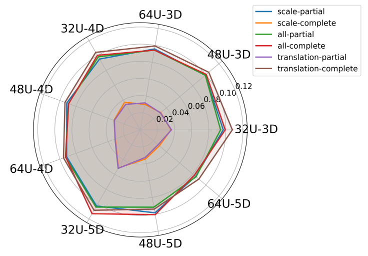
(a) Four Coupling Layers - Zeros.



(b) Six Coupling Layers - Zeros.

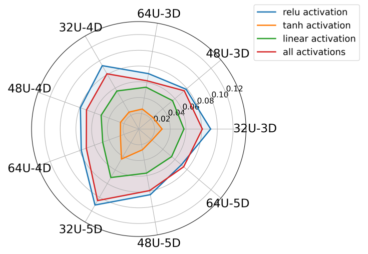


(c) Four Coupling Layers - Random.

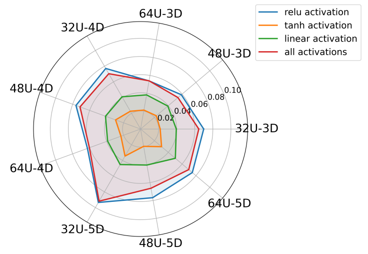


(d) Six Coupling Layers - Random.

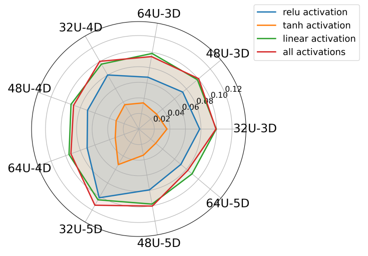
Figure 4.38: Layer Outputs Injections at different Variables.



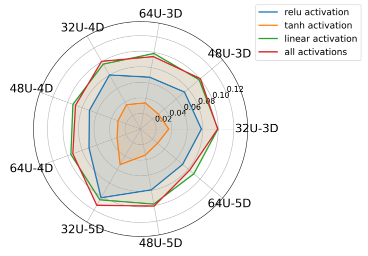
(a) Four Coupling Layers - Zeros.



(b) Six Coupling Layers - Zeros.

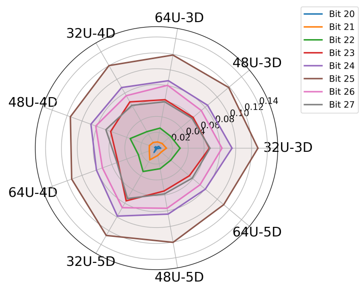


(c) Four Coupling Layers - Random.

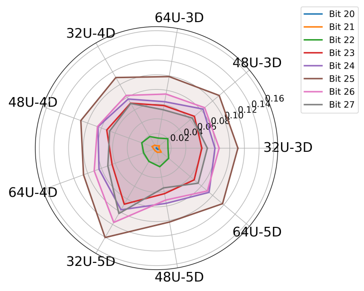


(d) Six Coupling Layers - Random.

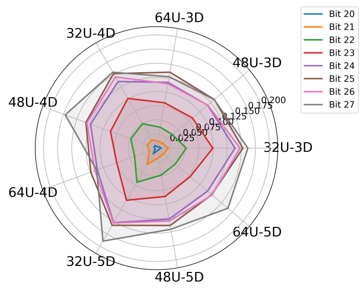
Figure 4.39: Layer Outputs Injections at different Activation functions.



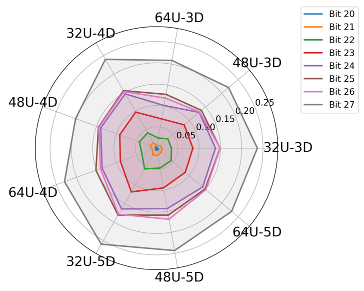
(a) Four Coupling Layers - Partial.



(b) Six Coupling Layers - Partial.



(c) Four Coupling Layers - Complete.



(d) Six Coupling Layers - Complete.

Figure 4.40: Layer Outputs Bitflips Injections from 20th to 27th bits.

4.3 Summary of Results

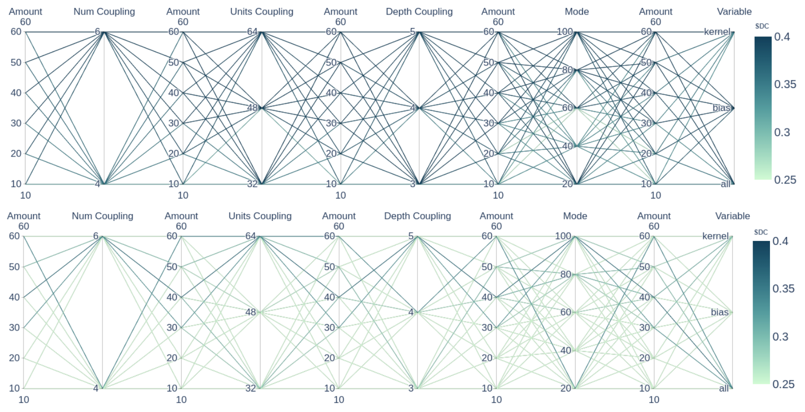
Here are summarized the results presented in this Chapter, we conducted several experiments, as shown in Figures 4.41a, 4.41b, 4.41c. Each type of injection affected the network differently. Fault injections revealed the degradation threshold where the model begins to fail. At this critical point, often associated with task saturation, the outputs become dependent solely on the inputs.

The effects of injections on layer states showed distinct patterns. Random noise primarily affected the bias of network states, but due to redundancy, these changes did not significantly disrupt the model's output. Zeros had a stronger impact on weights, breaking connections between neurons and degrading the model's ability to detect anomalies. Bitflips exhibited unique behavior influenced by bit distribution. Flips in the exponent bits were particularly sensitive, creating Gaussian degradation patterns in performance plots. For example, a specific flip in the 30th bit caused NaN or infinite values, leading to network failure.

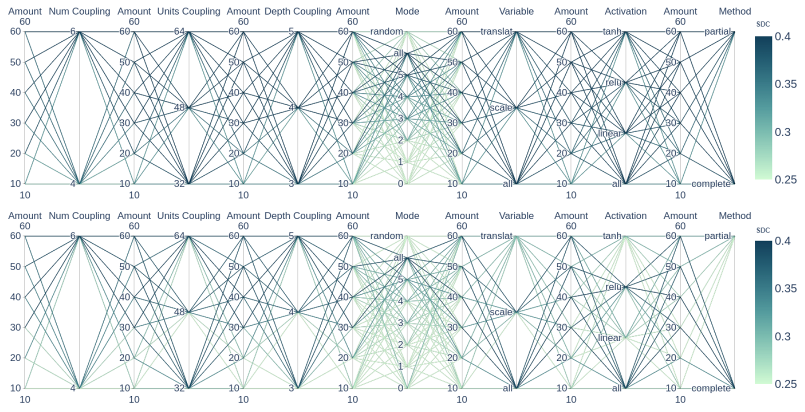
In terms of layer outputs, injections were applied either partially or completely, depending on whether a broad or focused fault injection was desired. The scale and translation layers reacted differently to these injections. For partial injections, translation layers demonstrated greater robustness against random noise, while scale layers were more resilient under complete injections. Zeros showed a similar trend, with translation layers performing better under partial injection and scale layers excelling under complete injection, illustrating their impact across the network. Bitflips revealed critical points in the exponents, where increasing the range of bits exposed vulnerabilities in the outputs.

Activation functions also influenced the network's resilience to fault injections. Linear and ReLU activation functions in the translation layer showed strong performance in limiting degradation, although their Silent Data Corruption (SDC) rates increased slightly with higher injection rates.

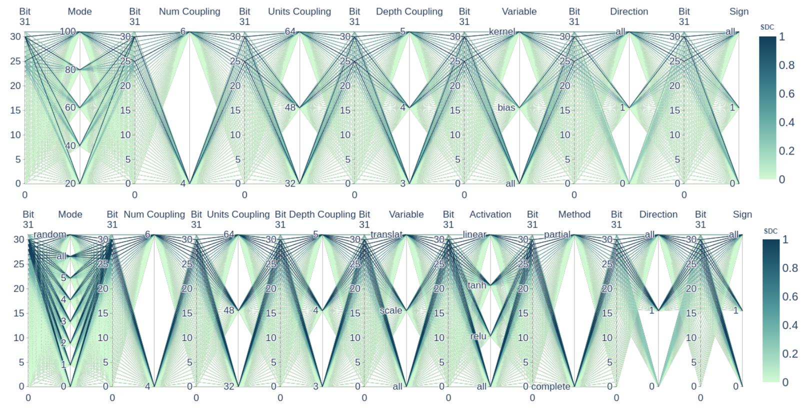
When comparing models trained with different parameters, achieving a balance between key architectural factors proved crucial. These factors include the length and depth of coupling layers and the number of units in the hidden layers of the coupling. Together, these parameters determine the total number of connections within the network. A sufficient number of units is essential to retain information, while minimizing propagation distance between outputs is critical to reduce the impact of fault injections.



(a) Random (up) and Zeros (down) layer states experiments.



(b) Random (up) and Zeros (down) layer outputs experiments.



(c) Bit-flips experiments for layer states (up) and outputs (down).

Figure 4.41: Overview of experiments on layer states, outputs, and bit-flips across all configurations.

Chapter 5

Conclusion

This chapter concludes our study by presenting a summary of the contributions made in this thesis in Section 4.3 and outlining potential future work in this field of study in Section 5.2.

5.1 Summary of Contribution

The goal of this thesis was to develop a framework for fault injection in a Normalizing Flow Real NVP network used in space applications. These networks require compactness, robustness, and resilience, which are essential for safety-critical environments like those encountered in space. To emulate these harsh conditions and thoroughly test networks, fault injection frameworks are developed at both hardware and software levels. Our primary contribution lies in studying the available technology in the software domain and creating a customized framework capable of analyzing the network in all its aspects.

Building upon existing frameworks, such as TensorFI [21], we designed a tailored fault injection framework specifically for Normalizing Flow networks. This framework supports two main injection targets: layer states injection, which injects faults into the weights before the inference phase, and layer outputs injection, which allows faults to be directly injected into the network's outputs during inference runs. These configurations enable both general fault injection and more granular approaches, such as disabling specific components to perform ablation studies. By doing so, we identified the network's more robust components and its weaker points.

The results of our experiments focused on two key aspects of the network. The first was the weights of the network, which include its total parameters determined during training. These parameters guide the model in extracting essential features from the input. The second aspect was the outputs of the network and its layers, which generate scores used to classify points as anomalous or not.

Three types of fault injections were analyzed: zeros, simulating values being set to zero and effectively removing information; random noise, introducing fluctuations in the network's parameters or outputs; and bitflips, emulating changes in individual bits, such as those caused by space radiation.

Through this framework, we were able to test the network's resilience under various fault scenarios and contribute to understanding its behavior under conditions akin to space environments. This comprehensive approach provides insights into improving the robustness of Normalizing Flow Real NVP networks for safety critical applications.

5.2 Future Work

Fault injection frameworks are powerful tools that help developers test the robustness of their networks, providing valuable insights into the optimal network dimensions, identifying the most resilient components, and determining the most suitable activation functions. To remain effective, these frameworks must be continuously updated and researched to facilitate efficient testing. They should be designed to perform experiments quickly, enabling developers to complete extensive testing within a short timeframe, while avoiding unnecessary complexity.

However, software-level fault injection alone is insufficient to provide a comprehensive understanding of how neural networks perform in real-world applications. It does not fully account for the behavior of these networks when deployed on hardware devices. For this reason, fault injection frameworks should adopt a hybrid approach, combining software-level injection with hardware-level fault injection. This hybrid method yields more reliable results and provides a clearer understanding of improvements needed at both the software and hardware levels.

Bibliography

[1] Agenzia Spaziale Italiana. *ESA's Earth observation mission.* 2025. URL: https://www.asi.it/en/earth-science/ers-1/ (visited on 03/25/2025) (cit. on p. 1).

[2] The European Space Agency. *BepiColombo: Investigating Mercury's mysteries.* 2025. URL: https://www.esa.int/Science_Exploration/Space_Science/BepiColombo (visited on 03/25/2025) (cit. on p. 1).

[3] NASA. *Space Radiation Effects on Electronic Components in Low-Earth Orbit.* 1999. URL: https://llis.nasa.gov/lesson/824 (cit. on p. 2).

[4] Bruce Pritchard, G.M. Swift, and A.H. Johnston. «Radiation effects predicted, observed, and compared for spacecraft systems». In: Feb. 2002, pp. 7–13. ISBN: 0-7803-7544-0. DOI: 10.1109/REDW.2002.1045525 (cit. on p. 2).

[5] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. *Density estimation using Real NVP.* 2017. arXiv: 1605.08803 [cs.LG]. URL: https://arxiv.org/abs/1605.08803 (cit. on p. 8).

[6] Zhuangwei Kang, Ayan Mukhopadhyay, Aniruddha Gokhale, Shijie Wen, and Abhishek Dubey. «Traffic Anomaly Detection Via Conditional Normalizing Flow». In: *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC).* 2022, pp. 2563–2570. DOI: 10.1109/ITSC55140.2022.9922061 (cit. on p. 8).

[7] Sondre Sørbø and Massimiliano Ruocco. *Navigating the Metric Maze: A Taxonomy of Evaluation Metrics for Anomaly Detection in Time Series.* 2023. arXiv: 2303.01272 [cs.LG]. URL: https://arxiv.org/abs/2303.01272 (cit. on p. 10).

[8] Alexis Huet, Jose Manuel Navarro, and Dario Rossi. «Local Evaluation of Time Series Anomaly Detection Algorithms». In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.* KDD '22. ACM, Aug. 2022, pp. 635–645. DOI: 10.1145/3534678.3539339. URL: http://dx.doi.org/10.1145/3534678.3539339 (cit. on p. 10).

[9] Paolo Rech. «Artificial Neural Networks for Space and Safety-Critical Applications: Reliability Issues and Potential Solutions». In: *IEEE Transactions on Nuclear Science* 71.4 (2024), pp. 377–404. DOI: 10.1109/TNS.2024.3349956 (cit. on p. 10).

[10] Ramon Canal et al. «Predictive Reliability and Fault Management in Exascale Systems: State of the Art and Perspectives». In: *ACM Computing Surveys* 53 (Oct. 2020), pp. 1–32. DOI: 10.1145/3403956 (cit. on p. 10).

[11] Giulio Gambardella, Nicholas J. Fraser, Ussama Zahid, Gianluca Furano, and Michaela Blott. «Accelerated Radiation Test on Quantized Neural Networks trained with Fault Aware Training». In: *2022 IEEE Aerospace Conference (AERO)*. 2022, pp. 1–7. DOI: 10.1109/AERO53065.2022.9843614 (cit. on p. 10).

[12] Alessandro Vallero, Dimitris Gizopoulos, and Stefano Di Carlo. «SIFI: AMD southern islands GPU microarchitectural level fault injector». In: *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2017, pp. 138–144. DOI: 10.1109/IOLTS.2017.8046209 (cit. on p. 10).

[13] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. «SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation». In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, pp. 249–258. DOI: 10.1109/ISPASS.2017.7975296 (cit. on p. 10).

[14] Michael A. Kochte, Christian G. Zoellin, Rafal Baranowski, Michael E. Imhof, Hans-Joachim Wunderlich, Nadereh Hatami, Stefano Di Carlo, and Paolo Prinetto. «Efficient Simulation of Structural Faults for the Reliability Evaluation at System-Level». In: *2010 19th IEEE Asian Test Symposium*. 2010, pp. 3–8. DOI: 10.1109/ATS.2010.10 (cit. on p. 10).

[15] Anderson Sartor, Pedro Becker, and Antonio Beck. «A fast and accurate hybrid fault injection platform for transient and permanent faults». In: *Design Automation for Embedded Systems* 23 (June 2019). DOI: 10.1007/s10617-018-9217-0 (cit. on p. 11).

[16] Sparsh Mittal. «A survey on modeling and improving reliability of DNN algorithms and accelerators». In: *Journal of Systems Architecture* 104 (2020), p. 101689. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2019.101689. URL: https://www.sciencedirect.com/science/article/pii/S1383762119304965 (cit. on p. 11).

[17] Austin P. Arechiga and Alan J. Michaels. «The Robustness of Modern Deep Learning Architectures against Single Event Upset Errors». In: *2018 IEEE High Performance extreme Computing Conference (HPEC).* 2018, pp. 1–6. DOI: `10.1109/HPEC.2018.8547532` (cit. on p. 11).

[18] Ralf Graafe, Qutub Syed Sha, Florian Geissler, and Michael Paulitsch. *Large-Scale Application of Fault Injection into PyTorch Models – an Extension to PyTorchFI for Validation Efficiency.* 2023. arXiv: `2310.19449 [cs.AI]`. URL: `https://arxiv.org/abs/2310.19449` (cit. on pp. 11, 24, 38).

[19] 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: *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis.* 2017, pp. 1–12 (cit. on pp. 11, 13, 14, 38).

[20] Annachiara Ruospo, Ernesto Sanchez, Marcello Traiola, Ian O'Connor, and Alberto Bosio. «Investigating data representation for efficient and reliable Convolutional Neural Networks». In: *Microprocessors and Microsystems* 86 (2021), p. 104318. ISSN: 0141-9331. DOI: `https://doi.org/10.1016/j.micpro.2021.104318`. URL: `https://www.sciencedirect.com/science/article/pii/S0141933121004786` (cit. on p. 11).

[21] P.W.D. DependableSystemsLab. *TensorFI 2: A fault injector for TensorFlow 2 applications.* `https://github.com/DependableSystemsLab/TensorFI2`. 2013 (cit. on pp. 11, 22, 79).

[22] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. «LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults». In: *2015 IEEE International Conference on Software Quality, Reliability and Security.* 2015, pp. 11–16. DOI: `10.1109/QRS.2015.13` (cit. on p. 11).

[23] Saurabh Jha, Subho S. Banerjee, Timothy Tsai, Siva K. S. Hari, Michael B. Sullivan, Zbigniew T. Kalbarczyk, Stephen W. Keckler, and Ravishankar K. Iyer. *ML-based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection.* 2019. arXiv: `1907.01051 [cs.LG]`. URL: `https://arxiv.org/abs/1907.01051` (cit. on p. 11).

[24] George Papadimitriou and Dimitris Gizopoulos. «Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers». In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* 2021, pp. 902–915. DOI: `10.1109/ISCA52012.2021.00075` (cit. on p. 11).

[25] Majid Sabbagh, Cheng Gongye, Yunsi Fei, and Yanzhi Wang. «Evaluating Fault Resiliency of Compressed Deep Neural Networks». In: *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. 2019, pp. 1–7. DOI: 10.1109/ICESS.2019.8782505 (cit. on p. 13).

[26] Sparsh Mittal, Himanshi Gupta, and Srishti Srivastava. «A survey on hardware security of DNN models and accelerators». In: *Journal of Systems Architecture* 117 (2021), p. 102163. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2021.102163. URL: https://www.sciencedirect.com/science/article/pii/S1383762121001168 (cit. on p. 14).

[27] Lei Ma et al. *DeepMutation: Mutation Testing of Deep Learning Systems*. 2018. arXiv: 1805.05206 [cs.SE]. URL: https://arxiv.org/abs/1805.05206 (cit. on p. 15).

[28] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC]. URL: https://arxiv.org/abs/1605.08695 (cit. on p. 16).

[29] Zitao Chen, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. *TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications*. 2020. arXiv: 2004.01743 [cs.DC]. URL: https://arxiv.org/abs/2004.01743 (cit. on pp. 17, 38).

[30] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. «BinFI: an efficient fault injector for safety-critical machine learning systems». In: Nov. 2019, pp. 1–23. ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356177 (cit. on p. 19).

[31] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. «Ares: A framework for quantifying the resilience of deep neural networks». In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465834 (cit. on p. 20).

[32] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703 (cit. on p. 23).

[33] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. «PyTorchFI: A Runtime Perturbation Tool for DNNs». In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2020, pp. 25–31. DOI: 10.1109/DSN-W50199.2020.00014 (cit. on p. 23).

[34] Carlo Cena, Umberto Albertin, Mauro Martini, Silvia Bucci, and Marcello Chiaberge. *Physics-Informed Real NVP for Satellite Power System Fault Detection*. 2024. arXiv: `2405.17339 [cs.LG]`. URL: `https://arxiv.org/abs/2405.17339` (cit. on p. 26).

[35] Ole Mengshoel, Adnan Darwiche, Keith Cascio, Mark Chavira, Scott Poll, and Serdar Uckun. «Diagnosing Faults in Electrical Power Systems of Spacecraft and Aircraft.» In: vol. 3. Jan. 2008, pp. 1699–1705 (cit. on p. 32).

[36] «IEEE Standard for Floating-Point Arithmetic». In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: `10.1109/IEEESTD.2019.8766229` (cit. on p. 35).

[37] Xun Jiao et al. *PVF (Parameter Vulnerability Factor): A Scalable Metric for Understanding AI Vulnerability Against SDCs in Model Parameters*. 2024. arXiv: `2405.01741 [cs.CR]`. URL: `https://arxiv.org/abs/2405.01741` (cit. on p. 38).

[38] Keras Team. *Keras 3 Documentation*. `https://keras.io/keras_3/`. Accessed: 2025-05-23. 2024 (cit. on p. 41).