Proceedings of the 33rd European Safety and Reliability Conference (ESREL 2023)
Edited by Mário P. Brito, Terje Aven, Piero Baraldi, Marko Čepin and Enrico Zio
 Published by Research Publishing, Singapore.
doi: 10.3850/978-981-18-8071-1_P400-cd

SynthiCAD: Generation of Industrial Image Data Sets for Resilience Evaluation of Safety-Critical Classifiers

Berit Schuerrle
IAS, University of Stuttgart, Germany. E-mail: berit.schuerrle@ias.uni-stuttgart.de

Venkatesh Sankarappan
IAS, University of Stuttgart, Germany. E-mail: st180516@stud.uni-stuttgart.de

Andrey Morozov
IAS, University of Stuttgart, Germany. E-mail: andrey.morozov@ias.uni-stuttgart.de

Due to their versatility, Deep Neural Networks are becoming increasingly relevant for the industrial domain. However, there are still challenges hindering their application, such as the lack of high-quality training data and suitable methods for assessing their robustness to internal computing hardware faults in safety-critical applications. To address these challenges, this paper introduces (i) a new data generation tool SynthiCAD for creating customisable image training data, along with an open-source industrial data set for classification generated by SynthiCAD. In addition, (ii) we categorized and compared existing approaches to fault injection and evaluated software-based fault injection using a VGG19 model trained on our new data set. Our findings show that software-based fault injection is a fast and scalable way to assess the reliability of DNNs under the presence of faults.

Keywords: Fault Injection, Robustness, Resilience, DNN, Synthetic Data, Computer Vision.

1. Introduction

With the continuous advancement of Artificial Intelligence (AI), the industrial sector has shown increasing interest in leveraging its potential advantages and applying it to various industrial applications. However, the implementation of AI in industry still faces significant challenges. One of them is the unavailability of suitable training data, especially for computer vision tasks such as object detection for quality control or classification for automated item picking. The reason for this is that collecting and labeling large quantities of training data can be both expensive and resource-intensive, and companies are often reluctant to share data due to confidentiality concerns.

A second challenge is ensuring the robustness of AI when implemented in safety-critical applications. Due to the black box nature of Deep Neural Networks (DNNs), formal methods are not applicable when assessing the reliability. A promising approach to overcome this is the targeted introduction of faults into the system. Fault injection tools allow the user to inject errors into a specific part of the network and observe the systems behaviour. At the moment, there is no structured overview of these tools and their methods, making it difficult to compare and evaluate the most suitable one.

To address these challenges, this paper makes two contributions:

(i) Industrial Data Set and Generative Tool: We introduce SynthiCAD, an open-source tool, that allows the creation of customisable image training data, which we used to generate a public data set specifically designed to fit industrial applications.
(ii) Categorization and Evaluation of FI Tools: We categorize existing fault injection methods for DNNs and evaluate software-based fault injection on our new data set.

2. Data Set

The availability of high-quality training data is one of the most critical bottlenecks limiting the application of neural networks for industrial tasks. Although there are several public data sets, suitable training data for the industrial domain is

Table 1. Comparison of related Data Sets for Computer Vision.

Data set	Size	Classes	RGB	Real	Labelled	Content	Industrial
ImageNet	14.2 Mio.	21,841	yes	yes	yes	Misc.	no
CIFAR100	60,000	100	yes	yes	yes	Misc.	no
CoCo	328,000	171	yes	yes	yes	Misc.	no
MVTEC ITODD	3500	28	no	yes	partially	Industrial	yes
Casting Product	7348	2	no	yes	yes	Industrial	yes
SORDI	800,000	80	yes	no	yes	Logistics	yes
SynthiCAD Data Set	**100,000**	**10**	**yes**	**no**	**yes**	**Industrial**	**yes**

scarce, as the collection and creation of this data, particularly images, can be expensive and time-consuming. Synthetic image generation is a promising alternative to extensive data collection. In this paper, we present SynthiCAD, a tool that enables the automated creation of customised data sets based on CAD models. We used this tool to generate a new labelled data set, which we also compare to other relevant computer vision data sets.

2.1. *Data Set Benchmark*

Table 1 summarizes several well known data sets for computer vision tasks. One of the most extensive data sets is ImageNet, which contains over 14 million images and features numerous classes ranging from common household objects to different animals and plants. Similarly, CIFAR-100 and the CoCo data set also focus on daily objects and classes. However, for industrial applications, these benchmark data sets are not suitable as they do not offer industry-specific content.

When analysing the few existing industrial data sets it becomes apparent, that they are often relatively small, may not fully represent the diverse range of industrial applications or lack sufficient labeling. The Casting Product data set published by Kantesaria et al. (2020), which aims to identify defective parts in a casting product, suffers from the common scarcity of negative or defect samples, resulting in a relatively small data set. Akar et al. (2022) released an extensive Synthetic Object Recognition Dataset for Industries (SORDI), with currently more than 800,000 images and 80 classes. However, as this data was generated in collaboration with the BMW group, it is highly specific for the automotive and logistic sector, making it not generalisable for other industry related tasks. The Industrial 3D Object Detection Dataset, introduced by Drost et al. (2017), provides a broad range of real images depicting different objects in various scenes. However, the labeling of objects in the images is only partially complete, with some instances missing labels.

This lack of suitable benchmark data sets for industrial applications has been a challenge for the computer vision community. Through the introduction of our newly generated data set and SynthiCAD for the customisable creation of individual data sets we want to contribute to addressing this shortcoming.

2.2. *SynthiCAD*

The foundation of our data generation tool is based on BlenderProc, a python library developed by Denninger et al. (2023), that utilises Blender's Python API to facilitate the automation of synthetic data generation. The library provides a rich set of functions and classes that users can leverage to create and manipulate 3D models, configure scenes, and produce data for computer vision tasks. BlenderProc is distributed under the MIT License, making it open-source and available for free to the community.

SynthiCAD, as illustrated in Fig. 1, imports 3D object models in various formats, such as *.obj, *.ply, and *.blend, and generates three different labelled data sets - train, test, and validation data. The user has the flexibility to adjust various parameters in the tool to tailor the generated data set to their requirements. This includes (i) the number of 3D models, (ii) the scene composition (e.g.

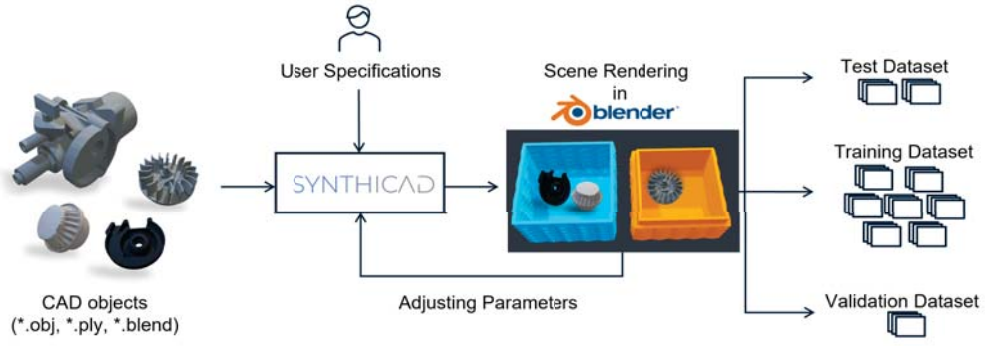

Fig. 1. This illustration presents the image rendering pipeline of SynthiCAD.

number of objects per scene), (iii) the splitting ratio of the resulting data sets, (iv) the image resolution and output formats (COCO or .hdf5), (v) different rendering settings for light and camera, as well as (vi) various object properties like position and material and (vii) the background scene. SynthiCAD provides an easy-to-use configuration file that allows selecting and adjusting these parameters. Through this, the user can customize the tool to meet their specific needs and generate high-quality synthetic data with ease.

2.3. *Industrial Data Set*

We utilized ten CAD models of industrial objects from Drost et al. (2017) as input for our tool. The selected objects include a diverse range of engine parts and components such as screws, injection pumps, and T-connectors. From these ten classes we generated 10.000 images each, resulting in a data set with 100.000 JPEG-images in total. The images are distributed among three subsets: one for training, one for testing, and one for validation, with a ratio of 0.7, 0.2, and 0.1, respectively. All images are RGB and have a resolution of 224x224. To ensure the suitability of our data set for various computer vision tasks, we included not only the class labels but also generated bounding boxes and semantic masks for each image, which are stored in a separate annotation file in the coco format. Each image contains one instance of the ten selected objects. Throughout the 10,000 images for each class, we randomly varied the position of the object in the x-y-z direction and the object's rotation to provide a diverse range of images. Figure 2 provides an example of the data set, showing three different classes (a, b, c) with three variations of the object's position and rotation. Additionally, we changed the object's surface to a smooth metallic texture, imitating real industrial components. Lastly, we varied the lighting conditions within each image, including the position of the light sources, their energy, and emission strength.

Our tool and the data set are open-source and publicly available at: `https://github.com/mbsa-tud/SynthiCAD`

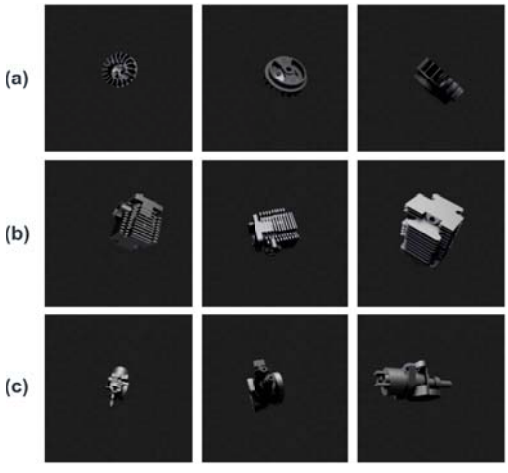

Fig. 2. These images give an insight in the data set showing three different objects (a, b, and c) with varied positions.

3. Fault Injection for Deep Neural Networks

As neural networks are by definition ”black-box” systems, formal approaches for the resilience assessment are not applicable. A promising approach to overcome this is the targeted introduction of faults into the system. These so-called fault injection tools allow the user to inject different errors into a specific part of the network and observe the systems behaviour. This section discusses the state of the art regarding fault injection, introduces concepts and methods and clusters the existing tools and approaches.

3.1. *Introduction to Fault Injection*

Failures in memory or other parts of the computing hardware for the deployment of a neural network can cause silent data corruptions, which can lead to perturbation of the output and result in unreliable predictions of the network. These faults can occur due to hardware or software failures as well as environmental influences resulting in erroneous outputs. Fault injection methods involve simulating such faults to identify potential vulnerabilities before deployment, allowing for a more resilient design of the system. This assessment of the error resilience is especially relevant for safety-critical systems, where even a single bit flip can have severe consequences.

3.2. *Classification of Fault Injection Methods*

There are various ways to classify the different methods of inducing faults within a neural network. One approach is to categorize them based on the types of faults that can be injected. Deep neural networks (DNNs) are primarily susceptible to faults in two areas: input data and network computation. For input data, researchers like Hendrycks and Dietterich (2018) have evaluated the impact of image corruptions on the network's robustness. When it comes to the computation, different fault types could occur, such as bit flips, quantization errors, as well as packet losses in the network or timing problems like jitter.

Another classification approach is based on the location of the fault injection, as discussed in Ruospo et al. (2020) and Ruospo et al. (2021). Figure 3 summarizes this classification. There are two broad categories of fault injection approaches: software-based and hardware-based. Hardware-based fault injection can be grouped into three categories: injecting simulated faults at the register transfer level, injecting simulated faults on an emulated physical device, and physically inducing faults on real hardware. These methods have been demonstrated in studies like Ruospo et al. (2020), De Sio et al. (2020), and Breier et al. (2018).

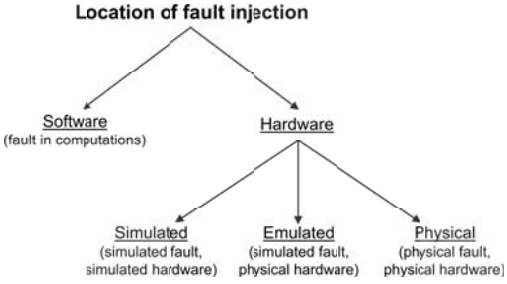

Fig. 3. Classification of Fault Injection Methods.

Software-based approaches do not require knowledge of the specific hardware used and either manipulate saved parameters of the network (static fault) or corrupt the layer output during inference (dynamic fault). In this paper we are focusing on software-based fault injection, as our further research aims to evaluate and uncover weaknesses of the architectures in general rather than a specific inference on certain hardware.

3.3. *Software-based Approaches*

We have identified and compared three openly available tools (see Table 2), that allow a software-based fault injection in DNNs. In the following few chapters, each tool and it's underlying principle will be introduced.

3.3.1. *Ares*

Ares is a DNN-specific fault injection framework first introduced by Reagen et al. (2018). Rather than only offering the injection of faults, Ares provides an entire framework for the resilience assessment, including the training, fault injection and evaluation. It addresses the three most prominent fault points within a network: weights, acti-

Table 2. This table compares the three software-based fault injection tools Ares, InjectTF and TensorFI.

Tool	FI Location	Fault types	Supported Models
Ares	SW	Bit Flips, Noise, others*	FC, Conv., GRU
InjectTF	SW (Operations)	Zero, Random Value, Bit Flips	Add, Sub, Mul, ReLU
InjectTF2	SW (Layer wise)	Bit Flips	Sequential Models
TensorFI	SW (Operations)	Zero, Random Value, Bit Flips	No limitations specified
TensorFI2	SW (Layers)	Zero, Random Value, Bit Flips	No limitations specified

*: extendable with own mathematical computations

vation and hidden states. It supports static faults occurring in the weights during the training as well as dynamic faults within the activations and hidden states during the inference of a DNN. For the static faults, Ares manipulates the saved weights of the model, while for dynamic injection of faults in the activation or state it uses element-wise tensor operations. To this date, it supports the injection in fully connected and convolutional layers as well as GRU's.

3.3.2. *InjectTF(2)*

InjectTF is a fault injection framework published by Beyer et al. (2020) and is designed for inducing faults into TensorFlow models. In accordance to the TensorFlow versions, there are two frameworks publicly available: InjectTF and InjectTF2.

InjectTF is a fault injector designed for TensorFlow 1 models. The faults are injected by manipulating operations such as Add, Sub, Mul or ReLU. To do this, InjectTF builds a corrupted counterpart of the initial graph by duplicating all operations and wrapping the injected ones to manipulate their results. Beyer et al. (2019)

The second framework, InjectTF2, is used for the fault injection in Keras models and allows layer-wise fault injection. To this date, it is limited to sequential models and bit flips. To inject the faults in the desired output layer, the values thereof are manipulated by flipping a specific or random bit.

3.3.3. *TensorFI(2)*

Similar to InjectTF, TensorFI is a framework to induce faults in TensorFlow models. It is also implemented for both TensorFlow versions.

TensorFI works on an operation level for TensorFlow 1 models. It builds a replica of the existing graph with new operators, which allow for the manipulation during the execution of these operations.

TensorFI2 allows the layer-wise injection of faults in Keras models. Besides static faults in the saved weights and biases of the network, it is also able to induce dynamic faults in the activations during inference. It supports regular bit flips, zeros or random values.

4. Evaluation

After illustrating the working principles of these FI tools, we want to evaluate the performances of software-based fault injection on our new data set. For this, we have created a testing scenario, that will be specified in the following chapter. Afterwards, the results are presented and lastly summarized.

4.1. *Methodology*

To evaluate the performance of this approach, we wanted to test it on a self-trained model. For this we defined the following testing scenario: *We have a VGG19, that was trained on our new data set and want to inject random bit flips in a specific layer.*

4.1.1. *VGG 19*

The VGG19 architecture is one of the most prominent versions of a convolutional network. The architecture consists of 16 convolutional and 3 fully connected layers and is characterized by the use of small convolutional filters. Its deep stacking of these layers and the smaller filters allows for the

extraction of more complex features. Additionally, VGG19 has a relatively simple and uniform architecture, making it easy to understand and modify. For this reason, it is well accepted and often used in the industrial sector. Due to its interpretability and the simple architecture, we decided to use this specific model for our test. Table 3 provides an overview of the most relevant training parameters.

Table 3. This table provides an overview about the most relevant training parameters.

Parameter	Value
Convolutional Layers	16
Fully Connected Layers	3
Trainable Parameters	139,611,210
Training Data	70,000 images
Validation Data	10,000 images
Batch size	32
Epochs	15
Optimizer	RMSProp
Learning rate	0.0001
Training duration*	95 min
Test Accuracy	96.06%

*: trained on a Geforce RTX 3090 Ti

4.1.2. *Testing Procedure*

In order to determine the most suitable software-based fault injection tool for our testing purposes, we evaluated the tools from Table 2. We found that the ares framework was not suitable for our use case, as it required significant changes to the source code in order to integrate our own model. We also encountered compatibility issues with InjectTF2. Ultimately, we decided to use TensorFI2 for our testing.

To conduct our fault injection tests, we statically induced a number of random bit flips in the third layer of the fourth convolutional block within the VGG19 network. This means, that a values within this layer is randomly selected and a bit in the memory of that value is flipped. We then evaluated and compared the accuracy of the resulting network using a set of 2,000 test images. To assess the impact of bit flips on the network's accuracy, we gradually increased the number of simultaneous bit flips from 1 up to 250 in increments of 10.

All of our tests were conducted on the same hardware setup, which consisted of a Geforce RTX 3090 Ti in combination with an AMD Ryzen 9 processor. By keeping the hardware consistent across all tests, we ensure that any differences in accuracy were solely due to the number of bit flips injected and not influenced by variations in hardware performance.

4.2. *Results*

4.2.1. *Accuracy Assessment*

When conducting our initial experiments, we observed that the accuracy of our model varied greatly when rerunning the tests, especially for smaller numbers of bit flips. This can likely be attributed to the random selection of the bits. Each run generated a new set of randomly selected bits, which may have had different levels of relevance for the computation, leading to differing levels of accuracy in the results.

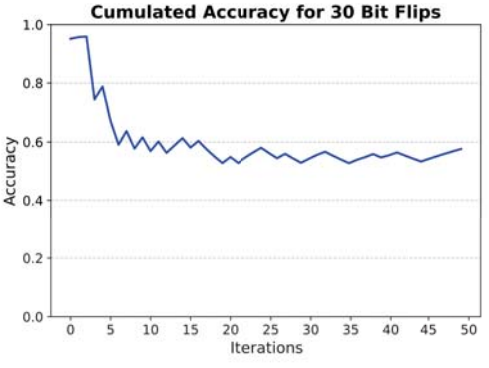

Fig. 4. Cumulative accuracy over the course of 50 iterations for 30 bit flips.

To address this issue and make our results more repeatable, we decided to rerun each experiment multiple times for each number of bit flips, and calculate the cumulative average over 50 iterations. Over this amount of iterations, the accuracy for the specific number of bit flips converged to a certain value. Fig 4 shows the cumulative aver-

aged accuracy over 50 iteration for the injection of 30 bit flips at once. We repeated these 50 iterations for all numbers of bit flips and determined the accuracy value that the experiment converged to. The resulting plot illustrating the accuracy for each number of bit flips can be seen in Fig 5.

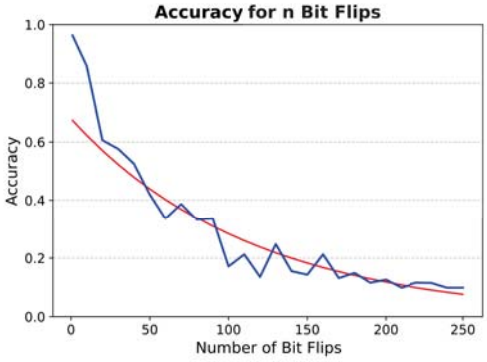

Fig. 5. Overall accuracy for n bit flips

By taking the converging value as the final result, we were able to mitigate any potential fluctuations or biases introduced by the random bit selection. This allowed us to obtain a more accurate representation of the impact of bit flips on our computation for each specific number of bit flips tested. As shown in Fig 5, it is clear that the overall accuracy of the network decreases exponentially as the number of bit flips induced increases. This trend is indicated by the red regression line in the graph. By analyzing this trend, we can gain important insights into the behavior of the network under different levels of bit flips, and use this information to optimize it for better performance.We can, for example, identify the critical threshold of bit flips beyond which the accuracy drops significantly.

4.2.2. *Time*

In addition to measuring the accuracy of the network's prediction under different levels of bit flips, we also recorded the time it took to inject the bit flips for each experiment. As shown in Fig 6, the graph displays a gradual and linear increase in time with the number of bit flips induced, with

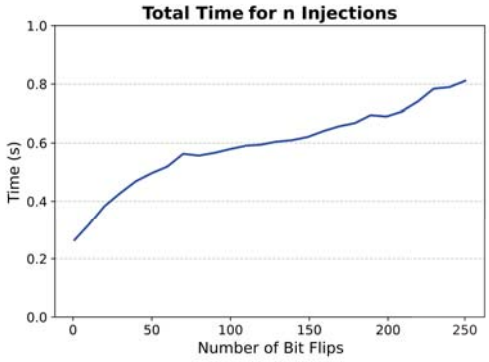

Fig. 6. Total time for injecting n bit flips.

a little over 0.2 seconds for a single bit flip, up to around 0.8 seconds for 250 bit flips. This shows, that the static injection of faults into a neural network hardly cause any additional overhead and is quite efficient.

4.3. *Summary*

In summary, the study on our new data set has demonstrated that software-based fault injection is a fast and scalable method for inducing bit flips in DNNs. We found that the accuracy of our classifier exponentially declines as the number of bit flips increases, which can have significant implications for the reliability and robustness of the system. To further evaluate the vulnerability of the entire network, we recommend repeating the experiments for all layers in order to identify the most vulnerable one.

5. Conclusion

In this paper we introduced SynthiCAD, a new data generation tool, as well as an open-source data set for classification tasks. With minor modifications, SynthiCAD also allows the creation of additional data sets, which feature multiple instances of different objects for object detection and segmentation tasks as well as higher resolution images. We plan to publish an example image data set for these computer vision tasks as well. In addition, our tool is constantly undergoing optimization, and we are actively exploring various post-processing methods to enhance the realism

of the generated output. Furthermore, we conducted a comprehensive analysis of existing fault injection approaches, comparing several software-based methods, including Ares, InjectTF(2), and TensorFI(2). To test these methods, we used a VGG19 network that had been trained on our newly generated data. In order to demonstrate the resilience analysis using the data set we implemented TensorFI2 to induce varying numbers of bit flips in a specific layer, and evaluated the effects on the network's accuracy. We also measured the time required for the fault injection. Such experiments allow to identify the most critical components and uncover areas for improvement, making them essential for the resilience analysis of DNNs.

References

Akar, C. A., J. Tekli, D. Jess, M. Khoury, M. Kamradt, and M. Guthe (2022, October). Synthetic object recognition dataset for industries. In *2022 35th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*. IEEE.

Beyer, M., A. Morozov, K. Ding, S. Ding, and K. Janschek (2019, oct). Quantification of the impact of random hardware faults on safety-critical ai applications: Cnn-based traffic sign recognition case study. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Los Alamitos, CA, USA, pp. 118–119. IEEE Computer Society.

Beyer, M., A. Morozov, E. Valiev, C. Schorn, L. Gauerhof, K. Ding, and K. Janschek (2020). Fault injectors for tensorflow: Evaluation of the impact of random hardware faults on deep cnns. *CoRR abs/2012.07037*.

Breier, J., X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu (2018). Practical fault attack on deep neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, New York, NY, USA, pp. 2204–2206. Association for Computing Machinery.

De Sio, C., S. Azimi, and L. Sterpone (2020). An emulation platform for evaluating the reliability of deep neural networks. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–4.

Denninger, M., D. Winkelbauer, M. Sundermeyer, W. Boerdijk, M. Knauer, K. H. Strobl, M. Humt, and R. Triebel (2023). Blenderproc2: A procedural pipeline for photorealistic rendering. *Journal of Open Source Software 8*(82), 4901.

Drost, B., M. Ulrich, P. Bergmann, P. Hartinger, and C. Steger (2017, October). Introducing MVTec ITODD — a dataset for 3d object recognition in industry. In *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*. IEEE.

Hendrycks, D. and T. G. Dietterich (2018). Benchmarking neural network robustness to common corruptions and perturbations. *CoRR abs/1807.01697*.

Kantesaria, N., P. Vaghasia, J. Hirpara, and R. Bhoraniya (2020). Casting product image data for quality inspection. `https://www.kaggle.com/datasets/ravirajsinh45/real-life-industrial-dataset-of-casting-product`.

Reagen, B., U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei (2018). Ares: A framework for quantifying the resilience of deep neural networks. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA. Association for Computing Machinery.

Ruospo, A., A. Balaara, A. Bosio, and E. Sanchez (2020). A pipelined multi-level fault injector for deep neural networks. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6.

Ruospo, A., L. M. Luza, A. Bosio, M. Traiola, L. Dilillo, and E. Sanchez (2021). Pros and cons of fault injection approaches for the reliability assessment of deep neural networks. In *2021 IEEE 22nd Latin American Test Symposium (LATS)*, pp. 1–5.