Abstract—Research on Deep Neural Networks (DNNs) has focused on improving performance and accuracy for real-world deployments, leading to new models, such as Spiking Neural Networks (SNNs), and optimization techniques, e.g., quantization and pruning for compressed networks. However, the deployment of these innovative models and optimization techniques introduces possible reliability issues, which is a pillar for DNNs to be widely used in safety-critical applications, e.g., autonomous driving. Moreover, scaling technology nodes have the associated risk of multiple faults happening at the same time, a possibility not addressed in state-of-the-art resiliency analyses.

Towards better reliability analysis for DNNs, we present enpheeph, a Fault Injection Framework for Spiking and Compressed DNNs. The enpheeph framework enables optimized execution on specialized hardware devices, e.g., GPUs, while providing complete customizability to investigate different fault models, emulating various reliability constraints and use-cases. Hence, the faults can be executed on SNNs as well as compressed networks with minimal-to-none modifications to the underlying code, a feat that is not achievable by other state-of-the-art tools.

To evaluate our enpheeph framework, we analyze the resiliency of different DNN and SNN models, with different compression techniques. By injecting a random and increasing number of faults, we show that DNNs can show a reduction in accuracy with a fault rate as low as $7 \times 10^{-7}$ faults per parameter, with an accuracy drop higher than 40%. Run-time overhead when executing enpheeph is less than 20% of the baseline execution time when executing 100,000 faults concurrently, at least 10× lower than state-of-the-art frameworks, making enpheeph future-proof for complex fault injection scenarios.

We release the source code of our enpheeph framework under an open-source license at https://github.com/Alexei95/enpheeph.

Index Terms—Deep Neural Networks, Resiliency, Spiking Neural Networks, Compressed Networks, Quantized Neural Networks, Sparse Neural Networks, Fault Injection

I. INTRODUCTION

In the last decade, Deep Neural Networks (DNNs) have seen an exponential increase in practical applications [1], [2], due to their ability to learn complex patterns beyond classical hard-coded algorithms. A possible application is autonomous driving, which is becoming more prominent at different capability levels [3]. However, strong error-tolerance and resiliency are required to reach high autonomous capability levels, as detailed in ISO 26262 [4], which indicates a Failure In Time (FIT) rate of fewer than 100 failures in 1 billion hours of operation for the highest safety level. The exponential increase of multiple upset events in advanced technology nodes [5], [6], makes this threshold complex to achieve and maintain.

Even though resiliency to faults and errors is of foremost importance, there have been few in-depth resiliency analyses for the effect of faults on DNNs. Some examples focus on permanent faults [7]–[12], while others only consider Convolutional Neural Networks (CNNs) [13]–[16]. These tools focus on analyzing a single fault happening at a certain time inside the model, a fault model which is bound to be superseded by multiple fault events due to the aforementioned technology scaling. Hence, state-of-the-art tools are not optimized for scalability, making it very difficult to inject multiple faults in the models without affecting the run-time.

Additionally, many new techniques and architectures for improving DNN efficiency have been developed, such as quantization [17], pruning [17] or Spiking Neural Networks (SNNs) [18]–[21], making them more challenging to analyze using traditional methodologies. As state-of-the-art tools are tailored to specific platform/model configurations, it proves that it is difficult to quickly adapt them to the constantly-evolving model space and optimization techniques.

A. Motivational Case Study

We show a comparison of run-time overhead when using different state-of-the-art fault injection frameworks in Fig. 1. By running from 1 to 100,000 injections, we can see how much overhead is incurred when using multiple frameworks, which grows exponentially. Hence, using these frameworks for multiple faults makes injection experiments very slow, which affects the system design phase. Additionally, these frameworks are not easily adaptable to new technologies or different deep learning libraries, as their code is tied to the specific framework and neural network on which they are implemented. Hence, we can see how a scalable and adaptable framework is necessary for making resiliency analysis of DNNs future-proof.

B. Research Questions

The aforementioned case study leads us to formulate the following research questions:

- How can we maintain performance when executing multiple fault injections simultaneously?
We can note the exponential growth pattern against the number of concurrently injected faults.

- How can we develop a generic fault injection framework capable of adapting to different models with minimal modifications?
- How can we carry out the resiliency analysis for SNNs and compressed DNNs?

C. Novel Contributions

To answer the research questions, we provide the following novel contributions:

- we develop enpheeph, a modern fault injection framework, capable of handling multiple fault injections with minimum overhead, and adaptable to all models and configurations with minimal-to-none modifications;
- we release enpheeph under an open-source license at https://github.com/Alexei95/enpheeph;
- we employ enpheeph to analyze the resiliency of different DNNs, as well as SNN for gesture recognition, employing different compression techniques;

After a brief background and related work analysis, in Sections II and III, we discuss the methodology and the implementation behind our enpheeph framework in Section IV. Then, we show our experimental setup in Section V, and analyze the fault injection results in Section VI. We draw the conclusion on our work in Section VII.

II. BACKGROUND

A. Fault Injection for Neural Networks

Fault injection is used to test the behaviour of a system when an unexpected state is erroneously reached. Faults are classified mainly into two types, transient, which disappear after a concise time interval, and permanent, which are not repairable. Also, depending on the outcome of the affected signals, they are categorized as bit-flip if the signal value is inverted or stuck-at if the signal value is stuck at a 0 or 1 in bit value. Our focus will be on transient faults, which are caused by particles interacting with the hardware and flipping the signal values. In the case of neural networks, these can happen in different locations. However, when using software-level injection methodologies, only a limited set of faults can be covered without prior knowledge of the inner workings of the software and hardware platforms. This means the only directly addressable elements are the tensor values and their indices for layer weights and outputs, representing faults happening in memory locations for the weights and the temporary layer outputs.

B. Spiking Neural Networks

Along the line of brain-inspired neural networks, Spiking Neural Networks (SNNs) have been developed recently [18], [19], which use temporal correlation and increase the amount of information learned from the inputs, due to the differential equation that governs the neuron outputs. In this way, fewer data inputs are required to reach a similar accuracy as of standard DNNs [22], while the complexity of each input is higher. This increase in input complexity leads to increased computational complexity, which also opens possibilities for resiliency issues. The main difference between a standard artificial neuron and a spiking neuron is that in the latter, the output is driven by a differential equation, which encodes additional information based on the time correlation of the inputs, as can be seen in Figure 2. SNNs have seen an enormous increase in practical applications, especially for complex tasks [23], [24]. However, they are still not well-integrated in the deep learning frameworks; hence, their development is based on custom hardware accelerators [25].

C. Compression Techniques

Many software techniques are developed to optimize the DNN requirements while maintaining similar accuracy, leading to compressed networks, where their memory and computational requirements are lower than their original counterparts.

An example of these techniques is pruning [17], which reduces the number of neuron connections, i.e., synapses or neurons, as shown in Fig. 3. However, it is most effective when the underlying hardware supports sparse execution [26], such that the operations can be executed on smaller matrices containing the coordinates and the values of the elements, therefore representing only the non-zero elements [27].

On the other hand, quantization [28] reduces the number of bits used to represent the data in the inputs, the outputs, and the weights. It is used during deployment to reduce 32-bit floating-point numbers to 8-bit integers with negligible loss in accuracy. In addition to reducing the memory footprint, the operations can be optimized further, allowing parallel operations to run faster than 32-bit floating-point numbers.
III. RELATED WORK

We compare existing state-of-the-art frameworks in Table I, also covering the features provided by enpheeph. Compared to the other state-of-the-art frameworks, enpheeph is not a collection of scripts, but it has been developed as a homogeneous set of software API primitives. With this approach, enpheeph can be easily adapted to different underlying libraries providing different functionalities, without having to modify its internal code-base. On the other hand, state-of-the-art frameworks do not provide a generic API for easy programmability, but simply a set of configuration knobs which can be accessed via predefined configuration files. This configuration method is limiting as it does not allow for configurations beyond the original scopes of the work. The easy-to-use API additionally enables any new user of the enpheeph framework to write their own customized injection while leveraging the setup/restore backbones provided with enpheeph. An example of this easy implementation is related to different hardware devices, as the original implementation covered only CPU execution, but was programmatically extended to GPU compatibility with almost zero code changes, while state-of-the-art frameworks require extensive updates to allow execution on GPU.

Of all the frameworks, only enpheeph has no implementation details tied with the underlying DNN framework, making it adaptable to different DNN frameworks with minimal-to-none internal code changes. Moving on to the targetability of the elements, enpheeph allows for custom targetability from the bit-level through the tensor-level to the layer-level, allowing a custom number of bit-precise injections during an execution. The state-of-the-art frameworks are limited to a single layer, and there are limits on executing a single specific fault or a random one over the whole tensor. enpheeph provides customizability for the fault and the target type, while providing implementations for the basic fault types, e.g., bit-flips and stuck-at. Other frameworks support only bit-flips, with the only exception being TensorFI2, also supporting stuck-at faults, and PyTorchFI providing only stuck-at injections. Similarly, state-of-the-art frameworks allow for targeting layer weights or outputs, but implementing other injections, e.g., on the temporary buffers, is not allowed, while enpheeph easily allows for such a possibility if required. Moving to compressed networks, enpheeph is the only framework providing full support for sparse tensors and quantization, beyond what is currently offered by DNN framework, as most of them lack direct support. Finally, most of the frameworks run only on CPU, and they are not easily extendable on GPU, with enpheeph having support for additional devices as well.

Overall, state-of-the-art frameworks are not customizable enough for the evolving scenario of models and techniques, and they do not guarantee multiple fault injection capabilities. enpheeph aims at addressing all of the aforementioned issues.

| Library | enpheeph | TensorFlow2 [14] | InjectTF2 [29] | PyTorchFI [30] | TorchFI [31] |
|---------|----------|------------------|----------------|----------------|------------|
| Bit     | TensorCustom | Single Random | Single Random | Single Random | Single Random |
| Targetability | Single Multiple Custom | Random | Random | Single Random | Single Random |
| Tensor | Single Multiple Custom | Single | Single | Single | Single |
| Targetability | Single Multiple Custom | Custom | Custom | Custom | Custom |
| Fault   | Bit-flip Custom | Stuck-at | Stuck-at | Bit-flip | Stuck-at |
| Type    | Weight Output | Weight Output | Output | Weight Output | Weight Output |
| Target   | Weight Output | Output | Output | Weight Output | Weight Output |
| Quantization Support | Full | No | No | Limited | Limited |
| Sparse Tensor Support | Full | No | No | No | Limited |
| Hardware Support | CPU GPU Custom | CPU | CPU | CPU | CPU GPU |

IV. METHODOLOGY

We approach the problem of lightweight fault injection and customizability outlined in Section I, by developing a generic fault injection framework for DNNs, enpheeph. The methodology is shown in Fig. 4, and it will be described in detail in the following sub-sections.

A. Inputs

The enpheeph framework requires 4 inputs:

- **One or more fault models**: they are required to generate the targets for the faults and the monitors to be inserted in the model.
- **The set of hardware and software platforms**: they are required to employ the correct injection handler and optimized operations, as each handler has a different internal implementation depending on the used DNN framework.
- **The target DNN model**: it is used to compute the dimensions of the faults to be injected.
- **The target dataset**: it is required to run the model executions and gather the injection results.

B. Model Setup

In Fig. 4 we can see the “Model Setup” step, which is necessary for setting up the faults and the monitors based on the fault model and the running hardware/software platform. Each fault represents the set of bits which need to be modified by the mask. Each monitor contains which values...
must be saved for each execution, as the modified bits are not automatically logged. In this way, there is full customizability of the values which need to be saved.

The enpheed framework initially creates an injection handler, which is entitled to generate the faults and the monitors based on the fault model and the dimensions of the target DNN model. The faults and the monitors are optimized based on the combination of software and hardware platforms, guaranteeing the lowest possible overhead. The instrumented DNN model, as shown in 2, is used for the “Model Execution” step in the following sub-section.

C. Model Execution

Continuing with Fig. 4 3, the instrumented DNN model is executed with the given data set, meant as a collection of data on which to test, e.g., a subset of the CIFAR10 dataset. All the targeted layers are backed up at the beginning of the execution, and the fault masks are created and placed in the layer execution flows. Then, as shown in the “Fault Injection Process”, the model is executed with the faults, and the results are gathered based on the monitor configurations. This process is repeated for each input in the dataset. When the dataset is extinguished, the injected layers are restored to their backed-up version, and the results are finalized in the output database.

D. Output

There is only one output from the enpheed framework, and it is a database containing the results of all the monitors, as well as the metrics used for evaluating the model, e.g., accuracy for a classification task.

E. Example Implementation: PyTorch Injection Handler Setup

In Algorithm 1, we show an example for the methodology implementation in the enpheed framework: the PyTorch [32] implementation of the setup phase of the injection handler.

This procedure requires the list of injections, where each injection is a structure containing the information of the target layer name, the target type, the fault type and the indices of the target elements.

After selecting the correct layer from the layer name in the injection, the correct target is selected, choosing between output and weight. Then, depending on whether the injection is a fault or a monitor, the mask creation process at line 17 consists of expanding the bit-wise mask to the shape of the target array, so it can be combined with the target at runtime. Finally, we add an execution hook to either update the target with the mask at line 22 or save the target at line 27, if the injection is a fault or a monitor respectively.

If we are injecting a fault, we have developed an automated interface capable of determining on which device type we are operating, e.g., CPU vs GPU vs TPU. In this way the mask injection is tailored to the specific device, employing different back-end libraries and further accelerating the execution.

In this particular implementation, we do not require backing up the layer before the injection handler setup. PyTorch allows using execution hooks that do not leave permanent modifications on the model once removed. The execution hooks are run right before or after the execution of the layer, depending on the chosen hook, allowing for updates at runtime before the execution starts. This enables further speed-ups and savings when compared to the complete backup and restore processes of the target layers.

Algorithm 1 PyTorch Implementation of the Injection Handler Setup

1: \* each injection in the input list is either a fault or a monitor \* 2: \* each injection is a structure containing the necessary information \* 3: procedure INJECTIONHANDLERSETUP(model, listInjections) \* 4: for injection in listInjections do \* 5: \* the layer with the same name as in the injection is selected \* 6: module ← layer from model using injection.layerName \* 7: \* the correct target is selected based on the injection type \* 8: if injection.target == output then \* 9: target ← module.output \* 10: else if injection.target == weight then \* 11: target ← module.weight \* 12: end if \* 13: if injection.type == fault then \* 14: \* if we have a fault we create a mask \* 15: \* from the type of fault and \* 16: \* expand it to the target size \* 17: maskElement ← injection.faultType \* 18: mask ← expand maskElement to target.shape \* 19: \* an execution hook is added to the module \* 20: \* to update the target \* 21: \* the update involves the mask to force the injection \* 22: add exec. hook in module, target ← target + mask \* 23: else if injection.type == monitor then \* 24: \* a similar hook is added to the module \* 25: \* if we are running a monitor, \* 26: \* in order to save the target \* 27: add execution hook in module, to save target \* 28: end if \* 29: end for \* 30: end procedure
V. EXPERIMENTAL SETUP

To evaluate our enpheeph framework, we implement the injection handler and the corresponding components detailed in Section IV. We also train and test multiple models and fault rates on different hardware platforms. All the trained models and their configurations will be available in the open-source release of the framework.

An overview of the experimental setup is given in Fig. 5, where we can see the training loop depending on the models, the chosen device and the different datasets to produce a trained model. The trained model is then fed as input to a similar testing loop, comprising enpheeph and requiring the datasets as well as the fault configurations, to execute the fault injections. The results are gathered into different file formats, namely a CSV file for the accuracy and all the model metrics, and a SQL database for the results gathered by the monitors and a trace of the faults which have been executed.

![Fig. 5: Overview of the used experimental setup for training the DNN models and for testing our enpheeph framework.](image)

A. Hardware

We test our injections on both CPU and GPU, running on AMD Ryzen Threadripper 2990WX [33] and NVIDIA RTX 2080Ti [34].

B. Software

We implement the low-level compatibility layer of the enpheeph framework in PyTorch [32]. However, the components can be easily implemented on other DNN libraries. The injections on the CPU are implemented via NumPy arrays [35], while for the GPU, we use CuPy [36]. We additionally use PyTorch Lightning (PTL) [37] to expedite the models’ training and testing due to its seamless switch between CPU and GPU execution. We employ PyTorch Lightning Flash (PTLF) [38] and PyTorch Lightning Bolts (PTLB) [39] for expediting access to task pipelines, models, and datasets. For SNNs, we employ the norse framework [40] and the tonic dataset collection [41] on top of PyTorch for easier SNN implementation.

C. Models

We choose the VGG11 [42] and ResNet18 [43] models as the backbone for the image classification task on CIFAR10 [44]. Each model is trained with the Adam [45] optimizer, using 0.001 as the learning rate for up to 60 epochs; the early stopping technique limits the training if the loss starts increasing from the minimum reached value [46].

For the SNN, we develop a shallow network with 2 convolutional layers and a fully-connected layer to be run on the DVS Gesture Dataset [47]. It is trained for 25 epochs with the Adam optimizer and 0.001 learning rate.

D. Experiments

We inject a variable number of faults, determined by the fault rate per number of parameters via sampling of a uniform random distribution. We choose to sweep the range from $1 \times 10^{-7}$ until 1, which means we start injecting one fault per ten million parameters until we reach one fault per parameter. For each decade we cover nine different points, e.g. we sweep from $1 \times 10^{-7}$ to $9 \times 10^{-7}$ via $1 \times 10^{-7}$ increases. For each fault rate we iterate over the model layers and inject the faults one layer at a time, selecting the lowest testing accuracy per each fault rate.

For the compressed networks, there is limited support at the time of writing in PyTorch; hence we target only a limited subset of operations. We inject faults only on the indices of the sparse outputs of each layer. To obtain a sparse representation, we convert the dense output to a sparse index-value representation [27], inject the faults on the index array and convert back to a dense format. Similarly, to inject faults on quantized networks, we convert the output of each layer to 32-bit integer from 32-bit floating-point, inject the faults, and convert them back to 32-bit floating-point. To increase the dynamic range in integer representation, as the conversion truncates the original floating-point numbers, we multiply the floating-point by $2^{24}$, so that the original range in floating point is $[-128, 127]$, while keeping a precision of $\approx 6 \times 10^{-8}$ in integer representation. These numbers were verified to be well-above the range of the tensors in the employed models, hence not affecting the output dynamic range.

E. Comparison

We run the AlexNet model [48] on CIFAR10 [44] for testing both TensorFI2 [14] and PyTorchFI [30]. The results for enpheeph are instead obtained averaging all the executions from the results, both on CPU and GPU.

VI. EVALUATION

We evaluate our enpheeph framework based on the experiments mentioned in Section V-D.

A. Execution Time Comparison

A comparison between enpheeph and other state-of-the-art fault injection tools is shown in Fig. 6. We run the fault injection with up to 100,000 faults, as we concurrently run multiple batches and we inject faults in multiple bits. The overhead is computed as percentage with respect to the baseline, which is the execution of the models without any injection. This scenario is realistic for faults that might happen in critical control logic, modifying many values at the same time. Our enpheeph framework is much faster for multiple faults, achieving less than 20% overhead at 100,000 faults, against the 200% of PyTorchFI and the 100,000,000% of TensorFI2. Additionally, enpheeph shows a linear trend for increasing number of faults, while PyTorchFI and TensorFI2 both show exponential increases.
C. Compressed networks

We will now analyze the results of the fault injections on the layer outputs of different compressed networks, namely the DNNs and the SNN models first with 32-bit integer quantized and then with sparse indices injection.

In Fig. 8 we show the resiliency of the quantized layer outputs, for ResNet18 in (6), VGG11 in (11) and our SNN model in (1). ResNet18 is the least resilient, with the first notable drop in accuracy around $9 \times 10^{-7}$, pointed by (10). VGG11 has a much bigger drop around $3 \times 10^{-5}$, which reaches to 25% accuracy, shown by (11). However, compared to ResNet18, VGG11 has an area between $8 \times 10^{-3}$ and $8 \times 10^{-2}$, as shown by (12), where the accuracy is higher than the random classification even though a higher number of fault is injected. This is related to lower-weight bits being targeted during the fault injection, leading to a lower impact on the total accuracy. For the SNN model, accuracy is constant until $7 \times 10^{-3}$, where it starts alternating low accuracy and high accuracy, as pointed by (13). This behaviour highlights the higher resiliency of the SNN model, even though it shows that quantization has an effect on the resiliency of all the models.

More specifically, when bits closer to the Most Significant Bit (MSB) are hit, the effects are increased compared to floating-point numbers, but if a bit closer to the Least Significant Bit (LSB) is hit, the effects are attenuated.

Regarding sparse networks, we can see the results in Fig. 9 (4). All the networks show limited effects of the injected faults, even at very high fault rates: (14), (15) and (16) show the range of accuracy at various fault rates for ResNet18, VGG11 and the SNN model. The range is slightly bigger for VGG11, being roughly around 20%. This shows that exchanging some tensor values in the output does not affect greatly the accuracy, even though further experiments are needed.

VII. Conclusion

The rising technology of neural networks for critical applications introduces the potential of many faults occurring concurrently, which is not included in state-of-the-art resiliency mitigations and analyses.

To assist in analyzing DNNs reliability, we propose enpheeph, a Fault Injection Framework for Spiking and Compressed DNNs. In enpheeph, fault injection may be executed on both SNNs and compressed networks with little to no modification to the underlying code, a feat that other state-of-the-art tools are incapable of accomplishing. To experiment with our enpheeph framework, we examine the resilience of various DNNs and SNNs when compressed using various approaches. By injecting a random and growing number of faults, we demonstrate that DNNs have their accuracy reduced by more than 40%, when receiving as little as $7 \times 10^{-7}$ random faults. Additionally, enpheeph exhibits 10x less run-time overhead than state-of-the-art fault-injection frameworks. We release the source code of our enpheeph framework under an open-source license at https://github.com/Alexei95/enpheeph.
Fig. 7: Comparison of testing accuracy against random faults for different models running an image classification task on CIFAR10. A shows output injection on ResNet18, B shows output injection on VGG11, C shows output injection on SNN, D shows weight injection on ResNet18, E shows weight injection on VGG11 and F shows weight injection on SNN. The SNN shows higher resiliency than the DNNs.

Fig. 8: Testing accuracy of quantized DNNs against increasing fault injection rate on the outputs. G shows ResNet18, H shows VGG11 and I shows our SNN model. Quantization affects the resiliency with non-trivial patterns when compared to full-precision networks.

Fig. 9: J shows ResNet18, K shows VGG11 and L shows our SNN model.

ACKNOWLEDGMENT
This work has been supported by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien’s Faculty of Informatics and the UAS Technikum Wien. This research is partly supported by the NYUAD’s Research Enhancement Fund (REF) Award on “eDLAuto: An Automated Framework for Energy-Efficient Embedded Deep Learning in Autonomous Systems”, and by the NYUAD Center for Artificial Intelligence and Robotics (CAIR), funded by Tamkeen under the NYUAD Research Institute Award CG010.

REFERENCES
[1] O. I. Abiodun et al., “State-of-the-art in artificial neural network applications: A survey,” Heliyon, vol. 4, no. 11, e00938, 2018-11-01.
[2] S. Dong et al., “A survey on deep learning and its applications,” Computer Science Review, vol. 40, p. 100 379, 2021-05.
[3] On-Road Automated Driving (ORAD) committee, “Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles,” SAE International.
[4] International Organization for Standardization. “ISO 26262-10:2018,” ISO. (n.d). Available: https://www.iso.org/cms/render/live/en/sites/isooorg/contents/data/standard/06/83/68392.html (visited on 2021-12-31).
[5] J. D. Black et al., “Physics of Multiple-Node Charge Collection and Impacts on Single-Event Characterization and Soft Error Rate Prediction,” IEEE Transactions on Nuclear Science, vol. 60, no. 3, pp. 1836-1851, 2013-06.
[6] A. Neale et al., “Neutron Radiation Induced Soft Error Rates for an Adjacent-ECC Protected SRAM in 28 nm CMOS,” IEEE Transactions on Nuclear Science, vol. 63, no. 3, pp. 1912–1917, 2016-06.
[7] J. Zhang et al., “Thundervolt: Enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators,” in Proceedings of the 55th Annual Design Automation Conference, San Francisco California: ACM, 2018-06-24, pp. 1–6.
[8] E. Ozen et al., “Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3250–3262, 2020-11.
[9] B. Reagen et al., “Ares: A framework for quantifying the resiliency of deep neural networks,” in 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018-06, pp. 1–6.
