This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

Fault Injection for TensorFlow Applications

Niranjhana Narayanan†, Zitao Chen†, Bo Fang‡, Guanpeng Li§ Karthik Pattabiraman†, Nathan DeBardeleben¶
†University of British Columbia, ‡Pacific Northwest National Laboratory, §University of Iowa, ¶Los Alamos National Laboratory
{nniranjhana, zitaoc, karthikp}@ece.ubc.ca, bo.fang@pnnl.gov, guanpeng-li@uiowa.edu, ndebard@lanl.gov

Abstract—As machine learning (ML) has seen increasing adoption in safety-critical domains (e.g., autonomous vehicles), the reliability of ML systems has also grown in importance. While prior studies have proposed techniques to enable efficient error-resilience (e.g., selective instruction duplication), a fundamental requirement for realizing these techniques is a detailed understanding of the application's resilience. In this work, we present TensorFI 1 and TensorFI 2, high-level fault injection (FI) frameworks for TensorFlow-based applications. TensorFI 1 and 2 are able to inject both hardware and software faults in any general TensorFlow 1 and 2 program respectively. Both are configurable FI tools that are flexible, easy to use, and portable. They can be integrated into existing TensorFlow programs to assess their resilience for different fault types (e.g., bit-flips in particular operations or layers).
We use TensorFI 1 and TensorFI 2 to evaluate the resilience of 11 and 10 ML programs respectively, all written in TensorFlow, including DNNs used in the autonomous vehicle domain. The results give us insights into why some of the models are more resilient. We also measure the performance overheads of the two injectors, and present 4 case studies, two for each tool, to demonstrate their utility.

Index Terms—Fault Injection, Machine Learning, TensorFlow, Error Resilience, Deep Neural Networks

1 Introduction

In the past decade, Machine Learning (ML) has become ubiquitous. It is being increasingly deployed in safety-critical applications such as Autonomous Vehicles (AVs) [1] and aircraft control [2]. In these domains, it is critical to ensure the reliability of the ML algorithm and its implementation, as faults can lead to loss of life and property. Moreover, there are often safety standards in these domains that prescribe the allowed failure rate. For example, in the AV domain, the ISO 26262 standard mandates that the FIT rate (Failures in Time) of the system be no more than 10, i.e., at most 10 failures in a billion hours of operation [3], in order to achieve ASIL-D levels of certification. ASIL-D refers to Automotive Safety Integrity Level D, which is the ISO standard's most stringent level of safety measures for avoiding life-threatening or fatal situations. Therefore, there is a compelling need to build efficient tools to (1) test and improve the reliability of ML systems, and (2) evaluate their failure rates in the presence of different fault types.

The traditional way to experimentally assess the reliability of a system is fault injection (FI). FI can be implemented at the hardware level or software level. Software-Implemented FI (also known as SWiFI) has lower costs, is more controllable, and easier for developers to deploy [4]. Therefore, SWiFI has become the dominant method to assess a system's resilience to both hardware and software faults and many tools such as NFTape [5], Xception [6], GOOFI [7], LFI [8], LLFI [9], PINFI [10] have been developed. However, most of these tools were not built with ML applications in mind, and hence are not well-suited for evaluating the reliability parameters associated with ML applications. Therefore, it is important to build FI tools for ML applications.

Due to the increase in popularity of ML applications, many frameworks have been developed for writing them. An example is TensorFlow [11], which was released by Google in 2017. Other examples are PyTorch [12] and CNTK [13]. Prior work has studied the error resilience of ML models by building customized fault injection tools [14], [15], [16], [17]. However, these tools are limited to specific ML programs or are platform-dependent. Mutation testing techniques have been used to create mutant ML models that are tested with certain inputs to improve the quality of test data [18], [19] and to localize bugs in the ML frameworks [20], [21]. Nevertheless, there is still a lack of FI tools that can be specifically configured and used for reliability assesment, with the focus on *resilience analysis of different ML models to various fault types*. To the best of our knowledge, PyTorchFI [22] has been developed to address this need for ML applications written in the PyTorch framework. In this paper, we build such a tool set for the TensorFlow framework.

There are three main challenges in developing a FI tool for a specific ML framework. The first challenge is finding the suitable place for injection. The common method of modifying the higher level operators in place with faulty versions and checking with a runtime flag to either inject faults or continue normally does not work since the underlying implementation of the ML frameworks is in C/C++ code or assembly for performance reasons and cannot be modified. The alternative of directly modifying the low level code would render the tool dependent on platform or framework version and harm its portability.

The second challenge is developing the method of injection. Even in the same ML framework, there can be conceptual differences between the versions, which requires

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

different methods. For example, while both TensorFlow 1 and 2 provide high level Python APIs for ease of use, the way the ML model uses those APIs, and runs in the background are quite different from each other.

The third challenge is choosing the right fault types in the tool. It is important that the tool is capable of abstracting a wide variety of fault types and injection modes so that the user can configure it for their particular use case.

To address these challenges, in this paper, we develop two tools, TensorFI 1 and 2 for injecting faults into TensorFlow 1 and 2 applications respectively*. They can inject both hardware and software faults in either the outputs of TensorFlow operators (TensorFI 1) or the states and activations of model layers (TensorFI 2). However, they differ significantly in their implementations, as explained below.

TensorFI 1 works by first duplicating the TensorFlow graph and creating a *FI graph* that parallels the original one. The operators in the FI graph mirror the functionality of the original TensorFlow operators, except that they have the capability to inject faults based on the configuration parameters specified. These operators are implemented by us in Python, thereby ensuring portability. Moreover, the FI graph is only invoked during fault injection, and hence the performance of the original TensorFlow graph is not affected (when faults are not injected). Additionally, because we do not modify the TensorFlow graph other than to add the FI graph, external libraries can continue to work as before.

Unlike TensorFlow 1, TensorFlow 2 applications do not necessarily have an underlying data-flow graph. TensorFI 2 addresses the challenge by using the Keras APIs to intercept the tensor states of different layers directly for fault injection. Graph duplication is avoided along with the overheads it incurs (we quantitatively evaluate the overheads later in Section 4). While TensorFI 1 can only inject into the results of individual operators in the TensorFlow graph, TensorFI 2 can also be used to inject faults into the model parameters such as weights and biases as well as the outputs of different activation or hidden layer states. *Overall, in TensorFI 2, we include support for weight injection, introduce new modes of injection, design and implement a new methodology for FI, conduct more experiments, and develop new case studies.* Like TensorFI 1, TensorFI 2 is also designed to be portable and fully compatible with external libraries. We outline further the design challenges specific to each tool in Section3.

Finally, in both tools, it is easy to specify different fault configurations and readily access the FI results. This is because unlike traditional SWiFi frameworks, TensorFI performs *interface-level FI* [24], [25], directly operating on either the graph nodes or the layer objects of a model. Both TensorFI 1 and 2 incorporate designs that are tailored to the specific version of the TensorFlow framework, and hence use different designs, to meet the same design goals.

We focus on TensorFlow as it is one of the most popular frameworks used today for ML applications [26], though our techniques of graph duplication in TensorFI 1 and layerwise injection in TensorFI 2 are not restricted to TensorFlow and can be applied to other ML frameworks that use the computational dataflow graph model or the layer object model respectively. TensorFI 1 and 2 are open source tools available at *github.com/DependableSystemsLab/TensorFI* and *github.com/DependableSystemsLab/TensorFI2* respectively.

Our tool set contains generic and configurable fault injection tools that are able to inject faults in a wide range of ML programs written using TensorFlow 1 and 2. With the help of our tools, users can conduct error resilience analysis to obtain the worst case estimates of reliability before deploying their model, especially for safety critical applications. Users can also identify the classes of inputs that are vulnerable to faults, and the important features for correct classification in different datasets. Finally, users can determine the operators/layers in a model that are susceptible to various fault types, and fine tune the parameters to improve the model resilience to such faults.

We make the following contributions in this paper.

- Propose generic FI techniques to inject faults in the TensorFlow 1 and 2 frameworks.
- Implement the FI techniques in TensorFI 1 and 2, which allow (1) easy configuration of FI parameters, (2) portability, and (3) minimal interference with the program.
- Evaluate the TensorFI 1 and 2 tools on 11 and 10 ML applications respectively, including deep neural network (DNN) applications used in AVs, across a range of FI configurations (e.g., fault types, error rates). From our experiments, we find that there are significant differences due to both individual ML applications, as well as different configurations. Further, ML applications are more vulnerable to bit-flip faults than other kinds of faults in both the tools. Finally, TensorFI 2 was more than twice as fast as TensorFI 1 for injecting similar faults.
- Conduct four case studies, two for each tool, to demonstrate some of the use cases of the tool. We find the most and least resilient image classes in the GTSRB dataset [27] from fault injection in a traffic sign recognition model. We consider layer-wise resilience in two of our case studies, and observe that faults in the initial layers of an application result in higher vulnerability. In addition, we visualize the outputs from layer-wise injection in an image segmentation model, and are able to identify the layer in which faults occurred based on the faulty prediction masks. These case studies thus provide valuable insights into how to improve the resilience of ML applications.

2 Overview and Fault Model

We start by explaining the general structure of ML applications, followed by related work in the area of ML reliability. We then introduce the fault model we assume in this paper.

2.1 Background

2.1.1 ML Applications

An ML model takes an input that contains specific features to make a prediction. Prediction tasks can be divided into classification and regression. The former is used to classify the input into categorical outputs (e.g., image classification). The latter is used to predict dependent variable values based on the input. ML models can be either supervised or unsupervised. In the supervised setting, the training samples are assigned with known labels (e.g., linear regression, neural

*TensorFI 1 was published in the ISSRE'20 conference [23] for TensorFlow 1 applications. This paper extends that work to also support TensorFlow 2 applications, and adds a more diverse set of injectors.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

network), while in an unsupervised setting there are no known labels for the training data (e.g., k-means, kernel density estimation).

An ML model typically goes through two phases: 1) training phase where the model is trained to learn a particular task; 2) inference phase where the model is used for making predictions on test data. The parameters of the ML model are learned from the training data, and the trained model is evaluated on the test data, which represents the unseen data.

2.1.2 TensorFlow 1 and 2

TensorFlow abstracts the operations in an ML application thus allowing programmers to focus on the high-level programming logic. In TensorFlow 1, programmers use the built-in operators to construct the data-flow graph of the ML algorithm during the *training phase*. Once the graph is built, it is not allowed to be modified. During the *inference phase*, data is fed into the graph through the use of placeholder operators, and the outputs of the graph correspond to the outputs of the ML algorithm. TensorFlow 1 version was difficult to learn and use for ML practitioners, with users having to deal with graphs, sessions and follow a meticulous method of building models [28].

With TensorFlow 2, eager execution was introduced making it more Pythonic. Graphs are not built by default but can be created with tf.function as they are good for speed. TensorFlow 2 embraces the Keras APIs for building models making it easier and more flexible for users. In TensorFlow 2, programmers define the ML model layer by layer and these layer objects have *training* and *inference* features. When data is fed into the ML algorithm, the operations in the layers are immediately executed.

Both versions of TensorFlow also provide a convenient Python language interface for programmers to construct and manipulate the data-flow graphs. Though other languages are also supported, the dominant use of TensorFlow is through its Python interface. Note however that the majority of the ML operators and algorithms are implemented as C/C++ code, and have optimized versions for different platforms. The Python interface simply provides a wrapper around these C/C++ implementations.

2.2 Related Work

In the *hardware faults* space, there has been significant work to investigate the resilience of deep neural networks (DNN) using fault injectors. The earliest such work was by Li et al., who build a fault injector by using the tiny-CNN framework [14]. Reagen et al. design a generic framework for quantifying the error resilience of ML applications [15]. Sabbagh et. al develop a framework to study the fault resilience of compressed DNNs [17]. Chen et al. introduce a technique to efficiently prune the hardware FI space by analyzing the underlying property of ML models [16]. However, the above FI techniques are either limited to the specific application being studied, or are dependent on the underlying hardware platform, unlike TensorFI that is able to perform FI on generic ML applications. The only exception is PyTorchFI [22], which is a generic FI tool for DNNs, for PyTorch applications. PyTorch has a different architecture than TensorFlow, and hence PyTorchFI cannot be easily applied to TensorFlow applications.

In the *software faults* space, researchers have employed conventional software techniques such as mutation testing in ML applications. DeepMutation [18], DeepMutation++ [19] are such frameworks specialized for ML applications written in TensorFlow. These tools support specific source and model level mutations and use their mutant models to evaluate the quality of test data; in contrast, in TensorFI, users can define different fault types, configure the fault location and injection modes for a wide variety of purposes. LEMON [20] and AUDEE [21] are also based on mutation testing, and support many ML frameworks (TensorFlow, Theano, CNTK, MXNet/PyTorch). However, their goal is to evaluate the ML frameworks themselves, while our goal is to evaluate the models.

In summary, the TensorFI tool set targets a broader range of ML applications, gives users the flexibility of emulating different faults, and can be used to inject both software and hardware faults in the ML application.

2.3 Fault Model

In this work, we consider both hardware faults and software faults that occur in TensorFlow programs.

TensorFI 1 operates at the level of TensorFlow graph operations. We abstract the faults to the operators' interfaces. Thus, we assume that a hardware or software fault that arises within the TensorFlow operators, ends up corrupting (only) the respective outputs. However, we do not make assumptions on the nature of the output's corruption. For example, we consider that the output corruption could be manifested as either a random value replacement [18] or as a single bit-flip [14], [15], [16], [17].

TensorFI 2 operates at the TensorFlow model level. We abstract faults either to the interfaces of the model layers or to the model parameters. TensorFI 2 can inject faults into the layer states i.e. the weights and biases. TensorFI 2 models two kinds of faults. (1) Transient hardware faults during computation, which can alter the activations or outputs of each layer. (2) Faults that can occur due to rowhammer attacks [29], i.e., an attacker performing specific memory access patterns can induce persistent and repeatable bit corruptions from software. The vulnerable parameters tend to be larger objects (greater than 1MB) in memory, and these are usually page aligned allocations such as weights, biases.

Table 1 shows the faults considered by both TensorFI 1 and 2, and how they are modeled. We make 3 assumptions about faults. First, we assume that the faults do not modify the structure of the TensorFlow graph or model (since TensorFlow assumes a static computational graph) and that the inputs provided into the program are correct, because such faults are extraneous to TensorFlow. Other work has considered errors in inputs [30], [31]. Second, we assume that the faults do not occur in the ML algorithms or the implementation itself. This allows us to compare the output of the FI runs with the golden runs, to determine if the fault has propagated and a Silent Data Corruption (SDC) has occurred. Finally, we only consider faults during the *inference* phase of the ML program. This is because training is usually a one-time process and the results of the trained

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

TABLE 1: Fault model for the TensorFI tool set

	TensorFI 1	TensorFI 2
Source of fault	Software faults and transient hardware faults	Software faults, transient hardware faults, rowhammer attacks
Modeling of fault	Operator outputs	Layer outputs and layer state (weights)
Fault types	Bitflips, zeros and random value replacement	Bitflips, zeros and random value replacement

model can be checked. Inference, however, is executed repeatedly with different inputs, and is hence much more likely to experience faults. This fault model is in line with other related work [14], [15], [16], [17].

3 Methodology

We start this section by articulating the common design constraints for the 2 tools. We then discuss the design alternatives considered, and then present the design of TensorFI 1 and 2 to satisfy the design constraints.

3.1 Design Constraints

We adhere to the following three constraints in the design of the TensorFI tool set.

- **Ease of Use and Compatibility**: The injectors should be easy-to-use and require minimal modifications to the application code. We also need to ensure compatibility with third-party libraries that may either construct the TensorFlow graph, or use the model directly.
- **Portability**: Because TensorFlow may be pre-installed on the system, and each individual system may have its own TensorFlow version, we should not assume the programmer is able to make any modifications to TensorFlow.
- **Minimal Interference**: First, the injection process should not interfere with the normal execution of the TensorFlow graph or model when no faults are injected. Further, it should not make the underlying graph or model incapable of being executed on GPUs or parallelized due to the modifications it makes. Finally, the FI process should be reasonably fast.

3.2 TensorFI 1

3.2.1 Design Alternatives

Based on the design constraints in the previous section, we identified three potential ways to inject faults in the TensorFlow 1 graph. The first and perhaps most straightforward method was to modify TensorFlow operators *in place* with FI versions. The FI versions would check for the presence of runtime flags and then either inject the fault or continue with the regular operation of the operator. This is similar to the method used by compiler-based FI tools such as LLFI [32]. Unfortunately, this method does not work with TensorFlow graphs because the underlying operators are implemented and run as C/C++ code, and cannot be modified.

A second design alternative is to directly modify the C++ implementation of the TensorFlow graph to perform FIs. While this would work for injecting faults, it violates the *portability* constraint as it would depend on the specific version of TensorFlow being used and the platform it is being executed on. Further, it would also violate the *minimal inference* constraint as the TensorFlow operators are optimized for specific platforms (e.g., GPUs), and modifying them would potentially break the platform-specific optimizations and may even slow down the process.

The third alternative is to directly inject faults into the higher-level APIs exposed by TensorFlow rather than into the dataflow graph. The advantage of this method would be that one can intercept the API calls and inject different kinds of faults. However, this method would be limited to user code that uses the high-level APIs, and would not be compatible with libraries that manipulate the TensorFlow graph, violating the *ease of use and compatibility* constraint.

3.2.2 Key Idea

TensorFI 1 creates a replica of the original TensorFlow graph but with new operators. The new operators are capable of being injected with faults during the execution of the graph and can be controlled by an external configuration file. Further, when no faults are being injected, the operators emulate the behavior of the original TensorFlow operators they replace.

3.2.3 Implementation

To satisfy the design constraints outlined earlier, TensorFI 1 operates directly on TensorFlow graphs. Because TensorFlow does not allow the dataflow graph to be modified once it is constructed, we need to create a copy of the entire graph, and not just the operators we aim to inject faults into. The new graph mirrors the original one, and takes the same inputs as it. However, it does not directly modify any of the nodes or edges of the original graph and hence does not affect its operator. At runtime, a decision is made as to whether to invoke the original TensorFlow graph or the duplicated one for each invocation of the ML algorithm. Once the graph is chosen, it is executed to completion at runtime.

TensorFI 1 works in two phases. The first phase instruments the graph, and creates a duplicate of each node for FI purposes. The second phase executes the graph to inject faults at runtime, and returns the corresponding output. Note that the first phase is performed only once for the entire graph, while the second phase is performed each time the graph is executed (and faults are injected). Figure 1 shows an example of how TensorFI 1 modifies a TensorFlow graph. Because our goal is to illustrate the workflow of TensorFI 1, we consider a simple computation rather than a real ML algorithm.

In the original TensorFlow graph, there are two operators, an ADD operator which adds two constant nodes “Const_1” and “Const_2”, and a MUL operator, which multiplies the resulting value with that from a placeholder node. A placeholder node is used to feed data from an external source such as a file into a TensorFlow graph, and as such represents an input to the system. A constant node represents a constant value. TensorFI 1 duplicates both the ADD and MUL operators in parallel to the main TensorFlow

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

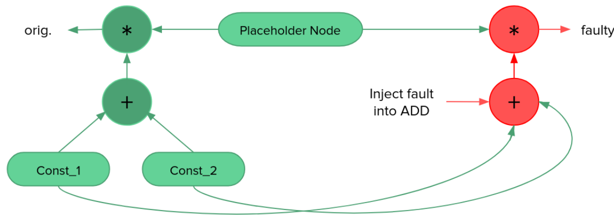

Fig. 1: Working methodology of TensorFI 1: The green nodes are the original nodes constructed by the TensorFlow graph, while the nodes in red are added by TensorFI 1 for FI purposes.

graph, and feeds them with the values of the constant nodes as well as the placeholder node. Note that however there is no flow of values back from the duplicated graph to the original graph, and hence the FI nodes do not interfere with the original computation performed by the graph. The outputs *orig.* and *faulty* represent the original and fault-injected values respectively.

Prior to the FI process, TensorFI 1 instruments the original TensorFlow graph to create a duplicate graph, which will then be invoked during the injection process. At runtime, a dynamic decision is made as to whether we want to compute the *orig.* output or the *faulty* output. If the *orig.* output is demanded, then the graph nodes corresponding to the original TensorFlow graph are executed. Otherwise, the nodes inserted by TensorFI 1 are executed and these emulate the behavior of the original nodes, except that they inject faults. For example, assume that we want to inject a fault into the ADD operator. Every other node inserted by TensorFI 1 would behave exactly like the original nodes in the TensorFlow graph, with the exception of the ADD operator which would inject faults as per the configuration.

3.3 TensorFI 2

3.3.1 Design Challenges

In TensorFlow 1, the session objects contained all the information regarding the graph operations and model parameters. In TensorFlow 2, there is no graph built by default as the eager execution model is adopted. This means that nodes in the graph can no longer be used as the injection target by the fault injector. Instead, the TensorFlow 2 models expose the corresponding layers that store the state and computation of the tensor variables in it. Since these layers are representative of the different operations in TensorFlow, they are chosen as the injection target in TensorFI 2.

In addition, TensorFlow 2 models can be built in three different ways - using the sequential, functional and the subclassing Keras APIs. The design of the FI framework should be such that faults can be injected into the model regardless of the method used to define it.

3.3.2 Design Alternatives

We considered two alternate approaches in the design of TensorFI 2. The first is to create custom FI layers that duplicate the original layers to inject the incoming tensors with the specified faults accordingly and pass it on to the next layer in the model. This mimics the TensorFI 1 approach of creating a copy of the FI operations in the graph. However, this approach incurs high overheads. While this was the only feasible approach for TensorFI 1 because of the static

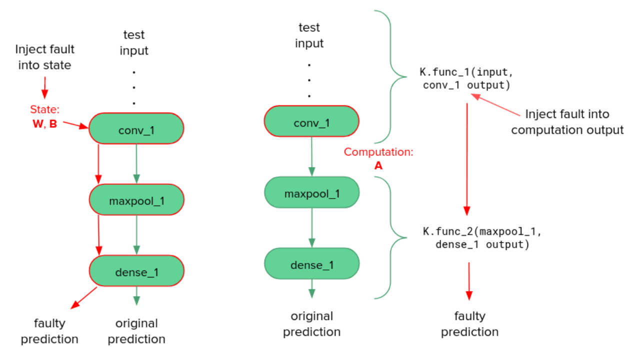

Fig. 2: Working methodology of TensorFI 2: The conv_1 layer is chosen for both weight FI (left) and activation state injection (right. The arrows in red show the propagation of the fault.

computation graph model adopted by TensorFlow 1, it is not so for TensorFlow 2. So we do not adopt this approach.

The second design alternative uses eager execution to inject faults. Once the model starts execution, each layer is checked whether it is chosen for FI. If a particular layer is chosen, the execution passes control to the injection function, which injects the specified faults into the layer outputs. Unfortunately, this approach only works for the sequential models, and not for models using non-linear topologies such as the ResNet model. So we do not adopt this approach.

3.3.3 Key Idea

TensorFI 2 avoids graph duplication and instead makes use of the layer object model to retrieve either the weights or the computations of the layer instance for injection.

3.3.4 Implementation

ML models use input data and weight matrices (that are learned during training) to compute activation matrices (that output a prediction at the last layer). Each layer instance thus has two components, both of which are possible injection targets in TensorFI 2. The first is the *layer state* or *weight matrices* that holds the learned model parameters such as the weights and biases. This is to allow emulation of hardware and software faults in these parameters. In TensorFI 2, we use the Keras Model API [33] to retrieve the trained weights and biases of the specified layer of the model given by the user, and use TensorFlow 2 operators (such as stack, flatten, assign) to retrieve and inject the parameters according to the specified faults and store it back to observe the faulty inference runs. By this method, the implementation is general enough to work with programs that use any of the three methods for building models in TensorFlow 2. The supported mutations include injecting bit-flips in these tensor values, replacing the tensor values with zeros or random values.

The second injection target is the *layer computation or activation matrices*, which hold the output states of the layers. This is to allow emulation of hardware transient faults that can arise in the computation units. In TensorFI 2, we use the Keras backend API to directly intercept the tensor states of the layers chosen for FI. For each layer where faults are

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

TABLE 2: List of fault types supported by TensorFI 1

Type	Description
Zero	Change output of the target operator into zeros
Rand-element	Replace *one* data item in the output of the target operator into a random value
bitFlip-element	Single bit-flip in *one* data item in the output of the target operator

TABLE 3: List of fault types supported by TensorFI 2

Type	Description	Amount
Zeros	Change specified amount of tensor values to zeros	Varies from 0% to 100%
Rand. Replacement	Replace specified amount of tensor values with random values in the range [0, 1)	An integer between 0 and total number of tensor values
Bit-flips	Single or multiple bit-flips in specified amount of tensor values	An integer between 0 and total number of tensor values

to be injected, two Keras functions are modeled before and after the injection call. The first contains the executed model outputs up to that particular layer for the given test input and is taken as the injection target to be operated on. These are the retrieved activation states, and faults are injected into these tensor values and passed into the second function that models the subsequent layers. For bit-flip faults, the bit position to be flipped can either be chosen prior to injection or determined at runtime.

Modifying the layer states is static and is done before the inference runs. This is illustrated in the left of Figure 2. The layers “conv_1”, “maxpool_1” and “dense_1” are part of a larger convolutional network. Let us suppose the first convolution layer “conv_1” states are chosen for injection. TensorFI 2 then injects the weights or biases of this layer and stores back the faulty parameters in the model. During inference, the test input passes through the different layer computations, and the fault gets activated when the execution reaches the “conv_1” layer outputs. The fault can then further propagate into the consecutive layer computations and result in a faulty prediction (i.e., an SDC).

On the other hand, modifying the layer computation is dynamic and is done during the inference runs. This is illustrated in the right of Figure 2. We have the same convolutional model but the “conv_1” activation states are chosen for injection here. The two Keras backend functions “K.func_1” and “K.func_2” work on the original model without duplication but with the inputs and outputs that we specify. During inference, TensorFI 2 passes the inputs to the “K.func_1” which intercepts the computation at the “conv_1” layer, injects faults into the outputs of the layer computation or the activation states and then passes the outputs into the next “K.func_2”, which feeds them to the immediate next layer, and continues the execution on the rest of the original model. Since “K.func_2” works with the faulty computation, faults can propagate to the model’s output, and result in a faulty prediction (i.e., an SDC).

3.4 Satisfying Design Constraints

- **Ease of Use and Compatibility**: To use the TensorFI tool set, the programmer changes a single line in the Python code of the ML model. Everything else is automatic, be it the graph copying and duplication in TensorFI 1 or the injection into the layer state and computation in TensorFI 2. Our method is compatible with external libraries as we do not modify the application’s source code significantly.
- **Portability**: We make use of the TensorFlow and the Keras APIs to implement our framework, and do not change the internal C++ implementation of the TensorFlow operators, which are platform specific. Therefore our implementation is portable across platforms.
- **Minimal Interference**: TensorFI 1 does not interfere with the operation of the main TensorFlow graph. Similarly, TensorFI 2 does not interfere with either the model or layer structure. Further, the original TensorFlow operators are not modified in any way, and hence they can be optimized or parallelized for specific platforms if needed.

3.5 Configuration

The TensorFI tool set allows users to specify the injection configurations such as fault type, error mode and amount of FI through a YAML interface. Once loaded at program initialization, it is fixed for the entire FI campaign. We further elaborate the listed fault types and injection modes for both tools below.

3.5.1 Fault types

We use three fault types namely *zeros, random value replacement* and *bit-flips*. These are used to replace the tensor value(s) in the operator or layer with zero(s), random value(s) in the range or with bit-flips in the value(s). In TensorFI 1, the bit position is chosen randomly each time. In TensorFI 2, we can also specify the position of the bit to be flipped in addition to random selection. The list of fault types and injection modes we have used in our experiments are described in Tables 2 and 3 for TensorFI 1 and 2, respectively. A full list of supported fault configurations can be found in the Wiki and README of the respective tools.

3.5.2 Injection modes

We use three injection modes in TensorFI 1 namely *errorRate, dynamicInstance* and *oneFaultPerRun*. These are used to inject different amount of faults in operators. In a model with multiple operators, there could be multiple instances of each operator. The *errorRate* takes a value between 0 and 1, which specifies the percentage of total operator instances to be injected. This can be specified for multiple operators. The operators are injected with faults based on the specified error rate. With *dynamicInstance*, one instance of each specified operator is randomly chosen and injected with faults. Finally, with *oneFaultPerRun*, one instance among all the operator instances is randomly selected for injection.

We specify injection modes in TensorFI 2 with *Layerwise* and *Amount* for both static and dynamic injections. We directly specify the number of faults we want to inject in *Amount*. This also allows for varying the range amount of values in a tensor in any layer instead of choosing either a single value (Rand-element, bitFlip-element) or all values (Zero, Rand, bitFlip-tensor) in a tensor as in TensorFI 1. We can emulate similar injection modes of TensorFI 1 by configuring these two in different ways. To emulate the

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

TABLE 4: List of analogous injection modes between TensorFI 1 and TensorFI 2

TensorFI 1	TensorFI 2
errorRate: Specify the error rate for different operator instances	*Amount:* Specify the error rate in tensors of different layers
dynamicInstance: Perform random injection on a randomly chosen instance of *each* operation	*Layerwise:* & *Amount:* Perform injection on specified amount of tensor values in *each* layer
oneFaultPerRun: Choose a single instance among all the operators at random so that only one fault is injected in the entire execution	*Amount: 1* Choose a random layer among all the layers and inject one fault into the tensor values of that layer

dynamicInstance mode of TensorFI 1, we specify *Layerwise* in addition to *Amount* so that TensorFI 2 can inject faults on the specified number of tensor values in each layer. Finally, for emulating the *oneFaultPerRun* mode, we specify a value of 1 in the *Amount* so that a random layer is chosen, and a single fault is injected into its tensor values.

Table 4 shows the mapping between the analogous injection modes of the two tools.

4 Evaluation

In this section, we first present the experimental setup for both the tools. We then present the research questions and results for TensorFI 1 followed by those for TensorFI 2. Finally, we discuss the performance overheads of both tools.

4.1 Experimental Setup

4.1.1 ML Applications

TensorFI 1: We use 11 ML applications for evaluation. These are supervised learning models listed in Table 5. Among these, we have an ML application used in the AV domain, i.e., comma.ai driving model.

In addition to the above supervised models, TensorFI 1 can be used to inject faults into unsupervised models. We use one such application, Generative Adversarial Networks (GAN) to show the effects of the injected faults visually. However, because GANs do not have an expected output label, we exclude this experiment from the other experiments and discuss its results separately in Section 4.2.4.

TensorFI 2: We choose a total of 10 ML applications including deep neural networks like ResNet, VGGNet, SqueezeNet that are commonly used in existing studies. We exclude two of the benchmarks we used for TensorFI 1 namely the Highway CNN and comma.ai models because their TensorFlow 2 implementation was not available. We still cover a wide range of applications to enable a comprehensive evaluation. We use VGG16 and ResNet-50 in TensorFI 2 experiments instead of VGG11 and ResNet-18 used in TensorFI 1, as they are more complex and recent architectures. Table 6 lists the applications with the respective datasets used.

4.1.2 ML Datasets

We use 6 public ML datasets in our experiments. *MNIST* dataset [35] is a handwritten digits dataset (with 10 different digits) with 28x28 pixel grayscale images. *Fashion-MNIST* [36] is a dataset of Zalando's article images consisting of a training set of 60000 samples and a test set of 10000 samples.

TABLE 5: ML applications and datasets used for TensorFI 1 evaluation. The baseline model accuracies are also provided.

ML model	Dataset	Accuracy
Neural Net	MNIST	85.42%
Fully Connected Net	MNIST	97.54%
Convolutional NN	MNIST	95.74%
LeNet	MNIST	99%
AlexNet	MNIST	94%
Recurrent NN	MNIST	98.40%
VGG11	GTSRB	99.74%
ResNet-18	ImageNet	62.66% (top-1) 84.61% (top-5)
SqueezeNet	ImageNet	52.936% (top-1) 74.150% (top-5)
Comma.ai model [34]	Driving frame	24.12 (RMSE) 12.64 (Avg. Dev.)
Highway CNN	MNIST	97.92%

TABLE 6: ML applications and datasets used for TensorFI 2.

ML model	Dataset	Accuracy
Neural Net	MNIST	97.34%
Fully Connected Net	FMNIST	88.28%
Convolutional NN	MNIST	98.47%
Convolutional NN	CIFAR10	70.87%
LeNet	MNIST	97.67%
AlexNet	CIFAR10	74.6%
Recurrent NN	MNIST	84.85%
VGG16	ImageNet	70.5% (top-1) 90.0% (top-5)
ResNet-50	ImageNet	77.8% (top-1) 94.2% (top-5)
SqueezeNet	ImageNet	58.5% (top-1) 74.2% (top-5)

Each sample is a 28x28 grayscale image, associated with a clothing or accessory label from 10 classes. The *CIFAR-10* dataset [37] consists of 60000 32x32 color images in 10 classes that include certain animals and vehicles, with 6000 images per class. *GTSRB* dataset [27] is a dataset consisting of 43 different types of traffic signs. In addition, we use a real-world *driving frame* dataset that is labeled with steering angles [38]. *ImageNet* [39] is a large image dataset with more than 14 million images distributed in 1000 classes. The image dimensions vary across the images, but are resized to 224x224 by the VGG, ResNet and SqueezeNet applications.

For models that use the ImageNet dataset (ResNet, SqueezeNet and VGG), we use the pre-trained models since it is time-consuming to train the model from scratch. For the other models, we train them using the following datasets.

TensorFI 1: MNIST, GTSRB, driving frame and ImageNet.

TensorFI 2: MNIST, Fashion-MNIST, CIFAR10, ImageNet.

In TensorFI 1, we used MNIST for the AlexNet model but have used CIFAR-10 in TensorFI 2 as it is more appropriate for a more complex architecture such as AlexNet. Similarly, we use ImageNet for VGG16 in TensorFI 2 instead of GTSRB. Finally, we evaluate the CNN model with the MNIST and CIFAR-10 datasets, and the NN and FCN models with the MNIST and Fashion-MNIST datasets in TensorFI 2.

4.1.3 Metrics

We use SDC rate as the metric for evaluating the resilience of ML applications. An SDC is a wrong output that deviates from the expected output of the program. SDC rate is the fraction of the injected faults that result in SDCs. For the

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

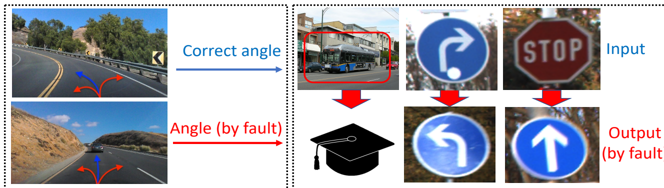

Fig. 3: Example of SDCs observed in different ML applications. Left box - steering model. Right box - image misclassifications.

classifier applications, an SDC is any misclassification. For the steering model *comma.ai* that produces a continuous value as output, we use different threshold values for the deviations of steering angles to identify SDCs: 15, 30, 60 and 120 degrees [16]. We use the RMSE (root mean square error) and average deviation per frame to evaluate the model's accuracy - these are commonly used in this domain [40].

4.1.4 Experiments

TensorFI 1: For each benchmark, we perform 1000 random injections per fault configuration and input. We choose 10 inputs for each injection, and because these injections are dynamic and occur in the duplicated graph nodes for each input, we perform a total of $10,000$ FIs per application and configuration. With 14 configurations (Section 4.2) and 11 applications (Table 5), this comes to around *1.55 million injections* in total with TensorFI 1.

TensorFI 2: We perform two kinds of injections.

1) *Injections in layer states:* The weight FIs (Fig 2) are *static*, and affect the stored states (weights or biases) of the layer. After each injection, we run the injected model with 10 inputs. Hence, we perform $10,000$ random injections per application and per fault configuration. With 11 configurations (Section 4.3) and 10 applications (Table 6), this comes to 1.1 *million injections*.
2) *Injections in layer computations:* The injections in the layer outputs are *dynamic*, and are analogous to operator output injections in TensorFI 1. For each benchmark, we perform 1000 injections per fault configuration and input. We choose 10 test inputs for each injection, and since these injections are dynamic, each test input has different faults occurring in its corresponding layer outputs. Thus we perform a total of $10,000$ FIs per application and configuration. With 2 configurations (Section 4.3) and 10 applications (Table 6), this comes to a total of *200,000 injections*.

Thus, we perform *1.3 million injections* with TensorFI 2.

Error bars: We calculate the error bars at the 95% confidence interval for each experiment in both tools.

Effects of SDCs: Figure 3 shows examples of some of the SDCs observed in our experiments for both the steering model and classification applications. These may result in safety violations in AVs if they are not mitigated. However, we do not distinguish hazardous outcomes in the SDCs.

4.2 TensorFI 1 Results

We use two of the fault types (bitFlip-element and Rand-element) and different configurations of TensorFI 1 for answering the following Research Questions (RQs):

- **RQ1:** What are the SDC rates of different applications under the *oneFaultPerRun* and *dynamicInstance* injection modes?
- **RQ2:** For the *errorRate* mode, how do the SDC rates vary for different error rates?
- **RQ3:** How do the SDC rates vary for faults in different TensorFlow operators in the same ML application?

We organize the results for the 11 ML models listed in Table 5 by each RQ, and then show the results of the FI experiments for GANs separately. For RQ1 and RQ2, we choose *all* the operators in the data-flow graph during the inference phase, which is a subset of operators in the TensorFlow 1 graph. This is because many of the operators in the TensorFlow 1 graph are used for training, and are not executed during the inference phase (we do not inject faults into these operators). We also do not inject faults into those operators that are related to the input (e.g., reading the input, data preprocessing), as we assume that the inputs are correct as per our fault model.

4.2.1 RQ1: Error resilience for different injection modes

In this RQ, we study the effects of two different injection modes, namely *oneFaultPerRun* and *dynamicInstance* (Table 4). We choose single bit flip faults as the fault type for this experiment. Figure 4 show the SDC rates obtained across applications. We can see that different ML applications exhibit different SDC rates, and there is considerable variation across the applications.

We can also observe that there are differences between the two fault modes. For the dynamicInstance injection mode, the SDC rates for all the applications are higher than those in the oneFaultPerRun mode. This is because in the dynamicInstance mode, each type of operator will be injected at least once, while in the oneFaultPerRun mode, only one operator is injected in the entire execution. Thus, the applications present higher SDC rates for the former fault mode than the latter.

We also observe significant differences between applications within the oneFaultPerRun mode. For example, the comma.ai driving model has a higher SDC rate than the classifier applications. This is because the output of the classifier applications are not dependent on the absolute values (instead classification probability is used). Thus, the applications are still able to generate correct output despite the fault occurrence, and hence have higher resilience. However, the comma.ai model predicts the steering angle, which is more sensitive to value deviations. For example, a deviation of 30 due to fault in the classification model will not cause an SDC as long as the predicted label is correct; whereas the deviation would constitute an SDC in the comma.ai model (when we use a threshold of 15 or 30).

In the oneFaultPerRun mode, we find that *RNN* exhibits the highest resilience (less than 1% SDC rate). This is because unlike feed-forward neural networks, RNN calculates the output not only using the input from the previous layer, but also the internal states from other cells. Under the single fault mode, the other internal states remain intact when the fault occurs at the output of the previous layer. Therefore, faults that occur in the feed-forward NNs are more likely to cause SDCs in this mode. However, under the dynamicInstance injection mode, more than one fault

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

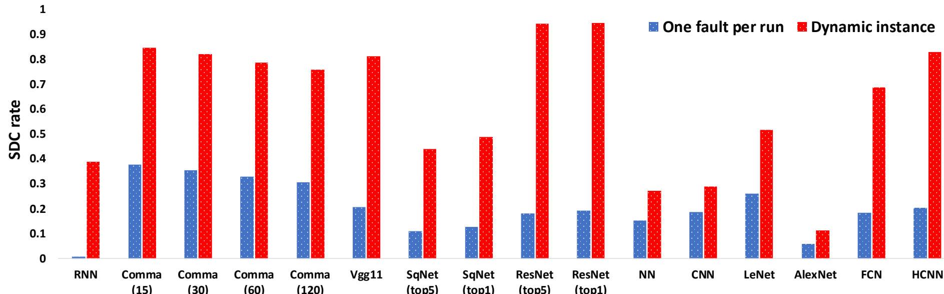

Fig. 4: SDC rates under single bit-flip faults (*from oneFaultPerRun and dynamicInstance injection modes*). Error bars range from $\pm 0.19\%$ to $\pm 2.45\%$ at the 95% confidence interval.

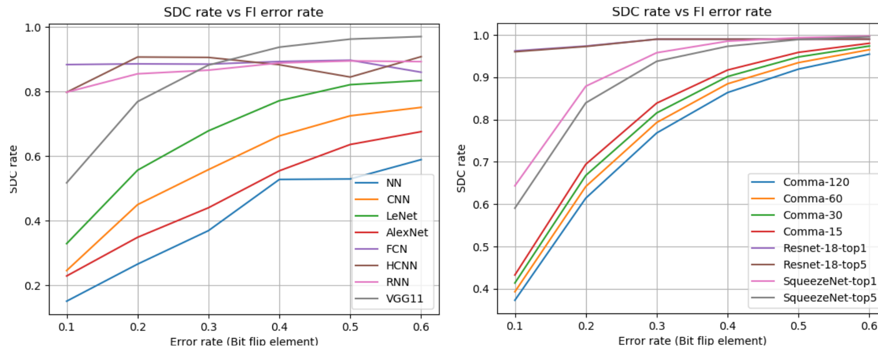

Fig. 5: SDC rates for various error rates *(under bit flip element FI)*. Error bars range from $\pm 0.33\%$ to $\pm 1.68\%$ at the 95% confidence interval.

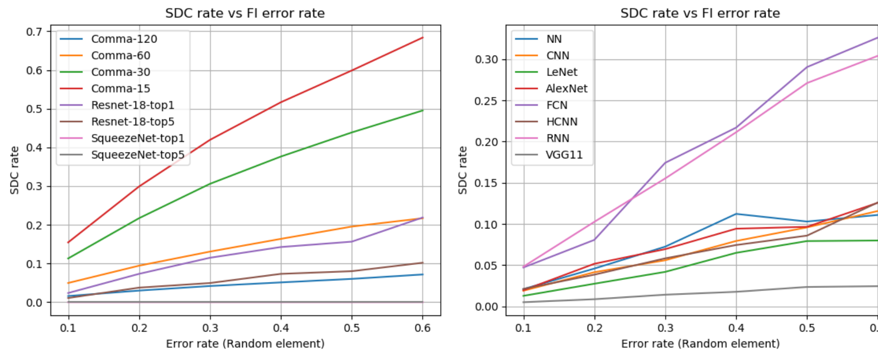

Fig. 6: SDC rates for various error rates *(under random value replacement FI)*. Error bars range from $\pm 0.13\%$ to $\pm 1.59\%$ at the 95% confidence interval.

will be injected. As a result, some of the internal states are also corrupted, thus making the results prone to SDCs (e.g., RNN has around 38% SDC rate).

We also find that *AlexNet* has the highest resilience among all the models in both the oneFaultPerRun and dynamicInstance injection modes. This is because AlexNet has a higher proportion of more resilient operators (i.e., the *add* and *multiply* operators) compared to the other models. The results from Fig. 7 show that the *add* and *multiply* operators are more resilient compared to *convolution* operators.

> ***RQ1.*** *With the bit-flip fault type, under oneFaultPerRun injection mode, RNN and comma.ai are the most and least resilient respectively; under dynamicInstance injection mode, AlexNet and ResNet are the most and least resilient respectively.*

4.2.2 RQ2: Error resilience under different error rates

In this RQ, we explore the resilience of different models for the *errorRate* injection mode (Table 4). This mode allows us to vary the probability of error injection on a per-operator basis. We choose 2 fault types for studying the effects of the error rate, namely bitFlip-element and Rand-element.

Figure 5 and Figure 6 show the variation SDC rates with error rates under both fault types. As expected, we can observe that larger error rates result in higher SDC rates in all the applications, as more operators are injected. However, compared with the results from the bit-flip FI, random value replacement results in lower SDC rates. This is likely because the random value causes lesser value deviation than the bit-flip fault type (in our implementation, we use the random number generator function from Numpy library). Thus, a lower value deviation in this mode leads to lower SDC rates [14], [16].

Figure 5 shows the variations of SDC rates of different ML applications with error rate under the bit-flip fault type. While it shows that the SDC rates of all the applications grow along with the increase of error rates†, we observe that different applications have different rates of growth of

†There are a few outliers such as HCNN in Figure 5, which exhibit oscillations, as SDC measurements are subject to statistical variations.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

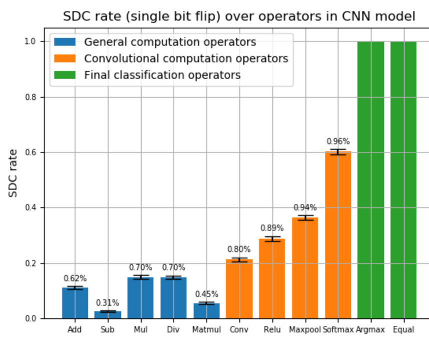

Fig. 7: SDC rates of different operators under bit-flip FI in the CNN model). Error bars range from ±0.3077% to ±0.9592% at 95% confidence interval.

SDCs. In particular, we find that there are four outliers in the results for the bit-flip fault model (Figure 5), RNN, HCNN, ResNet and FCN, which exhibit significantly higher SDC rates than the rest. This is because these models have higher number of operators, and hence higher number of injections.

Likewise, in the case of the random value replacement we find that the *SqueezeNet* application exhibits nearly flat growth in SDC rates with error rates, and that the SDC rates are consistently low. This is because faults need to cause large deviation in order to cause SDCs, which rarely occurs with the random value replacement fault type.

> ***RQ2.*** *Under the errorRate injection mode and with the bit-flip fault type, NN and ResNet are the most and least resilient respectively; with the random value replacement fault type, SqueezeNet and comma.ai are the most and least resilient respectively.*

4.2.3 RQ3: SDC rates across different operators

In this RQ, we study the SDC rates on different operators in the CNN model. The SDC rates are shown in Figure 7. It can be seen that faults in the convolution layer usually have higher SDC rates, compared with other operators (e.g., Sub).

Moreover, we can see that operators such as SoftMax, ArgMax, Equal exhibit the highest SDC rates. In fact, the SDC rates on the ArgMax and Equal operators are nearly 100%. This is because these operators are directly associated with the output, and thus faults in these operators are more likely to cause SDCs. We consider these operators as special cases, and hence exclude them from the other experiments.

On the other hand, operators such as *Sub*, *MatMul* have low SDC rates because faults in these operators are unlikely to propagate much, e.g., faults at the convolution layer are likely to propagate through the complex convolution operators, in which faults can quickly propagate and amplify.

However, faults in operators such as *add* and *multiply* might be masked *before* propagating to the convolution layer; or occur *after* the convolution layer. Therefore, faults in these operators are less likely to cause SDCs, due to limited fault amplification.

> ***RQ3.*** *In a CNN model, arithmetic operators such as Add, Mul, MatMul, Sub, Div are more resilient to bit-flip faults than convolution layer operators such as Conv, ReLU, MaxPool.*

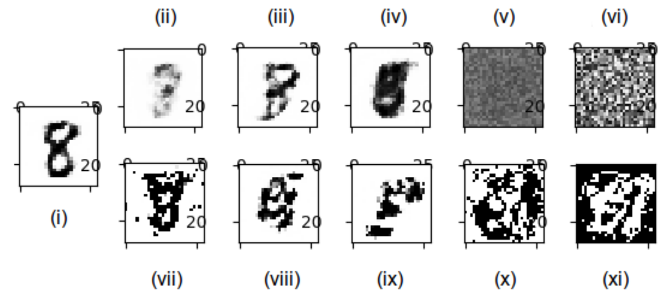

Fig. 8: Generated images of the digit 8 in the MNIST dataset under different configurations for GANs. Top row represents the Rand-element model, while bottom row represents the single bit-flip model. Left center is with no faults. The fault amount generally increases from left to right, with the rightmost images having the highest number of faults.

4.2.4 GAN FI results

The FI results on GAN is presented in Figure 8. The set of images in the top row, from columns (ii) through (vi), are generated from setting the fault type to Rand-element, and the set of corresponding images in the bottom row are generated from setting the fault type to Bitflip-element. Columns (ii) and (iii) are from *oneFaultPerRun*, and *dynamicInstance* error modes respectively. Columns (iv) to (vi) are generated from the *error rate* mode. Column (iv) is generated by setting the error rate to 25%, columns (v) and (vi) are generated from error rates of 50% and 100% respectively. Thus as the number of injected faults increases from left to right, we can see the corresponding fault progression as the images become more and more difficult to decipher.

The second row shows images obtained from similar configurations as the first row, with the only difference being that the fault type chosen is single bit flip. We observe that with bit flip in the operators, the resulting faults in images (vii) to (xi) tend to be more bipolar (i.e., have more black and white pixels than shades of grey). This is likely because with bit flips, the tensor values that store the image data are toggled between being present (1) at a pixel or being absent (0). As this error propagates into more operators, the computations performed amplify this effect, and the resultant end images have strong activated regions of black or white. In the random value replacement mode, the injected operators are replaced with values over the entire range, thus causing the error propagation, and consequently the generated pixels to also exhibit any values within the range.

4.3 TensorFI 2 Results

We use two of the fault types (Bitflips and Zeros) and different configurations of TensorFI 2, for answering the following RQs:

- **RQ1:** What are the SDC rates for faults in the layer states (i.e. weights and biases) for different models?
- **RQ2:** What are the SDC rates for faults in the layer computations (i.e. outputs) for different models?
- **RQ3:** What are the SDC rates for zero faults in the layer states (i.e., effect of weight sparsity)?
- **RQ4:** What are the SDC rates for zero faults in the convolutional layer states of CNN models?

Recall that TensorFlow 2 allows models to be defined in three ways (Section 3). In addition to the defined RQs, we

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

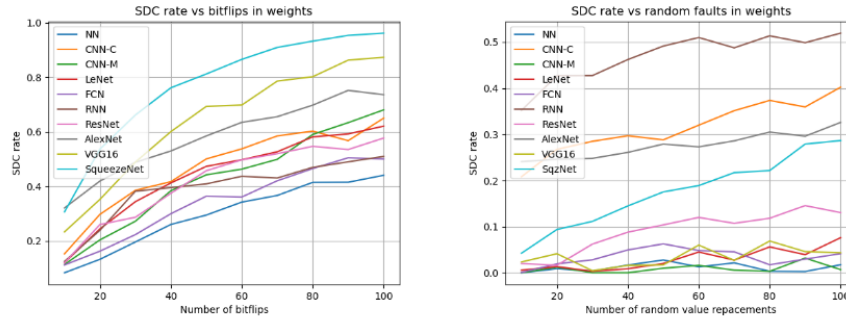

Fig. 9: SDC rates for bit-flips and random value replacement faults in the layer states. Error bars range from ±0.97% to ±3.09% for bit-flips and ±0.01% to ±0.98% for random value replacement faults at the 95% confidence interval.

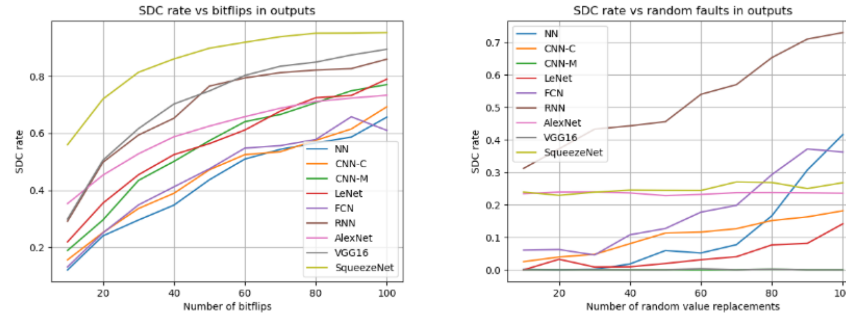

Fig. 10: SDC rates for bit-flips and random value replacement faults in the layer outputs. Error bars range from ±0.06% to ±3.08% for bit-flips and ±0.01% to ±2.31% for random value replacement faults at the 95% confidence interval.

also perform experiments to confirm that FI results agree for the same application irrespective of the way the model is defined in the application (Section 4.3.5).

4.3.1 RQ1: Error resilience for different injection modes and fault types in the layer states (weights and biases)

In this RQ, we study the effects of three different injection modes (Table 4) for the bit-flip and random fault types in the layer states (i.e. weights and biases) of different layers in the model. Recall that this is static injection in the stored states (Figure 2). This experiment gives us insights into how different models perform with faults in the memory and this type of injection into stored weights is different from the experiments we did in TensorFI 1.

Figure 11 shows the SDC rates across different models when a single bit-flip is injected over the entire model (analogous to *oneFaultPerRun* injection mode in TensorFI 1) and in each layer of the model (analogous to *dynamicInstance* in TensorFI 1). We observe that SDC rates vary considerably across the different applications - this is consistent with the FI results in TensorFI 1. As expected, injecting a bit-flip in each layer produces a higher SDC than when injecting a single bit-flip over the entire model because with more corrupted values, there are higher chances of error propagation.

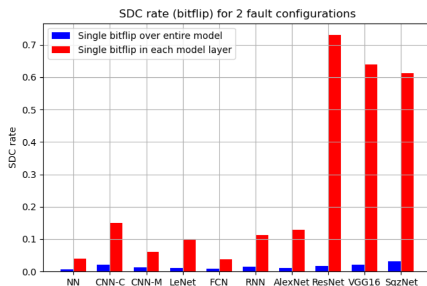

Fig. 11: SDC rates under bit-flip faults in weights and biases (*from single injection modes*). Error bars range from ±0.53% to ±3.02% at the 95% confidence interval.

The differences in the SDC rates of different applications is due to the number of layers in each application. For example, ResNet, which has the highest number of layers (50), gets injected with 50 bit-flips, and so has the highest SDC rate. This is followed by VGG16 and SqueezeNet with 16 and 18 layers respectively. The other networks have only 3 to 8 layers and so have relatively low SDC rates. However, the number of layers is not the only parameter that affects the SDC rate, as we see that the same CNN model with two different datasets, MNIST and CIFAR10, shows a significant difference in the SDC rate.

Figure 9 shows the effect of multiple bit-flips and random value replacement faults in randomly chosen layers of the network. We vary the number of faults injected from 10 to 100 for this experiment. For the bit-flip fault type, we also choose the bit position to be flipped randomly.

As expected, we find that SDC rates are higher with more number of bitflips across all the applications (Figure 9, left). We observe that NN and FCN, which are the simplest models, have the lowest SDCs. We find that SqueezeNet and VGG16 are the models with the highest SDC rates. ResNet, in contrast, has lower SDCs rates than either of them even though it is more complex. This is because ResNet has 50 layers compared to VGG16 and SqueezeNet, which have only 16 and 18 layers respectively. Because we injected bit-flips into a *single* randomly chosen layer's weights, and the influence of model's layer weights are inverse proportional

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

to the number of layers, there is a higher chance for bit-flips to be masked by the downstream layers in ResNet than in VGG16 or SqueezeNet.

We observe that there is considerable difference in the SDC rates for each model between the random value replacement fault type (Figure 9, right) and the bit-flips fault we considered above. This is consistent with our findings in the experimental results of TensorFI 1 (Section 4.2.2). In general, we find that lower SDC rates occur for random value replacement faults compared to bit-flips. This is because the value deviation that needs to occur to cause SDCs is not as drastic for random value replacements. We further find that RNN has the highest SDC rates for random value replacement faults, similar to our previous findings. The exact SDC values and the shape of the graph differ because in our TensorFI 1 experiments, we considered the percentage of values in the operator for FI, while in this experiment we consider the number of values in the layer (*errorRate* injection mode versus *Amount* in Table 4). Finally, we find that VGG16 and SqueezeNet have low SDC rates under random value replacement faults, with VGG16 having rates less than 10% even with 100 random value replacements. This is also consistent with our previous findings where SqueezeNet and VGG11 have the lowest SDC rates for random value replacement faults.

> ***RQ1.*** *Under weight injection, with a single bit-flip over the entire model, NN and SqueezeNet are respectively the most and least resilient; and with a single bit-flip in each model layer, FCN and ResNet are respectively the most and least resilient. When varying the amount of faults, with the bit-flip fault type, NN and SqueezeNet are respectively the most and least resilient; and with the random value replacement fault type, NN and RNN are respectively the most and least resilient.*

4.3.2 RQ2: Error resilience for different injection modes and fault types in the layer computations (outputs)

In this RQ, we study the SDC rates for single and multiple injection modes as before, but use a different injection target. We inject faults into the output tensors of each layer i.e. the activations or the layer computations instead of the layer states. Recall that this injection is dynamic (Figure 2) in the layer outputs, and is hence the closest in comparison to the injection in the operator outputs of TensorFI 1. This experiment gives us insights into the model resilience for faults arising during computations.

Figure 12 shows the SDC rates for different applications for a single bit-flip. We find that AlexNet and SqueezeNet exhibit the highest SDC rates in this experiment. With TensorFI 1, while SqueezeNet exhibited similar SDC rates, AlexNet exhibited the lowest SDC rate. This is because we used the CIFAR-10 dataset in this experiment, whereas earlier we used the model with the less complex MNIST dataset. Recall that we observed earlier in RQ1 of TensorFI 2, that more complex datasets often result in higher SDCs.

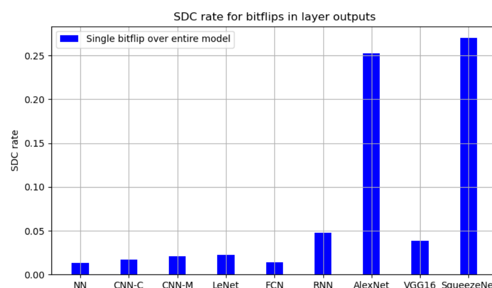

Fig. 12: SDC rates under single bit-flip faults in activations. Error bars range from ±0.22% to ±0.85% at the 95% confidence interval.

Further, we find that the SDC rates for different applications with bit-flips in the layer outputs are greater than those with bit-flips in the layer states. This is because the layer states use different dynamic test inputs (during inference) to produce the layer outputs, and so when faults are injected into the states, they are more likely to be masked compared to when faults are injected directly into the outputs.

Figure 10 shows the SDC rates for bit-flips and random value replacement faults in the layer outputs[‡]. We find that (1) SDC rates are higher for both fault types compared to injections in the layer states, as seen previously for the single FI mode, (2) SqueezeNet has the highest SDC rates for bit-flips in the outputs and is comparable to the SDC rates obtained in the TensorFI 1 experiments, (3) RNN has the highest SDC rates once again for random value replacement faults, (4) the CNN model with the MNIST dataset and VGG16 have the lowest SDC rates with almost no SDCs even when 100 random value replacements are injected and (5) both AlexNet and SqueezeNet have almost no variance between the number of random value replacements considered, suggesting that the interval might need to be larger than 10 faults for observing a difference.

> ***RQ2.*** *Under layer computation injection, with a single bit-flip over the entire model, NN and SqueezeNet are the most and least resilient respectively. When varying the amount of faults, with the bit-flip fault type, NN and SqueezeNet are the most and least resilient respectively; and with the random value replacement fault type, CNN-M and RNN are the most and least resilient respectively.*

4.3.3 RQ3: Error resilience under zero faults in the layer states (weight sparsity)

The goal of this RQ is to study the effect of zero faults in the layer states (i.e., weight sparsity) on the resilience of ML models. ML models can have sparsity i.e. having most of the values in the matrices as zeros. Exploiting sparsity allows the model to have reduced storage and computation needs. The zero fault type allows custom modeling of sparsity through different injection modes and in different targets. We use the zero fault type, and vary the amount of zeros in the layer states from 10% to 100% in steps of 10%. The layer is chosen randomly before injection in each FI trial.

Figure 13 shows the SDC rates for the different applications. We find that the maximum SDC rates even with replacing all of a chosen layers' weights or biases with zeros are less than 55% across all the models except RNN which exceeds 60% SDC rates at 80% sparsity. We find that AlexNet has high SDC rates even at 10% sparsity and continues to exhibit high SDC rates alongside RNN. On the other hand, the smaller models have SDC rates less than 20% even at 80% sparsity. Further, the SDC rates plateau for some of

‡We exclude the results for ResNet because we were unable to inject certain layers due to bugs in our implementation.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

the models in the mid-range of sparsity, before increasing again. For example, for the CNN model with the CIFAR-10 dataset, the SDC rates plateau at around 20% between 40% and 70% sparsity; while for the ResNet model, they plateau between 70% and 90% sparsity. For the RNN and SqueezeNet models, there are two and three plateau regions. This is likely because the stored layer weights are usually redundant. This result can be used to choose the optimal sparsity for the SDC rate. For example, if the user can have 30% SDC rates in the SqueezeNet model, they can choose 60% sparsity rather than 50% as both have similar resilience, and more sparsity typically means lower resource usage.

RQ3. *Under the zero fault type and with varying the amount of faults, CNN-M and RNN are the most and least resilient respectively.*

4.3.4 RQ4: Error resilience under zero faults in the convolutional layer states

The goal of this RQ is to understand how sparsity in the convolutional layer states affects model resilience. We consider only Convolutional Neural network (CNN) models for this experiment as only they have convolutional layers. We use the zero fault type as before, but vary the amount of zeros in the states of a particular layer. We choose the first convolutional layer to inject faults into. We vary the sparsity from 10% to 100% in steps of 10%.

Figure 14 shows the SDC rates for the 7 eligible models. We find that LeNet is the most resilient model, having low SDC rates even with 90% of the first convolutional layer states having zero faults. The CNN model with the MNIST dataset exhibits similar SDC rates, and is the only other CNN architecture used with MNIST. The high resilience is because the number of parameters required to learn the MNIST dataset is very small, and so even with as few as 10% or 20% of the convolutional layer states, the models are able to predict the test images correctly.

After LeNet and CNN-MNIST models, the AlexNet and VGG16 models exhibit the next higher SDC rates. These two models have an almost linear increase in SDC with increase in convolutional layer sparsity, e.g., they have 30% SDC rates even at 10% sparsity. This shows that the weight matrices learned by the models for the first convolutional layer are more dense than the LeNet and CNN-MNIST models, and so it is more important for correct classification. The models with the highest SDC rates are the CNN-CIFAR-10, ResNet and SqueezeNet. Even though the ResNet and SqueezeNet models have 30 and 18 convolutional layers respectively, this result shows that the first convolutional layer is more important for prediction than for all the other models. The CNN models have 3 convolutional layers. However, when the CNN model is used with the CIFAR-10 dataset, it exhibits high SDC rates. This is because the neural network encodes information differently for different datasets and for the CIFAR-10 dataset, the first convolutional layer is more important for correct classification than for the MNIST dataset.

RQ4. *In CNN models, under zero faults in the first convolutional layer states, LeNet and SqueezeNet are the most and least resilient respectively.*

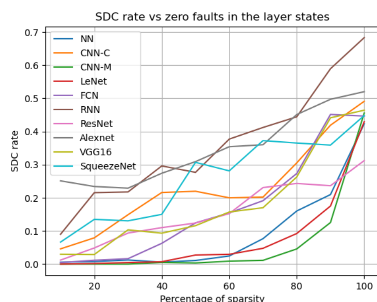

Fig. 13: SDC rates under zero faults in weights and biases. Error bars range from ±0.05% to ±3.06% at the 95% confidence interval.

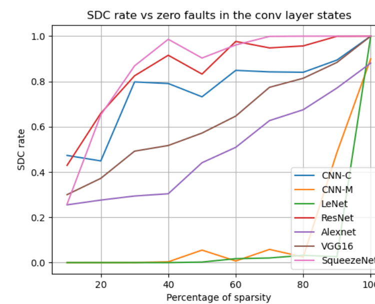

Fig. 14: SDC rates under zero faults in the convolutional layer states. Error bars range from ±0.01% to ±2.75% at the 95% confidence interval.

TABLE 7: SDC rates for bitflips in the NN-MNIST model

Bit-flips	**Sequential**	**Functional**	**Subclassing**
10	0.0882	0.0819	0.0906
20	0.1395	0.1402	0.1376
30	0.1981	0.2212	0.193
40	0.2504	0.2576	0.2459
50	0.2852	0.3077	0.3102
60	0.3090	0.3287	0.3284
70	0.3551	0.3709	0.368
80	0.3986	0.3987	0.393
90	0.4363	0.4283	0.429
100	0.4480	0.4594	0.4421

4.3.5 Error resilience for the same application defined using three different methods

For this experiment, we perform injections on one application using one fault configuration and define the model using the three different methods - using the Keras sequential, functional and subclassing APIs. We choose the neural network model with the MNIST dataset as the application, and inject multiple bit-flips in the layer states.

Table 7 shows the SDC rates when varying the number of bitflips from 10 to 100 in steps of 10 for the same model defined three ways. We perform 10000 injections for each model and configuration. We observe that the SDC rates are similar between the three model definitions and the variations are within the error bars. This shows that the model's resilience does not depend on which of the three high level APIs are used to define it in TensorFlow 2 (Section 3).

4.4 Overheads: TensorFI 1 vs 2

In this section, we present the injection overheads between the two frameworks on a per application basis.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

TABLE 8: Overheads for the program *(baseline)*; with instrumentation, without FI *(disable FI)*; with FI *(enable FI)*

ML model	Baseline (in s)	TFI 1 (in s)	TFI 2 (in s)	Overheads TFI 1	Overheads TFI 2
NN	0.16, 0.15	13.50	6.09	83.3x	39.60x
FCN	1.03, 0.37	86.53	14.13	83.0x	37.19x
CNN	0.23, 0.21	25.94	7.48	107.x	34.62x
LeNet	0.22, 0.21	17.44	8.13	78.3x	37.71x
AlexNet	0.58, 1.78	45.24	52.14	77.0x	28.29x
RNN	2.39, 0.59	145	18.48	59.6x	30.32x
VGG	0.82, 0.66	29.1	25.94	34.5x	38.30x
ResNet	3.76, 1.24	300	46.04	78.8x	36.13x
SqzNet	1.01, 0.45	22	16.13	21.0x	34.85x
Average	1.13, 0.63	76.08	21.62	77.8x	35.22x

We measure the average execution time for performing FI. We consider the execution time for the TensorFlow 1 and 2 programs as a baseline for 50 predictions. Then we measure the time taken for 50 predictions after the FI trials (single bit-flips) from TensorFI 1 and 2. Since there are some differences between the models used in TensorFI 1 and 2, we report the different baselines for each, and subsequently calculate the respective overheads. For TensorFI 2, we consider the dynamic injections (bit-flips in layer activations/outputs) as this incurs much more overhead than static injection, and is the type of injection comparable to TensorFI 1. These measurements are shown in Table 8.

As can be observed, the FI overheads are higher for TensorFI 1, ranging from 21x to 131x. This is because we are emulating the TensorFlow operators during FIs in Python, and cannot benefit from the optimizations and low-level implementation of TensorFlow. The TensorFI 2 overheads in contrast are on average, around 35x. This is because we have no duplication in TensorFI 2 and so the overheads are lower. Further, they do not vary based on how many operators or layers the model may have, and hence TensorFI 2 has lower variation in overheads than TensorFI 1 across different models. *Therefore, TensorFI 2 is more than twice as fast as TensorFI 1 for injecting similar faults.*

While the overheads may seem high, we report the actual time taken by the FI experiments to put these numbers in perspective. In our experiments, the most time-consuming experiment is on the ResNet and Highway CNN models, which took less than 16 hours to complete. However, on average, most of our experiments took 3-4 hours for injecting $10,000$ faults, which is quite reasonable.

4.5 Threats to Validity

Internal validity. We have chosen image classification models as these were one of the most popularly used ML applications. To mitigate the bias from application choosing, we have included experiments with diverse models such as GAN, object detection and comma.ai steering models.

External validity. While we have considered the common ML benchmarks used in reliability studies, real-world safety-critical ML systems typically have other components. For example, the software powering AVs employ a combination of perception, planning, control, localization and prediction modules. We have not considered these modules as our toolset is specific to ML applications.

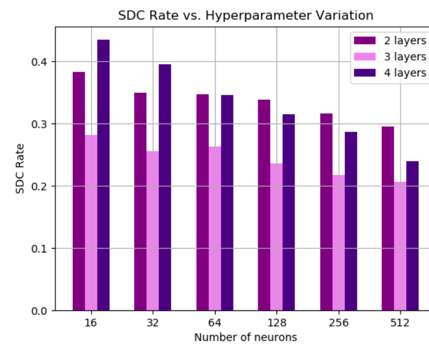

Fig. 15: SDC rates in different variations of the NN model. Error bars range from $\pm 0.7928\%$ to $\pm 0.9716\%$ at the 95% confidence interval.

Construct validity. We choose SDC as the standard metric for evaluating the resilience. This helps us avoid the cases where incorrect classification occurs due to reasons other than the particular fault under consideration.

Conclusion validity. To mitigate the effect of statistical anomalies on our conclusions, for each data point on the graph, we have performed 10,000 FI runs and report the values with error bars within a 95% confidence interval.

5 Case Studies

We present four case studies in this section, two for each tool to demonstrate their utility.

5.1 TensorFI 1

In this section, we perform two case studies to demonstrate the utility of TensorFI 1 in enhancing the error resilience of ML models. The first case study considers the effect of hyperparameter variations to tune for resilience, while the second considers the effect of per-layer protection in a DNN. We use the NN and CNN models from our TensorFI 1 experiments for these.

5.1.1 Effect of Hyperparameter Variations

In this first case study, we empirically analyze the effects of hyperparameter variation on the error resilience of a simple neural network model [41]. We consider three hyperparameters, namely: (i) number of layers - 2, 3, 4; (ii) number of neurons in each layer - 16, 32, 64, 128, 256 and 512; and (iii) optimizers for model training - Adam [42] and RMSProp [43]. This constitutes a total of 36 different models (3*6*2 = 36), on which we use TensorFI 1 to evaluate their error resilience. In this study, we consider the single bit-flip fault model, and *oneFaultPerRun* injection mode. We perform 10000 injections for each model configuration.

Fig. 15 shows the SDC rates for the NN model under different number of layers and neurons. The networks in Fig. 15 are all trained with the Adam optimizer. We observe a similar trend in the networks trained with the RMSProp optimizer, and hence do not report them. In other words, the choice of the optimizer does not affect the SDC percentages.

We first examine the results of increasing the number of neurons. As can be seen in Fig. 15, the SDC percentages decrease with the increase in the number of neurons. To understand the reason behind this, we studied the accuracy of the models trained with different number of neurons

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

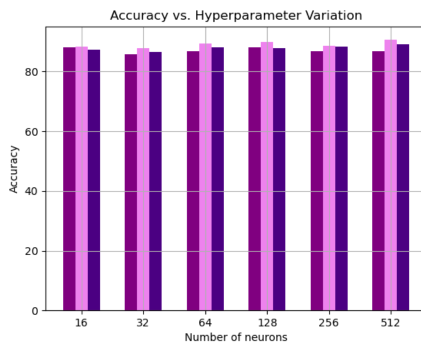

Fig. 16: Accuracy in different variations of the NN model.

TABLE 9: Layerwise resilience in a CNN model

Layer	Operators within the layer	Avg SDC rate
1st Conv	Add, Relu, Conv, Maxpool	0.3102
2nd Conv	Add, Relu, Conv, Maxpool	0.2275
Fully Connected	Add, MatMul, Relu	0.1429
Output	Add, MatMul	0.0499

ranging from 16 to 512. The results are shown in Figure 16. As can be seen, the accuracy remains more or less constant with the increase in the number of neurons. In other words, adding more neurons does not increase the accuracy, hence these additional neurons are redundant, which increases the resilience of the model. Therefore, adding more neurons is beneficial to resilience (in this case).

Second, we examine the results for varying the number of layers from 2 to 4 in Fig. 15. We find that the SDC rate initially decreases from 2 to 3 layers, but increases from 3 to 4 layers. This shows that unlike the number of neurons, there may not be a direct correlation between the SDC rate and the number of layers (however, we cannot determine whether the increase and decrease of the SDC rate is a statistical anomaly). Our tool set helps us to experimentally vary the hyperparameters and find the optimal values for different fault types and amounts in any ML model.

5.1.2 Layer-Wise Resilience

In the second case study, we study the layer-wise resilience in a CNN. The goal is to evaluate the resilience of the different layers in the network, and identify those that are most susceptible to transient faults. This could guide cost-effective resilience techniques to selectively protect them.

We inject faults into the instances of different operators, and coalesce the result based on the layers. We used the same single bit-flip model and the *oneFaultPerRun* fault mode as in the previous case study. For instance, the first layer consists of 4 operators: Add, Relu, Conv and MaxPool. We inject faults into these operators, and measure the SDCs.

We summarize these results in Table 9. As can be seen, the SDC rate decreases as the layer numbers increase. (i.e., the first layer has the highest SDC rate). This is because these layers consist of the operators (Relu, Conv and Maxpool) that are vulnerable to transient faults (Fig.7). Though the first two layers include the same type of operators, the former has higher SDC rate. This is because faults occurring in the earlier layers have a longer fault propagation path, and thus *more* values are likely to be corrupted during fault

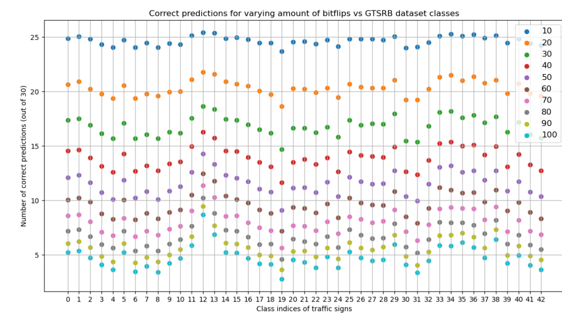

Fig. 17: The number of correct predictions for each class in GTSRB for different numbers of bit-flips (legend) in the first convolutional layer.

propagation. Therefore, based on the results, one should protect the earlier layers of the network first, if the protection overhead is limited.

5.2 TensorFI 2

In this section, we perform two case studies to show how TensorFI 2 can be applied for understanding and improving the resilience of applications. The first case study aims to understand how single bit-flips affect different classes of traffic signs. The second case study explores how bit-flips in different layers affects image segmentation and consequently object detection. We use two new models apart from the 10 ML applications we used in our TensorFI 2 experiments for the following case studies. We use bit-flip fault type for both studies. We use the injection in layer outputs for the first study, and injection in layer states for the second study.

5.2.1 Understanding resilience to bit-flips in a traffic sign recognition model: Are certain classes more vulnerable?

The goal of this experiment is to understand if certain image classes are more vulnerable than others to faults. For this experiment, we choose the GTSRB dataset, which has 43 distinct classes of traffic signs, and a CNN model with two convolutional layers. We choose dynamic injection of bit-flips in the first convolutional layer output. We vary the amount of bit-flips from 10 to 100 in steps of 10. After we train the model, we choose 30 test inputs from each class of 43 traffic signs in the dataset that are predicted correctly (in the absence of faults). We perform 1000 FI trials for each test input in each class, and plot the mean number of images that are predicted correctly in Figure 17.

From Figure 17, we can see that out of 30 test inputs, around 25 get classified correctly for almost all the classes when 10 bit-flips are injected. However, increasing the number of bit-flips decreases the number of images classified correctly (as expected). Further, the difference in the amount of correct predictions grows more pronounced among the different classes as we increase the number of bit-flips.

We find that certain classes of images are more vulnerable than others. We examine the top 5 most and least vulnerable classes of images respectively. Classes 12, 13, 38, 35 and 29 (in that decreasing order) are the most resilient, and have between 9 and 6 images out of 30 classified correctly even in the presence of a 100 bit-flips. Classes 19, 31, 8, 6

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

and 42 (in that increasing order), are the least resilient, and have between 2 and 4 images out of 30 classified correctly in the presence of 100 bit-flips. Also, there is a significant difference between the percentages of correctly predicted images of the most resilient (class 12 with 29.03%) and the least resilient (class 19 with 9.31%) classes.

Figure 18 shows some of the test images belonging to each of these 10 classes, chosen randomly. In general, we find that the more resilient traffic signs are those that are sufficiently unique, and have enough representation in the dataset (refer the relative class frequencies data in the GTSRB dataset [27]). For example, the most resilient Class 12 is the "Priority Road Sign" followed by Class 13, the "Yield Sign". We find that there is no other yellow diamond sign or inverted triangle sign in the dataset, and that both these classes have enough training images in the dataset [27]. While the mandatory blue road signs (Classes 33 to 40) all have above average resilience, we find the most resilient (Classes 38 and 35) have a high representation in the dataset. An exception is Class 29, which has relatively low occurrence of training images. However, it is only the fifth most resilient class, and its value is only slightly higher than the average.

Though some of the least resilient classes have moderately sufficient representation in the training dataset (Classes 31 and 8), we found that almost all the test images of these two classes as well as Class 19 had minimal brightness, low contrast or were blurry. This could have led to their misclassifications under faults. Classes 6, 31, 41 and 42 are all "No More Restrictions Signs" which are the only crossed out grayscale signs in the dataset. We find that these classes have low resilience values indicating that color is also important for the correct prediction of road signs.

5.2.2 Visualizing resilience to bit-flips at lower levels of object detection: Is it possible to identify the layer at which bit-flips occur from analysing the faulty masks predicted?

The goal of this experiment is to visualize how faults in different layers in DNNs affect the outcome of object detection. This is useful for identifying the layer in which the faults occurred. Further, it also provides us with a visual understanding of how the neural network "sees" in the presence of faults.

Image segmentation is an important part of object detection, and has applications in AVs, satellite and medical imaging [44]. We use image segmentation as the target application in our experiment. We use the modified U-Net

Fig. 18: Top 5 most (upper) and least resilient (lower) traffic signs and their GTSRB classes to bit-flips in the first convolutional layer.

from the TensorFlow tutorial [44] with the Oxford-IIIT Pet Dataset [45]. The dataset consists of 37 categories of different dog and cat breeds. To identify where an object is located in the image, the image segmentation component outputs a pixel-wise mask of the image. Each pixel of the image is assigned one of three labels, (i) belonging to the pet, (ii) bordering the pet, or (iii) surrounding pixel.

After we have trained the model, we inject faults into the different layer states of the decoder or upsamplerof the U-Net. The upsampler encodes the states back into the higher dimensional format using the reverse Conv2DTranspose layers. There are 4 blocks in the upsampler, followed by the last Conv2DTranspose layer which effectively reshapes the image into the original pixel dimensions by convolving over the upsampled data. We inject 100 bit-flips into either one of the 4 Conv2DTranspose layer weights in the upsample blocks, or in the final Conv2DTranspose layer. We will refer to the these Conv2DTranspose layers into which we inject faults as convolutional layers henceforth.

The resulting predicted masks for 5 test images in the presence of faults is shown in Figure 19. The first and second columns show the original images, and the correct masks for the images respectively. The third to the seventh columns show the faulty masks predicted when faults were injected into one of the five convolutional layers in order.

We make three main observations. First, we find that faults in the initial layers result in higher disruption (i.e., an unrecognizable mask) compared to faults in the latter layers. This is in line with a previous result from the second case study of TensorFI 2 (Section 5.1.2), where we found that faults in initial layers lead to higher SDC rates because they have a longer fault propagation path. This effect is especially pronounced when convolutional operations are involved, where faults in one value propagate to two or more values.

The second observation is that there are repeating units of faulty areas in the predicted masks. These are larger for faults in the initial layers, and smaller for faults in the latter layers. To understand this, consider the predicted masks from faults in the first layer. 4x4 dimensional tensors are upsampled to 8x8, 16x16, 32x32, 64x64 and finally 128x128 after each layer. This means that when 100 bit-flips injected into the first layer, they have a higher likelihood of spreading out more by the time they reach the final layer compared to them being injected in the final layer. Since all the five layers have the same filter sizes and strides in our model, we can observe that the dimension of the faulty units gets halved as we move each step to the right to the final layers.

The third observation is that we can identify the layer at which faults occur based on the final output. Although the faulty masks shown are the outcomes of a single injection, we show different predicted masks for the same fault configuration (of 100 bit-flips) in the first convolutional layer in Figure 20. Each instance produces a different pattern because these are random FIs, and the faults are propagated differently based on their values. However, across all these different images, the size of the faulty unit is the largest in the first layer. This size depends on the dimension and number of the upsample blocks used in the decoder. We have four blocks and the first upsample block converts a 4x4 to 8x8 image using the Conv2DTranspose operation. In the final faulty outputs in Figure 20, we can see the faulty unit

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

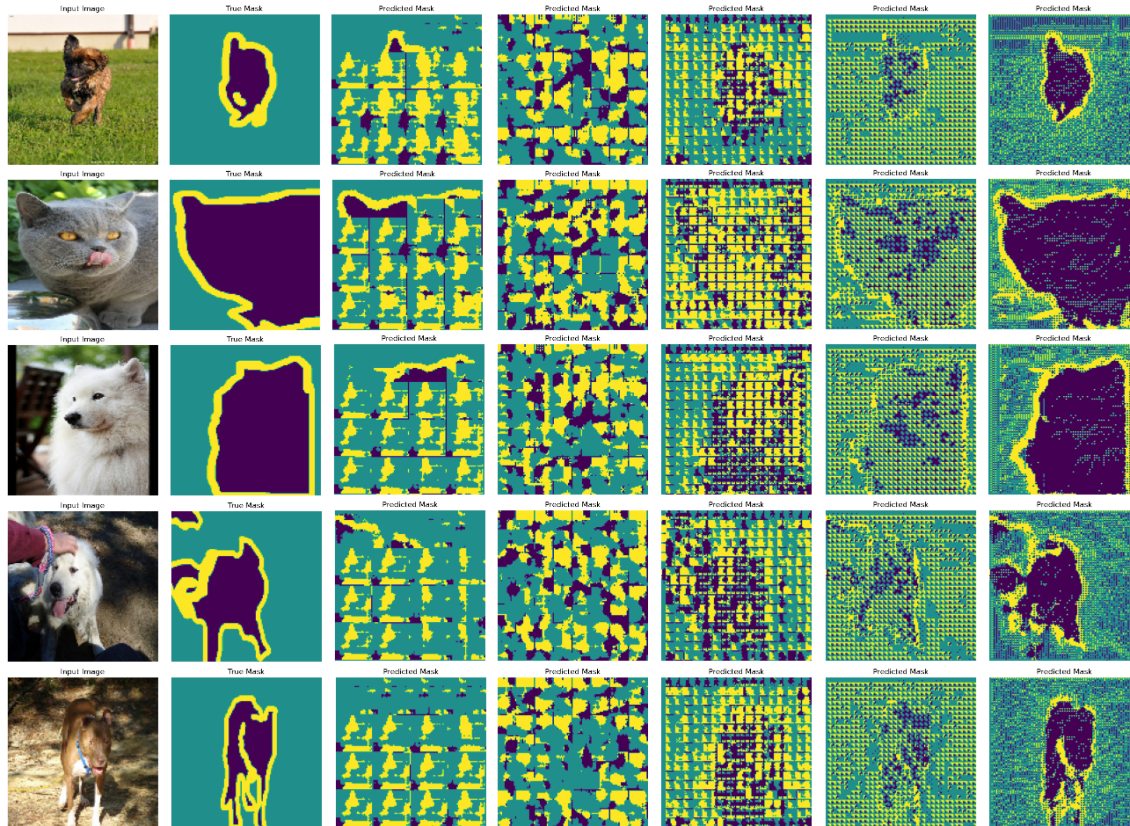

Fig. 19: Predicted faulty masks for bit-flips in different layers of the image segmentation model. The first column is the original image, the second column is the predicted mask in the absence of faults. The remaining columns show the predicted mask after a fault in the ith convolutional layer, where i ranges from 1 to 5.

repeated 4x4 times. Similarly, for the second upsample block that converts 8x8 images to 16x16, we see the faulty unit repeated 8x8 times in Figure 19 and so on. This observation helps us identify the layer in which the faults originated.

6 Conclusion

We present the TensorFI tool set, which consists of generic fault injection frameworks TensorFI 1 and TensorFI 2, for ML applications written using TensorFlow 1 and 2 respectively. They are configurable and can be easily integrated into existing ML applications. They are portable, and are also compatible with third party libraries that use TensorFlow. We use TensorFI 1 to study the resilience of 11 TensorFlow ML applications under different fault configurations, including one used in AVs, and also to improve the resilience of selected applications via hyperparameter optimization and selective layer protection. We use TensorFI 2 to study the resilience of 10 TensorFlow applications and illustrate its utility in understanding the resilience of a traffic sign recognition model and an image segmentation model.

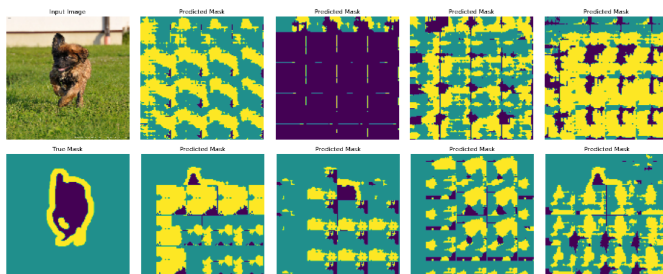

Fig. 20: 8 instances of faulty masks predicted for the same fault configuration in the first layer for the same test image (far left).

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and Huawei Corporation. This manuscript has been approved for unlimited release and assigned LA-UR-21-22618. This work has been co-authored by an employee of Triad National Security, LLC which operates Los Alamos National Laboratory under Contract No. 89233218CNA000001 with the U.S. Department of Energy/National Nuclear Security Administration. The publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Govt. purposes.

References

[1] S. S. Banerjee *et al.*, "Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018.

[2] K. D. Julian *et al.*, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016.

[3] "Functional safety methodologies for automotive applications." [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/solutions/automotive-functional-safety-wp.pdf

[4] M.-C. Hsueh *et al.*, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3175930

[5] D. T. Stott *et al.*, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*. IEEE, 2000, pp. 91–100.

[6] J. Carreira *et al.*, "Xception: Software fault injection and monitoring in processor functional units," *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 245–266, 1998.

[7] J. Aidemark *et al.*, "Goofi: Generic object-oriented fault injection tool," in *2001 International Conference on Dependable Systems and Networks*. IEEE, 2001, pp. 83–88.

[8] P. D. Marinescu *et al.*, "Lfi: A practical and general library-level fault injector," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 379–388.

[9] A. Thomas *et al.*, "Llfi: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[10] J. Wei *et al.*, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 375–382.

[11] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[12] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[13] https://docs.microsoft.com/en-us/cognitive-toolkit/.

[14] G. Li *et al.*, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[15] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[16] Z. Chen *et al.*, "Binfi: An efficient fault injector for safety-critical machine learning systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019.

[17] M. Sabbagh *et al.*, "Evaluating fault resiliency of compressed deep neural networks," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, 2019, pp. 1–7.

[18] L. Ma *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 100–111.

[19] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1158–1161.

[20] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 788–799. [Online]. Available: https://doi.org/10.1145/3368089.3409761

[21] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 486–498. [Online]. Available: https://doi.org/10.1145/3324884.3416571

[22] A. Mahmoud *et al.*, "Pytorchfi: A runtime perturbation tool for dnns," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2020, pp. 25–31.

[23] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "Tensorfi: A flexible fault injection framework for tensorflow applications," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2020, pp. 426–435. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISSRE5003.2020.00047

[24] N. P. Kropp *et al.*, "Automated robustness testing of off-the-shelf software components," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, 1998, pp. 230–239.

[25] A. Lanzaro *et al.*, "An empirical study of injected versus actual interface errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 397–408.

[26] https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a.

[27] S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing, and C. Igel, "Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark," in *International Joint Conference on Neural Networks*, no. 1288, 2013.

[28] [Online]. Available: https://www.tensorflow.org/guide/effective_tf2

[29] S. Hong *et al.*, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," *arXiv preprint arXiv:1906.01017*, 2019.

[30] K. Pei *et al.*, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[31] N. Akhtar *et al.*, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, 2018.

[32] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: an intermediate code-level fault injection tool for hardware faults," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, 2015.

[33] https://www.tensorflow.org/api_docs/python/tf/keras/Model.

[34] "comma.ai's steering model." [Online]. Available: https://github.com/commaai/research

[35] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[36] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017. [Online]. Available: http://arxiv.org/abs/1708.07747

[37] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[38] "Driving dataset." [Online]. Available: https://github.com/SullyChen/driving-datasets

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[40] S. Du, *et al.*, "Self-driving car steering angle prediction based on image recognition," *Department of Computer Science, Stanford University, Tech. Rep. CS231-626*, 2017.

[41] https://github.com/aymericdamien/TensorFlow-Examples.

[42] D. P. Kingma *et al.*, "Adam: A method for stochastic optimization," 2014.

[43] T. Tieleman *et al.*, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[44] [Online]. Available: https://www.tensorflow.org/tutorials/images/segmentation

[45] O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. V. Jawahar, "Cats and dogs," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

Author Biographies

- **Niranjhana Narayanan** received her MASc from the University of British Columbia in 2021 and her B. Tech from IIT Madras, India in 2017. She is currently a software engineer at Neat.
- **Zitao Chen** is a PhD student at the University of British Columbia (UBC). He received his BEng in Computer Sciences at the China University of Geosciences (Wuhan) in 2018, and MASc from UBC.
- **Bo Fang** received his MASc and Ph.D. degrees from the University of British Columbia. He is a computer scientist at the Pacific Northwest National Laboratory.
- **Guanpeng Li** received his BASc (2014) and PhD (2019) degrees from the University of British Columbia, and joined the Department of Computer Science at the University of Iowa as an assistant professor.
- **Karthik Pattabiraman** is a Professor of Electrical and Computer Engineering at the University of British Columbia (UBC).
- **Nathan DeBardeleben** is a senior research scientist at Los Alamos National Laboratory and co-executive director of the Ultrascale Systems Research Center.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Pacific Northwest National Laboratory. Downloaded on September 14,2022 at 18:01:12 UTC from IEEE Xplore. Restrictions apply.