TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications

Zitao Chen*†, Niranjhana Narayanan*†, Bo Fang†, Guanpeng Li‡ Karthik Pattabiraman†, Nathan DeBardeleben§
†University of British Columbia, ‡University of Iowa, §Los Alamos National Laboratory
{zitaoc, nniranjhana, bof, karthikp}@ece.ubc.ca, 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, a high-level fault injection (FI) framework for TensorFlow-based applications. TensorFI is able to inject both hardware and software faults in general TensorFlow programs. TensorFI is a configurable FI tool that is flexible, easy to use, and portable. It can be integrated into existing TensorFlow programs to assess their resilience for different fault types (e.g., faults in particular operators). We use TensorFI to evaluate the resilience of 12 ML programs, including DNNs used in the autonomous vehicle domain. The results give us insights into why some of the models are more resilient. We also present two case studies to demonstrate the usefulness of the tool. TensorFI is publicly available at https://github.com/DependableSystemsLab/TensorFI.**

***Index Terms*—Fault Injection, Machine Learning, Resilience**

I. Introduction

In the past decade, Machine Learning (ML) has become ubiquitous in many applications. ML is also 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 allowable 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. 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 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.

There has been a plethora of SWiFI tools such as NFTape [5], Xception [6], GOOFI [7], LFI [8], LLFI [9], PINFI [10]. These tools operate at different levels of the system stack, from the assembly code level to the application's source code level. In general, the higher the level of abstraction of the FI tool, the easier it is for developers to work with, and use the results from the FI experiments in practice [4].

Due to the increase in popularity of ML applications, there have been many frameworks developed for writing them. An example is TensorFlow [11], which was released by Google in 2017. Other examples are PyTorch [12] and Keras [13]. These frameworks allow the developer to "compose" their application as a sequence of *operators*, which are connected together in the form of a graph. The connections represent the data-flow and control dependencies among the operators. While the underlying implementation of these frameworks is in C++ or assembly code for performance reasons, the developer writes their code using high-level languages (e.g., Python).

In this paper, we introduce a SWiFI tool called TensorFI*, which injects faults into the data-flow graph used in TensorFlow applications. TensorFI performs *interface-level FI* [15], [16]. We focus on TensorFlow as it is *the most popular framework* used today for ML applications [17], though our technique is not restricted to TensorFlow. TensorFI can be used to inject both hardware and software faults in the outputs of TensorFlow operators, and study the effects of the faults on the ML application. The main advantage of TensorFI over traditional SWiFI frameworks is that it directly operates on the TensorFlow operators and graph, and hence its results are readily accessible to developers.

Building a FI tool for TensorFlow applications is challenging due to three reasons. First, because TensorFlow operators are implemented in C++ or assembly code and optimized for different platforms (i.e., different processors and operating systems), it is not practical to modify the implementation of these operators as doing so will hurt both portability and performance. However, in order to inject faults at the level of TensorFlow operators and the graph, one needs to intercept the operators at runtime to modify their execution results. Unfortunately, TensorFlow does not expose the operators once the graph has been constructed, and most of the execution

*Both authors contributed equally to this work.

*A preliminary version of this work was published in a workshop [14]. This work extends the workshop version to support more complex ML programs, as well as a richer set of fault injection configurations.

occurs "behind the scenes" in the low-level code. Therefore, it is not possible to intercept these operators. Secondly, the speed of execution of the TensorFlow graph should not be adversely affected when no faults are injected, as otherwise developers will avoid using the framework. Finally, there are many external libraries that are used by TensorFlow developers. These often rely on the structure and semantics of the TensorFlow graph, and hence these should not be modified.

TensorFI addresses the above challenges 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 their 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). Finally, because we do not modify the TensorFlow graph other than to add the FI graph, external libraries that depend on the graph's structure and semantics can continue to work.

While prior studies have studied the error resilience of ML models by building customized fault injection tools [18]–[21], these tools are usually tailored for a specific set of programs and might not be applicable to general ML programs. *In contrast, TensorFI is a generic and configurable fault injection tool that is able to inject faults in a wide range of ML programs written using TensorFlow.*

We make the following contributions in this paper.

- Propose a generic FI technique to inject faults at the level of the TensorFlow graph, without hurting portability,
- Implement the FI technique in TensorFI, a flexible tool, which allows easy configuration of FI parameters.
- Evaluate TensorFI on 12 ML applications in TensorFlow, including deep neural network (DNN) applications used in AVs, across a wide range of FI configurations (e.g., fault types, error rates). We find that there are significant differences due to both individual ML applications, as well as due to different configurations. We also measure the performance and memory overhead of TensorFI.
- Conduct two case studies to: (1) evaluate how different hyper-parameters (e.g., number of neurons, layers) affect the resilience of ML models; (2) identify the subset of computations that are most susceptible to faults. Both of these can be used to improve the models' resilience.

II. Background 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.

A. Machine Learning 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 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 will be evaluated on the test data, which represents the unseen data.

B. Related Work

Several studies have attempted to evaluate the error resilience of ML applications through fault injections [22], [23]. However, such FI techniques are limited to the specific application being studied, unlike TensorFI that is able to perform FI on generic ML applications.

More recent studies investigate the resilience of deep neural networks (DNN) to *hardware faults* by building fault injectors [18]–[21]. Li et al. build a fault injector by using the tiny-CNN framework [18]. Reagen et al. design a generic framework for quantifying the error resilience of ML applications [19]. Sabbagh et. al develop a framework to study the fault resilience of compressed DNNs [21]. Chen et al. introduce a technique to efficiently prune the hardware FI space by analyzing the underlying property of ML models [20].

In the software faults space, researchers have employed mutation testing in ML applications. DeepMutation [24] is one such framework specialized for DL systems which involves source, program or model level mutations. Their goal is to improve the quality of the test data through testing with the mutated models. PyTorchFI [25] is a very recent runtime perturbation tool developed for DNNs, used to mutate the weights or neurons in PyTorch applications. In contrast to the above studies, TensorFI targets a broader range of ML applications, and can be used to inject both software and hardware faults in the ML application.

C. TensorFlow

TensorFlow is an open-source framework for modeling large data-flow graphs and is widely used for building ML programs. TensorFlow allows programmers to represent the program in the form of a TensorFlow graph (see below). TensorFlow is flexible and can be better optimized as it exposes the underlying graph to the developer. Thus, TensorFlow is considered a low level framework. Many high level frameworks like Keras use TensorFlow as their backend for implementation.

To use TensorFlow, 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.

In this phase, the graph is typically executed directly in the optimized form on the target platform using custom libraries.

TensorFlow also provides a convenient Python language interface for programmers to construct and manipulate the dataflow 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.

D. Fault Model

In this work, we consider two types of faults, hardware faults and software faults that occur during the execution of the TensorFlow program. As TensorFI operates at the level of TensorFlow operators, 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 outputs of the operators. 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 (e.g., mutation testing [24]) or as a single bit-flip [18]–[21]. We also assume that the faults do not modify the structure of the TensorFlow graph (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 [26], [27]. Finally, we assume that the faults occur neither in the ML algorithms, nor in the model parameters. This allows us to compare the output of the FI runs with the golden runs, to determine if a Silent Data Corruption (SDC) has occurred.

We only consider faults during the *inference* phase of the ML program. Because training is usually a one-time process and the results of the trained 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 work in this area [18]–[21].

III. Methodology

We start this section by articulating the design constraints of TensorFI. We then present the design of TensorFI, and an example of its operators. Finally, we present its implementation and explain how to configure it.

A. Design Constraints

We follow 3 constraints in the design of TensorFI.

- **Ease of Use and Compatibility**: The injector 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 construct the TensorFlow graph.
- **Portability**: Because TensorFlow may be pre-installed on the system, and each individual system may have its own installation of TensorFlow, 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 when no faults are injected. Further, it should not make the main graph incapable of being executed on GPUs or parallelized due to the modifications it makes. Finally, the fault injection process should be reasonably fast.

We also make two assumptions in TensorFI. First, we assume that faults occur only during the execution of the TensorFlow operators, and that the faults are transient in nature. In other words, if we reexecute the same operator, the fault will not reappear. This is because studies have shown that the kinds of faults that are prevalent in mature software are often transient faults [28]. Second, we assume that the effect of a fault propagates to the outputs of the TensorFlow operators only, and not to any other state. In other words, there is no error propagation to the permanent state (since we assume faults in the inference phase), which is not visible at the TensorFlow graph level. Again, this is due to the structure of TensorFlow graphs, and our fault model (Section II).

B. Design of TensorFI

To satisfy the design constraints outlined earlier, TensorFI operates directly on TensorFlow graphs. The main idea is to create a replica of the original TensorFlow graph but with new operators. The new operators are capable of injecting faults during the execution of the operators 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. 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 works in two phases. The first phase instruments the graph, and creates a duplicate of each node for fault injection 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). We explain below how this satisfies the design constraints.

- **Ease of Use and Compatibility**: To use TensorFI, the programmer changes a single line in the Python code. Everything else is automatic, namely the graph copying and duplication. Because we duplicate the TensorFlow graph, our method is compatible with external libraries.
- **Portability**: We do not make any modifications to the TensorFlow code or the internal C++ implementation of the TensorFlow operators, which are platform specific. Therefore our implementation is portable across platforms.

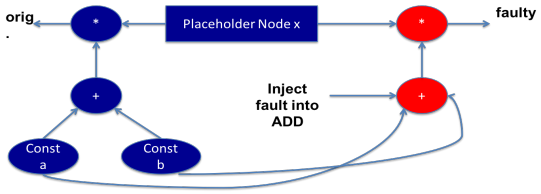

Fig. 1: Example of TensorFlow graph and how TensorFI modifies it. The nodes in blue represent the original nodes in the graph, while the nodes in red are those added by TensorFI for fault injection purposes.

- **Minimal Interference**: TensorFI does not interfere with the operation of the main TensorFlow graph. Further, the original TensorFlow operators are not modified in any way, and hence they can be optimized or parallelized for specific platforms. The only overhead introduced by TensorFI (when no faults are injected) is the check at runtime on whether to call the original graph or the duplicated graph, but this incurs modest performance overhead (Section IV).

C. Example of TensorFI's operation

We consider an example of TensorFI's operation on a small TensorFlow program. Because our goal is to illustrate the workflow of TensorFI, we consider a simple computation rather than a real ML algorithm. The example is shown in Figure 1. The nodes in blue represent the original TensorFlow graph, while those in red represent the duplicated nodes created by TensorFI.

In the original TensorFlow graph, there are two operators, an ADD operator which adds two constant node "a" and "b", and a MUL operator, which multiplies the resulting value with that from a place-holder node. A place-holder 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 duplicates both the ADD and MUL operators in parallel to the main TensorFlow graph, and feeds them with the values of the constant nodes as well as the place-holder node. Note however that there is no flow of values back from the duplicated graph to the original graph, and hence the fault injection 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 fault injection process, TensorFI 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 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 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 (Section III-F).

D. Implementation

We have implemented TensorFI using the Python language as TensorFlow primarily exposes a Python interface. Our implementation consists of about 2500 lines of heavily commented Python code, and is split into 5 modules. We have made TensorFI publicly available under a MIT license on Github (https://github.com/DependableSystemsLab/TensorFI), along with extensive documentation on its use and internals. We have also released all the benchmarks and scripts used in this paper. We currently support TensorFlow version 1.0.

TensorFI supports the following features:

- Comparing each FI result with the golden run
- Launching multiple FI runs in parallel (multi-threading)
- Support for visualizing the modified TensorFlow graphs
- Ability to specify how (i.e., fault type) and where (i.e., operators) to inject faults in a configuration file
- Automated logging of fault injection runs
- Support for statistics collection and analysis

E. Example

Figure 2 shows an example of how TensorFI is used. We consider a simple TensorFlow program that models a Perceptron neural network. For brevity, we omit the construction of the original TensorFlow graph via training as that is not relevant to TensorFI. The first line (line 104) initializes TensorFI on the TensorFlow graph with the current session. It also sets the debugging log level, and initially disables fault injections for obtaining the golden run (i.e., correct output). We then run the original TensorFlow graph with a set of test images and store the result in *correctResult* (line 107).

After performing a number of initializations (lines 111-120), we then launch fault injections using TensorFI in parallel using the *launch* function (lines 123-124). We set the total number of injections to 100, the number of threads to 5 (for parallel injections), and collect statistics for each thread in a list called *myStats*. We also use the *correctResult* to compare with the result of each fault injected run - this is done through the *difffunc* function, which is declared as an anonymous function (i.e., lambda function) in Python, and computes the difference between each fault injected run's result and the *correctResult*. If the difference is greater than 0, the FI run is classified as a Silent Data Corruption (SDC), i.e., incorrect output. Finally, we collate the statistics collected by each thread.

F. Configurations

TensorFI allows the user to configure it through a YAML interface. Figure 3 shows a sample file for configuring TensorFI in YAML format. This is loaded at program initialization, and is fixed for the entire fault injection campaign. The config file consists of the following fields:

- **Seed**: The random seed used in the fault injection experiments, for reproducibility purposes (this is optional).
- **ScalarFaultType**: The fault type to inject for scalar values (full list of types in Table I). We set this to *bitFlip-element*.
- **TensorFaultType**: The fault type to inject for tensor values (full list of types in Table I). We set this to *bitFlip-element*.

```
# Add the fault injection code here to instrument the graph
fi = ti.TensorFlowFI(sess, name="Perceptron",
        logLevel=50, disableInjections="True")

correctResults = sess.run(accuracy,feed_dict={X: minst.test.images, Y:minst.test.labels})

print("Test accuracy:", correctResults)

diffFunc = lambda x: math.fabs(x - correctResults)

# Initialize the number of threads and injections
numThreads = 5
numInjections = 100

# Now start performing fault injections, and collect statistics
myStats = []
for i in range(numThreads):
    myStats.append(ti.FIStat("Perceptron"))

# Launch the fault injections in parallel
fi.Launch(numberOfInjections=numInjections,
    numOfProcesses=numThreads,
    computeDiff=diffFunc, collectStatsList=myStats,
    timeout=100)

print(ti.collateStats(myStats).getStats())
```

Fig. 2: Example of TensorFI usage in a single ML application

```
# Sample YAML for FI configuration
Seed: 1000

ScalarFaultType: bitFlip-element
TensorFaultType: bitFlip-element

Ops:
  - ALL = 1.0

SkipCount: 1

InjectMode: "errorRate"
```

Fig. 3: Example configuration file in YAML format

- **InjectMode**: The mode of injection (list of modes in Table II). We set this to *errorRate*.
- **Ops**: This is a list of the TensorFlow operators that need to be injected, and the probability for injecting a fault into each operator when the mode is *errorRate*. Probability values can range from 0 (never inject) to 1 (always inject). We choose *ALL*, which represents all operators in the graph.
- **SkipCount**: This is an optional parameter for skipping the first 'n' invocations of an operator before injection.

IV. Evaluation

Our goal is to study how resilient are different ML applications (and datasets) to fault configurations of TensorFI, thereby demonstrating its utility. We first describe our experimental setup, followed by the research questions we ask in this study. We then present our experimental results†.

†All our experiments use TensorFI v2.0.0, which is available at https://github.com/DependableSystemsLab/TensorFI/releases/tag/v2.0.0.

TABLE I: List of fault types supported by TensorFI

Type	Explanation
None	Do not inject a fault
Zero	Change output of the target operator into all zeros
Rand	Replace *all* data items in the output of the target operator into random values
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
bitFlip-tensor	Single bit-flip in *all* data items in the output of the target operator

TABLE II: List of injection modes supported by TensorFI

Mode	Meaning
errorRate	Specify the error rate for different operator instances
dynamicInstance	Perform random injection on a randomly chosen instance of *each* operation
oneFaultPerRun	Choose a single instance among all the operators at random so that only one fault is injected in the entire execution

A. Experimental Setup

Hardware Our experiments were conducted on three different systems, namely (i) a Fedora 20 system, 2 GTX TITAN GPUs, 16 CPUs with 256 GB memory; (ii) an Ubuntu 16.04 system with 6 CPUs, 1 GeForce GT610 GPU with 16 GB memory and (iii) Nodes running Red Hat Enterprise Linux Server 6.4, with 12 CPUs cores and 64GB memory.

ML applications: To evaluate TensorFI, we choose 11 supervised learning applications listed in Table III (e.g., deep neural networks like ResNet, VGGNet) that are commonly used in existing studies. We also choose an ML application used in the AV domain, i.e., comma.ai driving model.

In addition to supervised models, TensorFI can be used to inject faults into unsupervised models. We use one such application, Generative Adversarial Networks (GAN) as our 12th ML application to show the effects of the injected faults visually. Because GANs do not have an expected output label, we exclude it from the other experiments.

ML datasets: We use 4 public ML datasets that are commonly used in ML studies. *MNIST* dataset is a hand-written digits (with 10 different digits). *GTSRB* dataset is a dataset consisting of 43 different types of traffic signs. *ImageNet* is a large image dataset with more than 14 million images in 1000 classes. In addition, we use a real-world driving dataset that is labeled with steering angles [29].

For models that use the ImageNet dataset (ResNet-18 and SqueezeNet), 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 corresponding datasets. The datasets are summarized in Table III. The baseline accuracy of each model (without faults) is also provided for comparison.

Metrics: We consider 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 classifier applications, an SDC is any misclassification.

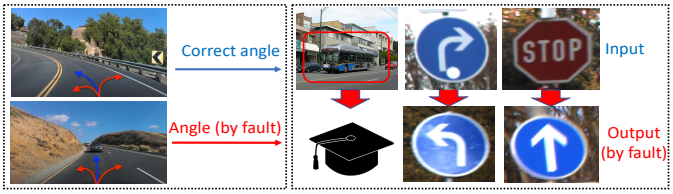

Fig. 4: Example of SDCs observed in different ML applications. Left box - Steering Model. Right box - Image Misclassifications.

TABLE III: ML applications and datasets used for evaluation. The baseline accuracy without faults is also provided.

ML model	Dataset	Accuracy
Neural Net	MNIST	85.42%
Fully Connected Net	MNIST	97.54%
LeNet	MNIST	99%
AlexNet	MNIST	94%
CNN	MNIST	95.74%
Highway CNN	MNIST	97.92%
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 [31]	Driving frame	24.12 (RMSE) 12.64 (Avg. Dev.)

However, the steering model *comma.ai* produces a continuous value as output. For this model, we use different threshold values for the deviations of steering angles to identify SDCs: 15, 30, 60 and 120 degrees [20]. For the steering model, we use RMSE (root mean square error) and average deviation per frame as metrics to evaluate the model's accuracy - these are commonly used in ML studies in the AV domain [30].

Experiments: For each benchmark, we perform 1000 random FI experiments per fault configuration and input. We choose 10 inputs for each application, and hence perform a total of $10,000$ fault injections per configuration and we use 14 different fault configurations. We also calculate the error bars at the 95% confidence interval for each experiment.

Fig. 4 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. With that said, we do not specifically distinguish hazardous outcomes in SDCs.

Research Questions We use different configurations of TensorFI (shown in Table I and Table II) for answering the following Research Questions (RQs):

RQ1: What are the SDC rates of different applications under the *oneFaultPerRun* and *dynamicInstance* error 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?

RQ4: What are the overheads of TensorFI?

B. Results

We organize the results for the 11 ML models listed in Table III 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 graph. This is because many of the operators in the TensorFlow 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.

1) RQ1: Error resilience for different error modes: In this RQ, we study the effects of two different fault modes, namely *oneFaultPerRun* and *dynamicInstance*. We choose single bit flip faults as the fault type for this experiment. Fig. 5 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 dynamic instance injection mode, the SDC rates for all the applications are higher than those in the one fault per run mode. This is because in the dynamic instance mode, each type of operator will be injected at least once, while in the one fault per run 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 one fault per run 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 one fault per run 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 dynamic instance injection mode, more than one fault 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* exhibits the highest resilience among all the models in both the one-fault-per-run and dynamic instance injection modes. This is because AlexNet has many operators such as ADD, MUL, which are more resilient to faults (see Fig. 8). Therefore, the proportion of operators that are more prone to SDCs (e.g., convolution operators, activation function) is not as high as that in other models.

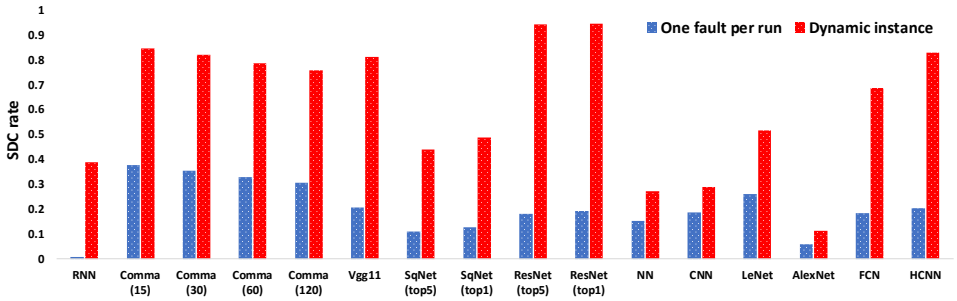

Fig. 5: SDC rates under single bit-flip faults (*from oneFaultPerRun and dynamicInstance injection modes*). Error bars range from $\pm 0.19\%$ to $\pm 2.45\%$ at the 95% confidence interval.

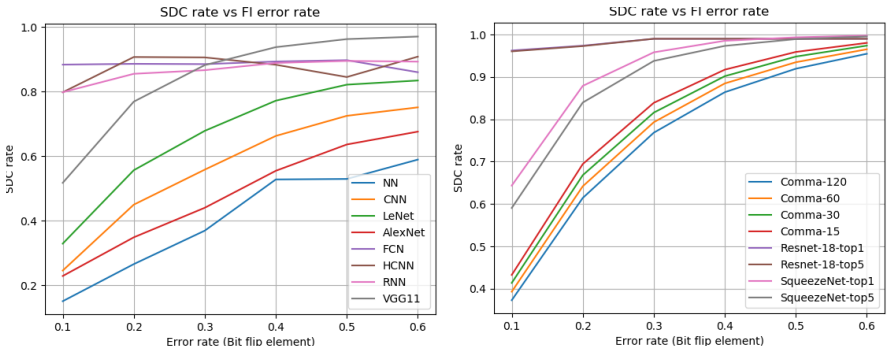

Fig. 6: SDC rates for various error rates *(under bit flip element FI)*. Error bars range from $\pm 0.33\%$ to $\pm 1.68\%$ at the 95% confidence interval.

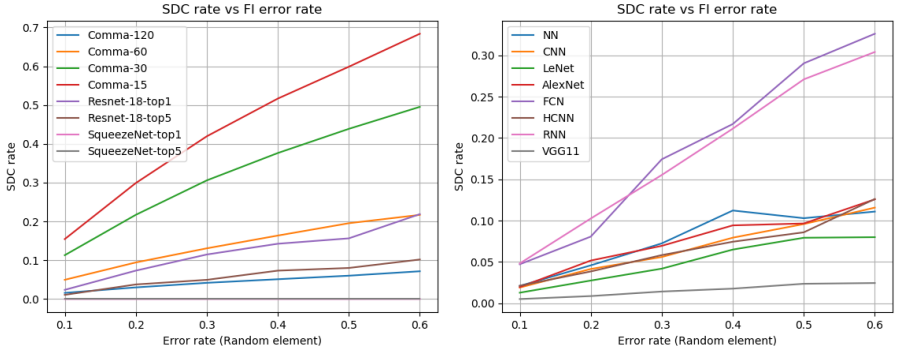

Fig. 7: SDC rates for various error rates *(under random value replacement FI)*. Error bars range from $\pm 0.13\%$ to $\pm 1.59\%$ at the 95% confidence interval.

2) RQ2: Error resilience under different error rates: In this RQ, we explore the resilience of different models for the *errorRate* injection mode. This mode allows us to vary the probability of error injection on a per-operator basis. We choose 2 different fault types for studying the effects of the error rate, namely bitFlip-element and Rand-element.

Fig. 6 and Fig. 7 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 [18], [20].

Fig. 6 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 SDCs. In particular, we find that there are four outliers in the results for the bit-flip fault model (Fig. 6), 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 have higher number of injections.

Likewise, in the case of the random replacement we find that the *SqueezeNet* applications exhibit 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 replacement fault type.

[‡]There are a few outliers such as HCNN in Fig. 6, which exhibit oscillations, as SDC measurements are subject to small statistical variations.

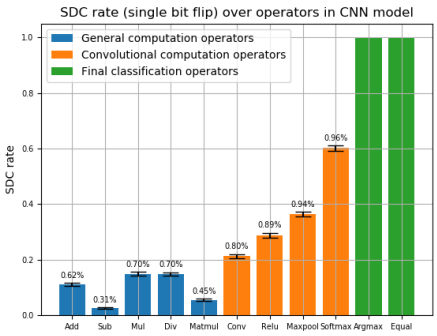

Fig. 8: SDC rates of different operators under bit-flip FI in the CNN model). Error bars range from $\pm 0.3077\%$ to $\pm 0.9592\%$ at 95% confidence interval.

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 Fig. 8. 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 three operators as special cases and we exclude them from all other injection experiments). On the other hand, operators such as *Sub*, *MatMul* have low SDC rates because faults in these operators are unlikely to propagate much. For example, faults at the convolution layer are likely to propagate through the complex convolution operators, in which faults can quickly propagate and amplify. However, faults at 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.

4) RQ4: Overhead.: In this question, we evaluate the memory overhead and the performance overhead. The memory overhead is mainly due to the graph computation, which is not duplicated, and is hence low. For FCN, which has the highest number of operations, the memory overhead is 6.25%.

For the performance overhead, we measure the execution time for the TensorFlow programs as a baseline for 50 predictions. Then we measure the time taken for 50 predictions after the TensorFI instrumentation phase, but with fault injections disabled. These measurements are detailed in the *Disable FI* column of the Table IV. We then measure the time taken for 50 predictions with a single bit flip fault injected per run, and report this time in the *Enable FI* column. The first subcolumn 'Inst.' in *Overheads*, is the instrumentation overhead (difference between Disable FI and Baseline over the Baseline) and the second subcolumn 'FI' gives the overhead incurred by fault injection alone (difference between Enable and Disable FI over Disable FI).

As can be observed, the instrumentation overheads are relatively small, and range from 0.1x to 6.9x across applications. The fault injection overheads are much higher, ranging from 21x to 131x. This is because we are emulating the TensorFlow operators during fault injections in Python, and cannot benefit from the optimizations and low-level implementation of TensorFlow. However, the instrumentation phase itself incurs only modest overheads, with an average of 1.5x, when faults are not injected, in keeping with our minimal interference goal - this overhead is due to the runtime check for choosing which version of the operator to invoke (Section III-D).

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

ML model	**Baseline** (in s)	**Disable FI** (in s)	**Enable FI** (in s)	**Overheads** Inst.	FI
NN	0.06	0.16	13.50	1.6x	83.34x
FCN	0.13	1.03	86.53	6.9x	83x
LeNet	0.105	0.22	17.44	1.1x	78.27x
AlexNet	0.51	0.58	45.24	0.1x	77x
CNN	0.23	0.23	25.94	0.1x	107x
HCNN	0.44	1.02	134.56	1.3x	131x
RNN	0.097	2.39	145	1.5x	59.66x
VGG11	0.19	0.82	29.1	3.3x	34.5x
SqueezeNet	0.85	1	22	0.2x	21x
ResNet	2.72	3.76	300	0.4x	78.78x
Comma.ai	0.3	0.47	46	0.6x	96.87x
Average	0.59	1.06	78.66	1.5x	77.3x

While the overheads may seem high, we report the actual time taken by the FI experiments to put this number 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 to complete for injecting $10,000$ faults, which is quite reasonable.

5) GAN FI results: The FI results on GAN is presented in Fig. 9 The set of images in the top row, (ii) to (vi), are generated from setting the fault type to Rand-element. (ii) and (iii) are for one fault per run, and dynamic instance respectively. (iv) to (vi) are generated from the *errorRate* mode. (iv) is from setting the error rate to 25%. We can see the fault progression clearly as the image becomes more difficult to decipher as the error rate increases. 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 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.

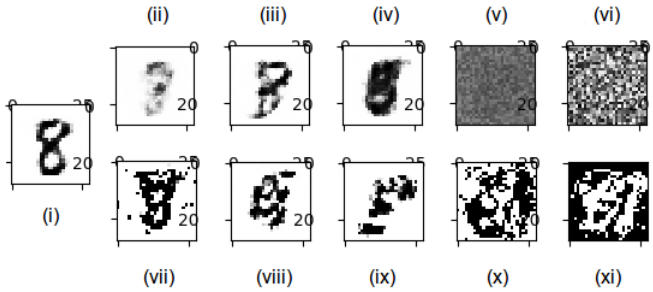

Fig. 9: Generated images of the digit 8 in the MNIST data-set under different configurations for GANs. Top row represents the Rand-element model, while bottom row represents the single bit-flip model. Left center is with no faults.

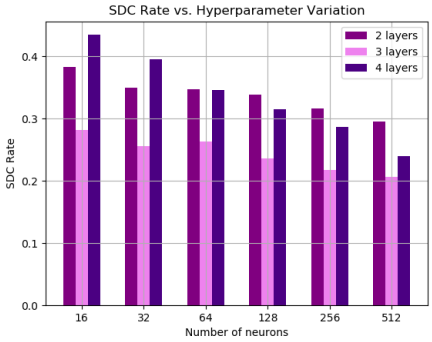

Fig. 10: SDC rates in different variations of the NN model. Error bars range from $\pm 0.7928\%$ to $\pm 0.9716\%$ at the 95% confidence interval.

V. Case Studies

In this section, we perform two case studies to demonstrate the utility of TensorFI in enhancing the error resilience of ML models. The first case study considers the effect of hyper-parameter variations to tune for resilience, while the second considers the effect of per-layer protection in a DNN.

A. Effect of Hyper-parameter Variations

In this first case study, we empirically analyze the effects of hyper-parameter variation on the error resilience of a simple neural network model [32]. We consider three hyper-parameters, 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 [33] and RMSProp [34]. This constitutes a total of 36 different models (3*6*2 = 36), on which we use TensorFI 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. 10 shows the SDC rates for the NN model under different number of layers and neurons. The networks in Fig. 10 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 decreasing the number of neurons. As can be seen in Fig. 10, 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 ranging from 16 to 512. The results are shown in Figure 11. 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).

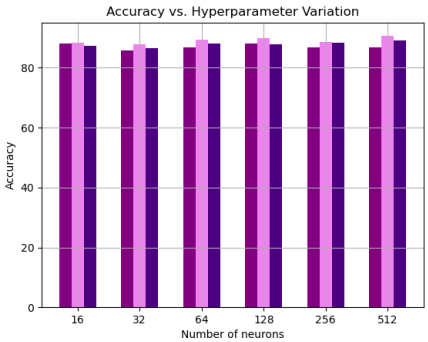

Fig. 11: Accuracy in different variations of the NN model.

TABLE V: Layerwise resilience in a CNN model

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

Second, we examine the results for varying the number of layers from 2 to 4 in Fig. 10. We find that the SDC rate initially decreases from 2 to 3 layers, but increases from 3 to 4 layers. This shows that while adding layers initially increases resilience, layer redundancy has an optimal point (in this case, 3 layers), beyond which the resilience falls again. Further, from the accuracy results in Figure 11, the addition of layers has little effect on the accuracy, and hence increasing the number of layers from 2 to 3 boosts the error resilience *without degrading accuracy.* However, we should not increase the number of layers beyond 3 as it degrades the resilience.

B. 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 the vulnerable layers.

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 SDC percentages.

We summarize these results in Table V. 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.8). 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 propagation. Therefore, based on the results, one should protect the earlier layers of the network first, if the protection overhead is limited.

VI. Conclusion

We present TensorFI, a generic fault injection tool for ML applications written using the TensorFlow framework. TensorFI is a configurable tool that can be easily integrated into existing ML applications. TensorFI is both portable and configurable. Further, it is also compatible with third party libraries that use TensorFlow. We use TensorFI to study the resilience of 12 TensorFlow ML applications under different fault configurations, including one used in AVs, and also to improve the resilience of selected applications via hyper-parameter optimization and selective layer protection. Our evaluation thus demonstrates the utility of TensorFI in measuring and improving the resilience of ML applications.

As future work, we plan to (1) mitigate the performance overhead of TensorFI by providing C/C++ variants of the TensorFlow operators that support FI, (2) extend TensorFI to inject faults during the training phase, and (3) consider ML frameworks other than TensorFlow such as PyTorch and Keras.

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). This manuscript has been approved for unlimited release and has been assigned LA-UR-20-26216. 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.

[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] "Keras." [Online]. Available: https://keras.io/

[14] G. Li *et al.*, "Tensorfi: A configurable fault injector for tensorflow applications," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018.

[15] N. P. Kropp *et al.*, "Automated robustness testing of off-the-shelf software components," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*. IEEE, 1998, pp. 230–239.

[16] 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.

[17] "Tensorflow popularity." [Online]. Available: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

[18] 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.

[19] B. Reagen *et al.*, "Ares: a framework for quantifying the resilience of deep neural networks," in *55th Annual Design Automation Conference*, 2018.

[20] 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.

[21] 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.

[22] C. Alippi *et al.*, "Sensitivity to errors in artificial neural networks: A behavioral approach," *IEEE Transactions on Circuits and Systems*, vol. 42, no. 6, 1995.

[23] S. Bettola *et al.*, "High performance fault-tolerant digital neural networks," *IEEE transactions on computers*, no. 3, 1998.

[24] 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.

[25] 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.

[26] 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.

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

[28] R. Iyer *et al.*, "Experimental analysis of computer system dependability," 1993.

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

[30] 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.

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

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

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

[34] 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.