TensorFI: A Configurable Fault Injector for TensorFlow Applications

Guanpeng Li[+], Karthik Pattabiraman[+], Nathan DeBardeleben[*]
[+]Department of Electrical and Computer Engineering, University of British Columbia
[*]Ultrascale Systems Research Center, Los Alamos National Laboratory[1]
Email: {gpli, karthikp}@ece.ubc.ca, ndebard@lanl.gov

Abstract

Machine Learning (ML) applications have emerged as the killer applications for next generation hardware and software platforms, and there is a lot of interest in software frameworks to build such applications. TensorFlow is a high-level data-flow framework for building ML applications and has become the most popular one in the recent past. ML applications are also being increasingly used in safety-critical systems such as self-driving cars and home robotics. Therefore, there is a compelling need to evaluate the resilience of ML applications built using frameworks such as TensorFlow. In this paper, we build a high-level fault injection framework for TensorFlow called TensorFI for evaluating the resilience of ML applications. TensorFI is flexible, easy to use, and portable. It also allows ML application programmers to explore the effects of different parameters and algorithms on error resilience.

I. Introduction

In the last decade, Machine Learning (ML) has become ubiquitous, with applications ranging from speech recognition to game playing. Much of this revolution has been fueled by the rise of deep neural networks, which automatically learn to classify objects such as images based on a few training samples and data. Simultaneously, there has been an explosion in the number of ML frameworks and software infrastructure such as TensorFlow [1] to facilitate the rapid development of ML algorithms. The main goal of these frameworks is to automate the more mundane aspects of developing ML programs, and allow programmers to focus on the core algorithms.

ML applications have also found their way to safety-critical domains such as self-driving cars (i.e., autonomous vehicles) and home robotics. In these domains, it is vitally important that the ML application provide reliability and safety guarantees on its outputs, even in the presence of faults and failures. While these can include both accidental (i.e., random) and malicious faults, we focus on the former in this paper. Therefore, it is necessary to evaluate the resilience of ML applications deployed in safety-critical contexts, in the presence of faults.

[1]This work was primarily performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract AC52-06NA25396. The publication has been assigned the LANL identifier LA-UR-18-27031. We also acknowledge support from the NSERC, Canada for this research.

Fault Injection (FI) is a widely used technique to evaluate the resilience of software applications to faults. While FI has been extensively used in general purpose applications, its use in ML applications presents three main challenges. First, because ML applications are often written using specialized infrastructures, it is difficult to inject faults at the level of individual program statements or variables as these are hidden inside the framework. Second, it is difficult to interpret the results of the FI experiments as they are dependent on the application and the inputs as well as the framework being deployed. Finally, performing FI in ML applications requires the programmer to understand where faults are likely to occur in the application and map them to its implementation.

In this paper, we build a fault injector for ML applications written using specialized frameworks. Because TensorFlow is the most widely used, publicly available software framework for writing ML applications today, we only support TensorFlow and we call our injector TensorFI. TensorFI has three main features. First, it does not rely on the internal implementation of TensorFlow, aiding its *portability* to different platforms and TensorFlow versions. Second, it requires minimal modifications for programmers to make to their applications and is hence *easy to use*. Third, it allows programmers to configure the injection process through an external interface without modifying the application (*flexible*).

The main abstraction used in TensorFlow is a computational data-flow graph, in which different ML operations are represented as graph nodes and the flow of data between them is represented as edges. Developers will typically construct the ML application as a TensorFlow graph, and allow the system to optimize it at runtime depending on the platform it is deployed on (e.g., CPUs, GPUs). TensorFI interposes on the TensorFlow graph to inject faults at the level of TensorFlow operators or nodes. In other words, it allows programmers to emulate both hardware and software faults in the computations of specific ML operators in TensorFlow, and evaluate the effects of the faults on the end-to-end result of the ML application. Furthermore, the faults can also be configured through an external configuration file to allow rapid exploration of the ML resilience space, and choosing appropriate parameters and algorithms for resilient ML.

Prior work has attempted to evaluate the effects of faults in ML algorithms [2], [4]. However, most prior work is limited to specific ML algorithms and their implementations,

rather than targeting an entire class of such algorithms (that use a framework). Other work has attempted to consider the effects of hardware faults in ML applications implemented on accelerator platforms [8]. However, this work is limited to specific hardware accelerators, and does not incorporate software faults. *To the best of our knowledge, we are the first to build a fault injector for both hardware and software faults in general ML applications built using software frameworks.*

We make the following contributions in this paper:

- We propose a systematic approach for performing fault injection in ML frameworks such as TensorFlow that use dataflow graphs to represent their computations,
- We build a fault injector called TensorFI, to inject faults in applications built using TensorFlow. We demonstrate the use of TensorFI on a simple example program.
- Using TensorFI, we evaluate different ML algorithms on the same data set in terms of their performance and resilience, to understand which algorithm(s) perform best under different fault types and fault probabilities.

II. Background and Fault Model

A. Fault Injection

Fault injection (FI) is the act of systematically injecting faults or perturbations in the system under test (SUT) and studying the effects of the fault. FI can be done in hardware or software, and can be used to model both hardware and software faults. Software Implemented Fault Injection (SWiFI) is the injection of faults through software-based techniques such as debuggers and instrumentation. SWiFI can be used for injecting either hardware or software faults. In this paper, when we say fault injection, we are referring to SWiFI techniques. SWiFI techniques can be further classified into compiler-based or runtime injections. In compiler-based techniques, the code is mutated to inject the fault prior to the program being run. In runtime fault injection techniques, the perturbation is done during the execution of the program with no changes to the code during its compilation. Runtime injection has the advantage that it can access the dynamic state of the program but is limited in that it does not have information about where the fault can occur in the program. Compile-time injection on the other hand does not have access to runtime information, but it can use knowledge of the program's code to seed the fault. It is also typically faster than runtime injections. Hybrid FI attempts to combine the advantages of compile-time and runtime injections by modifying the code at compile-time to insert hooks for runtime injections.

FI techniques can also be classified into interface-level injection and implementation-level techniques. In the former, faults are injected at the level of interfaces (e.g., function calls, Application Programmer Interface (APIs)), while in the latter, faults are injected into the implementation of the APIs or functions. The main advantage of interface-level fault injection is that it is independent of the implementation and is hence much more portable than implementation-level fault injection. On the other hand, interface-level injection is often more limited than implementation level injection as it cannot access the internal states of the SUTs. With that said, it is possible to emulate a wide variety of faults that propagate to the interface of the SUT, and hence interface-level injection is widely used in practice [6], [7]. In this paper, we primarily inject faults at the level of machine-learning operators in TensorFlow, which corresponds to a form of interface-level fault injection.

B. Machine Learning Applications

Typically, a machine learning (ML) application consists of two phases: a *training phase* where a ML model is trained using a certain ML algorithm (e.g., linear regression) and an *inference phase*, where the model performs a task. The parameters of the ML model are learned from *training data*, then in the *inference phase* the model is used for the actual ML task on *test data*. Note that the test data is typically independent from the training data. Both the training data and the test data constitute an ML dataset.

An ML model takes features that are used in the inference phase as input to make a prediction. Two general ML tasks are *regression* and *classification*. The former is used to predict the value of one or more outputs given a set of feature values, while the latter is used to classify the input feature values into one or more categories. ML models can be either *supervised* or *unsupervised*. Supervised algorithms are those in which the training samples have a known label assigned to them, while unsupervised algorithms are those in which there are no known labels for the training data. Examples of common supervised ML algorithms are linear regression and random forests. Examples of unsupervised ML algorithms are k-Nearest neighbors (kNN) and kernel density estimation.

The overall quality of a supervised ML model is determined by a variety of metrics related to accuracy, which is a quantitative meaure of how much the predicted value differs from the ground truth or correct value. Common metrics are precision, recall, F1 score, and Area under the ROC curve. In cases where the ground truth is not available (e.g., for unsupervised ML algorithms), quality is defined in terms of some other external metric such as the tightness of the clusters formed.

C. Tensorflow

TensorFlow is an open-source framework released by Google for modeling large data-flow graphs. It has been prominently used in ML algorithms' implementation and has more users than almost all other open-source frameworks such as Torch or Caffe. Unlike many other ML frameworks, TensorFlow does not make any assumptions about the implementation of the ML algorithms, and allows users to express their algorithm however they see fit as long as they use the dataflow graph abstraction (see below). This distinguishes it from higher-level frameworks such as Keras, which provides a high-level API for users to express the ML algorithms and abstracts away the implementation details completely. Thus, TensorFlow is considered a lower level framework as it allows users more flexibility, but has a steeper learning curve. It is

worth noting that many of the higher level frameworks themselves use TensorFlow as their backend for implementation, and thus TensorFlow can be used indirectly as well.

The central abstraction in TensorFlow is the data-flow graph, which is a high-level representation of the flow of information among the various ML operators [1]. Users of TensorFlow have to construct the ML algorithm using the standard operators provided by TensorFlow or define their own operators. Operators are not allowed to have side effects, i.e., affect the global state of the program (with some exceptions), and hence the only method for communicating data across operators is through the edges of the data flow graph.

To use TensorFlow, programmers first use one of the in-built learning algorithms to construct the dataflow graph of their ML algorithm during the *training phase*. Various optimizations on the graph are also performed based on the input data set. Once the graph is built and optimized, it is not allowed to be modified. The graph is then used for the *inference phase* in which 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

We consider two categories of faults in our work, hardware faults and software faults, that occur during the execution of the TensorFlow operators. Because our framework operates at the level of TensorFlow operators, we abstract the faults to the interfaces of the operators. In other words, we assume that a hardware or software fault has occurred and caused the output(s) of the TensorFlow operator to diverge from its correct output. We do not make any assumptions about the nature of the output's divergence though. We also assume that the faults do not modify the structure of the TensorFlow graph, and that the inputs provided to the graph are not faulty (i.e., errors in the datasets used) - such faults would be extraneous to TensorFlow and hence we do not consider them. Finally, we do not consider faults in the ML algorithm e.g., using the wrong ML algorithm or wrong parameters.

We assume faults occur with a fixed probability or rate that is known *a priori* in each operation of the TensorFlow graph. Since the errors we inject are interface errors, this fault injection rate represents the interface error rate. Projecting the error rates for other levels requires applying appropriate derating factors, which depends on the platform and the application - we do not consider these factors in this paper.

[1]Technically, an operator in a TensorFlow graph can be any computation.

III. Design and Implementation

We start this section by articulating the design constraints and assumptions of TensorFI, followed by the various alternatives in the design. We then present the design of TensorFI, and an example of its operation. Finally, we present some of the implementation details.

A. Design Constraints and Assumptions

We aim to respect three design constraints in the design of TensorFI as follows.

- **Ease of Use and Compatibility**: We aim to make an injector that is as easy to use as possible, with minimal modifications to the application code. We also need to ensure compatibility with third-party libraries and sophisticated training algorithms that may construct the TensorFlow graph using custom API methods.
- **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 or its libraries.
- **Speed of Execution**: The third and final principle is that 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, though it does not have to be as optimized as the main graph.

We also make the following assumptions about the faults injected by TensorFI.

- Faults are only injected during the inference phase, and not during the training phase. This is because training is a limited operation and done once, while inferencing is done thousands of times (typically) on the trained graph, and is hence more likely to experience faults at runtime.
- 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 may not appear. This is because studies have shown that the kinds of faults that are prevalent in mature software are often transient faults [5]. Further, we do not make any assumptions about specific implementation of the TensorFlow operators, and so we cannot model permanent faults easily.
- Finally, 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 implicit error propagation to the permanent state or to other operators which is not visible at TensorFlow graph level. Again, this is in keeping with the structure of TensorFlow graphs, and fault model considered (Section II).

B. Design Alternatives

Based on the design constraints and assumptions in the previous section, we identified three potential ways to inject

faults in TensorFlow graphs. The first and perhaps most straightforward method was to modify TensorFlow operators *in place* with fault injection versions. The fault injection 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 fault injection tools such as LLFI [9]. Unfortunately, this method does not work with TensorFlow graphs because the operators are implemented in C/C++ code, and cannot be modified once they are constructed [2].

A second design alternative is to directly modify the C++ implementation of the TensorFlow graph to perform fault injections. 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 *Speed of Execution* constraint as the TensorFlow operators are optimized for specific platforms (e.g., GPUs), and modifying them would potentially break the platform-specific optimizations.

The third design alternative we considered was 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 third party libraries or frameworks that manipulate the TensorFlow graph. This would violate the *Ease of Use* and *Compatibility* constraints. Further, we would be limited in the kinds of faults we can inject as we would not have visibility into the actual TensorFlow graph.

Due to the above mentioned problems, we did not consider the above design alternatives. We explain our design in the next section.

C. Design of TensorFI

To satisfy the design constraints outlined earlier, we built TensorFI as a two-phase injector that operates directly on TensorFlow graphs. The main idea is to create a replica of the original TensorFlow graph but with a different set of operators than the original ones. 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 operation. 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 (this is done by replacing the *session.run* method of the main session object in TensorFlow). Once the graph is chosen, it is executed to completion at runtime to produce the corresponding outputs.

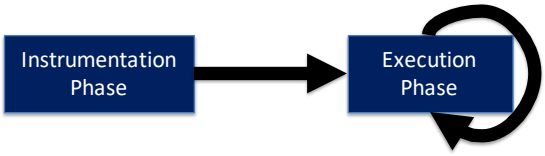

Fig. 1: TensorFI Phases of operation

Figure 1 shows the two phases of TensorFI's operation. 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 injected).

We explain below how this design satisfies the design constraints identified earlier.

- **Ease of Use and Compatibility**: To use TensorFI, the programmer has to change a single line in the Python code. Everything else happens behind the scenes, namely the graph copying and modifications. Because we operate at the level of the TensorFlow graph, our method is fully compatible with third-party APIs.
- **Portability**: We do not make any modifications to the TensorFlow code or the internal C++ implementation of the TensorFlow operators which are platform specific. We do perform *monkey patching*[3] of the TensorFlow session object (Section III-E), but this is done using the publicly documented TensorFlow methods from within the Python code. We do however depend on the interfaces of the TensorFlow operators being consistent across TensorFlow versions, which is reasonable as the TensorFlow interface is fairly stable across its versions and platforms.
- **Speed of Execution**: TensorFI does not interfere with the operation of the main TensorFlow graph, and hence does not slow it down in any way. 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 is the check at runtime on whether to call the original graph or the duplicated graph, but this overhead is minimal. Further, the check is executed at most once per inference procedure, and hence its cost can be amortized over the execution of the entire graph.

D. 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 operation of TensorFI, we consider a simple computation represented as a graph. The example is shown in Figure 2. The

[2]More accurately, the modification would have no effect on the runtime graph as it is compiled to C/C++ code and changes through the Python interface are not carried out in the C++ version of the code.

[3]This refers to the act of replacing the object's methods in-place without modifying the corresponding class declaration.

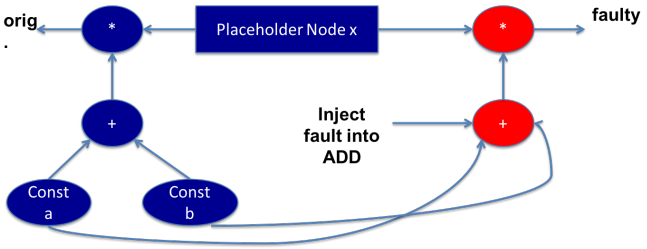

Fig. 2: Example of TensorFlow graph and how TensorFI modifies it. The nodes in blue represent the original nodes in the graph, while the nodes in red are those added by TensorFI for fault injection.

nodes in blue represent the original TensorFlow graph, while those in red represent the duplicated nodes 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. The graph is created before the fault injection process is launched after the original TensorFlow graph is created (i.e., after the training phase).

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 with a certain probability (specified by the user).

E. Implementation

TensorFI supports the following features:

- Launching multiple FI runs with support for 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 fault type etc. in a configuration file
- Automated logging of fault injection runs
- Support for statistics collection and analysis

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 as follows.

- **TensorFI**: The main module which interfaces with TensorFlow and intercepts the *session.run* method to interpose TensorFI's run method.
- **ModifyGraph**: This module is responsible for traversing the TensorFlow graph and making a mirror image of it.
- **InjectFault**: This module is responsible for actually injecting the faults into each operator of TensorFlow, using custom injection functions for each operator.
- **FIConfig**: This is responsible for parsing a fault configuration file, and configuring the initial fault injection state. We support Yaml format for the configuration file.
- **Statistics Collection**: This module collects the global statistics for the fault injection runs, and also logs the detailed fault information for each run.

We have made TensorFI publicly available under a MIT license on Github[4]. We have also added extensive documentation on the installation, use, and programming of TensorFI in the repository. We will also publicly release all the benchmarks and experimental setup used in this paper for generating the results, if the paper is accepted.

IV. Usage Model

In this section, we explain how a developer would use TensorFI in a TensorFlow application. We use an example in Figure 3 for illustration.

In the example, we consider a simple TensorFlow program that models a Perceptron neural network. For brevity, we omit the construction of the original TensorFlow graph. The first line initializes TensorFI on the TensorFlow graph with the current session. It also sets the debugging log level, and initially disables fault injections (during the execution of the graph). In the second line, we run the original TensorFlow graph with a set of test images and store the result in *correctResult* - since injections are disabled, there are no faults injected in this run of the TensorFlow graph, and hence the result corresponds to the golden run. We also make the new graph constructed by TensorFI available for visualization using *Tensorboard*, which is a graph visualization tool that comes with TensorFlow.

After performing a number of initialiations, we then launch fault injections using TensorFI in parallel using the *plaunch* function. 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*. Finally, we collate the statistics collected by each thread using the *collateStats* method, and print them to the console using *getStats*.

[4]https://github.com/DependableSystemsLab/TensorFI/

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

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

print("Testing Accuracy:", correctResult)

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

# Make the log files in TensorBoard
logs_path = "./logs"
logWriter = tf.summary.FileWriter( logs_path, sess.graph )

# Initialize the number of threads
numThreads = 5

# 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.pLaunch( numberOfInjections = 100, numberOfProcesses = numThreads, computeDiff = diffFunc, collectStatsList = myStats)

# Collate the statistics and print them
print( ti.collateStats(myStats).getStats() )
```

Fig. 3: Example of TensorFI usage in a simple ML application

Note that only the first line is mandatory for injecting faults using TensorFlow, namely that of attaching TensorFI to the TensorFlow graph. Everything else is either a convenience function (e.g., launching fault injections in parallel), or is there for ease of visualization and debugging. In fact, we could have used *sess.run* to launch a single fault injection of the TensorFlow graph after attaching TensorFI, exactly like how the original TensorFlow graph is executed.

Figure 4 shows a sample configuration file for configuring TensorFI in YAML format. This is loaded at application initialization, and is fixed for the entire fault injection campaign. The configuration 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 - acceptable types are None (no fault), Zero (zero out the value), and Random (set it to a random value).
- **TensorFaultType**: The fault type to inject for tensor values - - acceptable types are None (no fault), Zero (zero out the tensor), and Random (fill it with random values).
- **Ops**: This is a list of the TensorFlow operators that need to be injected, and the probability for injecting a fault into each operator. Probability values can range from 0 (never inject) to 1 (always inject).
- **SkipCount**: This is an optional parameter for skipping the first 'n' invocations of an operator before injection, where 'n' can be any integer value, greater than 0.

```
1  # This is a sample YAML file for fault injection configuration
2  # The fields here should correspond to the Fields in fiConfig.py
3
4  # Deterministic fault seed for the injections
5  # Seed: 1000
6
7  # Type of fault to be injected for Scalars and Tensors
8  # Allowed values are {None, Rand, Zero}
9
10 ScalarFaultType: Rand
11 TensorFaultType: Rand
12
13 # Add the list of Operations and their probabilities here
14 # Each entry must be in a separate line ad start with a '-'
15 # each line must represent an OP and it's probability value
16 # See fiConfig.py for a full list of allowed OP values
17 # NOTE: These should not be any tabs anywhere below
18
19 Ops:
20 # - ALL = 1.0 # Chooses all operations
21   - ADD = 1.0
22 # - DIV = 0.0 # This does not exist - and should be ignored (Test)
23 # - SUB = -0.5 # This should raise an exception
24
25 # How many times the set of above operations should be skipped before injection
26 # SkipCount: 1
```

Fig. 4: Example configuration file in YAML format

V. Experimental Setup

We study the effects of different experimental parameters on the error resilience of well-known ML algorithms using TensorFI. The goal is to demonstrate that TensorFI is able to perform fault injection studies with different parameters and experimental configurations. We first describe the experimental parameters and setup we consider, and then the metric(s) we measure in the study.

A. Experimental Parameters

We investigate how different ML algorithms using a variety of datasets react to errors under different fault injection rates.

- **ML datasets**: We make use of six publicly available machine learning datasets and tasks from the University of California at Irvine (UCI) Machine Learning repository [3]. They are popular ML datasets across different

TABLE I: Details of the Input datasets

Name	Domain	Description	Number of Output Classes	Number of Instances	Number of Features	Source URL
Adult	Econometrics	Census income prediction	2	48842	14	https://archive.ics.uci.edu/ml/data-sets/adult
Credit	Financial	Credit card application approval	2	690	15	https://archive.ics.uci.edu/ml/data-sets/adult
Marketing	Business	Prediction of direct marketing campaigns	2	45211	17	https://archive.ics.uci.edu/ml/data-sets/bank+marketing
MNIST	Image Analysis	Handwritten digit recognition problem	10	13500	5000	https://archive.ics.uci.edu/ml/databases/mnist/
Survive	Medical	Prediction of patient survival	2	306	3	https://archive.ics.uci.edu/ml/data-sets/Haberman's+Survival
Zoo	General Classification	Animal recognition	7	101	17	http://archive.ics.uci.edu/ml/data-sets/zoo

application domains. All six tasks can be modeled as either regression or classification problems. However, we choose to model them as classification problems in order to compare their resilience. The details of the input datasets are in Table I. As seen, the numbers of output classes (dependent variables) are 7 and 10 in *Zoo* and *MNIST* respectively, and 2 in the rest of input datasets. The number of features (independent variables) varies from 3 to $5,000$ across datasets. The number of instances (data points) vary from 101 to $48,842$ across datasets.

- **ML Algorithm**: We choose three machine learning algorithms in this study, namely (1) k-nearest-neighbour (kNN), (2) logistic regression (LR), and (3) neural networks (NN)[5]. We use the ML algorithms with the six UCI datasets as classification problems.
- **Fault Injection Rate**: To study the effect of different fault rates on the algorithms, we vary fault injection rates as follows: 0.5%, 1%, 5% and 10%. These correspond to the probabilities for injecting faults into TensorFlow graph nodes.

B. Metrics

In each fault injection run, we train one of our models. We first measure its prediction accuracy without any fault injection – we call it the original accuracy (OA). We then measure the prediction accuracy with fault injection using TensorFI (FA). We define the difference between OA and FA as *accuracy drop*. We repeat each fault injection run N times and compute the average accuracy drop for the trained ML model.

VI. Results

We present the results of the fault injection experiments in this section. They are categorized by the parameter considered.

A. Machine Learning Models & Datasets

Figure 5 shows the fault injection results of different ML models using different datasets under different error rates. As we can see, different ML models experience different average accuracy drops depending on which dataset is used. For example, the average accuracy drop varies from 0.74% to 33.67% at the error rate of 10%. We observe that accuracy drops are highly correlated with the number of the output classes in the dataset. Our intuition is that the total amount of output classes in the dataset limits the options of wrong results in an erroneous execution.

As seen, kNN has very low average accuracy drops in the datasets that have larger number of output classes. For example, at 10% error rate, *kNN* experiences 33.67% and 27.73% accuracy drops on average in *MNIST* and *Zoo*, and accuracy drops of 2.59%, 0.74% and 1.98% in *Survive, Marketing* and *Adult*. This is because *MNIST* and *Zoo* have 10 and 7 output classes, the rest of the datasets have only 2 output classes (Table I).

Among the algorithms, *kNN* has lower accuracy drops on average across different datasets under faults. In general, it is higher in *LR* and *NN*. *NN* is the worst algorithm in term of accuracy drop, except in the datasets of *Survive* and *MNIST*. In *MNIST* and *Zoo* which have larger number of output classes, the average accuracy drop of *LR* is as close as that of *kNN*, though both are lower than *NN* in most cases.

B. Error Rate

As expected, the average accuracy drop of the models grows as the error rate increases regardless of which dataset is used. We see the ranking of the average accuracy drop at different error rates is fairly consistent in most of the cases. However, the average accuracy drop of *kNN* grows more slowly than those of the other two models. The investigation on why certain models are more sensitive to the error rates are subjects of our future work. On the other hand, the average accuracy drop is almost flat in *kNN* with *Credit* and *Marketing*. Again, we speculate this is due to the smaller amount of output classes in these datasets, but more work is needed to better understand the reasons for the accuracy drop.

VII. Related Work

There has been a large body of work evaluating the error resilience of ML applications through fault injections [2], [4]. However, their FI procedure is tied to the specific application being studied, unlike TensorFI which aims to provide a generic FI framework for ML applications.

Li et al. [8] use the tiny-CNN framework to construct their fault injector. Reagen et al. [10] propose a generic framework

[5]The neural network consists of 2 hidden layers.

(a) Error Margin: +/-0.67% to +/-6.72%

(b) Error Margin: +/-0.00% to +/-9.17%

(c) Error Margin: +/-0.38% to +/-9.12%

(d) Error Margin: +/-2.76% to +/-9.78%

(e) Error Margin: +/-0.16% to +/-7.08%

(f) Error Margin: +/-2.47% to +/-9.75%

Fig. 5: Accuracy Drops for Different Error Rates, Machine Learning Algorithms and Datasets (Y-axis: Average Accuracy Drop; X-axis: Error Rate; Error margins are measured at 95% confidence interval)

for quantifying the resilience of ML applications. However, they focus on only deep neural networks (DNNs) and hardware faults, and hence these techniques cannot be applied to other types of ML algorithms or faults. In contrast, TensorFI targets a broad selection of ML algorithms, and can be used to study both software and hardware faults.

VIII. Conclusion

In this paper, we built TensorFI, a fault injection framework for TensorFlow for evaluating the error resilience of ML applications. TensorFI is flexible, easy to use, and portable. We use TensorFI to explore the effects of different parameters and algorithms on the resilience of ML applications.

Based on our results, we find that the error resilience of ML applications can be very different under different algorithms and input datasets. Hence, ML applications need to be evaluated on a per application basis before their deployment, in order to benchmark their operational resilience. Further, we find that the error resilience (i.e., accuracy drops) of ML applications depends on the amount of output classes available in the input dataset used. This should be taken into consideration when designing resilient ML applications.

In the future, we plan to explore how different structures and parameters of the ML models may affect their error resilience. We also plan to expand our study to include different types of faults incurred by different ML operators.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Cesare Alippi, Vincenzo Piuri, and Mariagiovanna Sami. Sensitivity to errors in artificial neural networks: A behavioral approach. *IEEE Transactions on Circuits and Systems*, 42(6).

[3] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[4] Simone Bettola and Vincenzo Piuri. High performance fault-tolerant digital neural networks. *IEEE transactions on computers*, (3), 1998.

[5] Ravishankar Iyer, Dong Tang, et al. Experimental analysis of computer system dependability. 1993.

[6] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239, 1998.

[7] Anna Lanzaro, Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, 2014.

[8] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[9] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik 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.

[10] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. Ares: a framework for quantifying the resilience of deep neural networks. In *55th Annual Design Automation Conference*, 2018.