Understanding Soft Error Sensitivity of Deep Learning Models and Frameworks through Checkpoint Alteration

Elvis Rojas*†, Diego Pérez*, Jon C. Calhoun¶, Leonardo Bautista Gomez‖, Terry Jones**, Esteban Meneses*§
*Costa Rica Institute of Technology, †National University of Costa Rica, ¶Clemson University
‖Barcelona Supercomputing Center, **Oak Ridge National Laboratory, §Costa Rica National High Technology Center
erojas@una.ac.cr, diperez@estudiantec.cr, jonccal@clemson.edu, leonardo.bautista@bsc.es, trjones@ornl.gov, emeneses@cenat.ac.cr

***Abstract*—The convergence of artificial intelligence, high-performance computing (HPC), and data science brings unique opportunities for marked advance discoveries and that leverage synergies across scientific domains. Recently, deep learning (DL) models have been successfully applied to a wide spectrum of fields, from social network analysis to climate modeling. Such advances greatly benefit from already available HPC infrastructure, mainly GPU-enabled supercomputers. However, those powerful computing systems are exposed to failures, particularly silent data corruption (SDC) in which bit-flips occur without the program crashing. Consequently, exploring the impact of SDCs in DL models is vital for maintaining progress in many scientific domains. This paper uses a distinctive methodology to inject faults into training phases of DL models. We use checkpoint file alteration to study the effect of having bit-flips in different places of a model and at different moments of the training. Our strategy is general enough to allow the analysis of any combination of DL model and framework—so long as they produce a Hierarchical Data Format 5 checkpoint file. The experimental results confirm that popular DL models are often able to absorb dozens of bit-flips with a minimal impact on accuracy convergence.**

***Index Terms*—deep learning, resilience, checkpoint, neural networks, high-performance computing, HDF5, fault injection**

I. Introduction

The recent wave of artificial intelligence (AI) techniques and tools has overtaken the scientific computing. Significant advances in complex challenges have been registered, such as identifying fake news after crunching massive text corpus to automating extreme weather detection in climate analysis. The impact of modern AI algorithms, particularly deep learning (DL) models, permeates many scientific disciplines. Moreover, the prospect of applying these models to other areas is promising [1], [2].

Contemporary DL frameworks [3]–[5] are heavily dependent on high-performance computing (HPC) hardware, particularly GPU-enabled systems. Neural networks with many internal layers and ensembles of those networks are computationally demanding [6], [7]. To satisfy their strong need for data crunching, many-core chips—usually in the form of GPUs—are exploited at a convenient balance of two rates: operations per second and operations per watt. Nevertheless, hybrid CPU-GPU architectures in HPC systems feature several sources of errors, particularly silent data corruption (SDC). SDC could occur by an alpha particle hitting a circuit and flipping a bit [8]. Sometimes, a bit-flip goes unnoticed until the end of execution, presumably rendering an erroneous or unexpected result. Reliability in HPC systems has become one of the primary recognized concern in keeping large-scale supercomputers productive [9]. Therefore, it is fundamental to address the effect of errors in scientific workflows that incorporate DL algorithms.

This paper provides an understanding of the impact of bit-flips on DL models. To construct an experimental environment in which we are able to try different DL models on different frameworks, a distinctive methodology is followed. Instead of creating a modified version of a framework for a particular model, we create a checkpoint-based fault injector. Therefore, by only altering a checkpoint file made by any DL framework for any model, the impact of SCD can be studied. Additionally, since we control the fault injection, we run tailor-made experiments to investigate the sensitivity of DL models to alterations made to particular bits on particular layers or examining the robustness to a varying amount of bit-flips. Our strategy is inclusive in that through a Hierarchical Data Format 5 (HDF5) checkpoint file, we inject faults and study their effects.

This paper makes the following contributions:

- a failure injection strategy based on checkpoint alteration to study the sensitivity of scientific codes to SDC (Section IV);
- an open-source Python implementation of a parameterized fault injector for HDF5 checkpoint files (Section IV); the injector provides several control variables for studying the effects of faults in scientific codes;
- a derived mechanism, called *equivalent injection*, that allows studying the effect of the same error injection with application independence across different frameworks (Section IV-C);

Notice: This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

- a study of the sensitivity and robustness of representative DL models and frameworks to silent data corruption (Section V); and
- a series of research directions in which the strategy and results presented can be extended (Section VI).

II. Related Work

Fault tolerance in neural networks has been an active research subject for many years. A substantial body of work was developed in the 1990s [10]–[14] and at the beginning of the next decade [15]–[17]. At that time, researchers studied artificial neural networks (ANNs), focusing on the analysis of the faults that affect them. Many of these studies delved into theoretical aspects to solve problems of optimization and the design of new neural networks. In recent years and with the reemergence of DL, researchers have focused on the study of fault tolerance for the new neural network models, current DL frameworks, and improvements on existing hardware. In most studies, researchers manipulate neural network models in some way to generate disturbances or failures that allow them to analyze how these neural networks react to such a situation. Hong et al. [18] explores the training and classification vulnerability of deep neural networks (DNN) under bit errors that can be caused by hardware failures. Their research focused on attack trainings with single bit-flip attacks that are most realistic for representing memory corruption problems in hardware attacks. Liu et al. [19] investigates the impact of failure injection attacks on DNNs by using two types of attacks: (1) a single bias attack and (2) a gradient descent attack in which they try to preserve the classification accuracy in input patterns distinct to the target. Related to that study, the fault sneaking attack is proposed in Zhao et al. [20] as a new method of fault injection attack in DNNs. This attack method is based on the alternating direction method of multipliers. Li et al. [21] evaluates the resilience characteristics of a DNN system. Their research characterizes the propagation of soft errors from hardware to the application software of DNN systems. Fault injection is done by modifying a DNN simulator framework called Tiny-CNN. Many previous studies implemented attacks based on single bit-flip injection, and some based their experimentation on simulators. Our study presents multiple bit-flip injection scenarios, accounting for several configurations of experiments executed in nonsimulated environments using current DL frameworks.

Other studies [22], [23] propose attacks on more specialized neural networks. Rakin, He, and Fan [22] proposes a novel DNN weight attack methodology called Bit-Flip Attack. The idea of this methodology is to destroy the functioning of a neural network by flipping a very small amount of the bits, forming the weights stored in memory. Rakin, He, and Fan [23] proposes a Targeted Bit Trojan method, which inserts a neural trojan into a DNN through bit-flip attacks. This method identifies vulnerable neurons so that an attacker can generate a trigger to force the neurons to generate large output values. These studies present specialized attack techniques that in real execution environments would be unlikely to occur. In our study, attacks are performed by injecting bit-flips, accounting for real scenarios in which failures occur.

The applicability of previous studies must be considered in light of the DL framework they were performed on. Those studies [24]–[27] focused on developing failure injectors in DNNs but only work specifically with a given DL framework. Mahmoud et al. [24] created a tool called PyTorchFI, which is a disturbance tool developed for the PyTorch framework. This tool performs perturbations in the weights during training. Zitao et al. [26] developed fault injector called TensorFI, which injects faults into the data-flow graph of TensorFlow applications. Chen et al. [25] took the TensorFI injector as a base to extend it and implement BinFi, which is an injector that identifies safety-critical bits in DL applications and measures overall resilience. Hu et al. [27] focused on TensorFlow applications and introduced a mutation testing-based tool for DNNs that allows a set of a mutant program to be generated by injecting failures into the original program. One special case is the fault injection framework, called Ares [28], which is an injector built on the Keras framework. All these studies present fault injection tools specifically designed for one or two DL frameworks, which restricts their use. In our study, we develop an injector capable of altering the HDF5 checkpoint files generated by any DL frameworks that support the HDF5 format. With this, we inject bit-flips into any DL framework without compatibility problems.

III. Background

A. Deep Learning Models

DL bases its operation on ANNs, which are algorithms that specialize in identifying relationships in datasets. One of the most notable characteristics of these ANNs is that they can adapt to input changes to learn again without requiring redesign or changing their output criteria. This feature helps neural networks emulate a biological neuron system. There are several neural network architectures with different characteristics, each focused on particular applications. We use convolutional neural networks and a residual neural network. A *convolution* is the application of a filter to an input that results in an activation that passes as a result to the next layer [6], [29]. Conversely, residual neural networks represent a type of network that uses shortcuts or skip connections to move between layers, meaning that the input of a previous layer is added directly to the output of another layer [23]. This allows more layers to be stacked to build deeper networks, allowing layers to be skipped by determining that they are less relevant in training.

Two convolutional neural networks were used in this experiment: AlexNet and VGG16. Additionally, one residual neural network, ResNet50, was used. AlexNet comprises eight layers (five convolutional and three fully connected) and was developed in 2012 for the ImageNet LSVRC-2012 competition [6]. This neural network has 61 million parameters. Based on AlexNet, VGG16 emerged later in 2014 and increases the depth of the convolutional networks used. VGG was developed by the Visual Geometry Group from the University of Oxford.

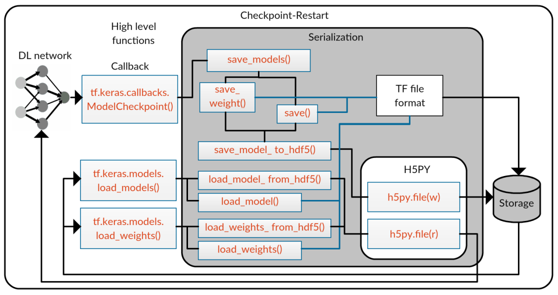


Fig. 1: TensorFlow checkpoint mechanism overview.

The name *VGG16* refers to its 16 layers (13 convolutional layers and three fully connected layers), and it has around 138 million parameters [7]. Finally, ResNet50 is one of the variants of the ResNet neural network and was introduced in 2015. The value 50 refers to the number of layers in the network. ResNet50 has more than 26 million parameters.

B. Deep Learning Frameworks

Neural networks can be implemented in a wide variety of DL frameworks. A DL framework is a tool that allows DL models to build and run easily and quickly without worrying about low-level details (i.e., algorithms) through a high-level programming interface. For this study, the following DL frameworks were included: Chainer, PyTorch, and TensorFlow.

Chainer [3], [30] is an open-source Python framework that was introduced in 2015. It uses a defined-by-run scheme (i.e. Chainer stores the computing history rather than the logic of programming). In the *define* phase, a computational graph is constructed (i.e., instantiation of a neural network object based on a model definition), and in the *run* phase, the model is trained by minimizing the loss function via optimization algorithms. To support GPUs, Chainer implements CuPy, an open-source matrix library accelerated with CUDA, that is a package similar to NumPy.

PyTorch [4] is a tensor DL framework based on the Torch framework [31] and is deeply integrated with Python. PyTorch performs executions of dynamic tensor computations with GPU acceleration and automatic differentiation to completely automate the difficult task of computing derivatives. Also, PyTorch provides an array-based programming model to implement the NumPy library.

TensorFlow [5], [32] was based on a system called DistBelief [33] that originated as part of the Google Brain project. The name *TensorFlow* is derived from multidimensional data arrays called *tensors* with which computations are performed to express them as dataflow graphs. This graph represents the computation in an algorithm and the state in which the algorithm operates.

C. Deep Learning Training Checkpoints

Because DL training can take hours, days, or even weeks running, all DL frameworks implement mechanisms to save and resume the state of a training. This mechanism is called *checkpointing*. The implementation of the checkpoint mechanisms is not automatic. However, DL frameworks do not incorporate sophisticated configuration or optimization options, so checkpoint are easily implemented. All DL frameworks studied use their own custom checkpoint format [34]. But, not all DL frameworks allow data to be saved in HDF5 format [35].

Chainer uses the `snapshot` extension to implement checkpoints. This extension allows the serialization of an object to save it to an output file. This DL framework saves checkpoints in native NPZ format (Numpy's compressed array format) and in HDF5 format. For serialization and deserialization processes, there are specific functions in Chainer called `save_hdf5()` and `load_hdf5()`.

Unlike Chainer, PyTorch implements checkpoint serialization only through the Pickle library [36]. Consequently, to store a checkpoint in HDF5 format, it is necessary to use other Python packages, such as H5PY [37]. For this study, it was necessary to program the serialization and deserialization processes in PyTorch to save a checkpoint in HDF5 format. Thus, we implemented our own HDF5 checkpoint tool.[1]

TensorFlow uses the callback mechanism to interact with the training process and allow the creation of checkpoints. Function `ModelCheckpoint()` is the main callback used to set parameters and manipulate the checkpoint process. TensorFlow also allows the programmer to save checkpoints in native format and HDF5 format. To serialize in HDF5 format, it is only necessary to set the extension of the output file name as `.h5`. Figure 1 illustrates generally the implementation structure of the process of saving and loading a checkpoint with TensorFlow.

IV. Fault Injection

A. Hierarchical Data Format

HDF5 is a file format for high-performance input/output [35]. It has become a standard for efficiently handling large amounts of data, throughout engineering and scientific communities due to its portable and extensible open-source design. "HDF" stands for *Hierarchical Data Format* because the way data are organized internally resembles a file system. An HDF5 file has a collection groups (i.e., folders), which are sets of objects (i.e., files) or other groups. This way, groups are attached together to build paths and organize the data hierarchically. Objects are common data types, such as strings, integers, floats, arrays, datasets, or even custom data types [35]. Many scientific applications use HDF5 to checkpoint, and in many DNNs, the checkpoint contains not only temporal results but the whole model. Models usually require a training phase in which values of the model are refined until a certain threshold is achieved or a fixed number of refinement steps are performed. After this, the trained model can be saved to a file for later use.

B. Fault Injector Design

Contrary to the common approach of injecting a fault during the execution of the application, soft errors are simulated by altering a previously saved checkpoint file. Thus,

[1]The source code is available at: https://github.com/elvinrz/Ckpt_Py_HDF5.

TABLE I: Settings for the HDF5 checkpoint file corrupter.

Setting	Description
hdf5_file	The path of the HDF5 file to corrupt
injection_probability	The probability that each injection attempt is successful
injection_type	Either "count" or "percentage"; the former allows an integer number of injection attempts, the latter allows a percentage of the file that can be corrupted
injection_attempts	The value for the *injection_type*
float_precision	16, 32, or 64 bit precision to use for each corruption of a floating-point number
corruption_mode	• *bit_mask*, A pattern of bits to flip (e.g., 101101), the first bit to apply the mask in each value is randomly selected from [0 to *float_precision* - length(*bit_mask*)], zeros are padded to both sides of the mask to match *float_precision*, then we XOR the mask against the floating-point value • *bit_range*, [first_bit, last_bit] a range of corruptible bits from 0 to *float_precision* -1 • *scaling_factor*, a scale factor to multiply each value
allow_NaN_values	If false, the corrupter does not transform a value to a NaN or INF
locations_to_corrupt	The list of locations to corrupt; all sublocations inside a location will be corrupted
use_random_locations	If true, it will ignore *locations_to_corrupt* and pick a random location every time

when the process loads the corrupted model, it continues execution normally as if nothing happened. We develop an HDF5 file corrupter application in Python for the experimental evaluation.[2] Some of its settings are described in Table I.

The first step of the injector's workflow is to have the definitive list of object's locations to corrupt. If the setting *use_random_locations* is false, then we use the value of *locations_to_corrupt*. Otherwise, we use all the full paths of the objects within the HDF5 file. The second step is to calculate the number of injections attempts. If the *injection_type* is "count", then the value of *injection_attempts* is such number, but if it is "percentage", then it must count the total number of entries in the file that can be corrupted to calculate how many of them represent the specified percentage. By *entries*, we mean the numerical values of all the objects in the file; in dataset objects, the product of their dimensions represents how many entries that object has. After this, the main loop occurs in which a random location is chosen from the list for each iteration, and the injection is attempted. At that point, the value to corrupt is obtained. If it is a simple numerical value, then it is the object itself; if it is a dataset, then it is a random index from its dimensions. We change the value with a probability of *injection_probability* in the following manner, depending on the *corruption_mode*. First, the binary representation of this value is calculated by using the given precision. If the *bit_mask* setting is used, then it is XOR against this value (Table I). If *bit_range* is the selected mode, then a bit within the range [*first_bit*, *last_bit*] is randomly flipped. Otherwise, the value is simply multiplied by the *scale_factor*. Finally, if the resulting value is a not a number (NaN) or infinity (INF) and the *allow_NaN_values* is false, then a new corruption attempt is performed until a valid value is obtained.

This algorithm describes how floating-point values are modified. However, if the value to corrupt is an integer data type, then the corruption process is different. Python has unlimited precision integer values, meaning that the number of bytes it takes to represent an integer value is not fixed. For this reason, we ask Python for the binary representation of the integer by using the built-in function *bin()*. After that, one of those bits is randomly flipped with a probability of *injection_probability* [38].

The key advantage of this error simulation process via checkpoint alteration is the fact that is application independent. No matter the programming language the application is written on or what it actually does, as long as it checkpoints using HDF5 files, it can be corrupted using this approach. This allows multiple corruption scenarios without needing to rerun the application. After a checkpoint is saved, several versions of it can be created by using different corruption configurations, and any of them can be used to restart the application. Moreover, the process enables error injection at specific stages of the application's life cycle, and at any time that a checkpoint can be created, it is corruptible.

Another benefit comes from being able to understand how the HDF5 checkpoint file is structured. As mentioned previously, in many applications, what is saved to these files is the model that allows the calculations to take place. Therefore, it is viable to identify the objects that correspond to each part of the model, and thus corruption can be targeted to specific sections. In the case of neural networks, for example, it is possible to corrupt the objects that represent a certain layer of the network. This error-simulation process can be extrapolated to other scientific applications, not only DL models.

C. Equivalent Injection

When comparing DL frameworks that run the same model, it is valuable to have an equivalent error injection (i.e., inject the same bit-flips on the same place of a particular model running on several frameworks). To do this, another feature was implemented in the fault-injector to save/load the sequence of bit-flips. We save, in a .json file, the specific bits that are changed for each value of a given location within the HDF5 file in a .json file. Using this feature in conjunction with *locations_to_corrupt* allows the user to have a file with the sequence of bits that are flipped in a specific part of framework A, such as the first layer. However, that same specific place on framework B can have a different HDF5 path. For example, the paths *"chpt_ch_vgg_e_5.h5/predictor/conv1_1"* and *"chpt_tf_vgg_e_5.h5/model_weights/_block1_conv1"* represent the first convolutional layer of model VGG using frameworks Chainer and Tensorflow, respectively. Once the sequence of bit-flips injected in a particular location for framework A is saved to a file, the same sequence is replicated for framework B at its equivalent location by changing the location string in the .json and loading it using the HDF5 checkpoint file that is produced by running framework B. This enables an accurate comparison between the frameworks.

We mean *equivalent* and not *equal* because each framework saves the weights of the network differently, and that is

[2] It is hosted on https://github.com/diepe28/hdf5_corrupter

TABLE II: Computer program versions

Computer tool	Version
Linux distribution	Red Hat Enterprise Linux (RHEL)
Linux kernel	4.14.0-115.21.2.el7a.ppc64le
Python	3.7.0
H5PY(HDF5 library)	1.10.4
PyTorch library	1.5.0
TensorFlow library	2.2.0
Chainer library	7.7.0
Horovod library	0.22.0

TABLE III: Experimental configurations and concepts

Concepts or configurations	Description
DL Frameworks	Chainer, PyTorch, and TensorFlow. They are used according to the type of experiment. In some cases, all 3 DL frameworks are used
Neural network models	ResNet50, VGG16, and Alexnet. They are used according to the type of experiment. In some cases, neural network types are used
Dataset	Cifar10. This dataset is used in all experiments
Restart epoch	In all experiments that require resuming training, a checkpoint from epoch 20 was used
Bit-flips	Column header. Indicates the number of bit-flips that are injected into the neural network
DL training	Column header. Indicates the number of trainings that were run in a given experiment

beyond our control. Therefore, saving the dataset and the index for each bit-flip is not very useful because it cannot be mapped to a different framework. Replicating a sequence from one framework to another ensures that the all bit-flips (same amount and order) occur in values that are part of the same location in the model, no matter which framework (e.g., the first convolutional layer of the VGG model using the frameworks Chainer and Tensorflow).

V. Experimental Evaluation

A. Experimental Setup

1) Machine and experiment configuration: All experiments in this paper used the Oak Ridge Leadership Computing Facility's Summit supercomputer at Oak Ridge National Laboratory. With a peak performance of 200 PFlops and power consumption slightly over 10 MW, Summit is listed as the second fastest supercomputer in the world, according to the latest TOP500 list of November 2020. The system contains 4,608 nodes, each comprising two IBM POWER9 CPUs and six NVIDIA Tesla V100 GPUs. Table II lists all versions of software used in the experiments. This section details the most relevant concepts and configurations regarding the experiments performed in Table III.

2) Bit-flip injection process in deep neural networks: We carry out bit-flip injections in neural network models by altering checkpoint files in HDF5. The checkpoint contains all the information of the weights and the structure of the neural network (i.e., layers); therefore, altering the bits that comprise the weights stored in the checkpoint files alters the weights of the neural network. With this, we generate a checkpoint of any DL framework and any neural network model during training to perform the injection process and later loaded the altered checkpoint file to resume execution of the training phase of a neural network model with errors.

3) Deterministic behavior of Deep Neural Networks: Deterministic training is a vital part of the experimental setup to measure differences between error-free training executions vs. training executions with errors. Neural network models used in DL are characterized by the randomness of their execution and results. However, the randomness of the DL models raises reasonable doubts about the reliability of the results. Additionally, it restricts the reproducibility of results and their evaluation [39]–[41]. Thus, to achieve a correct evaluation in the experimentation, we manipulate the DL frameworks to obtain deterministic results in the distributed DL training. Instructions that disable randomness in each of the DL frameworks are discussed here. Randomness is caused by factors related to the parallelism of distributed processes, libraries that are implemented, and the random nature of DL algorithms. Each framework provides different types of options to enable or disable randomness, so each DL framework must be modified according to its characteristics. Furthermore, to obtain deterministic behaviors in multiple executions, the same execution conditions must be maintained in the hardware (e.g., number of GPUs) and software (e.g., DL frameworks versions, DL hyperparameters).

Code 1: Instructions to set up deterministic behavior for the different DL frameworks

```
#Shared instructions between DL frameworks                    1
random.seed(SEED)                                             2
numpy.random.seed(SEED)                                       3
#PyTorch instructions                                         4
torch.manual_seed(SEED)                                       5
torch.cuda.manual_seed(SEED)                                  6
torch.backends.cudnn.deterministic = True                     7
os.environ['HOROVOD_FUSION_THRESHOLD'] = '0'                  8
#Chainer instructions                                         9
cupy.random.seed(SEED)                                        10
chainer.global_config.cudnn_deterministic = True              11
#TensorFlow instructions                                      12
tensorflow.random.set_seed(SEED)                              13
os.environ['TF_DETERMINISTIC_OPS'] = '1'                      14
```

Code 1 shows the instructions used in each of the DL frameworks to achieve deterministic executions. All frameworks share the first two instructions (lines 2–3). These two instructions initialize the seed of the random number generators. Therefore, libraries that use random numbers always generate the same initialization sequences. PyTorch mainly alters the behavior of the GPU-related libraries (lines 5–8). The `torch.manual_seed(SEED)` instruction controls the generation of random numbers within the framework, so initializing it forces all processes to follow deterministic patterns. With the instructions (lines 6–7), we make sure that the operations related to the GPU are deterministic. On the other hand, when Horovod is implemented for distributed training, it generates nondeterminism. Therefore, it is necessary to set the environment variable `HOROVOD_FUSION_THRESHOLD` to 0 to reduce the threshold of tensors that are combined when they are ready to be reduced. Chainer uses the CuPy library to speed up array-related operations with NVIDIA CUDA, so it must be initialized (line 10). It is also recommended to enable the determinism of the cuDNN library (line 11). To perform distributed training in Chainer, we use the ChainerMN library, which does not add additional nondeterminism. With TensorFlow, it

TABLE IV: Incidence of NaN and extreme values (N-EV).

		Chainer						PyTorch						TensorFlow					
		ResNet50		VGG16		AlexNet		ResNet50		VGG16		AlexNet		ResNet50		VGG16		AlexNet	
Bit-flips	Trainings	N-EV	%	N-EV	%	N-EV	%	N-EV	%	N-EV	%	N-EV	%	N-EV	%	N-EV	%	N-EV	%
1	250	1	0.4	0	0	0	0	1	0.4	1	0.4	0	0	1	0.4	0	0	1	0.4
10	250	18	7.2	7	2.8	15	6	22	8.8	17	6.8	12	4.8	17	6.8	7	2.8	7	2.8
100	250	122	48.8	32	12.8	96	38.4	142	56.8	163	65.2	119	47.6	167	66.8	83	33.2	106	42.4
1000	250	249	99.6	188	75.2	241	96.4	249	99.6	248	99.2	249	99.6	246	98.4	227	90.8	234	93.6

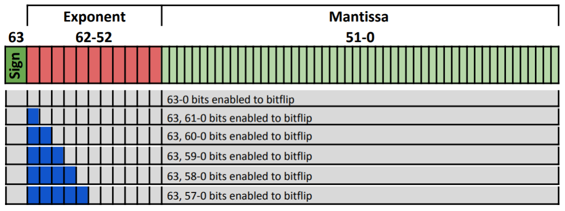

Exponent
Mantissa
63
62-52
51-0
Sign
63-0 bits enabled to bitflip
63, 61-0 bits enabled to bitflip
63, 60-0 bits enabled to bitflip
63, 59-0 bits enabled to bitflip
63, 58-0 bits enabled to bitflip
63, 57-0 bits enabled to bitflip

Fig. 2: Range of bits configured to perform bit-flips.

is only necessary to establish its own randomness generator (line 13). In environments that use GPU, it might be necessary to force the use of deterministic GPU algorithms through the environment variable `TF_CUDNN_DETERMINISTIC`. However, in this case, it was not necessary. Despite the fact that the TensorFlow implementation also uses Horovod, the `HOROVOD_FUSION_THRESHOLD` function does not influence the determinism of the executions. Finally, with TensorFlow, to obtain determinism, it is not necessary to modify the state of GPU-related libraries.

B. Not-a-Number (NaN) and Extreme Values

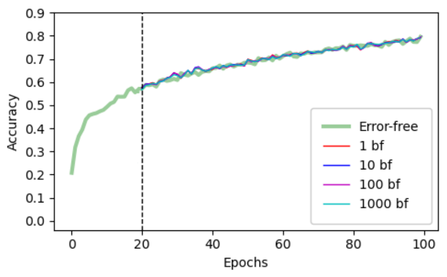
Weights of a DNN model are represented with floating-point values following the Institute of Electrical and Electronics Engineers (IEEE)-754 standard. This format is structured in three parts—sign, exponent, and mantissa—to represent exponential notation. However, one of the first considerations when modifying the bits of a floating-point value is the extreme vulnerability of these values both in single and double precision. Floating-point values can change to NaN values or extremely large values with a few bit changes of the exponent [42]. Sometimes changing a single bit in the float value could dramatically change the original value. For example, the number 0.25 represented in 64 bit IEEE-754 format has a binary exponent of `01111111101`. Performing a bit-flip on the most significant bit (i.e., flipping from 0 to 1) of the exponent would generate the new number 4.49423283715579e+307, which is an integer of 307 digits. We use the term *extreme values* to refer to integers or floats whose value is so large that it causes a neural network to collapse when computing with the value.

1) Bits that collapse a neural network: To determine which bits in the weights of a network can be flipped to collapse it, the fault injector was configured by changing the range of bits on which it could operate. In this way, the bits of the exponent that were considered harmful were excluded. Figure 2 shows how a 64 bit float value is composed in binary. The mantissa is structured from bit 0 (least significant) to bit 51 (most significant). The exponent is structured by 11 bits with bit 62 being the most significant. Bit 63 is used for the sign. For lower precision floating-point values, a 5 bit of exponent and a 10 bit mantissa are used in 16 bit floating-point values, and an 8 bit exponent and a 23 bit mantissa are used in 32 bit floating-point values. Also, this figure shows the configured ranges in the fault injector. Blue means it is not flipped, and gray means it can be flipped. For example, setting *first_bit* to 2 and *last_bit* to 63 tells the injector that the injection range starts at the second bit of the exponent (61 bit) and extends to the first bit of the mantissa (0 bit). Only bits of the exponent are excluded because they are the most significant bits and can generate the greatest numerical variations. A total of 170 training runs are performed per bit range. In each training, a checkpoint is loaded into which 1,000 random bit-flips were injected within the established ranges. The results show that the training collapses only when the injection range accounts for the most significant bit of the exponent (i.e., a probability of 1 in 64).

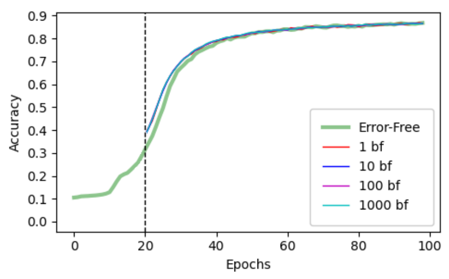
2) Looking for NaN and extreme values: To determine the probability of generating a NaN or an extreme value (N-EV) that collapses a neural network, we carried out another series of experiments with the three DL frameworks and the three neural network models. The bit range used includes all the bits of the floating-point number. The results are shown in Table IV. The table shows the number of trainings that collapsed when computing some N-EV and the percentage that they represent.

In each experiment, 250 trainings were executed and were injected between 1–1,000 bits-flips. One bit-flip is injected per weight of the network so that when injecting 1,000 bit-flips, 1,000 weights of the network are modified. The first row of the table shows that for each neural network model (three models per DL framework), 250 trainings are performed in which a bit-flip is injected into the neural network. As a result, the impact of a bit-flip is minimal, representing less than 0.5% of N-EV in all cases. By increasing the number of bit-flips, the percentage of undesirable values increases almost proportionally, reaching almost 100% of N-EV with 1,000 bit-flips in all frameworks. To determine whether this behavior is a generalizable constant, all neural network models and all DL frameworks are taken into account. In all cases, an ascending pattern is found between bit-flips and the N-EV rate. However, trainings that use VGG16 are less affected than those that use Chainer and TensorFlow. We believe that VGG's structure (i.e., large size, no skip connections) makes it less susceptible to the computation of a larger range of extreme values. For this reason, some trainings might not crash.

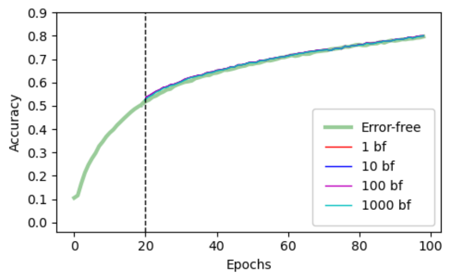
Based on our experimentation and the analyses performed elsewhere [18], [22], flipping certain bits generates extremely



(a) Chainer with ResNet50.



(b) PyTorch with VGG16.



(c) TensorFlow with AlexNet.

Fig. 3: Sensitivity to different bit-flip rates. The green line represents a full 100 epoch training without bit-flips injected.

TABLE V: Model sensitivity to 1 bit-flip. *RWC* stands for the number of trainings that restarted with no change in accuracy.

		Chainer		PyTorch		TensorFlow	
Model	Trainings	RWC	%	RWC	%	RWC	%
ResNet50	250	196	78.4	186	74.4	199	79.6
VGG16	250	134	53.6	194	77.6	240	96
AlexNet	250	226	90.4	115	46	247	98.8

large values. These values cause the total collapse of the network and thus the total collapse of any DL training. Furthermore, the experiments show that the extremely small values that could be generated in the weights of the network are not catastrophic.

C. Bit-Flip Impact

In the next experiments, we omit the most significant bit of the exponent to ensure that the training was executed without collapsing the neural network.

1) Sensitivity to a bit-flip: As long as the same hardware and software parameters were kept, deterministic trainings allows the same results to be obtained in each of the training executions. With this, we determine the sensitivity of the neural network models to the minimum change that is generated as a result of possible errors. Table V shows an experiment in which the sensitivity of the models was measured to 1 bit-flip. Sensitivity was calculated, accounting changes in accuracy. The experiment was performed with the three DL frameworks and with the three neural network models. Additionally, 250 trainings were performed per combination between a DL framework and a neural network model. The table shows the number of trainings that restarted with no change in accuracy (RWC) and its corresponding percentage. In the table, 77.7% of the results correspond to percentages of no change in accuracy higher than 70%. Only in two cases are percentages of 53.6% and 46% reported with Chainer-VGG16 and PyTorch-AlexNet, respectively. However, although these two percentages are low, they only correspond to minor changes in accuracy without degradation. Also, the DL framework with the least affectation is TensorFlow with percentages higher than 95% with VGG16 and AlexNet. Based on this experiment, we affirm that neural network models are very resilient with the ability to absorb a bit-flip.Additionally, if the bit-flip is not absorbed, the impact is shown to be minimal.

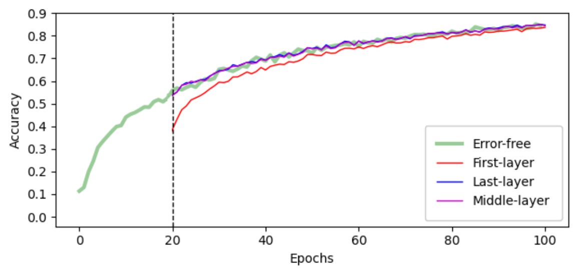


Fig. 4: Fault injection in different layers of AlexNet.

2) Sensitivity to different bit-flip rates: There is a high resilience capacity in the presence of a bit-flip. We develop other experiments in which a greater number of bit-flips was injected. The results are shown in Figures 3 and 4. In both figures, the lines represent the average accuracy of 10 trainings restarting from epoch 20. Figure 3 shows the behavior of the three DL frameworks with different neural network models and different amounts of injected bit-flips. Figures 3a, 3b, and 3c show that regardless of the number of bit-flips injected, the training shows no degradation in accuracy. Figure 3b shows a slight improvement in accuracy, but it is the result of not saving other types of optimization information at the checkpoint and not an improvement due to bit-flips. Figure 4 shows Chainer trainings with AlexNet but with bit-flips injected into the first layer, a middle layer, and the last layer. In the same way, there is little degradation of the accuracy when injecting in the intermediate and final layer. However, a training is affected by inserting the 1,000 bit-flips in the first layer. Although the training of the first layer degrades when restarting in the course of the epochs, the accuracy recovers and reaches almost the same percentage as the training without errors. After these experiments, we conclude that neural network models are highly resilient in the presence of errors.

3) Sensitivity to multi-bit masks: Finally, we run an experiment that injected multi-bits based on predetermined sequences. These multi-bit sequences are taken from Bautista-Gomez et al. [43] that studied the presence of multi-bit errors in memory systems. The experiment used the injector functionality called *bit_mask*, which allows a multi-bit mask to be added instead of a single bit-flip. Table VI shows the results of this experiment in which ResNet50 and five multi-bit masks were used. The average initial accuracy (AvgI-

TABLE VI: Multi-bit mask applied to DL framework training.

		Chainer		**PyTorch**		**TensorFlow**	
Bits	Mask	AvgI-Acc	N-EV	AvgI-Acc	N-EV	AvgI-Acc	N-EV
0	00000000	57.6		30.01		39.2	
3	10001010	57.3	1	29.9	1	36.8	0
4	01101010	57.1	3	29.9	0	36.6	0
4	10110010	57.4	0	29.1	1	36.7	1
5	11110001	53	0	27.2	0	36.5	3
6	11101101	57.4	1	29.9	2	36.8	3

Acc) and the number of N-EVs detected are reported. In this case, N-EVs can be generated because the multi-bit masks are applied randomly in any position within the bits that structure the weight. The row with 0 bit-flips represents the error-free accuracy. Additionally, each multi-bit mask is applied to 10 weights of the neural network, and each training is performed 10 times. In most results, the accuracy is not degraded because the multi-bit mask is applied in the mantissa or in lower priority bits within the exponent. In cases in which the multi-bit mask was applied in higher priority bits within the exponent, N-EVs were produced, as is the case with the multi-bit mask that contains 6 bits. This multi-bit mask generated an N-EV in all three DL frameworks. The AvgI-Acc is not altered because these trainings were exclude to calculate the average. Finally, an interesting case is the 5 bit multi-bit mask since it is the only one that generates an accuracy degradation in both Chainer and PyTorch. This is the result of the 4 bits activated within the mask that make the weight value change so that they generate large values that degrade the accuracy without generating N-EV. Although according to the literature, multi-bit errors are unlikely, it is a case that must be account for to create more robust error detection and correction systems [44]–[46].

D. Floating-Point Precision

Neural network models are becoming larger and more complex, demanding a higher level of computation with more power consumption in many cases due to the precision of the floating-point values [47]. This has resulted in many developers implementing precision less than 64 bit or supporting mixed precision to represent floating-point values. Many DL models can tolerate lower arithmetic precision without degrading their accuracy. This work experimented with bit-flip injections at different floating-point precisions.

1) Incidence of NaN and extreme values: The first experiment shown in Table VII is related to the incidence of N-EV values at different floating-point precisions. The results show the percentage of N-EV in 32 bit and 16 bit precision. Additionally, the trainings are run with Chainer by using all three neural network models. The 64 bit models were ignored because they are contained in Table IV. The percentages maintained a proportion in regards to the number of bit-flips performed(i.e., the higher the bit-flip rate, the higher the N-EV rate). This showed that the incidence of N-EV is not strictly linked to floating-point precision. On the other hand, there was a slight reduction in the percentage of N-EV when reaching 1,000 bit-flips in ResNet and AlexNet trainings in 16 bit and

TABLE VII: Incidence of NaN and extreme values in 16 bit and 32 bit precision.

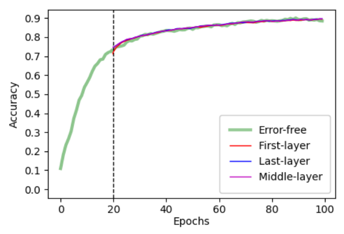
		16 bits			**32 bits**		
Bit-flips	DL Train	ResNet(%)	VGG(%)	AlexNet(%)	ResNet(%)	VGG(%)	AlexNet(%)
1	250	0.4	0	0.4	1.2	2.4	2.8
10	250	10.4	11.6	7.2	15.6	17.2	13.2
100	250	59.2	69.2	60	76.8	72.4	68
1,000	250	96	77.2	86	98	78	91.6

TABLE VIII: Prediction under different floating-point precisions and different bit-flip rates.

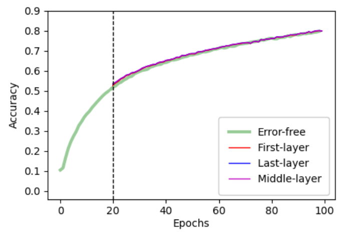
	16 bits			**32 bits**			**64 bits**		
Bit-flips	ResNet	VGG	AlexNet	ResNet	VGG	AlexNet	ResNet	VGG	AlexNet
0	75.6	84.5	83.1	75.6	84.5	83.1	75.6	84.5	83.1
1	75.75	84.16	84.5	76.1	82.95	83.5	74.65	84.9	83
10	74.6(1)	82.8	82.65	69.1	81	81.3	75.3(2)	82.6	82.2
100	60.2(8)	77.3	73.6	44.6(4)	79.1	80.95	56.4(6)	84.8	78.6
1,000	-(10)	42.6(1)	47.24	-(10)	58	66.2	-(10)	72.8	70.2

32 bit models compared with a 64 bit model. This reduction is the result of the generation of values that are not so large that the neural network collapses, such as those obtained in 64 bit models. In the case of VGG, although in 16 bit and 32 bit models do not present a reduction in the percentage of N-EV with respect to precision in 64 bit model, it is less susceptible to N-EV in most cases. VGG reaches with 1,000 bit-flips percentages of 77.2, 78, and 75.2% with precisions of 16 bit, 32 bit, and 64 bit models, respectively. This reduction could be associated with the large size of the neural network (138 million parameters) that manages to alleviate the effect of large values through its layers, resulting in a greater tolerance to degradation.

2) Prediction under different bit-flip rates: We explore the reliability of the prediction under different precisions. The predictions were made with Chainer and accounted for the three neural network models. The results are shown in Table VIII. A trained checkpoint was used up to epoch 100, which was injected with bit-flips at different rates and by using different floating-point precisions. Each result expresses the average percentage of 10 predictions (each prediction processed 1,000 different images), and the values in parentheses represent the amount of N-EV that are detected. The detected N-EVs produce incorrect prediction calculations. Row 1 with 0 bit-flips represents the average error-free prediction percentage. Unlike training, the prediction is affected by bit-flips. The impact is small at low rates of bit-flips but increases at high rates, going from prediction percentages greater than 80% without errors to less than 50% with 1,000 bit-flips in the cases of VGG and AlexNet in 16 bit precision. The neural network model most affected is ResNet because it is the only one that generates N-EV in all precisions from 100 bit-flips, and there is no prediction with 1,000 bit-flips generating N-EV in each of the 10 predictions. These results are related to the incidence of N-EV in ResNet that are present in Table IV in which ResNet with a rate of 100 bit-flips has the highest percentage in almost all cases. The VGG and AlexNet neuronal models are less susceptible to N-EVs under these conditions. Finally,



(a) PyTorch.



(b) TensorFlow.

Fig. 5: Equivalent injection in PyTorch and TensorFlow.

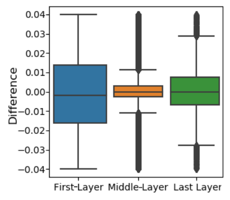
at 64 bit precision, VGG and AlexNet are not very sensitive to bit-flips with a degradation of no less than 70% with 1,000 bit-flips. Consequently, with lower precision, the degradation of the prediction percentage is greater. With the results of this experiment, we conclude that the bit-flips can directly affect the results of the predictions, depending on the precision of the floating-point values.

E. Soft-Error Reproducibility Across Frameworks

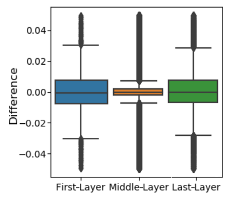
We contrast different frameworks using equivalent injection (Section IV-C). The injector saves a log with three types of information: (1) the number of weights that are modified with the bit-flips, (2) the position of the bit that is flipped, and (3) the layer in which the weight is located. The experiment is divided into three sub-experiments. In each sub-experiment, a different layer of the neural network was injected, and the generated log file was later loaded into other DL frameworks. The log file is generated with Chainer and AlexNet. Figure 4 shows the accuracy of these trainings with 1,000 bit-flips.

The log files generated with Chainer are loaded by the injector, which subsequently performs the injections.With this, it is possible to perform an injection referred to as *equivalent* in PyTorch and TensorFlow trainings. The exact same values cannot be modified in the two DL frameworks that loaded the log because they save the weights in different configurations. Figure 5 shows the results of the equivalent injection in PyTorch and TensorFlow. The bit-flips are injected into the three different layers of the AlexNet neural network model. Figures 5a and 5b show that there is no degradation in accuracy in the training after the injector loaded the log and injected the bit-flips. The bit-flips are clearly absorbed by the computation of the neural networks within the trainings.

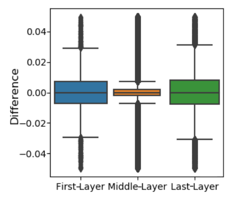
By analyzing the behavior of the DL frameworks for this specific experiment we determine that the least robust DL framework is Chainer (Figure 4). It was clearly was observed that the training that injects in the first layer shows the greatest degradation in the accuracy. Despite the fact that both PyTorch and TensorFlow were injected for the same number of bit-flips in the same layers and in the same bit positions, the accuracy degradation is minimal. Possibly increasing the bit-flip rate or using another technique, such as the scaling factor (Section VI), can generate notable differences between frameworks.



(a) First layer.



(b) Middle layer.



(c) Last layer

Fig. 6: Propagation of errors in a neural network.

F. Soft Error Propagation

In DL training, each time an epoch is executed, the training dataset is processed, and the weights of the neural networks are adjusted based on the calculated error until predictions with sufficient accuracy are obtained. This weight update is performed in each of the layers of the network. We implement an experiment with TensorFlow using AlexNet to understand how errors are propagated. This neural network was selected because it has the fewest number of layers of the three neural networks being studied. This allows us to explore fewer layers to analyze error propagation. The propagation of errors was analyzed by injecting into the first, intermediate, and final layers of the neural network. The weights of an error-free checkpoint from epoch 30 were compared with the weights of a checkpoint from the same epoch with 1,000 injected bit-flips. To compare the same epochs, the checkpoint was injected with the bit-flips in epoch 20 and was trained for 10 epochs until reaching epoch 30. The propagation was calculated based on the difference between the value of the error-free weights and the same weights of the checkpoint injected with the bit-flips.

AlexNet is built by eight layers, so layer 1 (convolutional) was the first layer, layer 4 (convolutional) was the middle layer, and layer 8 (fully connected) was the last layer. Figure 6 shows the propagation of the errors in three boxplots according to the layer in which errors are injected. The boxplots represent the differences that exist in the weights when comparing a checkpoint without alteration regarding a checkpoint with 1,000 bit-flips in a certain layer. Only weights with differences are used.

Figure 6a represents the injection of bit-flips in the first layer of the neural network. The boxplots reflect a greater range of differences, which indicates that the layer suffered the greatest value alterations. A different effect occurs in Figure 6b in which the intermediate layer shows a significant reduction in the range of differences. This indicates that affectation is minimal at this layer. This could be the result of its large size, compared with the first and last layers, which makes the bit-flips better absorbed. Figure 6c shows the last layer with a modest behavior among the three layers. These results suggest that by injecting the bit-flips into the intermediate layer, the absorption effect is generated, which reduces the effect of errors in other layers. The injection into the final layer shows a behavior similar to that of the intermediate layer, which is the result of a reduced propagation effect when performing the backpropagation stage.

VI. Discussion

1) Robustness of DL platforms: The experimental results evidence how solid DL models and frameworks are in the presence of SDCs. That strength comes mostly from the IEEE-754 floating-point number format and the value range of the weights for the models. There is practically only one critical bit. Flipping that bit would mean certain disaster, whereas inverting a subset of the other bits might not alter the final result. If the detection of N-EV was implemented at either the hardware or software level, then DL platforms would be virtually unbreakable.

2) Trade-off in floating-point number representations: The literature of the last decade is rich in studies of numerical methods and systems by using low or mixed precision. The motivation for using lower precision in codes is twofold. Not only are low-precision codes equally accurate in some cases but they are faster and more energy efficient. However, the results highlight the increased sensitivity of DL models when lower precision is used, which is an important trade-off to consider.

3) Dramatic neural network corruption: Because DL models appear to be highly resilient to bit-flips, we decided to explore the limits of such resiliency. Instead of injecting a bit-flip into a value, we used a *scaling factor* to alter that value. Depending on the scaling factor, the number of overturned bits might reach half of them. The heat map in Figure 7 represents the behavior of the accuracy under different scaling factors. The *y*-axis represents the number of weights affected by the scale factor per training. The trainings were executed with Chainer and ResNet50. Each cell depicts the average of 10 executions. The baseline accuracy is 0.576. The effect of scaling values is dramatic. Modifying 10 values with a scaling factor of 4,500 could cut accuracy in half.

4) Deterministic behavior of distributed training: The source of nondeterminism is usually considered helpful in DL trainings because it prevents undesirable effects, such as overfitting. Nevertheless, such nondeterminism prevent us from validating and reproducing the effects of error injection. In this study, we realized that the features offered by DL frameworks to generate deterministic results in training are not always reliable. In some cases, the framework developers do not ensure complete deterministic behaviors between versions and/or platforms (e.g., hardware, software). We believe that DL frameworks should provide easy-to-implement and reliable mechanisms that allow for the reproduction of results and the appropriate validation of the experiments.

5) Universal fault injector: The checkpoint alteration mechanism provides a flexible platform for studying the effects of SDC in scientific codes. It does not capture all possible errors on a real execution because bit-flips in the code segment are not emulated. However, we claim that bit-flips in the data segment are much more predominant as data, particularly for DL models, and represent an overwhelming fraction of all transistors occupied by program execution. Checkpoint alteration is a noninvasive strategy that can be used on any DL model and framework that the produces a checkpoint

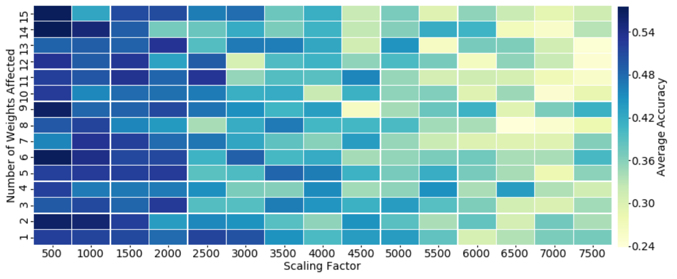


Fig. 7: Accuracy of a model altered with scaling factors.

file. It allows different models, frameworks, and even data representations (e.g. floating-point precision) to be evaluated. Additionally, it is possible to provide certain guarantees, such as equivalent injection across DL frameworks. We argue that checkpoint alteration is applicable to the whole spectrum of scientific codes. Traditional iterative solvers of systems of partial differential equations or particle-interaction codes are well-suited for this technique. Moreover, irregular graph algorithms would also fit our scheme if they produce a comprehensive checkpoint file.

6) Application-level vs. system-level checkpoint: This study focused on understanding the effects of bit-flips on different DL models and frameworks through checkpoint alteration. Checkpoint files are produced by either the frameworks or the programmer. In either case, both represent application-level checkpoints in which only the data required for a future restart are saved. Therefore, temporary and auxiliary data structures are not included in the checkpoint. Conversely, a system-level checkpoint includes the state of all hardware, including state-of-the-art memory hierarchy (e.g., caches, registers). Broadly speaking, checkpoint alteration could be applied to that scenario and model a wider spectrum of errors in the system.

VII. Final Remarks

As DL models continue to permeate many scientific disciplines, it becomes crucial to understand how they integrate with the remainder of the HPC ecosystem. In particular, SDC in hardware components of supercomputers has been a major concern for scientific codes. This paper leverages checkpoint alteration as a mechanism to delve into the effects of SDCs for DL models and frameworks. We create a parameterized HDF5-based checkpoint file fault injector that alters the values of a DL model by flipping bits according to several control knobs. As long as an HDF5 checkpoint file is produced, the injector works with any model and any framework without requiring any modification. This machinery was tested with three popular DL models and three main DL frameworks on Summit. We find the models are extremely resilient as long as a crucial bit in the IEEE-754 format of their values is not inverted. The results also highlight which scenarios provoke the most damage to model training and execution. We envision several directions in which this research can be extended. Not only more DL models and frameworks could be analyzed but different checkpoint file formats could also be explored. Our strategy could be applied to traditional iterative simulation codes or data science algorithms.

References

[1] K. Fagnan, Y. Nashed, G. Perdue, D. Ratner, A. Shankar, and S. Yoo, "Data and models: A framework for advancing ai in science," 12 2019. [Online]. Available: https://www.osti.gov/biblio/1579323

[2] T. Hey, J. Dongarra, F. Streitz, E. Deelman, G. Fox, J. Saltz, A. Sargent, D. Stanzione, and R. Willett, "Us department of energy, advanced scientific computing advisory committee (ascac), subcommittee on ai/ml, data-intensive science and high-performance computing, final draft of report to the committee," 9 2020.

[3] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Y. Vincent, "Chainer: A deep learning framework for accelerating the research cycle," 2019.

[4] A. Paszke and et al., "Pytorch: An imperative style, high-performance deep learning library," 2019.

[5] M. Abadi and et al., "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, Nov. 2016.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, May 2017. [Online]. Available: https://doi.org/10.1145/3065386

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[8] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech, "Experimental and analytical study of xeon phi reliability," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126960

[9] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[10] Y. Tohma and Y. Koyanagi, "Fault-tolerant design of neural networks for solving optimization problems," *IEEE Transactions on Computers*, vol. 45, no. 12, pp. 1450–1455, 1996.

[11] D. S. Phatak and I. Koren, "Complete and partial fault tolerance of feedforward neural nets," *IEEE Transactions on Neural Networks*, vol. 6, no. 2, pp. 446–456, 1995.

[12] C. H. Sequin and R. D. Clay, "Fault tolerance in artificial neural networks," in *1990 IJCNN International Joint Conference on Neural Networks*, 1990, pp. 703–708 vol.1.

[13] G. Bolt, "Fault models for artificial neural networks," in *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, 1991, pp. 1373–1378 vol.2.

[14] C. Neti, M. H. Schneider, and E. D. Young, "Maximally fault tolerant neural networks," *IEEE Transactions on Neural Networks*, vol. 3, no. 1, pp. 14–23, 1992.

[15] P. Chandra and Y. Singh, "Fault tolerance of feedforward artificial neural networks- a framework of study," in *Proceedings of the International Joint Conference on Neural Networks, 2003.*, vol. 1, 2003.

[16] V. Piuri, "Analysis of fault tolerance in artificial neural networks," *Journal of Parallel and Distributed Computing*, vol. 61, no. 1, pp. 18–48, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731500916630

[17] A. S. Orgenci, G. Dundar, and S. Balkur, "Fault-tolerant training of neural networks in the presence of mos transistor mismatches," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 48, no. 3, pp. 272–281, 2001.

[18] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraş, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 497–514.

[19] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 131–138.

[20] P. Zhao, S. Wang, C. Gongye, Y. Wang, Y. Fei, and X. Lin, "Fault sneaking attack: a stealthy framework for misleading deep neural networks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019.

[21] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," ser. SC '17, New York, NY, USA, 2017.

[22] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1211–1220.

[23] ——, "Tbt: Targeted neural network attack with bit trojan," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 13 195–13 204.

[24] A. Mahmoud, N. Aggarwal, A. Nobbe, J. Vicarte, S. Adve, C. Fletcher, I. Frosio, and S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," pp. 25–31, 06 2020.

[25] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "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*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[26] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "Tensorfi: A flexible fault injection framework for tensorflow applications," 2020.

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

[28] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[29] T. N. Sainath, A. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for lvcsr," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[30] S. Tokui and K. Oono, "Chainer:a next-generation open source framework for deep learning," 2015.

[31] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," 2011.

[32] M. Abadi and et al., "Tensorflow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[33] J. Dean and et.al., "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231.

[34] E. Rojas, A. N. Kahira, E. Meneses, L. B. Gomez, and R. M. Badia, "A study of checkpointing in large scale training of deep neural networks," in *18th International Conference on High Performance Computing & Simulation, HPCS 2020, Barcelona, Spain, March, 2021*. IEEE, 2021.

[35] T. Krijnen and J. Beetz, "An efficient binary storage format for ifc building models using hdf5 hierarchical data format," *Automation in Construction*, vol. 113, p. 103134, 2020.

[36] P. S. Foundation, "pickle - python object serialization." [Online]. Available: https://docs.python.org/3/library/pickle.html

[37] A. C. . contributers, "Hdf5 for python." [Online]. Available: http://www.h5py.org/

[38] D. Knuth. Built-in types. [Online]. Available: https://docs.python.org/3/library/stdtypes.htm

[39] P. Nagarajan, G. Warnell, and P. Stone, "Deterministic implementations for reproducibility in deep reinforcement learning," 2018.

[40] I. Icke and J. C. Bongard, "Improving genetic programming based symbolic regression using deterministic machine learning," in *2013 IEEE Congress on Evolutionary Computation*, 2013, pp. 1763–1770.

[41] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup, "Reproducibility of benchmarked deep reinforcement learning tasks for continuous control," 2017.

[42] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," 2013.

[43] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of dram raw error rate on a supercomputer," in *SC '16*, 2016, pp. 645–655.

[44] H. Farbeh, N. S. Mirzadeh, N. F. Ghalaty, S.-G. Miremadi, M. Fazeli, and H. Asadi, "A cache-assisted scratchpad memory for multiple-bit-error correction," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 11, pp. 3296–3309, 2016.

[45] S. Shukla and N. Bergmann, “Single bit error correction implementation in crc-16 on fpga,” in *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, 2004, pp. 319–322.

[46] M. Richter, K. Oberlaender, and M. Goessel, “New linear sec-ded codes with reduced triple bit error miscorrection probability,” in *2008 14th IEEE International On-Line Testing Symposium*, 2008, pp. 37–42.

[47] J. Johnson, “Rethinking floating point for deep learning,” 2018.