Understanding and Mitigating Hardware Failures in Deep Learning Training Accelerator Systems

Yi He
University of Chicago
Chciago, IL, USA
yiizy@uchicago.edu

Mike Hutton
Google
Sunnyvale, CA, USA
mdhutton@google.com

Steven Chan
Google
Sunnyvale, CA, USA
scchan@google.com

Robert de Gruijl
Google
Sunnyvale, CA, USA
rdegruijl@google.com

Rama Govindaraju
Google
Sunnyvale, CA, USA
govindaraju@google.com

Nishant Patil
Google
Sunnyvale, CA, USA
nishantpatil@google.com

Yanjing Li
University of Chicago
Chciago, IL, USA
yanjingl@uchicago.edu

ABSTRACT

Deep neural network (DNN) training workloads are increasingly susceptible to hardware failures in datacenters. For example, Google experienced "mysterious, difficult to identify problems" in their TPU training systems due to hardware failures [7]. Although these particular problems were subsequently corrected through significant efforts, they have raised the urgency of addressing the growing challenges emerging from hardware failures impacting many DNN training workloads.

In this paper, we present the first in-depth resilience study targeting DNN training workloads and hardware failures that occur in the logic portion of deep learning (DL) accelerator systems. We developed a fault injection framework to accurately simulate the effects of various hardware failures based on the design of an industrial DL accelerator, and conducted $> 2.9M$ experiments ($> 490K$ node-hours) using representative workloads. Based on our experiments, we present (1) a comprehensive characterization of hardware failure effects, (2) the fundamental understanding on how hardware failures propagate in training devices and interact with training workloads, and (3) the necessary conditions that must be satisfied for these failures to eventually cause unexpected training outcomes.

The insights obtained from our study enabled us to develop ultra-light-weight software techniques to mitigate hardware failures. Our techniques require 24-32 lines of code change, and introduce $0.003\% - 0.025\%$ performance overhead for various representative workloads. Our observations and techniques are generally applicable to mitigate various hardware failures in DL training accelerator systems.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ISCA '23, June 17–21, 2023, Orlando, FL, USA.
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589105

KEYWORDS

Deep learning accelerator systems, neural network training, resilience, reliability, hardware failures, silent data curroption

ACM Reference Format:
Yi He, Mike Hutton, Steven Chan, Robert de Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023. Understanding and Mitigating Hardware Failures in Deep Learning Training Accelerator Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3579371.3589105

1 INTRODUCTION

Hardware failures are a growing challenge in datacenters, as evidenced by the increasing number of hardware failures that have recently been reported by Google, Facebook, and more [7, 19, 20, 39, 55, 77]. The hardware failure rate is high – a few cores per several thousand server machines [20, 39]. Moreover, a wide variety of hardware failures have been reported, including transient failures such as soft errors and dynamic variations, and permanent failures such as early life failures, manufacturing defects that escape testing, and circuit aging/degradation [19, 20, 39, 55, 77].

As deep neural network (DNN) training workloads are becoming more and more prevalent in datacenters [28, 32, 62], they are increasingly susceptible to hardware failures. For example, through significant efforts, Google recognized and corrected multiple instances of hardware failures during the execution of DNN training workloads on TPUs, with a failure rate similar to previously reported numbers [20, 39]. These hardware failures resulted in not only easy-to-detect unexpected outcomes such as NaN values that corrupted the training process, but also "mysterious, difficult to identify problems" [7]. Due to the widespread use of ECCs (Error Correction Codes) in both on-chip and off-chip memories, these hardware failures predominantly occurred in the logic portion of these TPU systems. They mostly exhibited transient effects – some could not be reproduced at all, while others could only be reproduced intermittently (e.g., when running the same workload 10 times on a faulty machine, the unexpected outcome was only observed 3 times), and were root-caused to manufacturing defects, circuit degradation, voltage variations, environmental conditions, and soft errors, among others.

We have learned several lessons from these reported experiences. First, as many logic hardware failures that pose real threats have been found in datacenter systems, they are not rare and cannot be ignored. Second, contrary to the common belief that hardware failures (especially those that exhibit transient effects) can largely be tolerated by training algorithms, we now have the evidence that suggests otherwise. Third, when an unexpected training outcome occurs, it is critical to determine if the issue is caused by hardware failures or software bugs. Otherwise, significant software engineering efforts would be wasted on debugging a problem engineers incorrectly perceive to be in their software systems. Last but not least, although there is rich resilience literature, in practice, no solution exists to efficiently handle unexpected DNN training outcomes caused by hardware failures (see Sec. 6) – these issues have been termed "bugs from hell" [7], and the industry has issued urgent call-to-action to address them [7, 20, 39].

All of these lessons point to one important realization: there is an urgent and crucial need to devise efficient and effective hardware failure mitigation techniques for DNN training workloads. In order to create new solutions, the critical first step is to thoroughly understand the impacts of logic hardware failures on DNN training workloads. However, there is no such prior study in the literature.

To bridge these important knowledge gaps, we present the first study on hardware failures in DNN training systems. We focus on logic hardware failures that exhibit transient effects, which predominantly occur in datacenters today [7] – in the rest of the paper, *hardware failure* is used to refer to this class of failures unless specified otherwise. Moreover, we focus on deep learning (DL) training accelerator systems, since they are widely used and are currently undergoing rapid growth [26, 62]. Through in-depth analysis, we now have a comprehensive characterization of the hardware failure effects. We also fundamentally understand how hardware failures propagate, as well as the necessary conditions for these failures to eventually cause unexpected outcomes. These new insights enabled us to develop efficient hardware failure mitigation solutions that are readily deployable in practice.

The major contributions of this paper are:

(1) We present the first in-depth study on hardware failures in DNN training accelerator systems, which is enabled by a new fault injection framework that accurately models the behaviors of hardware failures. Using this framework (open-sourced [1]), we performed $> 2.9M$ fault injection (FI) experiments ($> 490K$ node-hours) in a distributed DNN training environment.

(2) Based on the experiment results, we present a complete characterization of the failure behaviors. In addition to known effects (e.g., a failure generates INFs/NaNs [7, 77]), we identified four new, intricate outcomes (Sec. 4.1), where failures resulted in abnormal convergence trends that persist for a long time (thousands of training iterations or more), without visible anomalies. Instances of one of the new outcomes (SlowDegrade, see Sec. 4.1) were later observed (and corrected) in DL training accelerator systems in datacenters.

(3) Deeper analysis led to a finding that large absolute gradient history values in optimizers, or large absolute moving variance values in normalization layers, are the necessary conditions for hardware failures to generate the new unexpected training outcomes revealed by our experiments. Moreover, these conditions always occur within two training iterations after hardware failures occur.

(4) Based on the necessary conditions, we devised (a) a new hardware failure detection technique that checks the absolute gradient history values and the absolute moving variance values against their respective bounds, where the bounds can be mathematically derived based on the properties of a given DNN training workload, coupled with (b) light-weight re-execution of the two most recent training iterations, which is sufficient to recover the training workload from hardware failures. Evaluation on Google Cloud TPUs shows that our detection and re-execution techniques together require $24 - 32$ lines of code change and introduce $0.003\% - 0.025\%$ performance impact for various DNN training workloads.

This paper is organized as follows. Background information is provided in Sec. 2. Our fault injection experiments and results are presented in Sec. 3 and Sec. 4. We present new mitigation techniques in Sec. 5, and discuss related work in Sec.6.

2 BACKGROUND

DNN training workloads are typically executed in a distributed manner using many training devices. For example, in synchronous distributed training [95], every device stores a separate copy of a given DNN model, and uses one mini-batch of the training data-set to compute a training loss through a forward pass, followed by a backward pass where the gradients of the trainable parameters (e.g., weights, biases, etc.) are computed with respect to a loss function using an optimizer. After each iteration, the weight gradients generated by all training devices are averaged (e.g., by a central server). The average gradients are then propagated back to all training devices to start the next iteration.

Hardware failures can pose various effects on DNN training workloads (some examples are shown in Fig. 1). Although it might appear that DNN training is resilient to hardware failures, industry reports have already shown that these failures are detrimental to training and not rare, as discussed in Sec. 1.

Some failures may be masked by hardware logic, e.g., if faulty values are AND'ed with 0's. They may also be masked by various operations performed during the training process, e.g., if a faulty value is multiplied by a 0, or is set to 0 by the activation function. Without the above masking effects, still the final training outcome (training/test accuracy and training time) may not be affected significantly because the training process may be able to recover from the effects of hardware failures. This presents an opportunity: if we can pinpoint the hardware failures that are likely to cause unexpected outcomes, we can devise optimized mitigation solutions.

3 METHODOLOGY AND FRAMEWORK

To study the resilience of DL training accelerator systems, we performed statistical fault injection (FI) experiments, the most widely-used approach for analyzing hardware failure behaviors [15–17, 46, 48, 58, 72, 78].

Existing FI methods suffer from the following limitations. Fast FI is typically achieved by injecting faults in software [48, 49, 93].

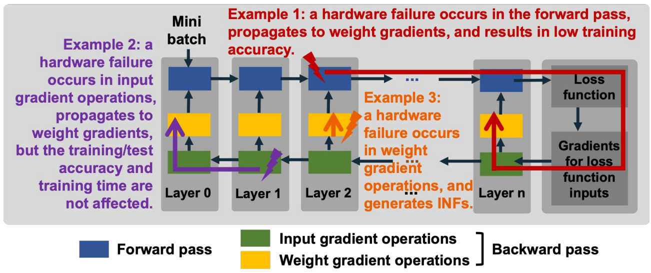

Example 1: a hardware failure occurs in the forward pass, propagates to weight gradients, and results in low training accuracy.
Mini batch
Example 2: a hardware failure occurs in input gradient operations, propagates to weight gradients, but the training/test accuracy and training time are not affected.
Example 3: a hardware failure occurs in weight gradient operations, and generates INFs.
Layer 0
Layer 1
Layer 2
...
Layer n
Loss function
Gradients for loss function inputs
Forward pass
Input gradient operations
Weight gradient operations
Backward pass

Figure 1: Hardware failure examples in DNN training.

However, the accuracy of software FI is low [17, 35, 69]. To achieve high accuracy, RTL-level FI needs to be performed; however, RTL simulation time is prohibitively long, especially for the already time-consuming DNN training workloads. For example, using Resnet18 with the Cifar10 data-set as the DNN training workload, it would take $46K$ years with 8 threads to perform $1M$ RTL FI experiments to achieve high statistical significance. Our methodology and framework, discussed in this section, explain how we overcame these challenges.

3.1 Accelerator Architecture for DNN Training

Detailed hardware information (e.g., RTL) is required to obtain accurate FI results. Although there are no DL training accelerators with open-source access, an inference accelerator can be adopted for training because the designs of DNN training and inference accelerators are similar. For example, TPU v4 (training) and v4i (inference) share the same design [45], and the same is true for Nvidia A100 (training) and A30 (inference) [63]. The training and inference versions differ mainly in the number of cores.

In our study, we adopted NVDLA, Nvidia's DL accelerator [64], as our base architecture (to the best of our knowledge, NVDLA is the only industrial DL accelerator with open-source RTL access). However, the key findings from our work can be generalized to other accelerator designs, because DL accelerators follow a similar dataflow architecture [12, 41, 45, 75], and are expected to experience similar hardware failure effects.

The major modules in NVDLA include (1) $512KB$ on-chip buffers to store layer inputs, weights, partial sums, and layer outputs, (2) sequencing units that control the dataflow of inputs and weights, (3) 16 parallel compute units that perform MAC (Multiply-ACcumulate) operations, and (4) compute units for element-wise, activation, and pooling operations.

To use NVDLA for training, on the hardware side, we augmented the datapath so that bfloat16 and FP32 are used for MAC and element-wise operations, respectively, which is a common precision setting for training [60]. On the compilation side, we introduced extra matrix transpose and rotation operations such that the order of gradient computations, which is fixed by NVDLA's dataflow algorithm, matches that required by the training algorithm [91].

3.2 Fault Injection Framework

We achieve accurate and quick FI in our framework by (1) deriving a set of software fault models through a systematic analysis of NVDLA's RTL, and (2) injecting software faults (i.e., instances of the software fault models) using Tensorflow [25], where each software fault accurately captures the behavior of a hardware failure. We have open-sourced our framework [1].

3.2.1 Hardware Fault Model. Hardware failures are modeled using single-cycle, single-FF (flip-flop) bit-flips. This fault model is widely used to study dynamic variations, unstable/marginal circuit behaviors, and soft errors [10, 15–17, 30, 40, 44, 46, 61, 78, 83, 88]. The observations obtained using this model may also provide insights for other failures that exhibit transient or intermittent effects. For example, the SlowDegrade outcome (Sec. 4.1), which was first revealed by our study, was later observed in real systems and root-caused to hardware failures that can be reproduced intermittently.

3.2.2 Software Fault Models. Based on the hardware fault model, we derived a set of software fault models, which can accurately represent the effects of hardware faults.

A subset of the software fault models in our framework draw similarities to those from our previous work called FIdelity [35], which provides software fault models for DNN inference workloads based on the same hardware fault model used in this study. These similar fault models are the ones used to represent bit-flips in a *datapath FF* (i.e., an FF in the accelerator's datapath) or a *local control FF* (i.e., an FF that controls exactly one datapath register) [35], because the dataflow and compute operations are the same in the forward/backward pass of training, and also during inference. Therefore, given a single-cycle bit-flip in one of these FFs (following the hardware fault model), the number of faulty elements in the output tensor, their relative positions, and their faulty values are derived in exactly the same way for all of the above operations.

The key difference between the new framework and FIdelity lies in how bit-flips in *global control FFs* are modeled. Global control FFs are FFs in the control logic that affect more than one datapath registers. Thus, a bit-flip in a global control FF can result in many faulty elements in the output tensor of the current DNN layer (see Table 1 for some examples). For inference, it is highly likely that the final prediction will be different from the fault-free prediction, so FIdelity simply models bit-flips in global control FFs as such [35]. In contrast, for training, many faulty output elements in a single DNN layer do not necessarily lead to unexpected training outcomes, because the training process may still be able to recover from the hardware failure effects. Therefore, accurate software fault models for bit-flips in global control FFs are required for training.

To this end, we systematically studied the functionalities of all global control FFs in NVDLA ($41K$ in total, corresponding to $7,531$ unique control variables) to derive the corresponding software fault models, as summarized in Table 1.

3.2.3 Validating the New Software Fault Models. We performed $40K$ RTL FI experiments, targeting global control FFs, for five layers arbitrarily selected from five representative DNN models: GoogleNet [84], Resnet [33], Transformer [87], Yolo [73], and LSTM [38]. For each RTL experiment where the injected fault is not masked by hardware ($11K$ total), we confirmed that the faulty output elements match those obtained by simulating the corresponding software fault. Given this result, we can estimate with 99% confidence that the accuracy of our software fault models is very high, with < 1 in $1M$ faults not modeled correctly.

Table 1: Fault injection framework and methodology.

DL accelerator:	**Software fault injection platform**:	**Hardware fault model**:
NVDLA [64], adopted for training	Tensorflow [25]	a single-cycle bit-flip in a single FF

Definitions and terminologies:

Layer_Output: output neurons in forward pass, input gradients or weight gradients in backward pass.

Layer_Input_1: input feature map in forward pass and for weight gradient operations, output gradients for input gradient operations.

Layer_Input_2: weights in forward pass and for input gradient operations, output gradients for weight gradient operations.

n: an integer ≥ 1 indicating how long the effect of a fault lasts in a given DNN layer. If the FF where the fault occurs has a feedback loop. n is randomly chosen between 1 and the max number of loop iterations. Otherwise, $n = 1$.

Layer_Outputs computed in one cycle: they belong to 16 consecutive channels, computed by 16 MAC units in parallel.

Layer_Outputs computed in n consecutive cycles: output elements across n cycles grow in the width dimension.

Layer_Inputs_1/Layer_Inputs_2 required in one cycle: they belong to 64 consecutive channels.

Layer_Inputs_1/Layer_Inputs_2 required in n consecutive cycles: input elements across n cycles grow in the width dimension.

Software fault models for datapath FFs and local control FFs: same as FIdelity [35]

Accurate software fault models for global control FFs	**For which bit-flips?**	% **FFs**
1. Random faulty values that can span the entire data precision dynamic range are set in all Layer_Outputs computed in one cycle, for n consecutive cycles.	A bit-flip in a configuration FF, or a valid signal for Layer_Output turns from 'invalid' to 'valid', affecting all 16 MAC units.	0.24%
2. All Layer_Outputs computed in one cycle are set to 0, for n consecutive cycles.	A valid signal for Layer_Output turns from 'valid' to 'invalid', affecting all 16 MAC units.	0.25%
3. One Layer_Output element is randomly chosen, and its value is set to a random faulty value in each cycle. This effect lasts for n consecutive cycles.	Same as group 1, but the bit-flips affect only one MAC unit.	0.48%
4. All Layer_Outputs computed in one cycle are written to incorrect, randomly chosen memory locations while maintaining their relative positions, for n consecutive cycles.	Bit-flips in FFs that control the memory addresses of Layer_Outputs.	2.36%
5 / 6. All Layer_Inputs_1 / Layer_Inputs_2 required in one cycle are read from incorrect, randomly chosen memory locations while maintaining their relative positions, for n consecutive cycles (from DRAM) or one cycle (from on-chip buffers).	Bit-flips in FFs that represent the memory addresses of Layer_Inputs_1 / Layer_Inputs_2.	1.31% / 0.96%
7 / 8. All Layer_Inputs_1 / Layer_Inputs_2 required in one cycle are set to 0, for n consecutive cycles (from DRAM) or one cycle (from on-chip buffers).	A valid signal for Layer_Input_1 / Layer_Input_2 turns from 'invalid' to 'valid'.	0.09% / 0.22%
9 / 10. All Layer_Inputs_1 / Layer_Inputs_2 required in one cycle use a random set of values from Layer_Input_1 / Layer_Input_2, while maintaining their relative positions, for n consecutive cycles (from DRAM) and 1 cycle (from on-chip buffers).	A valid signal for Layer_Input_1 / Layer_Input_2 turns from 'valid' to 'invalid'.	0.16% / 0.12%

3.3 Experiment Setup

We implemented the software fault models derived for NVDLA using Tensorflow APIs. The DNN models used in our study are summarized in Table 2. In the fault-free runs, we trained each workload for $430 - 50K$ iterations, which corresponds to $40 - 80$ epochs with 8 training devices (similar to typical training procedures in the literature [60]). For each workload, the final fault-free training/test accuracy reaches $> 95\%$ of that reported in the corresponding paper cited in Table 2.

We deployed our framework on Google Cloud TPUs, and conducted $> 2.9M$ ($> 490K$ node hours) FI experiments. Each FI experiment consists of the following steps: (1) randomly select an FF and a cycle to indicate where and when a bit-flip is to be injected; (2) use the corresponding software fault model to obtain the number and the positions of all faulty output elements in the current DNN layer; (3) obtain the faulty values of the faulty output elements based on the software fault model; and, (4) propagate the effects of the faulty output elements in the current DNN layer by continuing to train the DNN until either an error message (e.g., one that reports the occurrence of INFs/NaNs) is encountered, or until a predefined number of training iterations are completed.

For each workload, the upper bound of the training iterations used in our experiments is 2× the number of iterations in the fault-free run (reported in Table 2). In each FI experiment, we captured the convergence trend by recording the training loss and accuracy values in every training iteration, as well as the test accuracy once every 100 training iterations.

4 RESULTS

4.1 Characterization of Hardware Failure Effects

We observed two distinct categories of training outcomes from our FI experiments. In the first category, which accounts for $82.3\% - 90.3\%$ of all cases across the workloads, the injected faults did not significantly affect the final training/test accuracy for the same training time as the fault-free runs. In fact, the majority of them ($65.5\% - 86.3\%$ of all cases) yielded slightly higher training/test accuracy compared to the fault-free cases, perhaps because the

Table 2: DNN training workloads. Optimizer: Adam (except for Resnet_SGD). Momentum value in batch normalization (BatchNorm) layers: 0.9 (except for Resnet_LargeDecay).

DNN models	Data-sets	Num. iterations / epochs (fault free)	Num. experiments
Resnet [33] (4 configurations*)	Cifar10[47]	1960 / 80	>900K
DenseNet [42]			>400K
Efficientnet [85]			>400K
NFNet [8]			>100K
Yolov3 [73]	VOC12[21]	430 / 40	>200K
Multi-grid neural memory [43]	25*25 maze	50000 / N/A	>400K
Transformer [87]	WMT14 EN-DE [6]	50000 / 40	>100K

* Four configuration of Resnet18: (1) Resnet, a BatchNorm layer follows every convolution layer; (2) Resnet_NoBN, no BatchNorm layers; (3) Resnet_SGD, same as Resnet, except that SGD (stochastic gradient decent) is used as the optimizer; (4) Resnet_LargeDecay, same as Resnet, except that the momentum value in BatchNorm layers is 0.99.

faults created noises that introduced certain regularization effects. The rest of the cases in this category showed slight degradations (mostly within 2%, up to 6%) in training/test accuracy for the same training time compared to the fault-free runs. These cases by and large correspond to those where faults were injected late in the training process. For these cases, when we increased the training time by 10% / 17% to allow the training algorithm to recover the effects of the faults, the training/test accuracy differed by only less than 2% / 0.5% from that of the corresponding fault-free runs.

The remaining 9.7% − 17.7% of the FI experiments, belonging to the second category, all exhibit certain unexpected training outcomes. We characterized these outcomes based on (1) convergence trends (i.e., training/test accuracy values throughout the training process), and (2) occurrences of visible anomalies, as shown in Table 3. In addition to the occurrences of INFs/NaNs which have been reported by industry, we discovered four new unexpected outcomes: (1) SlowDegrade, (2) SharpSlowDegrade, (3) SharpDegrade, and (4) LowTestAccuracy. In Fig. 3, we report the percentage breakdown of different training outcomes normalized to the total number of experiments for each workload.

Based on the same statistics analysis methodology used in previous resilience studies [17, 54], we have achieved a 99% confidence level that the percentage of each outcome reported in this section is within a confidence interval of 0.1%. The probability of an unexpected outcome not exposed by our experiments is < 0.004% with a 99.5% confidence level. Moreover, after observing the SlowDegrade outcome in our experiments, this outcome was later observed in datacenters when training large DNN workloads using DL training accelerator systems.

4.2 Detailed Analysis

A detailed characterization of the fault propagation paths and effects are summarized in Fig. 4. We have also derived the necessary conditions for a fault to generate a latent unexpected outcome.

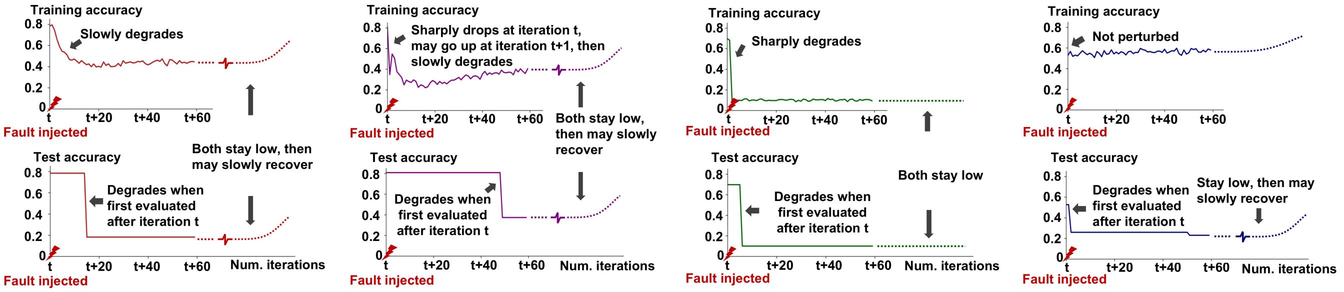
Table 3: Unexpected outcomes in DNN training workloads.

Symptoms	Descriptions
Manifestation latency: immediate	
INFs/NaNs	A fault in the forward pass: INFs/NaNs are observed in the forward or backward pass of the current iteration. A fault in the backward pass: INFs/NaNs are observed in either the backward pass of the current iteration, or the forward pass of the next iteration.
Low hardware utilization	Hardware resources are not fully utilized, resulting in sub-optimal performance, because a faulty control FF incorrectly disables a subset of hardware modules.
Accelerator hang	The accelerator fails to notify the host server that its task is completed within a pre-specified timeframe, because a fault causes some control logic to be stuck in an infinite loop.
Manifestation latency: short-term	
INFs/NaNs	INFs/NaNs show up within a few training iterations (2 in our experiments) after a fault occurs.
Manifestation latency: latent	
SlowDegrade (Fig. 2a)	Training accuracy slowly degrades for $10 - 100$ iterations, then stays at a low level. Training/test accuracy may recover after $10K - 100M$ iterations.
SharpSlow Degrade (Fig. 2b)	Similar to SlowDegrade, except that an additional sharp drop in training accuracy is observed at the iteration when a fault occurs.
SharpDegrade (Fig. 2c)	Training accuracy drops sharply at the iteration when a fault occurs, and stays at a low level. Test accuracy follows training accuracy.
LowTest Accuracy (Fig. 2d)	Training accuracy appears normal, but test accuracy shows visible degradation after a fault occurs. Test accuracy may recover after $10K - 100M$ iterations.

4.2.1 Analysis on the Immediate Outcomes. Immediate INFs/NaNs are generated by faults in the following FFs: (1) the datapath FFs that represent the high exponent bits, (2) a subset of control FFs that configure the data precision (e.g., if a fault in one of these FFs causes int16 MAC operations to be performed instead of bfloat16 operations, the results may overflow when they are converted to FP32 to undergo element-wise operations), and (3) FFs that correspond to valid/invalid signals (a fault in one of these FFs can result in incorrect logic functions that generate arbitrary datapath values, including INFs/NaNs).

The faults that can generate the other two immediate unexpected outcomes (low hardware utilization and accelerator hang) cannot be modeled using software-visible states. We observed these outcomes in our RTL FI experiments (see Sec. 3.2.3), and included them in Table 3 for completeness.

4.2.2 Analysis on the Short-Term INFs/NaNs Outcome. The fault propagation paths leading to this outcome are shown in Fig. 4. There are two major events along these paths. First, at the end of iteration t (i.e., the iteration when a fault occurs), weights with large absolute values are generated as a result of the fault and propagate to subsequent training iterations. Second, a history term combines the effects of large absolute faulty weight values across at least two



(a) SlowDegrade (b) SharpSlowDegrade (c) SharpDegrade (d) LowTestAccuracy

Figure 2: Four new unexpected latent outcomes observed from our experiments.

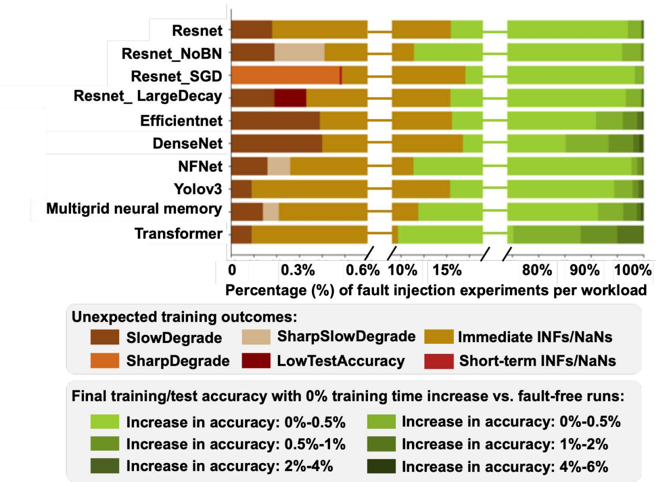


Figure 3: Percentages of training outcomes, normalized to the total number of experiments for each workload.

training iterations to generate a value that overflows after iteration $t + 1$.

History terms are used in normalization layers that are widely adopted in DNNs [4, 8]. For example, moving variance (*mvar*) in BatchNorm is such a history term, which combines the variance of the layer's inputs with the *mvar* obtained in the previous iteration, weighted using a decay factor: $mvar\ at\ iter\ (i{+}1) = decay\_factor * mvar\ at\ iter\ i + (1 - decay\_factor) * input\ variance$. The inputs of a BatchNorm layer are the outputs of the previous layer, which can contain large absolute neuron values because of the faulty weight values. The faulty BatchNorm layer inputs can result in a large absolute *mvar* value, which may overflow after iteration $t + 1$. For clarity, we use *mvar* to generally denote such a history term in normalization layers.

Short-term INFs/NaNs are rare. First, if an optimizer that normalizes gradients (e.g., Adam) is used, large absolute weight values can only be generated if a fault occurs during the weight update operation (i.e., the operation that adds gradients to current weight values), which is extremely unlikely because this operation takes a very small amount of time. This is why we observe this case for Resnet_SGD only in our experiments, since SGD does not normalize gradients.

Second, the absolute value of a faulty *mvar* across multiple iterations must lie in a specific range ($2.9e38 - 3.0e38$ from our experiments, as shown in Table 4) so that it does not overflow at iteration t, but overflows at a later iteration. Moreover, the overflow is expected to appear shortly after iteration $t+1$ because (1) a decay factor is applied to *mvar*, and (2) although quite slowly, the faulty weights are updated towards the correct direction (decreasing their absolute values) by the optimizer. Thus, it is not likely for INFs/-NaNs to occur beyond a small number of iterations after a fault occurs. Because of this decaying effect, the magnitude of *mvar* at iteration $t + 1$ must be very close to the max floating point value that can be represented (e.g., the max value of FP32 in our study). A large absolute *mvar* value therefore is a *necessary condition* for this outcome.

4.2.3 Analysis on the SlowDegrade and SharpSlowDegrade Latent Outcomes. The fault propagation paths that lead to these two outcomes are depicted in Fig. 4. Both outcomes are observed only if the optimizer uses gradient history values to normalize the gradients derived in the current iteration, which is common in DL training workloads (e.g., 134 such optimizers were developed out of a total of 154 between 2015 and 2021 [79]). SharpSlowDegrade can only occur if normalization layers are not present (e.g., Resnet_NoBN and NFNet) and if a fault occurs in the forward pass, while SlowDegrade can only occur if a fault occurs in the backward pass. Moreover, the convergence trends of these two outcomes exhibit three distinct phases. We mathematically explain each phase in Fig. 5 using Adam (Eq. 1) as an example.

Operations performed in Adam

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$

$$u_t = \eta \frac{\dfrac{m_t}{1-\beta_1^t}}{\sqrt{\dfrac{v_t}{1-\beta_2^t}+\epsilon}}, \quad w_t = w_{t-1} - u_t \tag{1}$$

g_t : gradient values computed in iteration t. β_1, β_2 : decay factors.
m_t : the history values of the gradients.
v_t : the history values of the square of the gradients.
u_t : the values used to update the weights. w_t : weight values.
η : learning rate. ϵ : a small value for numerical stability.

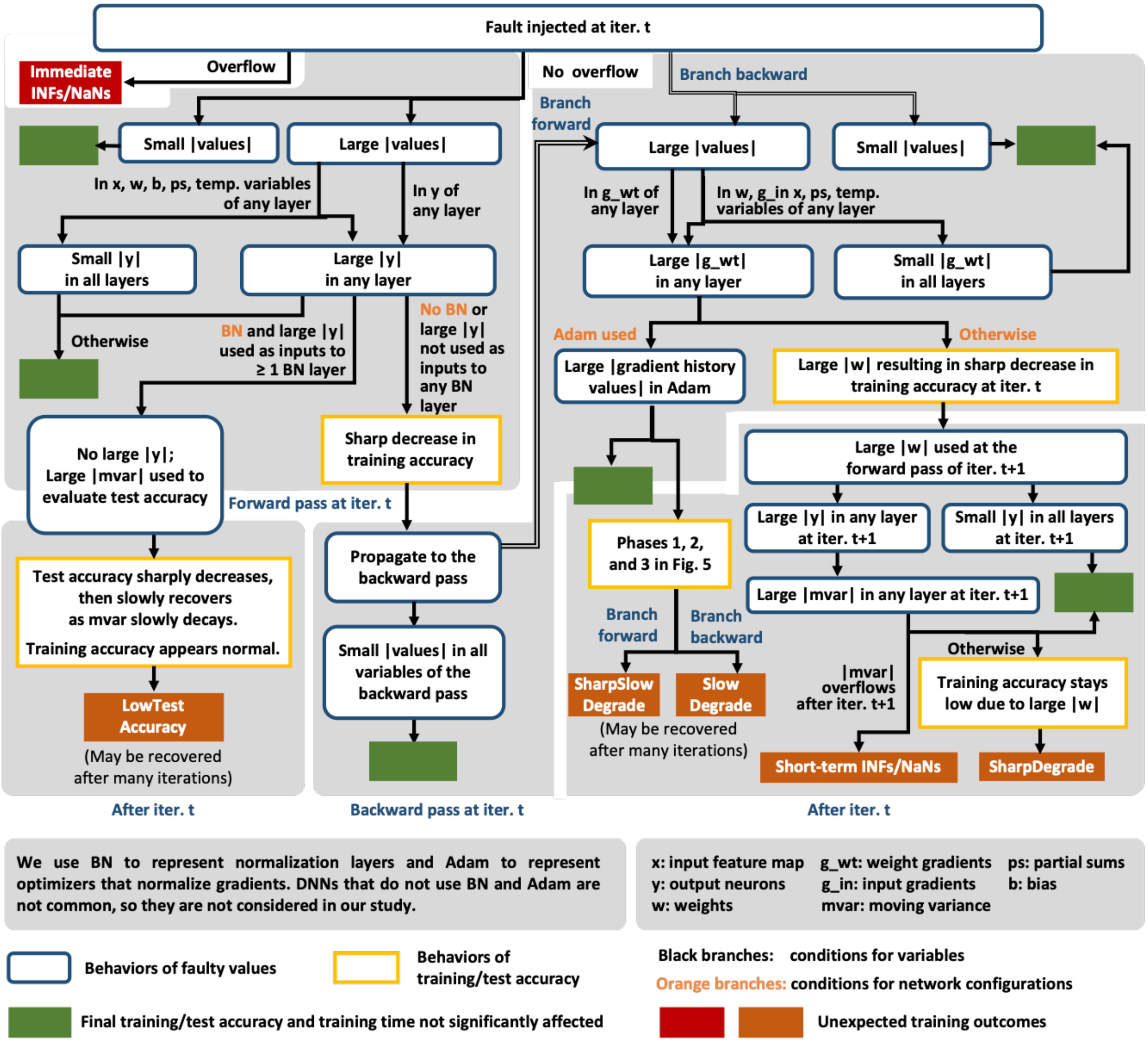

Fault injected at iter. t
Overflow
Immediate INFs/NaNs
No overflow
Branch backward
Branch forward
Small |values|
Large |values|
In x, w, b, ps, temp. variables of any layer
In y of any layer
Small |y| in all layers
Large |y| in any layer
Otherwise
BN and large |y| used as inputs to ≥ 1 BN layer
No BN or large |y| not used as inputs to any BN layer
No large |y|; Large |mvar| used to evaluate test accuracy
Sharp decrease in training accuracy
Forward pass at iter. t
Test accuracy sharply decreases, then slowly recovers as mvar slowly decays. Training accuracy appears normal.
LowTest Accuracy
(May be recovered after many iterations)
After iter. t
Propagate to the backward pass
Small |values| in all variables of the backward pass
Backward pass at iter. t
Large |values|
Small |values|
In g_wt of any layer
In w, g_in x, ps, temp. variables of any layer
Large |g_wt| in any layer
Small |g_wt| in all layers
Adam used
Otherwise
Large |gradient history values| in Adam
Large |w| resulting in sharp decrease in training accuracy at iter. t
Large |w| used at the forward pass of iter. t+1
Phases 1, 2, and 3 in Fig. 5
Large |y| in any layer at iter. t+1
Small |y| in all layers at iter. t+1
Large |mvar| in any layer at iter. t+1
Branch forward
Branch backward
SharpSlow Degrade
Slow Degrade
(May be recovered after many iterations)
|mvar| overflows after iter. t+1
Otherwise
Training accuracy stays low due to large |w|
Short-term INFs/NaNs
SharpDegrade
After iter. t
We use BN to represent normalization layers and Adam to represent optimizers that normalize gradients. DNNs that do not use BN and Adam are not common, so they are not considered in our study.
x: input feature map
y: output neurons
w: weights
g_wt: weight gradients
g_in: input gradients
mvar: moving variance
ps: partial sums
b: bias
Behaviors of faulty values
Behaviors of training/test accuracy
Black branches: conditions for variables
Orange branches: conditions for network configurations
Final training/test accuracy and training time not significantly affected
Unexpected training outcomes

Figure 4: Characterization of fault propagation paths and effects.

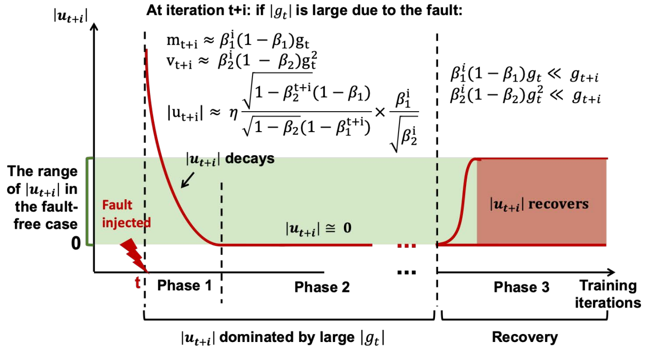

$|u_{t+i}|$
At iteration t+i: if $|g_t|$ is large due to the fault:
$m_{t+i} \approx \beta_1^i(1-\beta_1)g_t$
$v_{t+i} \approx \beta_2^i(1-\beta_2)g_t^2$
$|u_{t+i}| \approx \eta \frac{\sqrt{1-\beta_2^{t+i}}(1-\beta_1)}{\sqrt{1-\beta_2}(1-\beta_1^{t+i})} \times \frac{\beta_1^i}{\sqrt{\beta_2^i}}$
$\beta_1^i(1-\beta_1)g_t \ll g_{t+i}$
$\beta_2^i(1-\beta_2)g_t^2 \ll g_{t+i}$
$|u_{t+i}|$ decays
The range of $|u_{t+i}|$ in the fault-free case
0
Fault injected
$|u_{t+i}| \cong 0$
$|u_{t+i}|$ recovers
t
Phase 1
Phase 2
Phase 3
Training iterations
$|u_{t+i}|$ dominated by large $|g_t|$
Recovery

Figure 5: Explanation of the three phases in the convergence trends of SlowDegrade and SharpSlowDegrade. The math symbols are defined in Eq. 1.

From Fig. 4, we see that a *necessary condition* for both outcomes is that the absolute values of the faulty gradient history values of the optimizer (m_t and v_t in Eq. 1 for Adam) must be large enough to influence training accuracy, but not large enough to cause immediate or short-term INFs/NaNs (the range of faulty values obtained from our experiments is shown in Table 4).

However, this is a necessary condition but not a sufficient condition, because even if this condition is met, it is possible that the final training/test accuracy would not be affected significantly. For example, if only a few gradient history values are perturbed, it may not be significant enough to perturb the overall convergence trend. Moreover, training/test accuracy can start to improve again in Phase 3 (Fig. 5). However, for all of our experiments in which the convergence trend is perturbed due to large absolute gradient history values except for those training the Transformer workload, the final training/test accuracy values are still low even after the numbers of training iterations are doubled with respect to the corresponding fault-free runs. The takeaway is that, although theoretically there is a recovery phase, the final training outcome is largely dependent on the interactions between the magnitudes of the faulty values, the choice of hyperparameters (e.g., decay factors), and the training dynamics. In practice, the recovery phase may never be reached, or it may require millions of iterations to fully recover from the fault. For example, the latter can happen with a decay factor of 0.9999 (used in real datacenter workloads) and a faulty absolute gradient history value in the order of $1e19$ (observed from our experiments).

4.2.4 Analysis on the SharpDegrade Latent Outcome. The propagation path that leads to the SharpDegrade outcome is mostly the same as that for the short-term INFs/NaNs outcome, except that faulty *mvar*'s never overflow in the case of SharpDegrade. Instead, their absolute values must be large enough (which are generated by large absolute weight values due to a fault) for the training/test accuracy to show sharp degradations – this is thus a *necessary condition* for the SharpDegrade outcome. Afterwards, the absolute values of the faulty weights continue to stay large as they are updated very slowly by the optimizer, so the training/test accuracy stay low for a long time.

4.2.5 Analysis on the LowTestAccuracy Latent Outcome. From Fig. 4, we see that LowTestAccuracy can only occur in DNN workloads that satisfy two conditions. First, a history term is used, which is updated based on its values from previous iterations. Moreover, it is only used to evaluate test accuracy but not training accuracy. An example of such a history term is the moving variance in BatchNorm layers (*mvar* as defined previously), which we will use to generally denote such history terms for clarity. Second, the absolute value of *mvar* must be very large, such that it can visibly degrade the test accuracy (see Table 4 for the range observed in our experiments). Thus, this is a *necessary condition* for the LowTestAccuracy outcome.

Similar to the SlowDegrade and SharpSlowDegrade cases, there is typically a recovery phase for LowTestAccuracy, because a faulty absolute *mvar* value will decay over time given the common use of a decay factor (as explained in Sec. 4.2.2). However, whether the test accuracy can be successfully recovered depends on various factors, including the magnitudes of the faulty values, the decay factor, and the training dynamics. In our experiments, LowTestAccuracy is observed for the Resnet_LargeDecay workload, because the large decay factor (0.99, vs. 0.9 in other workloads) corrects the faulty *mvar*'s too slowly.

Moreover, only faults that occur in the forward pass can lead to LowTestAccuracy, because those in the backward pass can only perturb *mvar*'s in the forward pass of the next training iteration through faulty weight values. However, in this case, large absolute weight values will dominate the overall effect and create the SharpDegrade outcome instead.

4.2.6 Summary. We summarize the necessary conditions for a fault to generate each latent outcome (including short-term INFs/NaNs) in Table 4. The necessary conditions always occur within two training iterations after a fault occurs. Note that, these necessary conditions are not sufficient. In our experiments, we observed cases in which the training process is able to recover from the effects of faulty gradient history values in optimizers or faulty *mvar*'s in normalization layers, especially if the number of faulty values is small.

In addition to the necessary conditions, we have obtained the following three key observations from our analysis.

Table 4: Necessary conditions for short-term/latent unexpected outcomes. iter. t is the iteration during which a fault is injected.

Outcomes	Necessary conditions	When conditions observed	Ranges observed in experiments
SlowDegrade	Large absolute gradient history values in optimizer	iter. t	3.6e9-1.1e19
Sharp SlowDegrade		iter. t	2.7e8-1.2e19
SharpDegrade	Large absolute *mvar* values in normalization layers	iter. $t + 1$	6.5e16-1.2e38
LowTestAccuracy		iter. t	7.3e17-7.1e37
Short-term INFs/NaNs		iter. $t + 1$	2.9e38-3.0e38

Observation (1) Recovery effects of DNN training workloads. If the perturbations in all software variables that are affected by a fault are small, then the training process (provided that it is implemented correctly) is highly likely to be able to recover from the effects of the fault without posing high training time overheads. Even if the perturbations are large, given a long enough training time (which may not be practical), the training process may recover from the effects of the fault, unless INFs/NaNs are generated.

Observation (2) Necessary conditions for latent unexpected outcomes. For any hardware fault to cause a latent unexpected outcome, the effects of the fault need to last across multiple training iterations; otherwise, it is highly likely that the training process will recover. This observation is reflected in the necessary conditions, as both the *mvar*'s in normalization layers and gradient history values in optimizers can carry the effects of a fault from one iteration to the next.

The effects of faulty weight/gradient values will also last across multiple training iterations; however, they will propagate to the *mvar*'s and/or gradient history values, and subsumed in our necessary conditions. On the other hand, faulty *mvar* and gradient history values do not always imply faulty weights/gradients.

We also analyzed the behaviors of the training loss value to determine if it can serve as a necessary condition for the latent unexpected training outcomes. For faults that occur in the forward pass and subsequently generate the SharpSlowDegrade, SharpDegrade, and short-term INFs/NaNs outcomes, a sharp increase in the training loss value is observed at the iterations during which the faults first appear. However, for faults that occur in the backward pass, even if they eventually lead to latent unexpected outcomes (SlowDegrade or LowTestAccuracy), the training loss value appears normal throughout the training process.

Observation (3) Interactions between DNN configurations and hardware failures. Normalization layers in DNNs play an important role in the resilience of DNN training workloads. On the one hand, the occurrence of large absolute *mvar*'s is a necessary condition for various short-term and latent unexpected outcomes. On the other hand, the presence of normalization layers makes it more likely for a training workload to recover from the faults that occur in the forward pass. As shown in Fig. 4, if large absolute output neurons (large $|y|$) are generated in the forward pass, normalization layers will normalize the magnitudes of these output neurons, effectively alleviating the impacts of these faults.

The choice of optimizers and hyperparameters also play an important role. For example, the SlowDegrade and SharpSlowDegrade outcomes can only be generated if the optimizer normalizes gradients using gradient history values, while the SharpDegrade outcome can only occur if the optimizer does not.

4.3 Other Results and Discussions

4.3.1 Contributions to Unexpected Outcomes from Different FFs. In NVDLA, global control FFs whose bit-flips belong to groups 1 and 3 defined in Table 1 and local control FFs are more likely to generate large absolute *mvar*'s and large absolute gradient history values. Together they contribute to 55.7-68.5% of the total number of unexpected outcomes across different workloads in our experiments, even though these FFs only account for 9.8% of all FFs in the design.

For datapath FFs, bit-flips that correspond to the upper two exponents bits (5.5% of all FFs) contribute to 31.9%-44.3% of all unexpected outcomes across different workloads. These bit-flips are more likely to generate large absolute values that cause overflow or satisfy the necessary conditions reported in Table 4 than the bit-flips in other datapath FFs.

4.3.2 Generalization to other hardware fault models. The necessary conditions discussed in Table 4 were derived based on the single-cycle single-FF bit-flip hardware fault model. However, based on Observation (2), the same necessary conditions are applicable to any single hardware failure, regardless of the fault model.

Furthermore, given the hardware failure rate reported by industry, it is expected that at most one hardware failure would occur during the training process of mid-sized DNNs (i.e., DNNs with $< 1GB$ parameters), which account for the majority of all DNNs deployed in datacenters today [45]. For larger DNNs, even though multiple failures may occur during the training process, they are expected to occur far enough apart such that their effects are largely independent. Therefore, the same necessary conditions are also applicable to multiple hardware failures under the reported hardware failure rate.

4.3.3 Discussions on the number of training devices. We used 8 training devices in our experiments. With more training devices, our findings still apply. First, if a hardware failure occurs in a training device and generates an immediate unexpected outcome, the outcome will show up in the local device without affecting other devices, so the number of training devices is irrelevant. Second, the necessary condition for short-term INFs/NaNs, SharpDegrade and LowTestAccuracy is large absolute *mvar* values on a single training device, which is not affected by the number of devices. Last but not least, we consider the SlowDegrade and SharpSlowDegrade outcomes, for which the necessary condition is large absolute gradient history values. On the one hand, using more training devices results in a shorter training time, which makes it less likely for a workload to reach the recovery phase, or for the recovery phase to fully recover the training/test accuracy. On the other hand, since gradients are averaged among all training devices, absolute faulty gradient values (due to a hardware failure) would be smaller if more devices are used, making it less likely to meet the necessary condition of these two outcomes. These opposing factors balance out the sensitivity to the number of training devices.

4.3.4 Discussions on the sizes of DNNs and data-sets. How hardware failures propagate and affect DNN training workloads, as shown in Fig. 4, does not depend on the sizes of the DNN or the training data-set. The only consideration is that the sizes may influence when the three phases in SlowDegrade/SharpSlowDegrade and the recovery phase in LowTestAccuracy occur, and how long the different phases last.

5 TECHNIQUES TO TACKLE HARDWARE FAILURES IN DNN TRAINING SYSTEMS

In datacenters, when a potential issue is detected in a DL accelerator, a standard procedure is to decommission the accelerator for further investigation, revert all affected workloads to their previous checkpoints, and execute these workloads in other healthy devices [7]. Handling immediate and short-term NaNs/INFs is easy. However, for a latent unexpected outcome, its error detection latency, i.e., the time between when a hardware failure occurs and when the unexpected outcome is observed, can be very long – spanning thousands to millions of training iterations. The long error detection latency makes it challenging to recover an affected workload. For example, even though checkpointing is routinely used in DNN training, it is not clear how one could determine which checkpoint to revert to, not to mention that the available checkpoints may all have been corrupted.

Therefore, a detection technique that guarantees a short error detection latency is required. Although there exist a plethora of resilience techniques in the literature, these techniques are inadequate because they incur high performance/energy costs even in the absence of hardware failures (more details in Sec. 6). To this end, we leverage the necessary conditions revealed by our study to devise new, efficient techniques to mitigate hardware failures in DL training accelerator systems.

5.1 Detection

Our technique detects all hardware failures that are likely to lead to latent unexpected outcomes. The idea is to compare the gradient history values against a bound (for workloads trained by optimizers that use such history values), and also compare the *mvar*'s against a bound (for workloads with normalization layers). If any of these values is out of bound, an error message is generated. Since the necessary conditions occur within 2 training iterations after a failure occurs, the error detection latency of our technique is bounded. Further, we proved that these bounds can be mathematically derived based on the properties of a given DNN workload. As shown in Algorithm 1, the absolute gradient history values in the absence of hardware failures (and software bugs) are less than $20 \times \sqrt{n_l/m^2}$ with a probability larger than $(1 - 3 \times 10^{-89})$, where n_l is the number of partial sums used to generate one gradient value, and m is the batch size. Similarly, we derived a bound for the *mvar*'s in Algorithm 1.

Note that, a hardware failure detected by our technique does not always lead to unexpected training outcomes. However, it is still beneficial to decommission the accelerator for further analysis because it is highly likely (based on the probability shown in Algorithm 1) that the accelerator has encountered a hardware failure.

Algorithm 1: Derive bounds for gradient history values in Adam and moving variance values in BatchNorm. The bounds apply for various DNN layers including convolution, matrix multiplication, and fully-connected layers.

Without loss of generality, assume the following DNN properties [23, 34]:
(1)The mean of the outputs (before activation) and inputs of every DNN layer is 0, and the variance of all layers are approximately the same.
(2)The input data-set is normalized to zero mean and unit variance.
(3)Softmax-cross-entropy is used as the loss function, and Adam is used as the optimizer.
(4)The weight gradient values follow the Gaussian distribution.

I. Deriving the bound for absolute gradient history values in Adam.

Step 1: Let a_i and p_i := the i^{th} inputs and outputs of the softmax layer; y_i := the i^{th} one-hot encoded training target, $1 \leq i \leq I$; m := the number of mini-batches; L := Softmax-cross-entropy = $-\sum_i y_i log(p_i)/m$. We bound $\frac{\partial L}{\partial a_i}$, the input gradients of the last DNN layer $\forall i$.

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{I} e^{a_k}}, \qquad \frac{\partial L}{\partial a_i} = (p_i - y_i)/m.$$

$$\because \; p_i \in [0,1], \;\; y_i \in [0,1], \;\; \therefore \; \frac{\partial L}{\partial a_i} \in [-\frac{1}{m}, \frac{1}{m}].$$

Step 2: Let y_i^l := the i^{th} element of the output tensor of layer l. We bound $\frac{\partial L}{\partial y_i^l}$ $\forall l$. Given Property 1, $\frac{\partial L}{\partial a_i} \in [-\frac{1}{m}, \frac{1}{m}] \;\; \rightarrow$ $\frac{\partial L}{\partial y_i^l} \in [-\frac{1}{m}, \frac{1}{m}]$ $\forall l$, since a_i is the output of the last layer.

Step 3: Let w_i^l := the i^{th} element of the weight tensor of layer l; $\hat{x^l}$:= the transpose of layer l's input tensor; n_l := the number of the partial sums used to compute one gradient. We bound $\frac{\partial L}{\partial w_i^l}, \forall l$.

$$\frac{\partial L}{\partial w^l} = \hat{x^l}\frac{\partial L}{\partial y^l} \rightarrow Var[\frac{\partial L}{\partial w^l}] = Var[\hat{x^l} \times \frac{\partial L}{\partial y^l}].$$

Given Property 1, $\because \; \frac{\partial L}{\partial y_i^l} \in [-\frac{1}{m}, \frac{1}{m}], \;\; \forall l \;\; \therefore$ In the worst case,

$$\frac{\partial L}{\partial y_i^l} = \begin{cases} -\frac{1}{m} & x \in \hat{x^l}, x < 0 \\ \frac{1}{m} & x \in \hat{x^l}, x \geq 0 \end{cases} \text{ and } Var[\frac{\partial L}{\partial w^l}] \leq \frac{n_l}{m^2} Var[x_l].$$

Step 4: Based on the history value computation in Adam, shown in Eq. 1, $\because \;\; \beta < 1, \therefore$ the bound for $\frac{\partial L}{\partial w_i^l}$ can also be used for m_t.

Given Properties 1, 2, and 4, $E[\frac{\partial L}{\partial w^l}] = 0$, and also given the bound for $Var[\frac{\partial L}{\partial w^l}]$ shown in Step 3, $m_t \sim \mathcal{N}(0, \frac{n_l}{m^2})$,

$$\therefore \;\; Prob(|m_t| > 20 \times \sqrt{\frac{n_l}{m^2}}) < 3 \times 10^{-89}.$$

II. Deriving the bound for absolute moving variance values in BatchNorm.

Step 1: Let η := learning rate; g_t := $\frac{\partial L}{\partial w^l}$ in iteration t; $w^{l'}$:= the weight values of layer l in iteration $t+1$; N_l := the number of partial sums used to compute one output neuron in layer l.

$\because$ Property 1, $E[w^l] = 0, Var[w^l] = \frac{1}{N_l}$. Since Adam is the optimizer, let $k := \sqrt{1-\beta_2^t}/(1-\beta_1^t)$, then we have $u_t \sim \mathcal{N}(0, \eta^2 k^2)$ based on Eq. 1.

Step 2: $\because$ u_t and w_l are independent, $\therefore$ $E[w^{l'}] = 0$, and $Var[w^{l'}] \leq \frac{1}{N_l} + \eta^2 k^2$. $\because$ $Var[y^l] = N_l * Var[w^{l'}] * Var[y^{l-1}]$,

$$\therefore \; \frac{Var[y^l]}{Var[y^{l-1}]} = N_l * Var[w^{l'}] \leq 1 + N_l \eta^2 k^2.$$

$\because$ Properties 1 and 2, $\therefore$ $Var[y^l] \leq (1 + N_l \eta^2 k^2)^l$.

Step 3: Let $mvar_{l,t}$:= the moving variance of BatchNorm at layer l and iteration t; β := the decay factor.

$mvar_{l,t} = \beta \times mvar_{l,t-1} + (1-\beta) \times Var[y^l]$. $\because$ $\beta < 1$, $\therefore$ the bound for $Var[y^l]$ can be used to bound $mvar_{l,t}$.

$$\therefore \; mvar_{l,t} \leq (1 + N_l \eta^2 k^2)^l.$$

5.2 Recovery

We developed a light-weight recovery technique that re-executes the two most recent iterations of a DNN training workload on all training devices, which is sufficient to mitigate all immediate, short-term, and latent unexpected outcomes when coupled with our detection technique. The following changes to a DNN training program are required to implement our re-execution technique: (1) subtracting the gradients obtained in the last iteration from the current weight values to obtain the weight values used in the previous iteration; (2) reloading the mini-batch data-set used for the previous iteration; and (3) recording the seeds used to initialize random variables (if they are used) in the previous iteration, and applying them during re-execution.

5.3 Implementation and Evaluation

We implemented the detection and re-execution techniques in Tensorflow for the same set of workloads presented in Table 2, which requires only 24 – 32 lines of code change to the different DNN programs. The memory overhead is negligible since our detection technique only requires two new variables to bound the gradient history and *mvar* values, and our re-execution technique only requires a few seeds to be stored (if seeds are used). We evaluated the techniques on Google Cloud TPUs, using the cloud TPU profiler [27] to obtain performance/power/memory overheads. For each workload, the bounds-checking and re-execution operations were both executed 10K times.

If no out-of-bound values are detected, the performance impact is 0.003% – 0.025% on average (geomean) across different DNN training workloads. If re-execution is invoked once, the average (geomean) performance impact is $0.04\% - 0.15\%$. Also, the profiler reported a similar utilization of TPU resources between the modified and original programs for each workload, indicating that the power/memory overheads of our techniques are negligible.

Compared to the checkpointing approach where a checkpoint is saved at the end of each training epoch [60, 86], the performance/energy costs of our recovery technique are up to 500× lower (depending upon the number of iterations per epoch, which is typically $\sim 1,000$ iterations) assuming that 8 training devices are used.

6 RELATED WORK

Resilience analysis on DNN workloads. There is one study that targets memory errors in DNN training workloads [93]. Memory errors in datacenters are not a critical concern because ECCs are commonly supported in both on-chip and off-chip memories. Moreover, errors in memory behave differently from those in logic. For example, all the latent unexpected outcomes revealed by our study are unique to hardware failures in logic.

Many conclusions and findings from previous work on the resilience of inference workloads [2, 3, 9, 13, 24, 35, 48, 54, 59, 67, 68, 71, 80, 81, 90] cannot be extended to training workloads due to the fundamental differences in their respective algorithms and resilience requirements, as summarized in Table 5.

Table 5: Resilience properties of inference vs. training.

Inference	Training (details in Sec. 4)
Normalization layers effectively mask hardware failures [48].	Normalization layers can exacerbate or reduce the impact of hardware failures. See Observation (3) in Sec. 4.2.6.
Failures that occur in early layers are more likely to generate visible anomalies [48, 49].	We observed this trend only for the failures that lead to the SlowDegrade outcome.
Hardware failures that occur in certain output feature maps or input data samples are more likely to generate visible anomalies [54].	We did not observe such correlations in training.
INFs/NaNs are not observed.	INFs/NaNs are a major class of unexpected DNN training outcomes.

Hardware Failure Mitigation Techniques. There exist a plethora of resilience techniques across various system design layers [16], spanning algorithm [31, 94], compiler/software [18, 22, 51–53, 65, 66, 74, 76], architecture [18, 22, 29, 36, 50, 53, 56, 70, 82, 89], and circuit [5]. Selectively protecting FFs using circuit-level solutions (e.g., FF hardening) is a potential resilience solution, and our results in Sec. 4.3.1 can guide which FFs to harden; however, it requires hardware modifications, which may not be possible or desirable. Existing compiler/software techniques and architecture techniques were mostly developed for CPUs or GPUs, and they rely on specific properties in CPU/GPU applications or architectures; therefore, they do not lend themselves to be used in DL accelerators.

The authors in [54] focused on inference workloads, and proposed selective duplication in the weight kernel level or the inference task level. However, it is not clear how one would apply the kernel-level technique to training, since weight values are not static during training. Regarding the task-level technique, it is also not clear how to determine the importance of each input sample because that depends on the model and various training dynamics. Without selective redundancy, detection through duplication (or other redundancy-based techniques) incur high overheads. To recover from a failure, the overhead will be even higher since additional operations need to be executed upon detection.

In the algorithm level, algorithm-based fault tolerance (ABFT) techniques have been developed for DNN inference workloads [31, 94]. We extended the idea in [94] to cover training workloads, implemented it in Tensorflow for Resnet [33], Efficientnet [85] and DenseNet [42], and obtained the performance/energy results for these workloads using Google Cloud TPUs. This ABFT technique requires non-trivial software modifications ($463 - 485$ lines of code change), and incurs large ($5\% - 7\%$) performance/energy costs even in the absence of hardware failures.

Another line of work proposed to bound the activation outputs to improve the resilience of inference workloads [13, 14, 37, 48, 68]. This approach is inadequate for training because it can only detect a small fraction (33.7% from our experiments) of all latent unexpected outcomes.

Gradient clipping techniques in DNN training. Gradient clipping techniques [11, 57, 92] were proposed to boost test accuracy or reduce training time, without any resilience considerations. These techniques cannot be used to mitigate all unexpected training outcomes caused by hardware failures, because, as shown for the SlowDegrade, SharpDegrade, and LowTestAccuracy cases in Fig. 4, hardware failures can perturb gradient history / *mvar* values without affecting gradient values. Moreover, the bounds from previous work were heuristically determined. In contrast, our bounds were derived mathematically based on DNN properties to yield high detection coverage for hardware failures that are likely to generate latent unexpected outcomes.

7 CONCLUSIONS

We present the first in-depth resilience study on hardware failures in DL training accelerator systems. This study reveals the fundamental understanding on how hardware failures propagate in DL training devices and interact with DNN training workloads. We also present efficient and light-weight solutions to mitigate these failures.

Our work serves as a solid foundation for future work, which is essential because the impact of hardware failures is expected to increase as DL systems continue to scale and the complexity of DNNs continues to grow. We plan to extend our work to a broader set of hardware failures, DNN training workloads, and DL training systems such as GPUs and CPUs.

ACKNOWLEDGMENTS

We thank Rich Bonderson of Google, Michael Maire of the University of Chicago, and all anonymous reviewers for their assistance and comments. This research was partially supported by a Google Research Scholar Award. Computing resources for our experiments were provided by the Google TPU Research Cloud.

REFERENCES

[1] 2023. Fault injection framework. https://github.com/YLab-UChicago/FIdelityTraining.git.

[2] Khalid Adam, Izzeldin I Mohd, and Younis Ibrahim. 2021. Analyzing the resilience of convolutional neural networks implemented on gpus: Alexnet as a case study. *International journal of electrical and computer engineering systems* 12, 2 (2021), 91–103.

[3] Khalid Adam, Izzeldin Ibrahim Mohamed, and Younis Ibrahim. 2021. A Selective Mitigation Technique of Soft Errors for DNN Models Used in Healthcare Applications: DenseNet201 Case Study. *IEEE Access* 9 (2021), 65803–65823. https://doi.org/10.1109/ACCESS.2021.3076716

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[5] David Blaauw, Sudherssen Kalaiselvan, Kevin Lai, Wei-Hsiang Ma, Sanjay Pant, Carlos Tokunaga, Shidhartha Das, and David Bull. 2008. Razor II: In situ error detection and correction for PVT and SER tolerance. In *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*. IEEE, 400–622.

[6] Ondřej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Aleš Tamchyna. 2014. Findings of the 2014 Workshop on Statistical Machine Translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Baltimore, Maryland, USA, 12–58. https://doi.org/10.3115/v1/W14-3302

[7] Rich Bonderson. 2021. Training in Turmoil: Silent Data Corruption in Systems at Scale. International Test Conference Silicon Lifecycle Management Workshop. https://marcello.altervista.org/SLM.tttc-events.org/program.html#Keynote1

[8] Andrew Brock, Soham De, Samuel L. Smith, and Karen Simonyan. 2021. High-Performance Large-Scale Image Recognition Without Normalization. *CoRR* abs/2102.06171 (2021). arXiv:2102.06171 https://arxiv.org/abs/2102.06171

[9] N. Chandramoorthy et al. 2019. Resilient Low Voltage Accelerators for High Energy Efficiency. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 147–158.

[10] Athanasios Chatzidimitriou, Pablo Bodmann, George Papadimitriou, Dimitris Gizopoulos, and Paolo Rech. 2019. Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 26–38. https://doi.org/10.1109/DSN.2019.00018

[11] Xiangyi Chen, Steven Z Wu, and Mingyi Hong. 2020. Understanding gradient clipping in private SGD: A geometric perspective. *Advances in Neural Information Processing Systems* 33 (2020), 13773–13782.

[12] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. 2020. A survey of accelerator architectures for deep neural networks. *Engineering* 6, 3 (2020), 264–274.

[13] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. 2020. A Low-cost Fault Corrector for Deep Neural Networks through Range Restriction. https://doi.org/10.48550/ARXIV.2003.13874

[14] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. 2020. Ranger: Boosting Error Resilience of Deep Neural Networks through Range Restriction. *ArXiv* abs/2003.13874 (2020).

[15] Eric Cheng. 2018. *Cross-layer resilience to tolerate hardware errors in digital systems*. Ph. D. Dissertation. Stanford University.

[16] Eric Cheng, Shahrzad Mirkhani, Lukasz G. Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R. Stan, Klas Lilja, Jacob A. Abraham, Pradip Bose, and Subhasish Mitra. 2016. CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 68, 6 pages. https://doi.org/10.1145/2897937.2897996

[17] H. Cho et al. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference*. 1–10.

[18] J. Cong and K. Gururaj. 2011. Assuring application-level correctness against soft errors. In *Proceedings of the International Conference on Computer-Aided Design*. 150–157.

[19] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild. https://doi.org/10.48550/ARXIV.2203.08989

[20] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245* (2021).

[21] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. [n. d.]. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[22] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. 2010. Shoestring: probabilistic soft error reliability on the cheap. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 385–396.

[23] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 9)*, Yee Whye Teh and Mike Titterington (Eds.). PMLR, Chia Laguna Resort, Sardinia, Italy, 249–256. https://proceedings.mlr.press/v9/glorot10a.html

[24] Brunno F. Goldstein, Victor C. Ferreira, Sudarshan Srinivasan, Dipankar Das, Alexandre S. Nery, Sandip Kundu, and Felipe M. G. França. 2021. A Lightweight Error-Resiliency Mechanism for Deep Neural Networks. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. 311–316. https://doi.org/10.1109/ISQED51717.2021.9424287

[25] Google. 2019. Tensorflow. https://www.tensorflow.org.

[26] Google. 2021. Cloud TPU. https://cloud.google.com/tpu.

[27] Google. 2021. Profile your model with Cloud TPU tools. https://cloud.google.com/tpu/docs/cloud-tpu-tools.

[28] Prabhat K Gupta. 2016. Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, Vol. 2017. 20.

[29] S. K. S. Hari, S. V. Adve, and H. Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the International Conference on Dependable Systems and Networks*. 1–12.

[30] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 123–134.

[31] Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. 2022. Making Convolutions Resilient Via Algorithm-Based Error Detection Techniques. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2022), 2546–2558. https://doi.org/10.1109/TDSC.2021.3063083

[32] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629. https://doi.org/10.1109/HPCA.2018.00059

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR* abs/1502.01852 (2015). arXiv:1502.01852 http://arxiv.org/abs/1502.01852

[35] Y. He, P. Balaprakash, and Y. Li. 2020. FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 270–281. https://doi.org/10.1109/MICRO50266.2020.00033

[36] Yi He and Yanjing Li. 2019. Time-Slicing Soft Error Resilience in Microprocessors for Reliable and Energy-Efficient Execution. In *2019 IEEE International Test Conference (ITC)*. 1–10. https://doi.org/10.1109/ITC44170.2019.9000180

[37] Le Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. 2019. FT-ClipAct: Resilience Analysis of Deep Neural Networks and Improving their Fault Tolerance using Clipped Activation. *CoRR* abs/1912.00941 (2019). arXiv:1912.00941 http://arxiv.org/abs/1912.00941

[38] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[39] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. 2021. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 9–16.

[40] Ted Hong, Yanjing Li, Sung-Boem Park, Diana Mui, David Lin, Ziyad Abdel Kaleq, Nagib Hakim, Helia Naeimi, Donald S Gardner, and Subhasish Mitra. 2010. QED: Quick error detection tests for effective post-silicon validation. In *2010 IEEE International Test Conference*. IEEE, 1–10.

[41] Yunxiang Hu, Yuhao Liu, and Zhuovuan Liu. 2022. A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC. In *2022 14th International Conference on Computer Research and Development (ICCRD)*. 100–107. https://doi.org/10.1109/ICCRD54409.2022.9730377

[42] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. Densely Connected Convolutional Networks. arXiv:1608.06993 [cs.CV]

[43] Tri Huynh, Michael Maire, and Matthew R. Walter. 2019. Multigrid Neural Memory. *CoRR* abs/1906.05948 (2019). arXiv:1906.05948 http://arxiv.org/abs/1906.05948

[44] Younis Ibrahim, Haibin Wang, Junyang Liu, Jinghe Wei, Li Chen, Paolo Rech, Khalid Adam, and Gang Guo. 2020. Soft errors in DNN accelerators: A comprehensive review. *Microelectronics Reliability* 115 (2020), 113969.

[45] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. 2021. Ten lessons from three generations shaped Google's TPUv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[46] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez. 2017. MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 241–254. https://doi.org/10.1145/3079856.3080225

[47] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[48] Guanpeng Li et al. 2017. 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 (SC '17)*. 8:1–8:12.

[49] G. Li et al. 2018. TensorFI: A Configurable Fault Injector for TensorFlow Applications. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 313–320.

[50] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. 2008. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems - ASPLOS XIII*, Vol. 42. 265.

[51] D. Lin, T. Hong, Y. Li, E. S, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra. 2014. Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection. *IEEE Trans. Comput. Des. Integr. Circuits Syst.* 33, 10 (2014), 1573–1590.

[52] M. N. Lovellette, K. S. Wood, D. L. Wood, J. H. Beall, P. P. Shirvani, N. Oh, and E. J. McCluskey. 2002. Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed. In *IEEE Aerospace Conference Proceedings*, Vol. 5. 2109–2119.

[53] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. 2018. Optimizing Software-Directed Instruction Replication for GPU Error Detection. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 842–854. https://doi.org/10.1109/SC.2018.00070

[54] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W. Fletcher, Sarita V. Adve, Charbel Sakr, Naresh Shanbhag, Pavlo Molchanov, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. 2021. Optimizing Selective Protection for CNN Resilience. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 127–138. https://doi.org/10.1109/ISSRE52982.2021.00025

[55] J Markoff. 2022. Tiny Chips, Big Headaches. https://arxiv.org/abs/2203.08989

[56] A. Meixner, M. E. Bauer, and D. Sorin. 2007. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 210–222.

[57] Aditya Krishna Menon, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar. 2019. Can gradient clipping mitigate label noise?. In *International Conference on Learning Representations*.

[58] S. Mirkhani, S. Mitra, C. Cher, and J. Abraham. 2015. Efficient soft error vulnerability estimation of complex designs. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 103–108. https://doi.org/10.7873/DATE.2015.0367

[59] Sparsh Mittal. 2020. A survey on modeling and improving reliability of DNN algorithms and accelerators. *Journal of Systems Architecture* 104 (2020), 101689. https://doi.org/10.1016/j.sysarc.2019.101689

[60] MLCommons. 2021. v1.0 Results. https://mlcommons.org/en/training-normal-10/.

[61] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. 2013. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 77–88. https://doi.org/10.1109/FDTC.2013.9

[62] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. 2020. Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems. *CoRR* abs/2003.09518 (2020). arXiv:2003.09518 https://arxiv.org/abs/2003.09518

[63] Nvidia. 2021. Nvidia Ampere Architecture. https://www.nvidia.com/en-us/data-center/ampere-architecture.

[64] NVIDIA Corporation. 2018. NVDLA Open Source Project. http://nvdla.org/primer.html.

[65] N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Trans. Reliab.* 51, 1 (2002), 111–122.

[66] N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Error Detection By Duplicated Instructions in Spuper-Scalar Processor. *IEEE Trans. Reliab.* 51, 1 (2002), 63–75.

[67] Elbruz Ozen and Alex Orailoglu. 2020. Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3250–3262. https://doi.org/10.1109/TCAD.2020.3012209

[68] Elbruz Ozen and Alex Orailoglu. 2020. Just Say Zero: Containing Critical Bit-Error Propagation in Deep Neural Networks With Anomalous Feature Suppression. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[69] George Papadimitriou and Dimitris Gizopoulos. 2021. Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 902–915. https://doi.org/10.1109/ISCA52012.2021.00075

[70] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. 2007. Perturbation-based Fault Screening. In *IEEE 13th International Symposium on High Performance Computer Architecture*. 169–180.

[71] B. Reagen et al. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 267–278. https://doi.org/10.1109/ISCA.2016.32

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

[73] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. arXiv:1804.02767 [cs.CV]

[74] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. 2004. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the international symposium on Code generation and optimization*. 1–12.

[75] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2021. AI Accelerator Survey and Trends. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. https://doi.org/10.1109/HPEC49654.2021.9622867

[76] S. K. Sahoo, M. L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou. 2008. Using likely program invariants to detect hardware errors. In *Proceedings of the International Conference on Dependable Systems and Networks*. 70–79.

[77] Sriram Sankar, Rama Govindaraju, Arjan Van De Ven, Steven Hesley, and Subhasish Mitra. 2021. Panel: Hardware Operation at Scale Reliability to Address Silent Data Corruptions.

[78] S. K. Sastry Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. 2013. Relyzer: Application Resiliency Analyzer for Transient Faults. *IEEE Micro* 33, 3 (May 2013), 58–66. https://doi.org/10.1109/MM.2013.30

[79] Robin M Schmidt, Frank Schneider, and Philipp Hennig. 2021. Descending through a crowded valley-benchmarking deep learning optimizers. In *International Conference on Machine Learning*. PMLR, 9367–9376.

[80] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. 2018. Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 979–984. https://doi.org/10.23919/DATE.2018.8342151

[81] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. 2019. An Efficient Bit-Flip Resilience Optimization Method for Deep Neural Networks. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1507–1512. https://doi.org/10.23919/DATE.2019.8714885

[82] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel. 2013. Exploiting program-level masking and error propagation for constrained reliability optimization. In *Proceedings of the 50th Annual Design Automation Conference on – DAC13*.

[83] Eshan Singh, Clark Barrett, and Subhasish Mitra. 2017. E-QED: electrical bug localization during post-silicon validation enabled by quick error detection and formal methods. In *International Conference on Computer Aided Verification*. Springer, 104–125.

[84] Christian Szegedy et al. 2015. Going Deeper With Convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[85] Mingxing Tan and Quoc V. Le. 2020. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv:1905.11946 [cs.LG]

[86] Tensorflow. 2021. Training checkpoints. https://www.tensorflow.org/guide/checkpoint.

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[88] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V. Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. https://doi.org/10.1109/MICRO.2016.7783745

[89] N. J. Wang and S. J. Patel. 2006. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Trans. Dependable Secur. Comput.* 3, 3 (2006), 188–201.

[90] P. N. Whatmough et al. 2017. 14.3 A 28nm SoC with a 1.2GHz 568nJ/prediction sparse deep-neural-network engine with >0.1 timing error rate tolerance for IoT applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 242–243.

[91] Wikipedia. 2022. Backpropagation. https://en.wikipedia.org/wiki/Backpropagation.

[92] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. 2019. Why gradient clipping accelerates training: A theoretical justification for adaptivity. https://doi.org/10.48550/ARXIV.1905.11881

[93] Zhao Zhang, Lei Huang, Ruizhu Huang, Weijia Xu, and Daniel S. Katz. 2019. Quantifying the Impact of Memory Errors in Deep Learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–12. https://doi.org/10.1109/CLUSTER.2019.8890989

[94] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. 2021. Algorithm-Based Fault Tolerance for Convolutional Neural Networks. *IEEE Transactions on Parallel and Distributed Systems* (2021), 1–1. https://doi.org/10.1109/tpds.2020.3043449

[95] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. 2010. Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta (Eds.), Vol. 23. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2010/file/abea47ba24142ed16b7d8fbf2c740e0d-Paper.pdf

A ARTIFACT APPENDIX

A.1 Abstract

We provide our fault injection framework used in our study and the evaluation of our technique. The methodology to inject faults and apply our technique to the DNN training program are similar for all workloads. We will open-source the complete fault injection framework for all DNN workloads.

In each fault injection experiment, we pick a random training epoch, a random training step, a random layer (selected from both layers in the forward pass and the backward pass), and a random software fault model, and continue training the workload to observe the outcome. In order to inject faults to the backward pass and also correctly propagate the error effects, we manually implemented the backward pass for each DNN workload, which can be found in the `fault_injection/models` folder.

We have performed 2.9M fault injection experiments to obtain statistical results. In this artifact evaluation, we provide three reproducible examples of fault injections that correspond to three outcomes (Masked, Immediate INFs/NaNs, and SlowDegrade) reported in our paper. We also provide instructions for running more fault injection experiments.

Our technique's evaluation includes both detection and recovery. To measure detection performance, we perform detection operations 10,000 times in each training step and calculate the geomean of the overhead. For recovery performance, we re-execute the two most recent training iterations once for every 10 training iterations and calculate the geomean of the overhead.

A.2 Artifact check-list (meta-information)

- **Model: Resnet18.**
- **Data set: Cifar10.**
- **Run-time environment: Google Cloud TPU VM.**
- **Hardware: Google Cloud TPU.**
- **Output: Training / test accuracy.**
- **Experiments: Experiments performed by our fault injection framework.**
- **How much disk space required (approximately)?: 5GB.**
- **How much time is needed to prepare workflow (approximately)?: 10 minutes.**
- **How much time is needed to complete experiments (approximately)?: 20-30 minutes for all three examples of fault injection. 20-30 minutes for the evaluation of our technique.**
- **Publicly available?: Yes.**
- **Workflow framework used?: Tensorflow.**
- **Archived?: Yes.**
- **DOI: `https://doi.org/10.5281/zenodo.7952090`, and `https://doi.org/10.5281/zenodo.7952098`.**

A.3 Description

A.3.1 How to access. Our artifact can be accessed through this link: `https://doi.org/10.5281/zenodo.7952090`.

A.3.2 Hardware dependencies. Our experiments are run on Google Cloud TPUs (TPU versions v2-8 and v3-8).

A.3.3 Software dependencies. We require the following software tools:

- `Tensorflow 2.6.0`
- `Numpy 1.19.5`
- `Gdown 4.6.4`

A.3.4 Data sets. We use the Cifar10 dataset in this AE.

A.3.5 Models. We use Resnet18 in this AE.

A.4 Installation

The step for creating cloud TPU VMs and download the checkpoints.

(1) Step 1. Create Google Cloud TPU VM.

```
export PROJECT_ID=${PROJECT_ID}
gcloud alpha compute tpus tpu-vm create
    ${TPU_NAME} --zone=${TPU_LOCATION}
    --accelerator-type=${TPU_TYPE}
    --version=v2-alpha
```

`PROJECT_ID`: The Google cloud user ID.
`TPU_NAME`: A user-defined name.
`TPU_LOCATION`: The cloud region, e.g., `us-central1-a`.
`TPU_TYPE`: The type of the cloud TPU, e.g., `v2-8`.

(2) Step 2. SSH to the TPU VM.

```
gcloud alpha compute tpus tpu-vm ssh
    ${TPU_NAME} --zone=${TPU_LOCATION}
    --project ${PROJECT_ID}
```

(3) Step 3. Check Numpy and Tensorflow versions.

```
import numpy
numpy.__version__
import tensorflow
tensorflow.__version__
```

Make sure that the version of numpy is 1.19.5, and the version of tensorflow is 2.6.0. If the versions don't match, please install the correct versions.

(4) Step 4. Download files from this link: `https://doi.org/10.5281/zenodo.7952090`.

(5) Step 5. Download checkpoints from Google Drive.

```
cd fault_injection
pip install gdown
gdown --folder
    https://drive.google.com/drive/folders/
    1HVRFWY7NI5xr5qzR8yNeSKCRVnJNnqFf
    ?usp=sharing
```

The commands above download checkpoints to folder `fault_injection/ISCA_AE_CKPT`. Please keep this folder name. If gdown cannot be found, specify the full path where gdown is installed, mostly likely in `~/.local/bin`.

A.5 Experiment workflow

A.5.1 Fault injection. The `reproduce_injections.py` file is the top-level program to perform the entire workflow of a fault injection experiment, which takes in one argument `--file`, which specifies the injection configs, e.g., the target training epoch, target training step, target layer, faulty values, etc. The configs of our three examples are provided in the `injections` folder.

For each injection, the program generates an output file named `replay_inj_TARGET_INJECTION.txt` file under the `fault_injection` directory, which records the training loss, training accuracy for each training iteration, and test loss and test accuracy for each epoch. For Example 1, the file will also record when INF/NaN values are observed.

To execute each example, run:

i Example 1 (takes approximately 5 minutes).

```
cd fault_injection
python3 reproduce_injections.py --file
    injections/inj_immediate_infs_nans.csv
```

ii Example 2 (takes approximately 10-15 minutes).

```
cd fault_injection
python3 reproduce_injections.py --file
    injections/inj_masked.csv
```

iii Example 3 (takes approximately 10-15 minutes).

```
cd fault_injection
python3 reproduce_injections.py --file
    injections/inj_slow_degrade.csv
```

A.5.2 Evaluation of our technique. To evaluate the overhead of detection (takes approximately 15 minutes): we first execute the `detection.py` script without the `--check` flag. This trains the workload without detection. Once this step is complete, we execute the `detection.py` script with the `--check` flag, enabling detection and executing detection operations 10,000 times for every training iteration. These two commands generate two files: `train_recorder_no_check.txt` and `train_recorder_check.txt`, which record the total elapsed time of the training process without and with detection, respectively. Finally, we execute the `calc_overhead.py` script to obtain the overhead.

```
cd technique/detection
python3 detection.py
python3 detection.py --check
python3 calc_overhead.py
```

We follow a similar methodology for recovery (takes approximately 15 minutes), and execute the `replay.py` script without and with the `--rerun` flag. These two commands generate two files: `train_recorder_no_rerun.txt` and `train_recorder_rerun.txt`, which record the total elapsed time of the training process without and with re-execution, respectively. Finally, we execute the `calc_overhead.py` script to obtain the overhead.

```
cd technique/replay
python3 replay.py
python3 replay.py --rerun
python3 calc_overhead.py
```

A.6 Evaluation and expected results

A.6.1 Fault injection. Once each experiment is done, the output file should contain the following information:

i Example 1. NaN values are reported in the same training iteration (epoch 19, iteration 38) where the fault is injected.
A sample expected output file for this example can be found in `fault_injection/expected_results/replay_inj_immediate_infs_nans.txt`.

ii Example 2. After the fault is injected, both the training and test accuracy keeps improving as the training process continues. The final training accuracy is within 2% compared with the fault-free run.
A sample expected output file for this example can be found in `fault_injection/expected_results/replay_inj_masked.txt`.
A sample fault-free output file can be found in `fault_injection/expected_results/fault_free.txt`.

iii Example 3. After the fault is injected, the training accuracy degrades, then stays at a low level through. The final accuracy is significantly lower than that of the error-free runs.
A sample expected output file for this example can be found in `fault_injection/expected_results/replay_inj_slow_degrade.txt`.

A.6.2 Evaluation of our technique. The command `python calc_overhead.py` prints out the overheads for detection and recovery. Please note that due to the varying throughputs of different versions of TPUs, the overhead may vary slightly. However, as reported in our paper, we expect the overhead for detection to be less than 0.025% and the overhead for recovery to be less than 0.15%. For reference, we provide our `train_recorder_xx.txt` files under the `detection/expected_result` and `replay/expected_result` folders. These files report a detection overhead of 0.016%, and a recovery overhead of 0.12%.

A.7 Experiment customization

To run other examples, one can modify the three example injection files under the `injection` folder and specify different training epochs, training steps, target layers, and faulty values. Checkpoints that belong to other epochs can be downloaded through:

```
gdown --folder
    https://drive.google.com/drive/folders/
    1B4ptjedCX4e1PbzZWVe5Ydfe48BvSwzt?
    usp=sharing
```

The evaluation process is similar to the examples provided.

To run other workloads, download additional files from `https://doi.org/10.5281/zenodo.7952098` and follow the README. The entire workflow and evaluation procedure are the same as the examples we provided.