FT-DeepNets: Fault-Tolerant Convolutional Neural Networks with Kernel-based Duplication

Iljoo Baek
Carnegie Mellon University
ibaek@andrew.cmu.edu

Wei Chen
Purdue University
chen2732@purdue.edu

Zhihao Zhu
Carnegie Mellon University
704242527zzh@gmail.com

Soheil Samii
Motional / Linköping University
soheil.samii@motional.com / soheil.samii@liu.we

Ragunathan (Raj) Rajkumar
Carnegie Mellon University
rajkumar@andrew.cmu.edu

Abstract

Deep neural network (deepnet) applications play a crucial role in safety-critical systems such as autonomous vehicles (AVs). An AV must drive safely towards its destination, avoiding obstacles, and respond quickly when the vehicle must stop. Any transient errors in software calculations or hardware memory in these deepnet applications can potentially lead to dramatically incorrect results. Therefore, assessing and mitigating any transient errors and providing robust results are important for safety-critical systems. Previous research on this subject focused on detecting errors and then recovering from the errors by re-running the network. Other approaches were based on the extent of full network duplication such as the ensemble learning-based approach to boost system fault-tolerance by leveraging each model's advantages. However, it is hard to detect errors in a deep neural network, and the computational overhead of full redundancy can be substantial.

We first study the impact of the error types and locations in deepnets. We next focus on selecting which part should be duplicated using multiple ranking methods to measure the order of importance among neurons. We find that the duplication overhead for computation and memory is a trade-off between algorithmic performance and robustness. To achieve higher robustness with less system overhead, we present two error protection mechanisms that only duplicate parts of the network from critical neurons. Finally, we substantiate the practical feasibility of our approach and evaluate the improvement in the accuracy of a deepnet in the presence of errors. We demonstrate these results using a case study with real-world applications on an Nvidia GeForce RTX 2070Ti GPU and an Nvidia Xavier embedded platform used by automotive OEMs.

1. Introduction

Safety-critical systems require high robustness against systematic or modular faults caused by either hardware or software. Sometimes, even errors within only a small portion of the software system can lead to drastically different results.

1.1. Related Work

There have been numerous studies in the recent past to assess the resilience of hardware accelerator-enabled systems and applications using fault injection [17, 22, 32, 36]. They injected errors at the instruction-level by selecting registers and flipping random bits to study the soft error resilience of applications running on hardware accelerators. Similarly, [7, 24, 34] presented in-depth studies of low-level hardware failures on large-scale GPU-enabled systems and analyzed the behavior of applications in the presence of hardware errors. Other approaches inject errors and faults in the software and hardware states, such as environmental conditions and faults affecting sensor data on open-source or proprietary AV software stacks [20, 21, 31]. These previous studies provide significant insights into the impact of low-level software and hardware faults. However, they do not consider the impact of transient memory and computational errors in deepnet applications. In this work, we focus on transient memory-related errors that can affect the accuracy of deepnet applications. We also study the impact of the error locations in deepnet components.

Software-based approaches for mitigating deepnet errors can be categorized into two kinds. In the first category, the approach [5] detects the existence of an error, tries to locate the error and replicates the computation from the point of the error. Whenever the primary deepnet fails, a replica retrieves the checkpoints either from the frame-level or the layer-level, and continues processing from that checkpoint one step prior to the detected error location. This method, however, focuses on reducing the recovery time from a permanent failure and does not protect output accuracy in the presence of transient errors. Compared with the first category, the second approach skips the error detection step, and leverages the output from multiple deepnets to increase robustness. This approch trains multiple different models, where, in the best case, the trained models together produce a collective decision to make the system less fault-sensitive [16] [10] [33] [37]. This simple approach is, however, inefficient in terms of memory, since the whole model needs to be duplicated. Instead, one can duplicate only certain parts of the network and reduce the impact of the errors, which means that duplication of much more fine-grained modules inside the neural network is required. In particular, we highlight D2NN [27] which duplicates at the neuron level in order to generalize dual modular redundancy for deepnets to tackle hardware vulnerabilities and security threats at the algorithmic level. However, there is a core limitation in this baseline approach: the criterion for selecting which neurons to be duplicated is determined only by the magnitude of the neuron weights, which does not produce correct outputs in many cases. One such situation is where larger neuron weights are more sensitive to noise and thus end up increasing the model's vulnerability. Moreover, this approach requires complex changes to the network in order to achieve fine-grained neuron-level duplication and to share common neurons. We will explicitly compare the performance of our approach with this method in Section 4.4.

1.2. Our Contributions

In this paper, we first investigate the impact of the error type and its location in deepnets. We next focus on selecting the important neurons which should be duplicated using multiple ranking methods. Then, we discuss two different error-protection mechanisms that duplicate selected parts of the network using critical neurons to achieve higher robustness with less system overhead. Finally, we substantiate the practical feasibility of our approach and evaluate the performance of our solutions.

The main contributions of this paper are as follows:

1. We present an application-level fault-tolerant mechanism for deepnet applications to overcome transient errors and protect output accuracy by duplication.

2. We provide practical insights into the vulnerability of the weight and feature maps of deepnet applications and the impact of error types in memory.

3. We present and compare various methods to rank the importance of deepnet neurons.

4. We demonstrate the straightforward integration of our solution into the widely used PyTorch framework and deepnet applications.

5. We evaluate the usefulness of the recovery mechanisms with real-world perception applications: SSD-VGG / SSD-MobilNetV1 on the Nvidia Xavier board popular in the automotive industry.

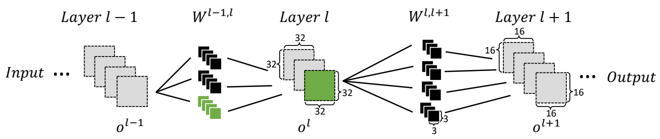


Figure 1: The structure of a general convolutional neural network. In the plot, $o^l \in \mathbb{R}^{C^l \times h_f^l \times w_f^l}$ is the feature map of layer l, $o^{l+1} \in \mathbb{R}^{C^{l+1} \times h_f^{l+1} \times w_f^{l+1}}$ is the feature map of layer $l+1$, and $W^{l,(l+1)} \in \mathbb{R}^{C^l \times C^{l+1} \times h \times w}$ is the kernel mapping o^l to o^{l+1}. Specifically, $C^l(=3)$ and $C^{l+1}(=4)$ are the number of channels in layer l and layer $l+1$, $h_f^l \times w_f^l (= 32 \times 32)$ and $h_f^{l+1} \times w_f^{l+1} (= 16 \times 16)$ are the feature map size of layer l and layer $l+1$, and $h \times w (= 3 \times 3)$ is the size of a convolutional kernel.

2. Background

2.1. Convolutional Neural Network Architecture

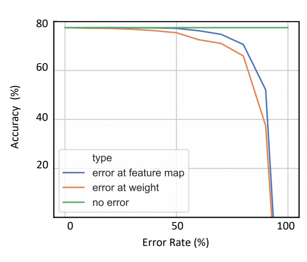
A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from another. As shown in Figure 1, a ConvNet consists of multiple kernels (= weights)[1], and each kernel connects its input layers and transforms the input to a desired output-layer shape. $W^{l,(l+1)} \in \mathbb{R}^{C^l \times C^{l+1}}$ are the weight-mapping neurons from layer l to layer $l+1$, C^l is the length of the input and C^{l+1} is the length of the output of the weight.

Each kernel can be represented by $C^l \times C^{l+1} \times h \times w$, where C_l and C_{l+1} are the lengths of the channels in layer l and layer $l+1$ respectively, and $h \times w$ are the dimensions of the kernel. The output from multiplying the kernel with the input data once creates a single value. As the kernel is applied multiple times to the entire input data, the result is a two-dimensional data of output values that represent a convolution of the input. We call the direct output of ConvNets the feature map o^l, the shape of which is determined by the kernel dimension, as well as its padding and stride size.

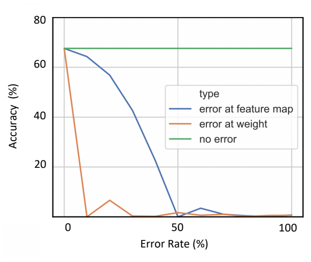
2.2. Error Type

We classify software and hardware errors into permanent and transient errors. Permanent errors are caused by device wear-out or environmental effects, and are present under specific conditions causing applications to permanent

[1]We will use the terms 'kernel' and 'weight' interchangeably because kernels consist of weights.



(a) SSD-VGG



(b) SSD-MobileNet

Figure 2: A comparison between kernel error and feature map error rates. The plots show the different impact of error location for (a) SSD-VGG and (b) SSD-MobileNet. Kernel errors cause more damage to accuracy than what the same feature map error rate does.

faults such as crashing or hanging [3, 34][2]. Transient errors can be random events caused by radiation particles affecting memory elements in the system [14, 15] or faults stemming from extremely low-voltage operation [8]. Error correction-code (ECC)-enabled memory can prevent some of the transient faults in DRAM [28]. Cyclic redundancy checks in the GDDR interface prevent faults from occurring during transfers across the memory bus [30]. However, these mechanisms incur extra cost, and not all devices employ them. They also do not cover all failure models, such as *transient faults* in computing or control logic [26]. A transient error can be classified as a zero error, a random error or a bit-flip error [11,35]. A zero error happens when the spot in memory is stuck and turns the value into all zeros. A random error means that the value turns into a random value. The most significant error is the bit-flip error; it reverses the value bit-wise and further leads it in the opposite direction. In this work, we only consider zero errors, which are more common [2, 12][3].

2.3. Error Location

The inference result of a neural network depends on the convolution between a kernel and its input feature map. Therefore, we consider the error in these two different locations: kernel error and feature map error.

Figure 2 shows the results from a kernel error and a feature map error using SSD-MobileNet and SSD-VGG. Both error locations affect and degrade the accuracy of the models as the error rate increases. However, we observe that the same percentage of the kernel error has a different impact on accuracy than the feature map error. Some deepnet models are more robust to the errors in the feature map than in the kernel. For example, SSD-MobileNet with a 10% feature map error shows about 62% accuracy, but a 10% kernel error makes the accuracy of SSD-MobileNet drop to 0%. Compared with SSD-MobileNet, SSD-VGG suffers less from a kernel error. For example, the accuracy of SSD-VGG remains the same (77%) for both 10% kernel and feature map errors. This phenomenon can be explained by the number of model parameters in the respective deepnets. SSD-VGG has a bigger feature map size and more parameters than the feature maps in SSD-MobileNet. Hence, there are built-in redundancies which make the feature maps in SSD-VGG more robust to errors.

[2]Recovering from permanent errors is beyond this paper's scope. The permanent faults constitute an orthogonal problem and need to be handled by other mechanisms such as full replication of HW/SW or the hot/cold standby approaches in [5,6].

[3]Our approarch can be extended to the other error types.

3. Our Proposed Approach and Algorithms

3.1. Ranking Neurons

Weight-Sum: Weight-sum is a straightforward method, which is commonly used in the model compression of neural networks. The weight-sum score indicates the importance of a neuron or convolutional kernel [25, 27]. It can also be used to select important kernels for duplication. The score s of the i^{th} neuron in layer l is calculated by the sum of its absolute weights:

$$s_i^l = \sum_{j=1}^{C^{l+1}} |W_{i,j}^{l,(l+1)}| \tag{1}$$

where $|\cdot|$ calculates the absolute value, $W^{l,(l+1)} \in \mathbb{R}^{C^l \times C^{l+1}}$ is the weight mapping neurons from layer l to layer $l+1$, C^l is the length of the input and C^{l+1} is the length of the output of the weight.

To calculate the i^{th} weight-sum score in convolutional layers, we consider the weight as a 4-dimensional (4D) tensor ($W^{l,(l+1)} \in \mathbb{R}^{C^l \times C^{l+1} \times h \times w}$), with C^l input channels, C^{l+1} output channels, where the height of each kernel is h, and the width of each kernel is w. It calculates the score s of the i^{th} kernel in layer l by the sum of its absolute weights:

$$s_i^l = \sum_{j=1}^{C^{l+1}} \sum_{p=1}^{h} \sum_{q=1}^{w} |W_{i,j,p,q}^{l,(l+1)}| \tag{2}$$

Figure 3(a) illustrates that the weight-sum score of a neuron is calculated based on the sum of the absolute values of its corresponding kernels.

Second-Order Derivative: The second-order derivative method is based on the Taylor expansion of the error (between the true label and the predicted output) with respect to weights or feature maps [18, 23]. As a larger score for the second-order derivative means more influence on the final loss, this approach also helps in selecting more important neurons or channels:

$$s_i^l = \frac{1}{B} \sum_{j=1}^{B} \frac{1}{2} \frac{(o_{i,j}^l)^2}{[\mathbf{H}^{-1}]_{i,i}^l} \tag{3}$$

where o^l is the output of layer l, and $\mathbf{H} = \frac{\partial^2 E}{\partial o^2}$ is the Hessian matrix of the loss E w.r.t. the output o. The score s_i^l of the i^{th} neuron in layer l needs to be averaged over a batch B of selected samples.

Usually the Hessian matrix and the inverse of a matrix are hard to calculate. To simplify the computation, we use the gradient to approximate the Hessian matrix [18]:

$$\mathbf{H} = \frac{\partial^2 E}{\partial o^2} \simeq (\frac{\partial E}{\partial o})^2 \tag{4}$$

so that Equation 3 becomes:

$$s_i^l = \frac{1}{B}\sum_{j=1}^{B}\frac{1}{2}(o_{i,j}^l)^2 \cdot (\frac{\partial E}{\partial o_{i,j}^l})^2 \tag{5}$$

To calculate the second-order derivative score in convolutional layers, we consider the 4D feature map ($o^l \in \mathbb{R}^{B\times C^l\times h_f^l\times w_f^l}$), with C^l channels, with B samples in a batch, feature map height h_f^l, and feature map width w_f^l, yielding

$$s_i^l = \frac{1}{B}\sum_{j=1}^{B}\sum_{p=1}^{h_f^l}\sum_{q=1}^{w_f^l}\frac{1}{2}(o_{i,j,p,q}^l)^2 \times (\frac{\partial E}{\partial o_{i,j,p,q}^l})^2 \tag{6}$$

Figure 3(b) illustrates that the second-order derivative score is calculated based on the feature map and the gradient of the loss w.r.t. this feature map.

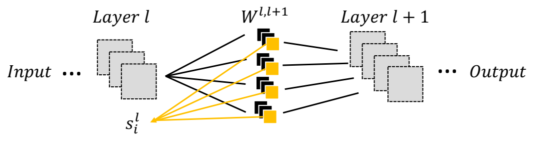
Entropy-Based Approach: Entropy is well-known in information science to measure the information in random variables [19]. It is also used in different areas in machine learning, such as model compression [29] or deepnet regularization [9]. We now propose an entropy-based approach for channel ranking.

Given a discrete random variable $x_i \in X$, and its corresponding probability $p(x_i)$, the entropy $\mathbb{H}$ of X is defined as:

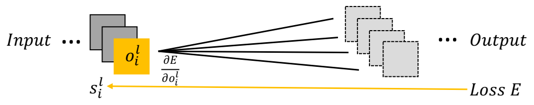
$$\mathbb{H}(X) = -\sum_{i=1}^{n} p(x_i)\cdot log(p(x_i)) \tag{7}$$

A higher value of entropy indicates more information, which is more robust to errors. Therefore, we aim to duplicate the channels with the lowest entropy, which are the most vulnerable to errors. To calculate entropy, we consider the 4D feature map $o^l \in \mathbb{R}^{B\times C^l\times h_f^l\times w_f^l}$, with B samples in a batch, feature map height h_f^l, and feature map width w_f^l. The i^{th} sample is represented as o_i^l. The j^{th} channel in the layer of the feature $o_{i,j}^l$ is flattened as an activation vector $a_{i,j}^l \in \mathbb{R}^{h_f^l\cdot w_f^l}$. Then, the entropy-based approach generates the probability distribution of the activation vector with softmax:

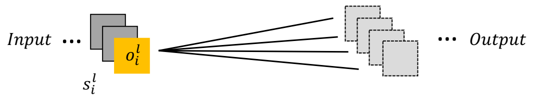
$$p_{i,j}^l = softmax(a_{i,j}^l) \tag{8}$$



(a) Weight-sum



(b) Second-order derivative



(c) Entropy-based approach

Figure 3: The illustration of our neuron-ranking methods. (a) weight-sum approach: the score of an input channel is the summation of its corresponding convolutional kernels; (b) second-order derivative approach: the score of a channel is correlated with both the feature map value o_i^l and its gradient $\frac{\partial E}{\partial o_i^l}$; (c) entropy approach: the score of a channel is correlated with the entropy of its flatten feature map value.

With the probability distribution of the activation vector, the score is given by:

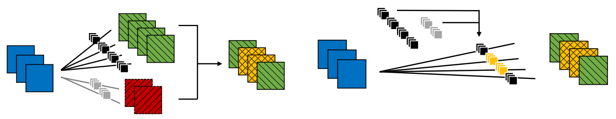
$$s_i^l = -\frac{1}{B}\sum_{j=1}^{B}\mathbb{H}(p_{i,j}^l) \tag{9}$$

where $\mathbb{H}(\cdot)$ is the entropy. A lower entropy for a neuron leads to higher importance, so we add the negative sign in front of the formula. Figure 3(c) illustrates that the entropy-based score of a neuron is calculated based only on its feature map.

3.2. Partial Duplication Approaches

We consider two duplication approaches to recover from any zero errors in a neural network. The first approach, called *kernel recovery* (KR), only recovers the weights when the model makes the inference. Another approach, called *feature map recovery* (FMR), recovers the feature maps during inference time. Both approaches duplicate the selected kernels at the beginning of the process. KR only averages the kernel values and uses the duplicated kernel to calculate the feature map during inference time, as shown in Figure 4(b). Since this approach focuses only on the kernel weights, the resulting accuracy can still be affected by errors in the newly generated feature map.

FMR calculates two streams of the feature map separately; one from the original kernels and the other from the duplicated kernels. The final output feature map is the av-



(a) Feature map recovery (b) Kernel recovery

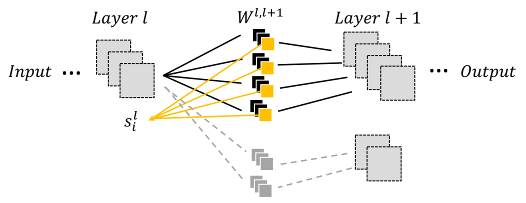
Figure 4: Illustration of the duplication approaches: (a) FMR duplicates kernels and calculates the feature map separately. The final output feature map is the average of both the original and the duplicated feature maps. (b) KR duplicates kernel only. It averages the original and duplicated kernels first and then generates a final feature map from the averaged kernel.

erage of both the original feature map and the duplicated feature map, as shown in Figure 4(a). This can reduce the impact of the errors in the feature maps. However, this approach requires more computational and memory overhead to calculate and store the feature map of duplicated kernels. We therefore mainly use the averaging approach to recover from kernel and feature map errors. However, our underlying observations and framework can also be used for other types of recovery with the duplication approach.

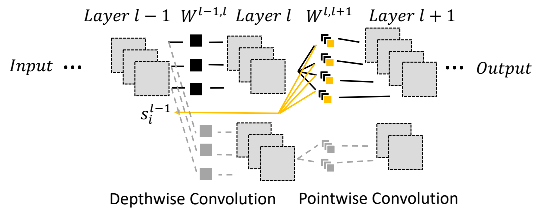
3.3. Implementation

To implement the neuron-ranking approach of Section 3.1, we first train the whole neural network without any duplication until it converges. Next, we select the target kernels to be duplicated according to the ranking scores and the replication percentage. However, the ranking methods in SSD-VGG and SSD-MobileNet have different implementation details, due to their deepnet structures. For example, Figure 5 illustrates the difference between SSD-VGG and SSD-MobileNet when the weight-sum ranking method is used. The implementation of other ranking methods are similar to the weight-sum ranking method. For SSD-VGG Figure 5(a), we directly get the score from the convolutional kernels. SSD-MobileNet contains a unique architecture, including depthwise convolutional kernels and pointwise convolutional kernels. Since the SSD-MobileNet decomposes the convolutional kernel into a depthwise convolutional kernel and a pointwise convolutional kernel, we treat the pair as a unit. Furthermore, we find that the pointwise convolutional kernel contains more information than the depthwise convolutional kernel. To preserve the correlation between the depthwise convolutional and pointwise convolutional kernels, we duplicate a part of the pointwise convolutional kernel and the entire depthwise convolutional kernel as shown in Figure 5(b). For example, in Figure 5, we duplicate the two kernels of $W^{l,l+1}$. Since layer l has three channels, we need to duplicate all the three kernels of $W^{l-1,l}$.

After selecting the target kernels, the recovery process of SSD-MobileNet is similar to SSD-VGG. As shown in



(a) SSD-VGG



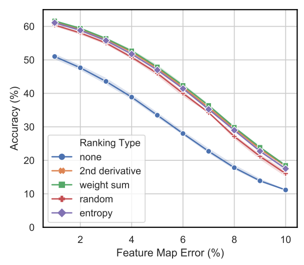
(b) SSD-MobileNet

Figure 5: Implementation of the neuron-ranking method for (a) SSD-VGG and (b) SSD-MobileNet: SSD-MobileNet decomposes a convolutional kernel into a depthwise convolutional kernel $W^{l-1,l}$ followed by a pointwise convolutional kernel $W^{l,l+1}$. In the weight-sum approach, we calculate the score of layer $l-1$ for SSD-MobileNet with the summation of only pointwise convolutional kernels.

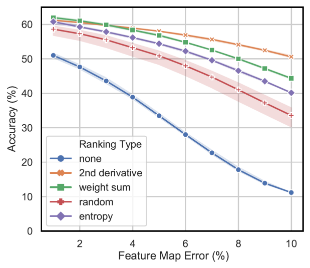
the executing inferencing stage, the duplicated kernels continually recover from the original deepnet error. By design, KR only recovers from the kernel errors. Specifically, in our work, we average the original and the duplicated kernels to mitigate the influence of errors. The averaged kernels are then used to generate a feature map. FMR focuses on protecting the feature maps. The original and the duplicated kernels first generate their own feature map independently; then we average the two sets of feature maps to mitigate the influence of errors. The averaged feature maps will be passed to the input of the next layer.

4. Evaluation

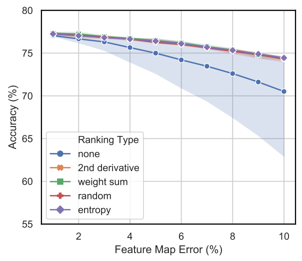
This section presents the evaluation of our error recovery mechanisms using different types of neuron-ranking methods as well as the choice of different recovery methods. We evaluate our approach with two deepnets: SSD-MobileNet and SSD-VGG, which are executed on an RTX 2070 Ti GPU. We further implement the experiments in an Nvidia Jetson Xavier to compare the overhead of our approach on different platforms. We use 64-bit Ubuntu 18.04, CUDA 10.0, and PyTorch in our experiments. The dataset we use in our experiments is PASCAL VOC2007 [13]. We first train SSD-VGG and SSD-MobileNet on the training data until convergence. SSD-VGG reaches 77.26% accuracy and SSD-MobileNet has 67.5% accuracy. Errors are next injected into both the kernel and feature maps of SSD-VGG or SSD-MobileNet. The error type is zero error; thus, when



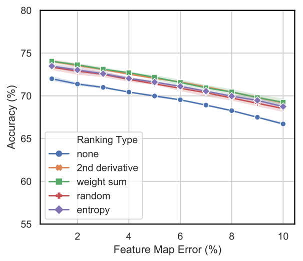
(a) SSD-MobileNet + KR duplication with 1% weight error



(b) SSD-MobileNet + FMR duplication with 1% weight error



(c) SSD-VGG + KR duplication with 1% weight error



(d) SSD-VGG + FMR duplication with 10% weight error

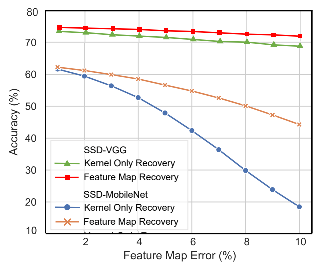
Figure 6: The accuracy of SSD-MobileNet and SSD-VGG using different neuron-ranking methods.

an error happens, the value in the kernel or feature map is stuck at 0. Since a transient error can happen randomly in different positions of the kernel and feature maps, we perform our experiments with three different random seeds to show a clear trend. Therefore, in Figure 6, the solid line and shade respectively represent the mean value and the standard deviation of the three experiments. In the evaluation, we present the results when using the weight-sum, second-order derivative and entropy-based approaches. We repeat all experiments three times with different random seeds for the error injections to generalize the results.

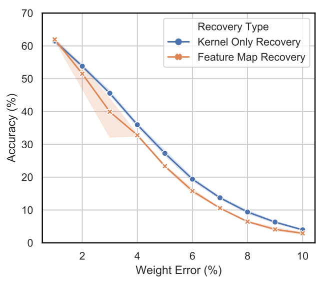
4.1. Overall Accuracy In The Presence of Errors

We first evaluate the overall accuracy using the KR approach w.r.t. the kernel weight and feature map errors. In the experiments, the errors are injected into all layers of the feature extraction layer of SSD-MobileNet (13 layers) and SSD-VGG (16 layers). As noted earlier, we use only zero errors in our experiment, and the errors are placed randomly in kernel and feature maps.

The duplication percentage is 50% uniformly distributed among all the layers, which means half of the feature extraction layers are duplicated. We now present the results of the weight-sum, second-order derivative and entropy-based approaches to select duplicated kernels. The duplicated kernels are selected from the original kernels with the highest score in descending order. We test two kernel weight error



(a) SSD-MobileNet / SSD-VGG with varied feature map error



(b) SSD-MobileNet with varied weight error

Figure 7: Comparison of kernel recovery and feature map recovery.

rates: 1% or 10%, and vary the feature map error rate from 1% to 10% to show how the different ranking types protect accuracy against different error rates. The results of SSD-MobileNet and SSD-VGG are shown in Figure 6. All our duplication methods provide 10% better accuracy compared with the original baseline without duplication. To show the clear benefit of the ranking methods, we also present the results with a random selection as a comparison. The random selection provides better accuracy than the baseline without duplication, but all of our ranking methods outperform the random method. This is more obvious in Figure 6(b) FMR duplication. Finally, we observe that SSD-MobileNet is more sensitive to the error rate. As shown in Figure 6(a), the 10% feature map error leads to nearly 10% accuracy of SSD-MobileNet, but SSD-VGG (Figure 6(c)) maintains over 74% accuracy with the same error rate. Figure 6(d) shows that SSD-VGG maintains over 69% accuracy with 10% weight error. The reason for this outcome is that SSD-VGG has many more parameters than SSD-MobileNet, with SSD-MobileNet designed for running on edge devices that are less powerful and easier to get affected by errors.

4.2. Kernel Recovery vs. Feature Map Recovery

Accuracy can be affected by the duplication methods, as discussed in Section 3.2. We now compare the performance between kernel recovery (KR) and feature map recovery (FMR). For KR, we duplicate the selected kernels, and during inference time, we recover the weights by averaging the original kernels and duplicated kernels. For FMR, we also duplicate the selected kernels. Instead of averaging the original kernels and the duplicated kernels, we use both kernels to generate the feature maps and average them.

Figure 7(a) shows that FMR yields better overall accuracy than KR w.r.t feature map errors. This is because FMR recovers the errors in both kernel weights and feature maps. As the feature-map error rate increases, FMR provides better accuracy than KR. However, FMR suffers from a bigger computational overhead to calculate the feature maps

SSD-VGG	Original	Kernel Recovery	Feature Map Recovery
x86	31.39 ms	32.02 ms	60.35 ms
Nvidia Xavier	106.09 ms	104.84 ms	181.01 ms
SSD-MobileNet	Original	Kernel Recovery	Feature Map Recovery
x86	7.53 ms	9.00 ms	13.54 ms
Nvidia Xavier	36.68 ms	37.03 ms	51.43 ms

Table 1: Computational Overhead

from duplicated kernels and to average them with the feature maps from the original kernels. Besides, the newly calculated feature maps require extra memory space. These overheads also increase the recovery time. We note that KR can provide similar accuracy as FMR when the feature-map error rate is expected to be lower with less overhead. Meanwhile, Figure 7(b) illustrates that KR provides overall better accuracy than FMR w.r.t. the weight error. This is because KR focuses on protecting the kernel weights by averaging the kernel values and recovering from weight errors. To maintain the best inference accuracy against severe errors, FMR with a higher duplication percentage is recommended.

4.3. Overhead Evaluation

We further evaluate the overhead of our method on an x86 system with a discrete GPU and an Nvidia Xavier embedded platform. The whole process can be separated into three different stages: training stage, ranking stage and executing stage. Once the model is deployed on the devices, only the executing stage accounts for the overhead. We analyze both computational and memory overheads. The KR approach requires memory space to keep the duplicated and averaged kernels. It also needs additional computational power to average the original and duplicated kernels.

The FMR approach needs more memory to store the duplicated kernels, newly calculated feature maps and averaged feature maps from the original and duplicated kernels. Therefore, the convolution operation from the duplicated kernels and averaging feature maps also impose more computational overhead.

The final overhead for both approaches is shown in Table 1. We calculate the mean and standard deviation of whole inference times for different methods: original, KR, and FMR. SSD-MobileNet needs less inference time than SSD-VGG because the latter has more parameters, thereby consuming more time to calculate convolutions. The inference times for KR and FMR of SSD-MobileNet show similar results on the x86 platform because of the faster CPUs. Meanwhile, FMR costs more time than KR on Xavier.

We further estimate the memory consumption of each method here. SSD-VGG has about 36.1 million parameters in total, including 14.7 million parameters in the feature extraction layers. Both KR and FMR duplicate 50% kernels from the original feature extraction layers, containing 7.4 million parameters. Assuming that each parameter is a 32-bit float, KR consumes 29.6MB more space in memory. FMR requires extra space for feature maps, 12.1 million (48.4MB) for SSD-VGG, so the overhead for FMR is 78MB. The SSD-MobileNet has about 5.1 million parameters, including 3.2 million parameters in feature extraction layers. Both KR and FMR duplicate half of the kernels in the feature extraction layers, so the duplicated kernels contain 1.6 million parameters. Therefore, KR has about 6.4MB memory overhead. FMR requires extra space for feature maps, which is 2.6 million (10.4MB) for SSD-MobileNet, so the overall overhead for FMR is 16.8MB. The comparison of estimated memory overhead is shown in Table 2.

	Kernel Recovery	Feature Map Recovery
SSD-VGG	29.6MB	78MB
SSD-MobileNet	6.4MB	16.8MB

Table 2: Estimated Memory Overhead

4.4. Case Study

We compare our performance with the baseline fault-tolerant approach D2NN [27]. We choose D2NN for comparison because it takes advantage of full network duplication and reduces overhead by sharing neurons between the original and duplicated networks.

Targeted Saftey-Critical Application: Our case studies are motivated by the software system of a self-driving car. Among various safety-critical applications of the car, we chose a Thermal infrared (TIR) camera-based real-time object detector [4]. TIR images lack the texture and details that come with the visible spectrum and have less resolution than RGB images. Hence, a TIR camera-based object detection deepnet can be more vulnerable to errors in software calculations or hardware memory. We trained SSD-MobileNet V1 deepnet with 14,000 images from the FLIR TIR dataset [1] and our own 35,000 data with manually labeled 8 class annotations.

Dual Modular Redundancy (D2NN) Solution Implementation: To get the evaluation results of D2NN on the same experiments, we re-implemented the D2NN method in a different fashion but kept the same logic. First, we split the entire neural network into two parts as indicated in [27]. The first *unshared* part contains neurons (or kernels in our case) that we will duplicate, and the second part contains the *shared* neurons (kernels). We find the neurons to be duplicated by weight-sum ranking, described earlier. This neuron selection method is also consistent with the "sensitivity"-based selection method proposed in D2NN. Then, we duplicate the whole network into two: Primary Network and Secondary Network. Next, we use different strategies to inject errors for the *shared and unshared* parts of the two neural networks. We inject the same random errors in the second *shared* parts of the two networks We next introduce different random errors only in the *unshared*

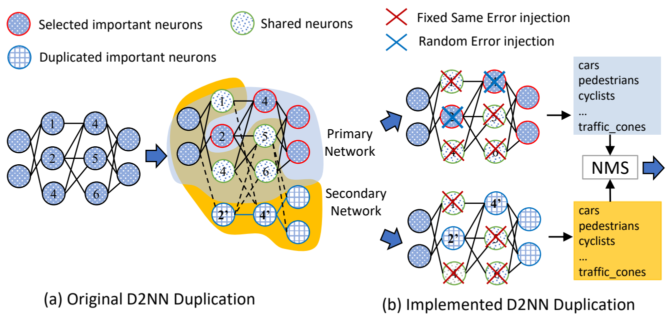

Selected important neurons
Shared neurons
Duplicated important neurons
Fixed Same Error injection
Random Error injection
Primary Network
Secondary Network
cars
pedestrians
cyclists
...
traffic_cones
NMS
(a) Original D2NN Duplication
(b) Implemented D2NN Duplication

Figure 8: D2NN implementation: (a) Original D2NN implementation. The primary and secondary networks share neurons and provide two outputs. (b) We use two identical networks to mimic the D2NN recovery technique to simplify the implementation. We apply non-maximum suppression (NMS) to suppress the bounding boxes and average the two outputs.

Dupliation Type	No Error	No Dupliation	FMR	FMR	KR	KR	D2NN	D2NN
Duplication Ratio	NA	0%	50%	20%	50%	20%	50%	20%
Car	0.56	0.41	0.50	0.46	0.49	0.48	0.46	0.44
pedestrian	0.26	0.17	0.22	0.20	0.21	0.21	0.20	0.19
Cyclist	0.28	0.15	0.20	0.18	0.21	0.20	0.18	0.17
Van	0.73	0.38	0.56	0.48	0.54	0.51	0.49	0.45
Truck	0.69	0.40	0.53	0.47	0.55	0.54	0.46	0.44
Bus	0.74	0.38	0.61	0.53	0.60	0.58	0.50	0.45
Traffic cone	0.39	0.11	0.18	0.14	0.17	0.16	0.19	0.15
Channelizer	0.28	0.06	0.19	0.10	0.16	0.14	0.17	0.15
mAP	0.49	0.26	0.37	0.32	0.36	0.35	0.33	0.30
FPS	119	119	69	65	100	103	NA	NA

Table 3: Accuracy Results with Weight Error 1% + Feature Map Error 1%

parts of the primary network. Finally, we run inferencing on both networks and take the average of their outputs. In this manner, we are able to replicate the propagation method proposed in D2NN without the need to hack into Pytorch or Tensorflow's encapsulated implementation of their linear/-convolutional layers.

Experiment: We inject the same 1% error rate in both weights and feature maps and compare the accuracies between the FMR, KR and D2NN approaches with different duplication ratios. In this experiment, we use only the weight-sum ranking approach. Table 3 capture the per-class accuracies and speed performance of the deepnet with respect to these different error scenarios and duplication methods. Based on our experiments, we infer the following.

Observation 1. KR duplication provides a similar accuracy protection capability and better speed performance compared to the FMR method. We note that KR drops only by 10% in speed performance. To maintain the best inference speed with reasonable accuracy protection against software, we recommend the use of KR with more duplication for a safety-critical application with limited resources. Each object class has a different vulnerability and recovery ratio with duplications against errors. Car and pedestrian classes are more robust than other classes overall, with 27% and 35% accuracy drops respectively. This is because these classes have been trained with more data than other classes and the trained weight parameters with the classes become less sensitive against errors. Car, pedestrian and bus classes show better accuracy recovery from duplication than other classes. For example, FMR with 50% duplication in all the experiments provide more than 50% recovery with these classes.

Observation 2. Our approaches out-perform D2NN recovery. FMR and KR improve the accuracy to 24% and 22% with 50% duplication, respectively. D2NN yields only 15% improvement with the same amount of duplication. Our recovery strategy is to cancel out the errors in weights and feature maps by compensation using partial duplication. The D2NN approach reduces the impact of the errors by attempting to duplicate the network's robustness itself. Hence, the performance of D2NN relies on the original network's robustness.

D2NN requires many changes in the original network implementation because the duplicated network shares the primary network. Since we simplified the implementation of the D2NN approach, it may not be entirely fair to compare the higher overhead of the D2NN implementation with that of our recovery methods. The overall overhead of the D2NN approach is similar to that of our FMR approach because both need to recalculate the feature maps from the duplicated parts.

5. Concluding Remarks

In this paper, we presented several findings on the vulnerabilities of deep-learning applications to transient faults. We proposed neuron-based network duplication approaches to improve the accuracy of a deepnet against these faults. Specifically, we tested and compared various methods to rank the importance among neurons. We next presented two fault-recovery mechanisms that duplicate a partial network by picking critical neurons in the inference stage to improve accuracy in the presence of transient faults. We demonstrated the benefits of our fault recovery solutions by presenting case studies using real-world convolutional neural network applications on multiple NVIDIA platforms: a GeForce GTX 2070Ti GPU and the popular Nvidia Xavier embedded platform. Our analysis of the impact of errors and recovery mechanisms can be useful to incorporate into highly automated vehicles with stringent safety and robustness requirements. As a next step, we plan to expand our solution to a fault-tolerance framework that handles both transient and permanent faults under real-time constraints.

References

[1] Free flir thermal dataset for algorithm training.

[2] Jacob A Abraham and W Kent Fuchs. Fault and error models for vlsi. *Proceedings of the IEEE*, 74(5):639–654, 1986.

[3] Rizwan A Ashraf, Saurabh Hukerikar, and Christian Engelmann. Shrink or substitute: Handling process failures in hpc systems using in-situ recovery. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 178–185. IEEE, 2018.

[4] Iljoo Baek, Wei Chen, Asish Chakrapani Gumparthi Venkat, and Raj Rajkumar. Practical object detection using thermal infrared image (tir)sensors. In *2021 IEEE Intelligent Vehicles Symposium (IV)*, 2021.

[5] Iljoo Baek, Zhihao Zhu, Sourav Panda, Nandha Kishore Srinivasan, Soheil Samii, and Ragunathan Raj Rajkumar. Error vulnerabilities and fault recovery in deep-learning frameworks for hardware accelerators. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2020.

[6] Anand Bhat. *Practical Solutions for Fault-Tolerance in Connected and Autonomous Vehicles (CAVs)*. PhD thesis, Carnegie Mellon University, 2019.

[7] Chun-Kai Chang, Sangkug Lym, Nicholas Kelly, Michael B Sullivan, and Mattan Erez. Evaluating and accelerating high-fidelity error injection for hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 45. IEEE Press, 2018.

[8] Kevin K Chang, A Giray Yağlıkçı, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, Donghyuk Lee, Mike O'Connor, Hasan Hassan, and Onur Mutlu. Understanding reduced-voltage operation in modern dram devices: Experimental characterization, analysis, and mechanisms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):1–42, 2017.

[9] Wei Chen, Kartikeya Bhardwaj, and Radu Marculescu. Fedmax: Mitigating activation divergence for accurate and communication-efficient federated learning. *arXiv preprint arXiv:2004.03657*, 2020.

[10] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE, 2012.

[11] Vasisht Duddu, N Rajesh Pillai, D Vijay Rao, and Valentina E Balas. Fault tolerance of neural networks in adversarial settings. *Journal of Intelligent & Fuzzy Systems*, (Preprint):1–11, 2020.

[12] Ashkan Eghbal, Pooria M Yaghini, Nader Bagherzadeh, and Misagh Khayambashi. Analytical fault tolerance assessment and metrics for tsv-based 3d network-on-chip. *IEEE Transactions on computers*, 64(12):3591–3604, 2015.

[13] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[14] Tilak Gaitonde, Shi-Jie Wen, Richard Wong, and Mark Warriner. Component failure analysis using neutron beam test. In *2010 17th IEEE International Symposium on the Physical and Failure Analysis of Integrated Circuits*, pages 1–5. IEEE, 2010.

[15] Brian Greskamp, Smruti R Sarangi, and Josep Torrellas. Threshold voltage variation effects on aging-related hard failure rates. In *2007 IEEE International Symposium on Circuits and Systems*, pages 1261–1264. IEEE, 2007.

[16] Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(10):993–1001, 1990.

[17] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258. IEEE, 2017.

[18] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

[19] Edwin T Jaynes. Information theory and statistical mechanics. *Physical review*, 106(4):620, 1957.

[20] Saurabh Jha, Subho Banerjee, Timothy Tsai, Siva KS Hari, Michael B Sullivan, Zbigniew T Kalbarczyk, Stephen W Keckler, and Ravishankar K Iyer. Ml-based fault injection for autonomous vehicles: a case for bayesian fault injection. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 112–124. IEEE, 2019.

[21] Saurabh Jha, Subho S Banerjee, James Cyriac, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Avfi: Fault injection for autonomous vehicles. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 55–56. IEEE, 2018.

[22] Saurabh Jha, Timothy Tsai, Siva Hari, Michael Sullivan, Zbigniew Kalbarczyk, Stephen W Keckler, and Ravishankar K Iyer. Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors. *arXiv preprint arXiv:1907.01024*, 2019.

[23] Seulki Lee and Shahriar Nirjon. Subflow: A dynamic induced-subgraph strategy toward real-time dnn inference and training. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 15–29. IEEE, 2020.

[24] Guanpeng Li, Karthik Pattabiraman, Chen-Yang Cher, and Pradip Bose. Understanding error propagation in gpgpu applications. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 240–251. IEEE, 2016.

[25] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[26] Si Li, Naila Farooqui, and Sudhakar Yalamanchili. Software reliability enhancements for gpu applications. In *Sixth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2013)*, 2013.

[27] Yu Li, Yannan Liu, Min Li, Ye Tian, Bo Luo, and Qiang Xu. D2nn: a fine-grained dual modular redundancy framework for deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 138–147. ACM, 2019.

[28] Caio Lunardi, Fritz Previlon, David Kaeli, and Paolo Rech. On the efficacy of ecc and the benefits of finfet transistor layout for gpu reliability. *IEEE Transactions on Nuclear Science*, 65(8):1843–1850, 2018.

[29] Jian-Hao Luo and Jianxin Wu. An entropy-based pruning method for cnn compression. *arXiv preprint arXiv:1706.05791*, 2017.

[30] Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. A high-performance fault-tolerant software framework for memory on commodity gpus. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[31] Abu Hasnat Mohammad Rubaiyat, Yongming Qin, and Homa Alemzadeh. Experimental resilience assessment of an open-source driving agent. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 54–63. IEEE, 2018.

[32] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R Johnson, David Nellans, Mike O'Connor, and Stephen W Keckler. Flexible software profiling of gpu architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 185–197. IEEE, 2015.

[33] Shizhao Sun, Wei Chen, Jiang Bian, Xiaoguang Liu, and Tie-Yan Liu. Ensemble-compression: A new method for parallel training of deep neural networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 187–202. Springer, 2017.

[34] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. IEEE, 2015.

[35] Cesar Torres-Huitzil and Bernard Girau. Fault tolerance in neural networks: Neural design and hardware implementation. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2017.

[36] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.

[37] Hui Xu, Zhuangbin Chen, Weibin Wu, Zhi Jin, Sy-yen Kuo, and Michael Lyu. Nv-dnn: towards fault-tolerant dnn systems with n-version programming. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 44–47. IEEE, 2019.