PyTorchFI: A Runtime Perturbation Tool for DNNs

Abdulrahman Mahmoud[1], Neeraj Aggarwal[1], Alex Nobbe[1], Jose Rodrigo Sanchez Vicarte[1], Sarita V. Adve[1], Christopher W. Fletcher[1], Iuri Frosio[2], Siva Kumar Sastry Hari[2]

[1]University of Illinois at Urbana-Champaign, [2]NVIDIA Corporation

***Abstract*—PyTorchFI is a runtime perturbation tool for deep neural networks (DNNs), implemented for the popular PyTorch deep learning platform. PyTorchFI enables users to perform perturbations on weights or neurons of DNNs at runtime. It is designed with the programmer in mind, providing a simple and easy-to-use API, requiring as little as three lines of code for use. It also provides an extensible interface, enabling researchers to choose from various perturbation models (or design their own custom models), which allows for the study of hardware error (or general perturbation) propagation to the software layer of the DNN output. Additionally, PyTorchFI is extremely versatile: we demonstrate how it can be applied to five different use cases for dependability and reliability research, including resiliency analysis of classification networks, resiliency analysis of object detection networks, analysis of models robust to adversarial attacks, training resilient models, and for DNN interpertability. This paper discusses the technical underpinnings and design decisions of PyTorchFI which make it an easy-to-use, extensible, fast, and versatile research tool. PyTorchFI is open-sourced and available for download via pip or github at:**
https://github.com/pytorchfi

I. Introduction

With the recent advances in machine learning (ML) alongside enabling hardware (such as GPUs [22] and custom ML processors [7], [14]), deep neural networks (DNNs) have quickly become a dominant player in the application space. Today, DNNs are heavily used across many application domains and hardware platforms, ranging from entertainment devices such as personal phones, to stringently safety-critical systems such as perception software in self-driving vehicles.

With the ubiquitous utilization of DNNs across many domains, it is crucial that DNNs operate reliably in the face of errors. There is mounting evidence that even tiny perturbations, such as transient cosmic ray particle strikes causing a bit flip (called *soft errors*), can cause a DNN to output an incorrect result at the software level [8], [19], [23], [37]. Additionally, recent work in adversarial machine learning has shown that malicious perturbations in the input (and more advanced attacks such as rowhammer within a network), can alter a DNNs execution [2], [3], [12], [27]–[30], [32], [42]. On one hand, most of the time an error has a negligible impact on the computation because it either gets masked out entirely (e.g., due to activation functions such as ReLU layers) or does not cause the DNN's decision to cross a decision boundary (a misclassification). On the other hand, there *are* errors which manifest into observable output corruption. It is therefore crucial that developers understand the dependability and reliability characteristics and limitations of their models before deployment in the field.

Developers are currently lacking the tools to study the impact of perturbations on DNNs. In order to detect and mitigate hardware errors which can propagate and affect DNN outcomes or malicious adversarial perturbations in the network, researchers and developers alike need accurate tools for assessing DNN reliability in the face of different error types. Such tools must be easy-to-use (for widespread adoption by researchers and developers), extensible (to keep up with the fast moving field of deep learning, while also allowing for the study of different perturbation models), and fast (since DNNs can become very large and there are many possible places within a network for an error to manifest).

In this paper, we introduce **PyTorchFI**, a runtime perturbation tool for DNNs developed in the popular PyTorch [31] framework. PyTorchFI allows users to perform neural network perturbations in weights and/or neurons in convolutional operations of DNNs during execution. Therefore, it enables the study of the manifestation and propagation of different perturbations at the application level. PyTorchFI is designed to be programmer-friendly and easy-to-use: it minimizes the programmer overhead by streamlining the installation process through the `pip` package manager, and provides a simple and intuitive implementation for performing perturbations at runtime. In addition, PyTorchFI is extremely fast with negligible runtime overhead due to its native implementation. Further, PyTorchFI is very versatile: by abstracting the notion of an "error" to that of a "perturbation" and designing for the latter, we believe a tool such as PyTorchFI can have many additional research applications beyond just reliability.

This paper focuses on presenting and discussing the technical underpinnings and design decisions of PyTorchFI, which are key to making it an easy-to-use, extensible, fast, and versatile tool. Additionally, we showcase multiple use cases for PyTorchFI, including (1) reliability analysis of convolutional neural networks (CNNs), (2) reliability analysis of object detection networks, (3) resiliency analysis of models designed to be robust to adversarial attacks, (4) training error-resilient models, and (5) an early exploration of using PyTorchFI for the interpretability of DNNs.

In summary, the primary contributions of this paper are:

- A programmer-friendly and easy-to-use perturbation tool implemented in the PyTorch framework.
- An extensible perturbation tool, allowing researchers to implement their own perturbation models or select one from a library of provided models. Further, it is tightly incorporated

This material is based upon work supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. A portion of this work was performed while Abdulrahman Mahmoud interned at NVIDIA.

within the PyTorch framework, allowing for future extensions to keep up with the fast moving field of deep learning.
- A fast and light-weight perturbation instrumentation, running at the speed of silicon and tested on both CPUs and GPUs.
- An exploration of PyTorchFI's versatility as a research tool, by studying five different use cases in the areas of reliability, security, and interpertability.

PyTorchFI is publicly available at https://github.com/pytorchfi and is also downloadable via Python's `pip` package installer.

II. Background and Related Work

A. CNN Background

A convolutional neural network (CNN) is a class of deep neural networks (DNNs) used for tasks such as image recognition and object detection. Figure 1 shows a general overview of a CNN performing *inference* (or *forward-pass*), an execution of a CNN to determine the class of an input image. A CNN is composed of many *convolutional layers*, which convolve the pre-trained weights/filters on *input feature maps* (or input fmaps) to produce *output fmaps*. A non-linear activation function is typically applied element-wise to the output fmaps and is considered part of the layer. Output fmaps of one layer form the input fmaps to the next layer.

Operations such as pooling and batch-normalization can also be applied after convolutional layers. These layers are connected using a human-selected topology to form a network. Simple DNN architectures consist of a series of convolutional layers (with pooling/batch-normalization layers in between) followed by some fully-connected layers. The final layer in the classification model is typically a *softmax* layer, which provides a probability distribution for each possible class the network is trained to predict. The class with the highest probability (the *Top-1* class) is the chosen prediction of the network during an inference. During training, these outputs are used to compute a loss between the expected and inferred output. That loss is then backpropagated through the network by a training algorithm, such as Stochastic Gradient Descent (SGD). Updates are computed during backpropagation to minimize the loss. It is common practice to compute and average updates across *batches* of inputs simultaneously.

The most fundamental computational unit in a CNN is a neuron (also commonly referred to as an *activation value*). A *neuron* is the result of a dot product between a filter of *weights* and an equal sized portion of the input. An output fmap is a plane of many neurons, and is obtained by performing a convolution operation over an input fmap.

B. Related Work

DNN dependability research that addresses hardware resilience, robust model training, security, and interpretability of models has largely been performed using custom techniques and tools [13], [23]. All these research areas require modifying state (by injecting random error or noise, or calculated perturbations) during DNN execution to evaluate and develop new techniques. This paper presents a tool, PyTorchFI, that enables easy customization to the target study (increasing productivity) and runs on state-of-the-art silicon (fast), making it a desirable tool for different dependability research studies.

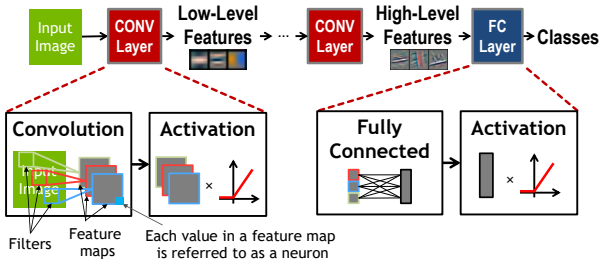

Fig. 1: A CNN inference example [7].

Two related error injection tools that offer similar capabilities are Ares [35] and TensorFI [9], designed to operate within the Keras [10] and TensorFlow [1] frameworks, respectively. These two tools allow modifying the state of the layers in DNNs as they are executing. However, Ares requires changes to the Keras inference computation to introduce dynamic perturbation, which is required for most dependable research studies. TensorFI requires the users to update a configuration file in addition to making modifications to the TensorFlow program. While Keras and TensorFlow are commonly used deep learning frameworks, PyTorch has emerged as a popular framework for DNN research for its ease-of-use [16] and its use of dynamic graphs for DNN computations, which is extremely powerful for understanding and debugging DNN models.

Apart from filling the gap for the PyTorch framework, we addressed the limitations of the prior tools and offer a tool that is fast and easy-to-use. Specifically, we address the issue of portability and longevity of the tool by implementing PyTorchFI in Python 3 rather than Python 2 [9] (Python 2 is no longer support as of January 1, 2020 [33]); we support injecting errors during both inference and training (we present a use case in Section IV-D); and we minimize the programmer overhead via a simplistic API which does not require model modifications. Additionally, PyTorchFI is still extremely fast as it operates at roughly the same native speed of PyTorch on silicon.

III. PyTorchFI: Tool Description

At a system level, PyTorchFI is a lightweight tool built on top of PyTorch [31] which enables perturbations on weights and neurons of DNN models for perturbation analysis with use cases including hardware resiliency, adversarial attacks, robust DNN design, and intepretability. This section explains the design choices and implementation details which make PyTorchFI an easy-to-use, extensible, fast, and versatile tool for error perturbations in DNNs. An overview of PyTorchFI is illustrated in Figure 2.

A. Design Choices

Dynamic perturbations for neurons can be implemented in several ways within the PyTorch framework. The simplest implementation is to append an intermediate layer after every convolutional layer, and apply a transformation layer to perturb output values before proceeding to the next layer in the network. Studying the effects of different perturbation models using this method would require major alterations to the network configuration. For deep

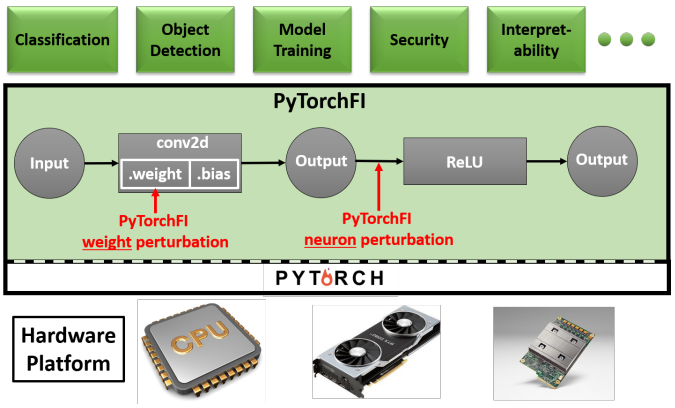

Fig. 2: PytorchFI is a lightweight tool built on top of PyTorch [31] that enables error perturbations for research into different domains of deep learning. Perturbations in weights are performed offline by modifying the weight tensor, while neuron perturbations are implemented using `hooks` on convolution operations.

networks with many layers, or networks with custom layers in-between convolutions, making the modifications to the model for this approach will require non-trivial effort for the user.

Another option is to modify the PyTorch source code to intercept the computation of the neuron to perturb it. This method suffers from a lack of portability because it may require separate implementations for convolutions on CPU, GPU, and other backends. It would require patching scripts and developer maintenance for future versions of PyTorch.

Rather than modifying the network topology or the PyTorch source code, we utilize PyTorch's `hook` functionality to perturb neuron values during the forward pass of a computational model. By leveraging the `hook` API to instrument error, PyTorchFI avoids altering any source code of PyTorch while also enabling compatibility with future PyTorch versions. Furthermore, it allows the perturbation to run at the native speed of PyTorch, with minimal introduced overheads (overheads depend only on the code introduced for perturbation – the instrumentation methodology introduces nearly no runtime overhead).

B. Implementation

Identifying hooks as the best candidate for instrumenting neuron perturbations is an important step to ensure that PyTorchFI is fast, extendable, and easy to integrate with existing implementations. For weights, we further optimize PyTorchFI by providing wrapper functions that directly modify the weight tensor before an inference, effectively perturbing weights offline and away from the critical path (Figure 2). This optimization translates to no runtime overhead for weight perturbations.

PyTorchFI was designed from the ground up for minimal programming overhead for the programmer. As a result, a researcher can begin using PyTorchFI by following just three steps: (1) importing PyTorchFI, (2) initializing their model, and (3) performing a perturbation with a custom or provided default error model. The following are the steps to install and use the tool.

1) **Installing and importing PyTorchFI**: PyTorchFI has been published to the `pip` package manager of Python, an extremely popular method for managing libraries such as `numpy` and `scikit-tools`. This makes PyTorchFI easily accessible, and requires no compilation or configuration scripts. Importing the tool is as easy as using *import* in the beginning of the code.
2) **Initialization:** Initializing PyTorchFI takes the model for which perturbations will be performed. Other arguments include input image height and width, and optional parameters such as batch size, model data type (e.g., FP32 or FP16), and whether to run on the CPU or GPU. PyTorchFI then performs a single, dummy inference to profile the model and gathers all the hyperparameters of the network, such as the number of layers, filter sizes, and feature map sizes. This information is used for ensuring that perturbations are legal, and to provide detailed debugging messages to the end user.
3) **Perturbation:** The third step involves selecting a perturbation model and a perturbation location. We provide a default set of perturbation models for the user to select from, such as a random value, a single bit flip, or zero value. The user can also easily implement their own perturbation model. Along with the perturbation model, the user needs to specify the location of the weight/neuron that will be perturbed. This can be a single location (specified by the layer, feature map, and neuron's coordinate position in the tensor) or multiple locations to incur multiple perturbations across the network. The user can also specify whether to have the same perturbation across all elements in a batch, or a different perturbation per element.

The actual perturbation occurs during runtime by taking in the location of the erroneous neuron/weight and appending it to a list of positions in the tensor to change. Then, on every layer, the forward hook will iterate through all of the locations and corrupt the corresponding value based on the selected perturbation model.

C. Evaluation

PyTorchFI has been tested on PyTorch versions 1.0 through 1.4 (latest at the time of submission). We expect long term support for PyTorchFI, as hooks are becoming first class objects in the PyTorch environment: they have been explicitly mentioned in every PyTorch release [34] and are widely used within the PyTorch ecosystem.

PyTorchFI's implementation has extremely low overhead since there is only a single check on every layer. If there are no perturbations defined, then there is no overhead. It also scales very well, since the same hook can be used to inject single or multiple error within the same operation.

To evaluate the runtime overhead introduced by PyTorchFI, we measured the runtimes of pretrained DNNs with and without perturbations introduced by PyTorchFI. We ran our experiments on two hardware platforms – for CPU, we used an AMD EPYC 7401 processor with 1 TB of RAM and for GPU we ran on an NVIDIA Titan Xp with 12 GB of RAM. We used the default perturbation model provided by PyTorchFI (a uniform, random value between [-1,1]) on a random neuron location for random input images. We averaged the runtime across 1000 trials for each network.

Figure 3 shows the runtime results. We see that all inferences (with and without PyTorchFI) typically take less than 0.2 seconds for both CPUs and GPUs. As GPUs are known to offer

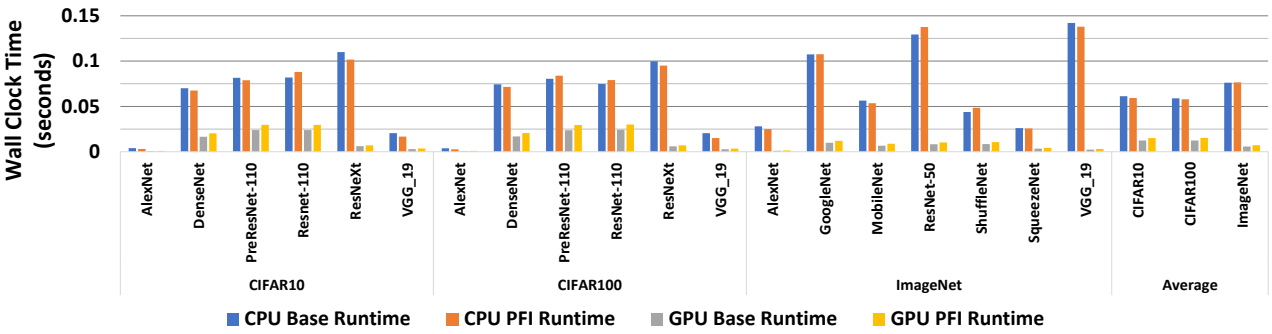

Fig. 3: Average runtime for 19 networks across three datasets, with and without PyTorchFI (PFI), for a single neuron injection with batch size = 1. PyTorchFI effectively runs at the same native speed on both CPU and GPU with negligible overhead.

higher throughput for deep learning workloads compared to CPUs, the GPU runtimes we observed were a lot faster. More importantly, what we find is that the runtime with perturbations differs by less than 10 millisecond in wall-clock time across both platforms, all models, and datasets. Further, we also performed a study of PyTorchFI using inference batching (a common practice for some DNN inference applications). We swept the batch size from 1 to 512. We observed the same trend: the wall clock time overall went up (as batching takes longer to run than for a single inference), while the runtimes with and without PyTorchFI were comparable and within the error margins, indicating an amortized cost per model for instrumenting perturbations. Thus, PyTorchFI is extremely fast, effectively operating on the native speed of the underlying hardware platform.

D. Limitations

PyTorchFI operates at the application level of DNNs, which is useful for modeling high level perturbations and understanding their effect at the system level. Lower level perturbation models, such as register-level faults, cannot be captured at this level. However, we can still use PyTorchFI to model lower level faults by mapping them to either single- or multiple- bit flips (in single or multiple neurons). Recent studies have shown that high level models can be used to study the effect of errors at the system level [6], [25]. At the same time, higher level models can run 4-6 orders of magnitude faster [15] compared to low-level implementations [5]. We show that PytorchFI runs at the native speed of silicon, as it requires no code instrumentation for error modeling. This enables a faster exploration of the large state space which is crucial for understanding real-life aspects of errors in safety-critical applications.

IV. PyTorchFI Use Cases

We demonstrate PyTorchFI's versatility as a research tool by showcasing five different use cases: 1) resiliency analysis of a classification task, 2) resiliency analysis of an object detection task, 3) resilience analysis of models robust to adversarial attacks, 4) training error-resilient models, and 5) DNN interpretability. While these are not the only uses of PyTorchFI, we show these to illustrate the importance and generality of the tool. Our goal is to demonstrate the uses of the tool and not to fully address challenges in each of the areas covered by the use-cases.

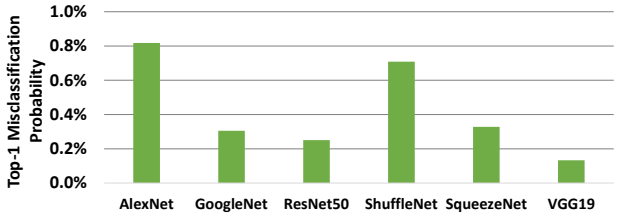

Fig. 4: Top-1 Misclassification probability for different quantized networks trained on ImageNet [11], using a single-bit flip error model of neurons.

A. Resiliency Analysis of DNNs used for Classification

DNNs are trained and optimized for accuracy, size, and speed, but not typically for resiliency against errors. We employed PyTorchFI to study the reliability of several popular networks. We performed large error perturbation campaigns across six networks with INT8 neuron-quantization [38] for the ImageNet [11] dataset. In each inference run, we inject a single-bit flip in a randomly selected neuron in the DNN to emulate a computational hardware error that may occur during inference. We only select images that are correctly classified by the model without perturbations. After conducting an error injection campaign for a model, we measure the total number of output corruptions observed, defined as a Top-1 misclassification due to the perturbation. We performed more than 107 million error injection experiments in total, which provides us 99% confidence interval error bars of $< 0.2\%$ for each network.

Figure 4 summarizes the results. All the networks display output corruptions and are not 100% reliable – overall, a little less than 1% of all errors manifested as Top-1 misclassifications. Results show that some networks are more resilient than other. For example, although AlexNet has a much lower classification accuracy than ShuffleNet (and is also a much smaller network in terms of size), both display a similar susceptibility to producing output corruptions due to single-bit flips. This suggests that network topology plays a role in resiliency of networks, also noted by prior work [23] done in the context of an accelerator.

While we demonstrate that some networks are more resilient than the others, several other resilience studies can be performed using PyTorchFI. Some examples are (1) evaluating resilience of a model at coarser granularity (via layer or feature map level error injections) to gain insights into why some models are more resilient than others, and use the results for low-cost selective

(a) No perturbation (b) PyTorchFI perturbations

Fig. 5: Perturbations on YOLOv3 object detection network

protection, (2) studying the effect of quantization on resilience, and (3) studying network vulnerability based on different output corruption criteria (e.g., top-1 misclassification vs. Top-1 not in Top-5 vs. significant confidence change between Top-3). Performing these studies using PyTorchFI can provide significant insights into DNN resilience and are interesting future research directions.

B. Resiliency Analysis of CNNs used for Object Detection

We used PyTorchFI to study resilience of object detection networks, exploring another class of DNNs widely used in autonomous vehicle systems. Object detection is more complex than image classification: it combines both the object localization and classification problems. Thus, the definition of an output corruption in this context changes dramatically from the Top-1 misclassification for a classification network.

Using PyTorchFI, we perturb multiple neuron values (one neuron perturbation per layer, each with a uniformly chosen random FP32 value) and study the effect on the inference output. Figure 5 illustrates the observed differences. Figure 5a depicts a correct inference with the YOLOv3 network [36] on an image from the COCO dataset [24]. In this image, the network identifies two objects (a car and a truck) by placing a border around each object and classifies each of the detected objects. Figure 5b shows that the perturbed network can behave irrationally, identifying many phantom objects each of which are classified seemingly arbitrarily. This example illustrates that PyTorchFI can be used to perform perturbations on DNN tasks beyond classification networks and with a different error model than the one used in Section IV-A. More importantly, it illustrates that perturbations can cause egregious outputs which must be studied for building resilient object detection networks for many safety-critical applications.

Using PyTorchFI, researchers can study the effect of perturbations across different error models on emerging DNN tasks. As nearly all the perception tasks in autonomous system are being performed by DNNs, it is important to have a versatile tool which can be used to perform detailed resiliency studies (including the ones described at the end of the previous section).

C. Resilience Analysis of Models Robust to Adversarial Attacks

In a traditional adversarial attack setting for classification networks, small perturbations in the input layer of a DNN typically propagate through subsequent layers and eventually lead to an incorrect classification [4]. Some of the defense strategies developed to protect a DNN from adversarial attacks aim at limiting the propagation of the perturbation from one

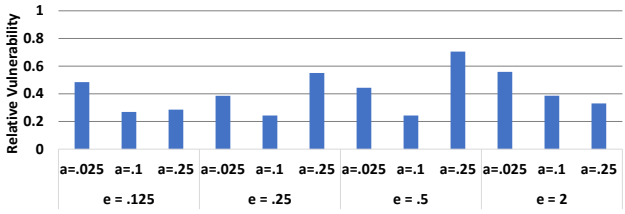

Fig. 6: Relative vulnerability (compared to a baseline model without IBP) of the first two layers of AlexNet when trained with different IBP parameters.

layer to the last one. To that end, PyTorchFI can be used to validate that protection against adversarial attacks should make a network inherently more resilient.

We consider the case of AlexNet on CIFAR-10 [21], and train a version of AlexNet through the Interval Bound Propagation (IBP) approach [13]. For a perturbation with a maximum $L_\infty = \epsilon$ norm in input, IBP computes the corresponding minimum and maximum probability of each class in output. Training is performed by minimizing the cost function

$$J = \sum(1-\alpha)log(p_{win}) + \alpha log(p_{win} - \delta p_{win}(\epsilon)), \quad (1)$$

where $\sum log(p_{win})$ is the traditional cross-entropy loss function, whereas $\sum log(p_{win} - \delta p_{win}(\epsilon))$ is the worst-case cross entropy, computed when the DNN is under attack and the magnitude of the attack is ϵ. For training, we follow the procedure for AlexNet in [43], but minimize the cost function in Equation 1. To guarantee stable convergence, we use curriculum learning as described in [13], and we scale linearly both α and ϵ from 0 to their respective maximum values from iteration 41 to iteration 123. We consider different values of $\alpha = \{0.025, 0.1, 0.25\}$ and $\epsilon = \{0.125, 0.25, 0.5, 2.0\}$ as these two parameters affect the robustness of the trained DNN in a different way: increasing ϵ leads to networks that are resistant to large input perturbations, while increasing α gives more importance to the worst case entropy, potentially penalizing the accuracy on clean data.

We used PyTorchFI to analyze the effect of IBP on the resiliency of the network, using the methodology similar to the one used in Section IV-A. The results showed improvement in the total resilience after training with IBP. While performing the per-layer vulnerability analysis, we discovered that the first two layers of AlexNet developed higher resilience compared to the rest of the layers. Figure 6 summarizes this key finding. This figure shows the vulnerability of the first two layers (defined using Top1-misclassifications) relative to a baseline AlexNet that is not trained with IBP. The analysis with PyTorchFI shows that the IBP is capable of improving resilience by up to 4×: this is a positive side-effect of adversarial training that, on the other hand, decreases the accuracy on clean data by approximately 3%. Our results also show that not all models trained to be robust to adversarial attacks are equally resilient. PyTorchFI enables us to investigate the reason for such differences and eventually develop a method that is robust to adversarial attacks and also highly resilient to hardware errors.

D. Training for Inherently Error-Resilient Models

Most use cases presented so far assume a trained model, which is then vetted using PyTorchFI for robustness to errors.

TABLE I: Training ResNet18 with and without PyTorchFI for resiliency.

	Baseline	**PyTorchFI**
Training time	2h 8m 33s	2h 8m 57s
Test accuracy	95.50%	95.34%
Post-training output misclassifications (out of 24 million)	10,543	7,701

A different approach towards DNN resiliency is to attempt to build reliability inherently into the network *while* training.

We propose a training procedure where we inject errors during training using PyTorchFI to increase the robustness of the network to errors once deployed. Injecting errors/noise during training can reduce the converged accuracy of the model and increase the training time. Models trained to be robust to traditional adversarial attacks commonly observe such a behavior. In contrast, our initial experiments show that some resilience can be built into the models via an injection-based training method with nearly no change in the model accuracy and training time.

Training, as described in section II-A, consists of many forward and back-propagation passes. PyTorchFI can be used to inject errors during forward passes during training, where the error model selection can be part of the training protocol.

Incorporating PyTorchFI into training requires minimal modifications — three additional lines of code as described in Section III-B. We integrate one of the built in error models for training, namely, a random neuron per layer is changed to a uniformly random value between [-1, 1] during the forward pass. Evaluations are presented on ResNet18 [17] trained on CIFAR10 [20]. Two models are trained for comparison: a baseline without PyTorchFI, and one with it. Both models are trained from the same initialization conditions for a clean comparison, and no other hyper-parameters are varied.

Table I summarizes some of the key elements between the two models, which were trained on an Nvidia Titan V. We find that training with PyTorchFI has a negligible impact on training time, where both models completed the same number of iterations on the dataset in the same amount of time. Importantly, integrating PyTorchFI into training does not adversely affect convergence. We find that training with PyTorchFI reduces the accuracy of the final model by 0.16%. Note that convergence time, unlike training time, describes the number of epochs required to reach the final accuracy; training with PyTorchFI does not affect convergence time either.

After training, we performed error injections on a separate test set to compare the resiliency of both networks. We measured the number of Top1-misclassifications due to perturbations, and found that the number of misclassifications are *reduced* for the ResNet18 model trained with PyTorchFI.

While these encouraging results show that some robustness can be introduced with no practical change in training time and model accuracy, selecting a different error model and the frequency with which we injection errors during the forward pass (during training) may likely provide different robustness, accuracy, and training time trade-offs. Studying this trade-off space is an interesting future research direction.

(a) No perturbation (b) Low sensitivity (c) High sensitivity

Fig. 7: Visualization of error injections in DenseNet using Grad-CAM [39]. a) shows the original visualization with no perturbation, b) perturbation in the least sensitive feature map, c) perturbation in the most sensitive feature map.

E. Interpretability

One important research question which can provide insight into the reliability and dependability of neural networks is to interpret *how* a DNN works. While prior research has looked into DNN interpretability [26], [40], [41], the field is still evolving and state-of-the-art techniques cannot fully explain the predictions made by the models. We propose a technique which can work alongside the state-of-the-art techniques to assist in DNN interpretability.

One popular technique for visualizing the important input pixels which contributed to a DNN inference is Guided-GradCAM [39], [44]. Guided-GradCAM performs backpropagations starting at different layers to generate gradients for the input, which are then aggregated and visualized based on magnitude. We perform error injections using PyTorchFI in the forward pass of GradCam on specific feature maps, to highlight the effect of a neuron firing and the affect that a specific feature has on classification. Figure 7a shows the superimposed heatmap generated by the Guided-GradCam technique on a correct inference using DenseNet [18]. Figure 7b shows the effect of injecting an egregiously large value of 10,000 in a feature map which has little impact on the classification as defined by the gradient values of the feature map. As shown, although the neuron value in this feature map is extreme, the visualization technique shows little difference in the output; this is also corroborated in the softmax where the Top-1 class does not change. On the other hand, perturbing a neuron of a "highly vulnerable" feature map skews the heatmap as portrayed in Figure 7c. Thus, an error-injection technique can be tuned to shed insight into the mapping between important input pixels and important feature maps. This simple experiment can perhaps guide a more rigorous iterative algorithm: perturbing a network at different feature maps and observing the effect on the heatmap along with the Top-1 network classification to extract which regions of the input pixels are picked up during inference to arrive at the correct classification of the image. This is an interesting future research direction.

V. Conclusion

We present PyTorchFI, an open-source runtime perturbation tool for DNNs implemented for the PyTorch deep learning framework. PyTorchFI is an easy-to-use, extensible, fast, and versatile tool for performing perturbations in neurons and weights of DNNs during execution. This paper describes the technical underpinnings of the tool, and demonstrates five different use cases enabled by PyTorchFI across multiple domains.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: http://tensorflow.org/

[2] B. Biggio, B. Nelson, and P. Laskov, "Poisoning Attacks against Support Vector Machines," in *The International Conference on Machine Learning (ICML)*, 2012.

[3] N. Carlini and D. Wagner, "Towards Evaluating the Robustness of Neural Networks," in *IEEE Symposium on Security and Privacy (SP)*, 2017.

[4] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin, "On Evaluating Adversarial Robustness," *ArXiv*, vol. abs/1902.06705, 2019.

[5] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[6] C.-K. Chang, G. Li, and M. Erez, "Evaluating Compiler IR-Level Selective Instruction Duplication with Realistic Hardware Errors," *The 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, 2019.

[7] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *The International Symposium on Computer Architecture (ISCA)*, 2016.

[8] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardelenben, "BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[9] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications," *ArXiv*, vol. abs/2004.01743, 2020.

[10] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[12] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *The International Conference on Learning Representations (ICLR)*, 2015.

[13] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. A. Mann, and P. Kohli, "On the Effectiveness of Interval Bound Propagation for Training Verifiably Robust Models," *ArXiv*, vol. abs/1810.12715, 2018.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and B. Dally, "Deep compression and EIE: Efficient inference engine on compressed deep neural network," in *The International Symposium on Computer Architecture (ISCA)*, 2016.

[15] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluation," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[16] H. He, "The State of Machine Learning Frameworks," https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/, 2019.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[18] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[19] S. Jha, S. S. Banerjee, T. Tsai, S. Hari, M. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "ML-based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection ," in *International Conference on Dependable Systems and Networks (DSN)*, 2019.

[20] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," *Computer Science Department, University of Toronto, Tech.*, 2009.

[21] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 (Canadian Institute for Advanced Research)."

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Neural Information Processing Systems (NIPS)*, 2012.

[23] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[24] T.-Y. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," *ArXiv*, vol. abs/1405.0312, 2014.

[25] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2015.

[26] A. Mahendran and A. Vedaldi, "Salient Deconvolutional Networks," in *European Conference on Computer Visio (ECCV)*, 2016.

[27] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. P. Rubinstein, U. Saini, C. A. Sutton, J. D. Tygar, and K. Xia, "Exploiting Machine Learning to Subvert Your Spam Filter," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[28] A. Newell, R. Potharaju, L. Xiang, and C. Nita-Rotaru, "On the Practicality of Integrity Attacks on Document-Level Sentiment Analysis," in *Artificial Intelligent and Security Workshop (AISec)*, 2014.

[29] J. Newsome, B. Karp, and D. X. Song, "Paragraph: Thwarting Signature Learning by Training Maliciously," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2006.

[30] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Neural Information Processing Systems (NeurIPS)*, 2019.

[32] R. Perdisci, D. Dagon, W. Lee, P. Foglat, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *IEEE Symposium on Security and Privacy (S&P)*, 2006.

[33] Python, "Sunsetting Python 2," https://www.python.org/doc/sunset-python-2/, 2020.

[34] PyTorch, "Releases pytorch/pytorch," https://github.com/pytorch/pytorch/releases, 2020.

[35] B. Reagen, U. Gupta, L. Pentecost, P. N. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *Design Automation Conference (DAC)*, 2018.

[36] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *ArXiv*, vol. abs/1804.02767, 2018.

[37] A. H. M. Rubaiyat, Y. Qin, and H. Alemzadeh, "Experimental Resilience Assessment of an Open-Source Driving Agent," *Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2018.

[38] C. Sakr and N. R. Shanbhag, "An Analytical Method to Determine Minimum Per-Layer Precision of Deep Neural Networks," *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[39] R. R. Selvaraju, A. Das, R. Vedantam, M. Cogswell, D. Parikh, and D. Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," in *International Conference of Computer Vision (ICCV)*, 2017.

[40] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps," *CoRR*, vol. abs/1312.6034, 2014.

[41] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller, "Striving for Simplicity: The All Convolutional Net," *CoRR*, vol. abs/1412.6806, 2014.

[42] H. Xiao, B. Biggio, G. Brown, G. Fumera, C. Eckert, and F. Roli, "Is feature selection secure against training data poisoning?" in *The International Conference on Machine Learning (ICML)*, 2015.

[43] W. Yang, "Pytorch-classification," https://github.com/bearpaw/pytorch-classification, 2017.

[44] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.