DeltaNN: Assessing the Impact of Computational Environment Parameters on the Performance of Image Recognition Models

DeltaNN: Assessing the Impact of Computational Environment Parameters on the Performance of Image Recognition Models

Nikolaos Louloudakis
n.louloudakis@ed.ac.uk
University of Edinburgh

Perry Gibson
perry.gibson@glasgow.ac.uk
University of Glasgow

José Cano
jose.canoreyes@glasgow.ac.uk
University of Glasgow

Ajitha Rajan
arajan@ed.ac.uk
University of Edinburgh

***Abstract*—Image recognition tasks typically use deep learning and require enormous processing power, thus relying on hardware accelerators like GPUs and TPUs for fast, timely processing. Failure in real-time image recognition tasks can occur due to sub-optimal mapping on hardware accelerators during model deployment, which may lead to timing uncertainty and erroneous behavior. Mapping on hardware accelerators is done using multiple software components like deep learning frameworks, compilers, and device libraries, that we refer to as the computational environment. Owing to the increased use of image recognition tasks in safety-critical applications like autonomous driving and medical imaging, it is imperative to assess their robustness to changes in the computational environment, as the impact of parameters like deep learning frameworks, compiler optimizations, and hardware devices on model performance and correctness is not yet well understood.**

In this paper we present a differential testing framework, `DeltaNN`, that allows us to assess the impact of different computational environment parameters on the performance of image recognition models during deployment, post training. `DeltaNN` generates different implementations of a given image recognition model for variations in environment parameters, namely, deep learning frameworks, compiler optimizations and hardware devices and analyzes differences in model performance as a result. Using `DeltaNN`, we conduct an empirical study of robustness analysis of three popular image recognition models using the ImageNet dataset. We report the impact in terms of misclassifications and inference time differences across different settings. In total, we observed up to 72% output label differences across deep learning frameworks, and up to 81% unexpected performance degradation in terms of inference time, when applying compiler optimizations.

I. Introduction

Much of the existing literature for assessing robustness and safety of image recognition models has focused on testing the Deep Neural Network (DNN) structure and addressing biases in the training dataset through adversarial examples and data augmentation [1]–[3]. However, the impact of computational environment aspects related to the DNN model deployment process, post training, has not yet been explored. In particular, existing techniques fail to consider model output errors that could potentially be caused by interactions of the DNN model with the underlying computational environment – conversions between Deep Learning (DL) frameworks (e.g., TensorFlow, PyTorch, TensorFlow Lite), compiler optimizations (e.g., operator fusion, loop unrolling, etc.), and the hardware platforms they run on (e.g., CPUs, GPUs, etc.). Figure 1 shows potential sources of error in the computational environment when a DNN model is deployed. These environment aspects are important considerations in model maintenance and evolution.

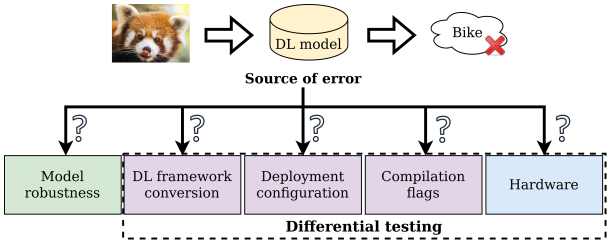

Fig. 1. Possible sources of errors when deploying DNN models.

To understand the impact of the computational environment on model deployment, we present a differential testing framework, `DeltaNN`[1], that helps evaluate the robustness of image recognition models to changes in specific aspects of the computational environment (Figure 2). `DeltaNN` takes as input a trained DNN model defined in a given DL framework and produces different implementations by changing the following parameters in the computational environment:

DL frameworks: transforming a model defined in one DL framework to the model format of another framework. Studying the impact of model conversion between DL frameworks is important, as developers may often convert their models to support resource constrained environments on mobile and IoT devices. Automated conversion processes suffer from faults, mainly caused by unsupported operations in the target framework, or by the converter. We generate different implementations of every trained model with several popular DL frameworks.

Compiler optimizations: considering different levels of compiler optimizations with each generating a distinct code implementation. The focus of our experiments is on graph-level optimizations like operator fusion, eliminating common subexpressions, data layout transformations for better memory utilization and access patterns on target devices, or potentially unsafe optimizations such as "fast-math". Compiler optimizations are expected to improve model performance, sometimes

Authors, Ajitha Rajan and Nikolaos Louloudakis, would like to acknowledge the support received from funding sources, UKRI Trustworthy Autonomous Systems Node in Governance and Regulation (EP/V026607/1) and Royal Society Industry Fellowship, for this work.

[1]The source code is available at: https://github.com/luludak/DeltaNN

at the cost of model accuracy. We study this parameter to understand the extent of impact on model performance and correctness.

Hardware devices: we generate implementations for a range of GPU accelerators, from a resource constrained mobile GPU to a powerful server-class GPU [4]. We consider different types of devices to check if GPU specifications can impact model output.

We assess the robustness of the DNN models with respect to consistency in output labels for changes in each of the three computational environment parameters. Note that it is important to check changes in the output label, as it directly affects model accuracy. Additionally, we monitor model inference times for different settings of computational environment parameters to understand the extent of variation among them. Inference times are an important consideration for timing safety in real-time perception systems in applications like self-driving cars, where there is a performance requirement for object detection models to return results within a fixed time budget [5].

We assess the robustness of three widely used image recognition models – MobileNetV2 [6], ResNet101V2 [7], and InceptionV3 [8], on the ImageNet object detection test dataset (ILSVRC2017) [9]. We chose the three models based on their popularity but also to provide variety on layer architecture and model size. The dataset we selected is a competition test dataset designed to extensively test models. The `DeltaNN` framework uses the Apache TVM [10] machine learning compiler stack, as it allows importing models from all major DL frameworks while providing fine-grained control over compilation configurations and execution of the DNN models, as well as a wide range of hardware backend support.

Overall, we find that conversions between DL frameworks significantly impacts output labels of the DNN models by up to 72%. We identify the source of the label-impacting error – small amounts of noise introduced in the weights by the framework conversion tools. The weight differences, although small, may be caused by floating-point rounding errors which can cause label changes when accumulated across the layers. On the other hand, we found that varying hardware accelerators and compiler optimizations do not affect model output but can lead to a non-negligible performance degradation with respect to inference time under specific scenarios. We observed up to 81% unexpected performance degradation in model inference times when applying certain compiler optimizations.

In summary, we make the following contributions:

1) Assess robustness of image recognition model *outputs*, post training, with respect to changes in the computational environment: DL frameworks, compiler optimizations, and hardware devices using a differential testing framework, `DeltaNN`.
2) Assess robustness of model *inference time* with respect to changes in the computational environment: DL frameworks, compiler optimizations, and hardware devices.
3) Analyze and identify sources of *label discrepancy* when converting between DL frameworks.

II. Background

Figure 4 gives an overview of the typical layers in the deep learning systems stack [11]. Much of the existing work on DL model robustness has focused on testing robustness with respect to the top two layers, *Datasets* and *Models*. In this paper, we consider robustness with respect to the bottom three layers which make up the computational environment required for executing a given DNN model, which includes the deep learning framework, the related systems software, and the underlying hardware.

A. Deep Learning Frameworks

Deep Learning Frameworks (the third layer in Figure 4), provide utilities such as model declaration, training, and inference to machine learning engineers. For our study, we consider four widely used DL frameworks: *Keras*, *PyTorch*, *TensorFlow (TF)*, and *TensorFlow Lite* (*TFLite*).

Keras [12] is a high-level DL framework that provides APIs for effective deep learning usage. It acts as an interface for TensorFlow, and we aim to observe potential overheads and bug introductions from the extra layer of complexity.

PyTorch [13] is an open source machine learning framework based on the Torch library. It supports hardware acceleration for tensor computing operations.

TensorFlow (TF) [14] is an open-source DL framework developed by Google, widely used for training and inference of DNNs.

TensorFlow Lite (TFLite) [14] is a lightweight version of TF, and part of the full TF library, focused only on the inference of DNNs on mobile and lightweight devices.

B. Framework Conversions

Developers and researchers convert DNN models from one DL framework to another for one of the following two reasons, 1) portability, to enable the compilation and execution of models on devices of varying capabilities, and 2) differing framework capabilities. For example, some DL frameworks may provide detailed debugging and profiling capabilities for development while others may be better suited for optimized deployment.

Conversion of DNN models between DL frameworks is facilitated by automated conversion processes enabled by tools such as `tf2onnx` [15], `onnx2keras` [16], `onnx2torch` [17], and `MMdnn` [18]. The conversion process can, however, suffer from errors in the model parameters and graph representation which can potentially affect the output labels. We refer to models defined within a given DL framework as a "*native model*", and models that have been converted to another DL framework as a "*converted model*". Systems such as ONNX [19] and *MMdnn* [18] attempt to provide common intermediate formats for conversion between DL frameworks. However, these systems can still be error-prone and have issues supporting bespoke operators, motivating our investigation of framework conversion errors.

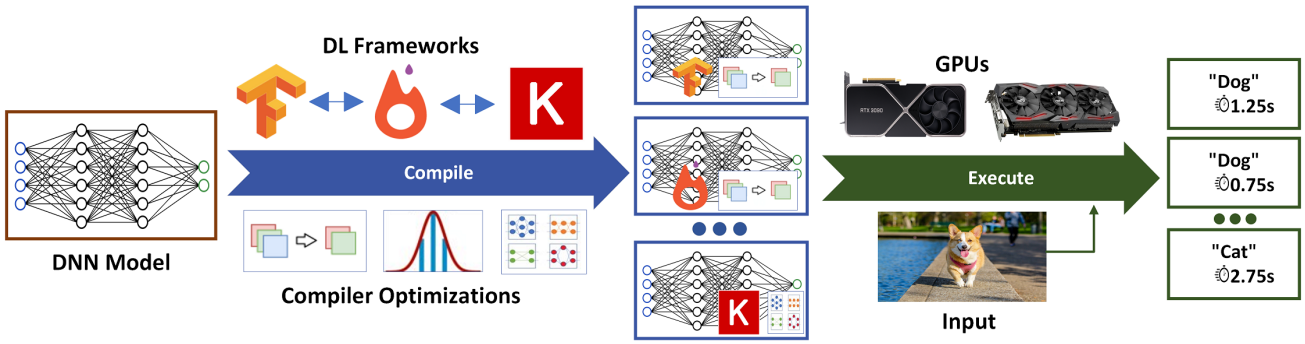

Fig. 2. Differential Testing applied by `DeltaNN` for a DNN model, varying deep learning frameworks, compiler optimizations, and hardware devices.

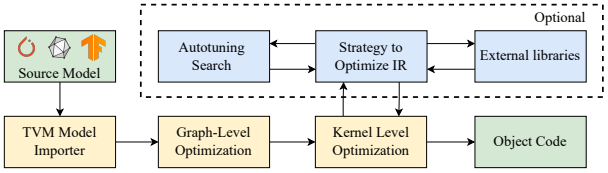

Fig. 3. Overview of DNN compilation in Apache TVM.

TABLE I
INFERENCE ACCURACY OF NATIVE MODELS ON THE IMAGENET DATASET.

DNN Model / Framework	PyTorch	Keras	TF	TFLite
ResNet101	81.9	76.4	77.0	77.0
InceptionV3	77.3	77.9	78.0	78.0
MobileNetV2	72.2	71.3	71.9	71.9

C. Systems Software: Apache TVM

Apache TVM [10] is an end-to-end machine learning compiler framework for CPUs, GPUs, and specialized accelerators, actively used and supported by a wide community of developers and researchers. It generates optimized code for specific DNN models and hardware backends, allows us to import DNN models from a range of DL frameworks, and provides profiling utilities such as per-layer inference times. A simplified representation of Apache TVM can be seen in Figure 3. TVM's support of several DL frameworks, optimization settings, and hardware accelerators makes it a suitable choice to leverage within `DeltaNN`. It also provides direct importers for models from most popular DL frameworks, which load the models as a TVM computation graph that can be optimized and compiled. TVM also provides a set of graph-based optimizations for DNNs, such as operator fusion, elimination of common subexpressions, loop re-ordering and unrolling, tiling, vectorization, and potentially unsafe precision optimizations such as fast math. TVM applies such optimizations automatically via the use of an `-O[0-4]` flag. Finally, TVM supports third-party operator kernel libraries such as cuDNN [20] and the Arm Compute Library [21].

D. The Perception AI Models

A common benchmark for Perception AI models is the ImageNet image classification dataset [9], which requires assigning one of 1000 possible class labels to RGB images of 224×224 pixels. For solving Perception AI problems, such as classification and semantic segmentation, convolutional neural networks (CNNs) are commonly used, which are DNNs characterized by convolutional layers. Transformer-based architectures [22] have begun to provide competitive results in recent years [23], [24], however are still maturing. Thus, for our evaluation we explore three widely used CNN models: MobileNetV2 [6], ResNet101V2 [25], and InceptionV3 [8]. These models are widely known and extensively used for classification and semantic segmentation operations, and are the "backbone network" for other tasks such as object detection [26]. All three models have native definitions within the DL frameworks under study. The accuracy of the native version of each model is shown in Table I. It is expected that the same model may have varying accuracy between frameworks, as each framework will define and train their own version of the model from scratch, which produce different parameters (since training is stochastic), and there may even be small differences in the graph definition (e.g., different padding parameters). We observe that TF and TFLite models have the same accuracy, suggesting that the latter models were converted from the former by developers.

III. Related Work

Existing work has primarily focused on the robustness of the dataset and model architecture layers, the top two layers in Figure 4. In particular, a survey by Zhang et al. [27] comprehensively presents existing testing techniques in machine learning by exploring a number of contributions in terms of correctness, robustness, and fairness, primarily focusing on model training and validation datasets. DeepXPlore [28] applies whitebox testing by measuring neuron coverage, identifying similar DNNs for cross-reference and generating adversarial inputs to detect faults. DLFuzz [3] attempts to minutely mutate inputs to improve neuron coverage.

DeepTest [2] modifies images using linear & affine transformations, and generates inputs simulating different weather conditions and phenomena to stress-test DNNs utilized for autonomous driving. DeepRoad [1] applies in the same context, while using GAN-based metamorphic testing that simulate extreme weather conditions, such as heavy rain and snow. Ayaz et al. [29] propose to improve the robustness against adversarial attacks with deeply quantized DNNs. For a more comprehensive overview of adversarial inputs of DNNs, we refer the readers to a survey [30].

In comparison, robustness with respect to the computational environment (last three layers of Figure 4) has received little attention. With respect to the DL Frameworks layer, some research has been conducted in the direction of analysis of model training and inference performance [31]–[35]. In addition, a recent survey [36] explores various parameters and their effect towards model accuracy and execution time. In terms of automated testing of DL frameworks, there are some works aiming to detect and localize inconsistencies between models sourced from different DL frameworks. Under this context, CRADLE [37] applies output and model execution comparison, while LEMON [38] utilizes mutation testing to detect bugs in DL frameworks. Similarly, Audee [39] aims to detect logical, not-a-number bugs, and crashes by applying an exploratory approach in combination with mutation testing. However, all these contributions conduct experiments that do not consider DL framework conversions, and focus on a specific set of bugs instead of broadly exploring potential issues related to DL frameworks. They also do not consider the impact of other computational environment aspects, such as optimizations and deployment on different hardware acceleration devices - aspects explored in our contribution.

In terms of DL Framework conversions, there is a variety of tools available in the community for that purpose. *MMdnn* [18] is a tool focusing on the process of library conversions, using an intermediate representation. There are many other tools for DNN framework conversions such as `tf2onnx` [15], `onnx2keras` [16], `onnx2torch` [17], as well as native APIs of TFLite and PyTorch. However, the error proneness of the process is overlooked in the literature, as there is only one empirical study of DL framework conversions [40] which focuses only on conversions between ONNX and CoreML, finding prediction accuracy of converted models to be similar to the original ones. We explore the effects of DL framework conversions extensively in our work.

Regarding the systems software layer in Figure 4, a recent study [41] examined bugs introduced by different DL compilers. Incorrect optimization code logic accounted for 9% of the bugs introduced by compilers. Other compiler bugs presented in the study include misconfiguration, type problem, API misuse, incorrect exception handling, and incompatibility. In our contribution, we primarily examine the effect of changing compiler optimizations on model performance, in terms of accuracy and inference time.

Finally, for the hardware layer in Figure 4, a taxonomy of faults encountered in DNNs used in object detection has been established [42]. The authors surveyed commits, issues and pull requests from 564 GitHub projects and 9,935 posts from Stack Overflow and interviewed 20 researchers and practitioners. The study revealed *GPU related bugs* to be one of the five main categories faults in deep learning tasks like object detection. The study, however, did not explore the impact of these bugs on model performance. The other four categories of faults were *API*, *Model*, *Tensors and Inputs*, and *Training* which relate to the top two layers in Figure 4.

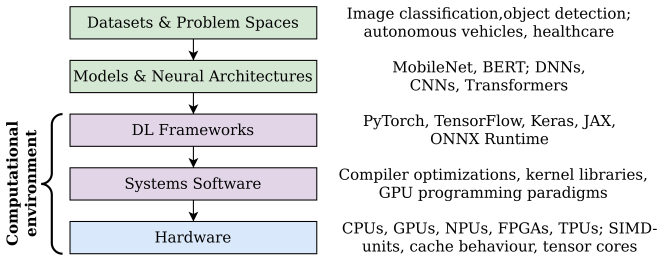

Fig. 4. Relevant layers in the deep learning systems stack [11].

We explore in-depth the effect of DL framework conversions in terms of output label predictions and execution times under different configurations of DNN models, DL frameworks, and hardware acceleration devices capabilities during model deployment. This extends our previous work [43]–[45] with further experiments and exploration of library conversions in detail. To the best of our knowledge, this is the first work that assesses the effects of the computational environment aspects on image recognition models, post training.

IV. Methodology

`DeltaNN` comprises three stages, as shown in Figure 5: (1) *Model Variant Generator* that generates different equivalent model implementations when changing DL frameworks and compiler optimizations; (2) *Differential Execution* that executes each of the model implementations with images from a test dataset; and (3) *Analysis* that compares the output labels, inference time, and other data from the different implementations, and aids in localization of discrepancy sources, if any.

A. Model Variant Generator

As seen on the left-hand side of Figure 5, the **Model Variant Generator** takes as input a pre-trained image recognition model (e.g., InceptionV3) sourced from a given DL framework (e.g., PyTorch). If we use one of these pre-trained models "as-is", and pass it directly to the **Model Importer**, we refer to it as a *native* model. However, if we convert it using the **DL Framework converter**, we refer to the original model as the *source*, and the converted model as the *target*. For example, we could convert an InceptionV3 model sourced from PyTorch to the TensorFlow model format. Across the four DL frameworks we support in `DeltaNN`, we have conversion paths from every framework to every other one.

To implement the conversions, we use the popular ONNX format [19] as an intermediate representation when a direct

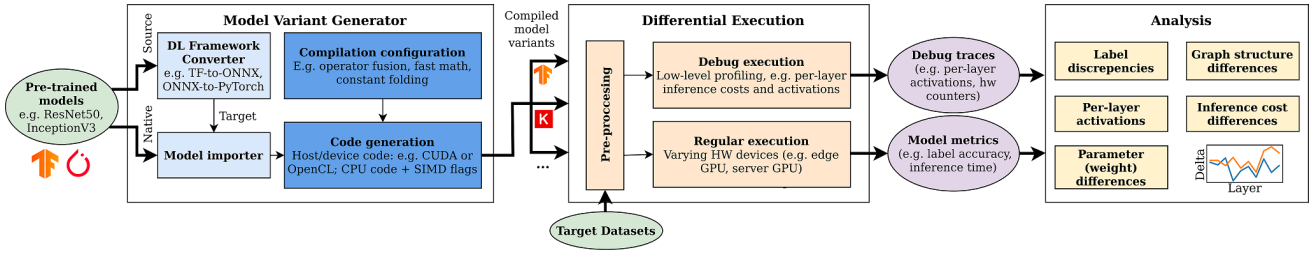

Fig. 5. Architecture of the DeltaNN framework: (1) *Model Variant Generator* for generating different model implementations when changing and converting DL frameworks and compiler optimizations; (2) *Differential Execution* for executing the various model implementations on images from a target dataset; and (3) *Analysis* for comparing output labels and inference time between executions while analyzing source of discrepancy.

conversion is not possible. Some DL frameworks, such as PyTorch and TFLite, have native tools for this conversion; whereas for others, such as TensorFlow, we leverage popular third-party conversion tools like `tf2onnx` [15]. We then convert from ONNX to the target DL framework model format using a number of widely used libraries, such as `onnx2torch` [17], `onnx2keras` [16].

a) Compiler Configuration: DeltaNN generates model implementations with different levels of TVM graph-level compiler optimizations: basic, default, and extended variants. **Basic** (`o0`) applies only "inference simplification", which generates simplified expressions with the same semantic equivalence as the original DNN. **Default** (`o2`) applies all optimizations of `o0`, as well as operator fusion for operations such as ReLU activation functions, as well as constant and scale axis folding. The optimizations are applied to TVM's Relay intermediate representation (IR) [46]. **Extended** optimization (`o4`) applies all optimizations from Default, as well as additional ones such as eliminating common subexpressions, applying canonicalization of operations, combining parallel convolutions, dense matrix and batch matrix multiplication operations, and enabling "fast math" (which allows the compiler to break strict IEEE standard [47] compliance for float operations if it could improve performance). We can also enable and disable specific optimizations at a fine-grained level, which can be useful for localization. In addition, kernel-level optimizations such as schedules, auto-tuning [48], third-party libraries (such as cuDNN [20]), and auto-scheduling [49], [50] can be explored, but are not the focus of this study.

b) Code Generation: The final part of the model variant generator takes the selected compiler configuration and imported model format and generates both host and device code, with the option to explore different programming paradigms (e.g., CUDA and OpenCL), CPU-side optimization flags (e.g., enabling vector instructions), and hardware devices (e.g., different GPU devices). The code generation step produces the outputs of the whole **Model Variant Generator** stage, namely several model variants, each with a different setting for compiler optimization, DNN model source or target, and host/device code configuration.

B. Differential Execution

The next stage of DeltaNN is **Differential Execution** of the model variants from the previous stage. It consists of three main steps: (1) the **Pre-processing** module responsible for normalizing inputs for better model performance (with a variety of pre-processing functions to choose from); (2) the **Regular Execution** module that executes the model for different target devices; and (3) the **Debug Execution** module, that executes the model similar to the *Regular Execution* module but additionally generates execution profiling information that can be used for deeper performance and error insights.

a) Pre-processing: It is a common practice to pre-process the inputs from the dataset before inference, similar to training. Examples of pre-processing include image resizing, input image pixels normalization, and more. By default, the module pre-processes the inputs based on the model architecture and source DL framework.

b) Regular Execution: This module executes the model on a specified target device to perform inference against a specific input and generate an output prediction. Execution encompasses model loading, setting up execution parameters from configuration, and experiment management (i.e., multiple runs). The output from this module is **Model metrics**, namely label accuracies and inference times for each image executed on the model. This module orchestrates and performs model execution in bulk, executing inference of a whole dataset against the numerous module variants generated by the **Model Variant Generator** module.

c) Debug Execution: As the main purpose of DeltaNN is differential testing, generating execution-based metadata is vital for analyzing possible sources of error in the model variants. The **Debug Execution** module performs model execution similar to the **Regular Execution** module. Nevertheless, during execution, the module generates profiling metadata and debug metrics associated with the inference process, such as tensor outputs of each layer, per-layer inference time, and hardware counters. This information is passed on as *debug traces* to the **Analysis** stage for fault localization.

C. Analysis

For every pair of model variants, the **Analysis** stage compares labels and inference time from *Model metrics* for all images in the dataset. To compare labels, we compare the top

ranked predictions between the model variants or performing rank-biased overlap to compare rankings for top-K elements. This means not only can we detect divergence, but also measure the level of divergence. When an image generates different labels or inference time between a pair of model variants, the **Analysis** module compares the *debug traces* from the model variants inspecting differences in *per-layer activations*, *weights*, and the *graph structure*. For *per-layer activations*, we compare mean, max, and standard deviations statistics of the layer activations between the pairs of models. The **Analysis** module also provides the capability to visualize the differences observed in layer activations and weights.

V. Experiments

fv We consider three widely used image recognition models of various sizes: MobileNetV2 [6], ResNet101V2 [7], [25], and InceptionV3 [8]. We use models pre-trained on ImageNet [51], using native model definitions and pre-trained parameters/weights sourced from 4 different DL frameworks' repositories: *Keras* [12], *PyTorch* [13], *TensorFlow(TF)* [14], and *TFLite* [14]. Each model is run through `DeltaNN` to generate model variants with different compiler optimization levels and *target* DL frameworks, and executed on 4 GPU devices (discussed in Section V-B). In total, we evaluate a combination of 3 models, 12 DL framework conversions, 4 devices, and 3 optimization levels.

A. Research Questions

Our experiments are aimed at evaluating: (1) Robustness of model output, by recording the top-1 output label for every combination of environment parameters and performing pairwise comparisons; and (2) Robustness of model execution time, by measuring average inference time across executions in our dataset and comparing across different configurations. We investigate the following research questions for evaluating robustness:

Output Label Robustness

RQ1. Label Sensitivity to DL Framework Conversions *Are the output labels of an image recognition model affected when converting the model from a source to a target DL framework?* Both source and target frameworks are one among PyTorch, TF, TFLite, or Keras with $source \neq target$. All conversions are through the intermediate ONNX format. We compare output labels of target against the source for each image to check if any errors were introduced by model conversion. We do this for each of the three image recognition models – MobileNetV2, ResNet101V2, and InceptionV3.

RQ2. Label Sensitivity to Compiler Optimizations *Are the output labels of an image recognition model affected when changing the level of compiler optimization?* We vary the optimization level within TVM between Basic, Default, and Extended and observe if there are any difference in the output label for images in the dataset.

Inference Time Robustness

RQ3. Time Sensitivity to DL Framework Conversion *Are the inference times of an image recognition model affected when converting the model from a source to a target DL framework?*

RQ4. Time Sensitivity to Compiler Optimizations *Are the inference times of an image recognition model affected when changing the level of compiler optimization?* We are aware that differences in inference time is to be expected to some extent when changing compiler optimizations. The goal here is to identify unexpected performance degradation and extent of change with the different compiler optimization levels.

B. Devices

We used four different hardware devices, featuring high-end to low-end GPU accelerators:

- an Intel-based server featuring an Nvidia Tesla K40c (GK11BGL) GPU (*Server*),
- a Nvidia AGX Xavier featuring an Nvidia Volta GPU (*Xavier*),
- a Laptop featuring an Intel(R) GEN9 HD Graphics NEO (*Local*),
- and a mobile-class Hikey 970 board featuring an Arm Mali-G72 GPU (*Hikey*).

For all GPU devices, we generate OpenCL device code, except for the Xavier device where we generate CUDA code, since it does not support OpenCL. We found no accuracy impact between the two programming paradigms, and OpenCL vs. CUDA trade-offs are already explored [52]. We run the test dataset through the model configurations, and take the average inference time.

C. Dataset

We use the ImageNet object detection test dataset [9] in our experiments, consisting of 5500 RGB images that are generally resized to 224×224 pixels, and perform classification of 1000 possible labels and measure inference time on each image. For models native to TensorFlow and TFLite, we observed that models actually used input size of 299, rather than the typical 244. In general, using larger input sizes could increase the potential accuracy of the models, but also the computational requirements they need to perform.

D. Execution Issues

All environment parameter combinations could not be executed with all models due to the following incompatibility issues. First, for ResNet101 sourced from PyTorch, we selected the V1 version the model instead of V2 as the V2 version was not provided in the official PyTorch repository. The version difference may have a larger effect on model inference time when we compare across DL frameworks. Second, regarding MobileNetV2, we experienced problems when executing it on the Xavier device, as we received a `CUDA_ERROR_INVALID_PTX` error, in all cases except when natively sourced from PyTorch. Thus, we do not consider

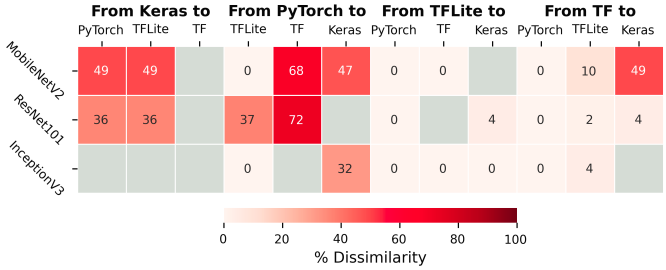

Fig. 6. Pairwise comparison of output labels between *source* and *target* for a given model architecture across all images in the dataset.

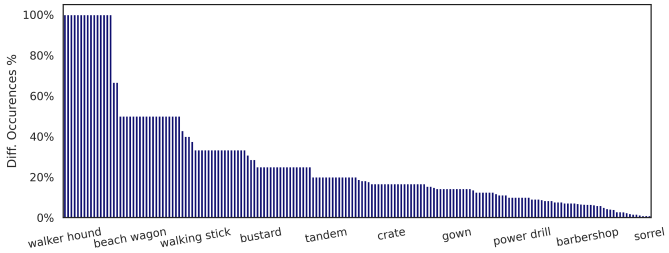

Fig. 7. Percentage of affected images due to library conversions, InceptionV3, TF-to-TFLite conversion.

this device configuration for MobileNetV2 in our experiments. Third, while utilizing the conversion process, we encountered various cases where the source model conversion failed, as presented in Figure 6. This happened due to incompatibility either of the source model with the conversion tool or the generated model operations in TVM.

VI. Results

We present results with respect to discrepancies observed in output labels and inference time over the different DNN model configurations in the context of the research questions presented in Section V.

A. Robustness of Output Label Prediction

1) RQ1: Label Sensitivity to DL Framework Conversions: The results are presented in Figure 6, showing the degree of dissimilarity between source and target models. As can be seen from the empty gray boxes, the conversion tool crashes in 10 out of the 36 conversions across the three DNN models, indicating that the conversion process failed. This happened due to compatibility issues between the conversion tool and a given model architecture, or the source or target DL framework. For instance, we could not convert any Keras models to TensorFlow, due to the `tf2onnx` tool being unable to handle some tensor element values. Additionally, we observe further 10 cases where the conversion succeed without crashing, but the target model gave considerably different labels (over 35%) from the source model. In particular, we observe a 72% discrepancy in the output labels when converting the ResNet101 model from TF to PyTorch.

For conversions between either TF or TFLite and PyTorch, we observe no errors introduced by the conversion process across all models, while when converting TF to TFLite, we see relatively small discrepancies, 0-10%, demonstrating more reliable conversion. For TFLite to TF we had no discrepancies, but had one conversion failure (ResNet101). This relative success is reasonable to expect, since TFLite has overlap with the TensorFlow codebase. However, ideally the differences should all be 0%. Table I shows that the native accuracy of the TensorFlow and TFLite models are all identical, implying that (1) the models are the same, and thus (2) the TFLite authors had 100% success with their conversions. However, we observe divergences using common open source conversion tools with default configurations.

Finally, the conversion of TF models to Keras gives varying results across models, with MobileNetV2 having 49% dissimilarity, ResNet101 having 4%, and InceptionV3 giving a model crash. This points to weaknesses in the conversion tool with certain DNN model architectures.

a) Fault Analysis: We use the **Analysis** part of `DeltaNN` to explore in greater detail the cause of discrepancies across DNN model conversions. To illustrate the analysis in depth, we select one of the models and conversions that results in a discrepancy, InceptionV3 with TF as the source DL framework converted to target TFLite. The discrepancies observed across *source-target* is 4%, and in Figure 7 we show the class breakdown of the images which demonstrated differences, sorted by what proportion of that class showed discrepancies. We highlight a subset of the class labels on the x-axis. We observe that some classes are impacted more than others, with some classes such as "walker hound" disagreeing on 100% of the images. However, this graph is not indicating test set accuracy, instead it is about agreement between the *source* and converted models, which under ideal circumstances we would expect to have equal agreement in all cases.

We also performed inference using the intermediate ONNX format, which is used as part of the conversion and model loading process to TVM. We used *ONNXRuntime* [53] for that purpose. For all selected images, we found that the TF model differed from the TFLite inference results, whereas the TFLite model results were identical to ONNX, as seen in Table II for five images as an example. The ground truth for these images matches the results from the source model, and deviates from TFLite and ONNX. This narrows down the source of the error mainly to the conversion tool from the source TF model to TFLite, i.e., `TFLiteConverter` [54] of the native TFLite API, and less to `tf2onnx` [15], which is widely used in the community (1.8k stars on GitHub). We store the tensor outputs from the source and target models for further analysis.

Next, we performed execution on the *source* and the *target* model utilizing `DeltaNN`'s Debug execution, which relies on TVM's debugger and provides metadata about the execution. Following this process, we perform per-layer activation analysis combining the debugger metadata with metadata of the build process for the *source* and the *target* models. We compare the average differences between the models across layers utilizing parameters (i.e., weights and biases from the convolution layers), per-layer tensor outputs (i.e., activations),

TABLE II
INFERENCE (TOP-1 PREDICTION) OF 5 IMAGENET IMAGES POSING DIFFERENT RESULTS BETWEEN MODELS USING TF, TFLITE CONVERTED FROM TF, AND COMPLEMENTARY ONNX APPLIED ON INCEPTIONV3 AND RUN ON LOCAL DEVICE USING DEFAULT OPTIMIZATION.

Image ID	TF	TFLite (TF)	ONNX
00001219	scooter	moped	moped
00002078	cottontail	llama	llama
00002439	wallet	purse	purse
00003928	black grouse	bee	bee
00004898	wallaby	It. greyhound	It. greyhound

as well as the hyperparameter values for the respective layers. We illustrate this for two images in Figure 8, focusing on the convolutional layers, where *Image 1* generated the same output label across source and target models, whereas *Image 2* produced completely different labels across source and target models for the top-5 predictions. We observe that both images have divergences in their activations, but for *Image 2* the divergences are higher for later layers (layer 170 onwards).

b) Per Layer Activation and Model Parameter Analysis: Figure 8 highlights the difference between intermediate activation maps (i.e., the outputs of individual layers during execution), as well as the differences in the parameters. We would expect the models to behave the same, since they should be the same model architecture and parameters. Our observation is however that the output labels are not always consistent. For *Image 1*, both *source* and *target* versions of the model produce the correct label, even though their intermediate activations are between 0.0 and 0.06 on average. However, for *Image 2* the models disagree on the output label, and for later layers, we observe a higher average difference, up to around 0.13. This may imply that whatever error has occurred may have impacted later layers more, but this does not explain why this is not the case for *Image 1*.

To identify the source of these discrepancies, we examine the differences in the parameters of the models (seen as the green line in Figure 8), which indicates a possible source of the error. Across parameters, we observe a divergence of 0.0003 on average and 0.011 at most. In principle, when we run our conversion tool the parameters should be unchanged, i.e., bit-wise identical from TF to TFLite. Furthermore, we presume that the model parameters are incorrectly copied at some stage of the DL framework model conversion process. With this bug identified and fixed in the conversion tool, we could expect that the difference between the models goes away. We demonstrate this by replacing the parameters of the converted model with the source model within TVM, and observe that 100% of our divergence disappears. However, we cannot assume that these bugs will always be fixed. Thus, we could also mitigate the impact of the error during training by simulating the conversion tool noise into our parameters, so that the model learns to be robust against it.

Therefore, the confirmation of our hypothesis still does not explain why we observed non-uniform divergence in the activation maps and output labels; despite the fact that the parameters had small, relatively uniform noise, and were identical between *Image 1* and *Image 2*. However, if we reason about the underlying operations and mechanics of a DNN model, we can begin to make sense of it. We observe that the impact of these weight errors are cumulative, since weights are generally used repeatedly in multiply-accumulate operations. Then, layers with more such operations (e.g., ReLU activation functions) will be more likely to have higher errors.

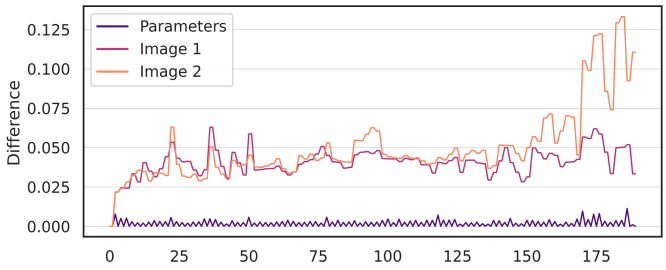

Fig. 8. Layer-wise evaluation of the differences between a model sourced from TensorFlow, and converted to TFLite. "Parameters" shows the mean difference between their weights and biases. *'Image 1'* and *'Image 2'* show models' differences in activations for two inputs.

By complementary examination of the conversion tools involved, we identify that `tf2onnx` did not introduce any weight deviations, but that `TFLiteConverter` was responsible for introducing the fault.

2) RQ2: Label Sensitivity to Compiler Optimizations: We conducted experiments across all DL frameworks and device combinations described in Section V, using only the native DNN model definition and varying the optimization level (Basic, Default and Extended), in order to observe inference time and output label discrepancies.

We found that varying compiler optimization levels causes no discrepancies in output labels for all three models. The lack of discrepancies/sensitivity is notable, since the Extended (`-o4`) level enables unsafe math optimizations that allow code violating IEEE `float` conventions to be generated. The conclusion is that these potential unsafe perturbations were small enough that all three models were resilient to them. It is however worth considering robustness checks with respect to optimization levels in safety-critical domains, in case that unsafe optimizations result in undesirable model outputs. It is also not a foregone conclusion that the ostensibly semantic preserving optimizations of Basic and Default optimization levels would have produced no label divergences, as bugs in compilers are a common occurrence. However, TVM's optimizations do not introduce any errors in our experiments.

B. Robustness of Model Inference Time

1) RQ3: Time Sensitivity to DL Frameworks: Between Keras and PyTorch native models, we observed a 4-16% difference in inference time using the MobileNetV2 model with the Default optimization, deployed on the Server across models, with the largest difference of 16% being seen between Keras and PyTorch, as shown in Figure 10. The differences were confirmed to be significant using one-way ANOVA with

5% significance level. We believe the difference is due to the different graph representation after framework conversion, e.g., one framework may represent a fully-connected layer as a "dense" operation and another may represent it as a "batch matmul"; or some conversion tools may apply some of the graph-level optimizations such as batch-normalization fusion, so that even with a Basic optimization level the model is simplified. We plan to investigate this further in future work.

2) RQ4: Time Sensitivity to Compiler Optimizations: We observed a maximum speedup of 114% in inference time with increasing optimization levels. As part of our statistical analysis, we confirmed the observation to be significant using one-way ANOVA with 5% significance level. This is not surprising, as different optimization settings have a direct effect on code efficiency. Interestingly, there were instances where increased optimization led to a slowdown in inference time. For instance, MobileNetV2 from Keras and Extended optimization was 81% slower than Basic on the Hikey device. We also confirmed our observations using one-way ANOVA.

To explore the impact of the compiler in greater detail, we enabled optimization passes individually, taking each optimization concept and applying it to the model separately, rather than using multiple optimizations together as the $-O\mathrm{X}$ bundles that we use at a high level. We conducted an analysis on 100 images, using one optimization pass per-case, to understand which optimizations contributed to speedup or slowdown in model inference time for ResNet101 and InceptionV3 using TensorFlow and PyTorch DL frameworks[2].

We found that no single optimization led to a significant inference time change for this experiment, suggesting that the non-trivial interactions between optimization passes are what contribute to these changes, making analysis and performance optimization more challenging. For ResNet101, the optimization which provides convolution operators' scale axis folding degraded the performance by 2.71% on the Local device, while the combination of parallel operators had a positive impact of up to 2.82% compared to using Basic optimizations alone. With InceptionV3, constant folding had a positive impact of 4.9%, while combining parallel operators *degraded* the performance by up to 5.47% compared to Basic optimizations alone. The difference in effect of optimizations between InceptionV3 and ResNet101 is likely due to the difference in their model architecture and data flows. We plan to analyze the reasons behind this in future work.

However, we observed that the combination of optimization strategies can lead to significant performance degradation under certain contexts, such as low-end hardware acceleration devices. Figure 9 shows the percentage difference in inference time for Basic versus Extended optimization on different devices and models with PyTorch, as an indicative example. For each device, we find that times generally improve with increased optimization in the range of 3.8-8.4% for Server, and 17-54% for Local. Increased optimizations on Hikey, however, had a 81.8% slowdown (confirmed with One-way ANOVA

[2]Our results can be found at: https://github.com/luludak/deltann-results

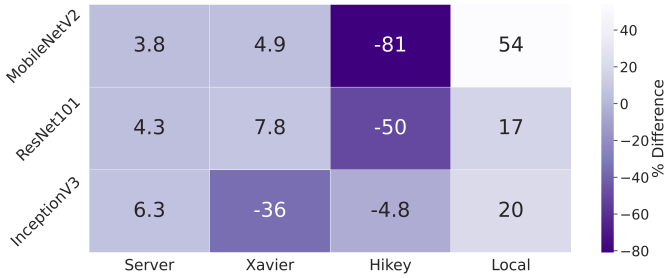

Fig. 9. Inference time differences (%) between Basic and Extended optimizations across devices, with native models from PyTorch.

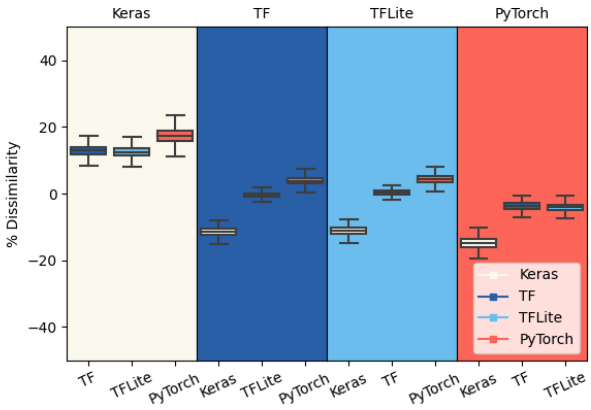

Fig. 10. Inference time differences (%) between DL frameworks on Server, for MobileNetV2, with Default Optimization.

5%). The Xavier device also had a 36% slowdown (confirmed with One-way ANOVA 5%) when increasing the optimization level from Basic to Extended on InceptionV3 model. For low to mid-range devices, Xavier and Hikey, we experienced a slowdown with increased optimization, and believe that the limited GPU memory poses a problem for the optimizations with parallel operations in the Extended optimization setting, leading to additional wait times, context switches, and GPU data transfer time, which result in a slowdown. Investigation of cache behavior, data transfer times between the CPU and GPU, and processor idle times to clearly identify the reasons for slowdown is subject for future work.

C. Threats To Validity

There are five main threats to validity in our experiments:

1) We only evaluate robustness using three image recognition models that are widely used. The results are model dependent as seen in our experiments and will likely vary on other models;
2) We use the ImageNet [9] object detection test dataset for our experiments, which we believe adequately stresses configurations. Other datasets may yield different robustness results on the models considered;
3) Model pre-processing is crucial for model performance [55], and models may give suboptimal performance if given data with ineffective and erroneous pre-

processing. We use the recommended pre-processing for each model and DL framework from the official repositories extracted;

4) Beyond the DL framework conversions explored in our results, we also have a "hidden" conversion step, i.e., importing models into Apache TVM, which itself may introduce errors. To ensure that errors are not introduced before loading each model into TVM, we generate "target outputs" from their source framework using an indicative number of random image samples. After importing into TVM, we confirmed that we match the target outputs, however this may not guarantee that the import process is entirely bug-free;
5) We consider the potential deviations of inference time measurement. To ensure that time deviations are taken into account, we repeat inferences 10 times for each image and use the average inference time across each run across a small-scale test dataset, verifying that no deviations happen on scaling. Note that non-trivial medium-term cache behavior may cause the inference time to change over time with repeated inferences, and depending on the deployment scenario of interest, only the "first" inference time may be of interest, or the inference time of the 'Nth' sample where N is a large value.

VII. Lessons Learned

Our empirical study exploring the effect of changing computational environment parameters revealed the following findings:

a) Failures in DL Framework Conversion: Automated conversion of models between DL frameworks can introduce significant output label discrepancies. We observed up to 72% output label dissimilarities when converting PyTorch to Keras for MobileNetV2. In addition, converting Keras to TF generated failures for all three models under test. These errors can be introduced in model weights, parameters, graph and architecture representation during the conversion process. Our analysis revealed errors in model weights introduced by `TFLiteConverter` when converting from the source model (TF) to TFLite, which can be fixed by correctly copying over the source model weights.

b) DL Framework Conversion - Impact on Model Inference Time: Changing the DL framework used to generate the model can have a considerable effect on model inference time. The extent of this impact depends on other environment parameters. Inference time impact varied from 1-16% with the largest impact (16%) observed between Keras and PyTorch.

c) Performance Degradation from Compiler Optimizations: Compiler optimizations are generally expected to improve the performance of a model. However, our findings indicate that for certain scenarios defined by the device, model and library, compiler optimizations can be detrimental to model inference time. In our experiments, we observed this performance degradation to the greatest extent when applying Extended optimization to MobileNetV2 for the Hikey device, which resulted in 81% performance degradation when compared to a lower optimization level using Basic.

Finally, it is important to note that in safety-critical applications, the consequences of the above sensitivities can be crucial. Therefore, it is essential that framework, compiler, and hardware communities, along with the developers of DNN models are aware of these sources of error, and test their systems for robustness to computational environment changes. Currently, there is no regulation or benchmarking of DNN model performance and accuracy for environment parameter configurations. The results from our study indicate that assessing sensitivity to environment parameters is an important consideration during model development and use.

VIII. Conclusion

We introduced the `DeltaNN` Differential testing framework to explore the impact of computational environment parameters on image recognition models. In particular, we study the effect of converting between popular deep learning frameworks (TensorFlow, Keras, TFLite, PyTorch), compiler optimization settings, and hardware devices on the output labels of three widely used image recognition models. We also monitor the impact of these parameters on model inference time. Overall, we find that conversions between DL frameworks significantly impact output labels of the DNN models by up to 72%. Our framework also provides analysis capabilities for label discrepancies stemming from framework conversions.

References

[1] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems," in 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2018, pp. 132–142.

[2] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in Proceedings of the 40th international conference on software engineering, 2018, pp. 303–314.

[3] J. Guo, Y. Zhao, H. Song, and Y. Jiang, "Coverage Guided Differential Adversarial Testing of Deep Learning Systems," IEEE Transactions on Network Science and Engineering, vol. 8, no. 2, pp. 933–942, Apr. 2021.

[4] V. Yaneva, A. Rajan, and C. Dubach, "Compiler-Assisted Test Acceleration on GPUs for Embedded Software," in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 35–45.

[5] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "VERIFAI: A toolkit for the Design and Analysis of Artificial Intelligence-Based Systems," arXiv preprint arXiv:1902.04245, 2019.

[6] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation," CoRR, vol. abs/1801.04381, 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," CoRR, vol. abs/1603.05027, 2016. [Online]. Available: http://arxiv.org/abs/1603.05027

[8] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," CoRR, vol. abs/1512.00567, 2015. [Online]. Available: http://arxiv.org/abs/1512.00567

[9] O. Russakovsky, J. Deng et al., "ImageNet Large Scale Visual Recognition Challenge," International Journal of Computer Vision (IJCV), vol. 115, no. 3, pp. 211–252, 2015.

[10] T. Chen, T. Moreau et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Oct. 2018, pp. 578–594.

[11] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O'Boyle, and A. Storkey, "Characterising Across-Stack Optimisations for Deep Convolutional Neural Networks," in IISWC, 2018, pp. 101–110.

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

[13] A. Paszke, S. Gross et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," CoRR, vol. abs/1912.01703, 2019. [Online]. Available: http://arxiv.org/abs/1912.01703

[14] M. A. et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[15] "TF2ONNX," https://github.com/onnx/tensorflow-onnx, 2022, [Accessed 15-Feb-2023].

[16] "onnx2keras," https://github.com/gmalivenko/onnx2keras, 2023.

[17] "onnx2torch," https://github.com/ENOT-AutoDL/onnx2torch, 2023, [Accessed 15-Feb-2023].

[18] Y. Liu, C. Chen, R. Zhang, T. Qin, X. Ji, H. Lin, and M. Yang, "Enhancing the Interoperability between Deep Learning Frameworks by Model Conversion," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Nov. 2020, pp. 1320–1330.

[19] "Open Neural Network Exchange," https://onnx.ai/, 2022.

[20] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: Efficient Primitives for Deep Learning," arXiv preprint arXiv:1410.0759, 2014.

[21] "Compute Library," Arm Software, Aug. 2022.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in Advances in Neural Information Processing Systems 30, 2017, pp. 5998–6008.

[23] Z. Dai, H. Liu, Q. V. Le, and M. Tan, "CoAtNet: Marrying Convolution and Attention for All Data Sizes," in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 3965–3977.

[24] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer, "Scaling Vision Transformers," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 12 104–12 113.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," CoRR, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[26] Y.-C. Chiu, C.-Y. Tsai, M.-D. Ruan, G.-Y. Shen, and T.-T. Lee, "Mobilenet-SSDv2: An Improved Object Detection Model for Embedded Systems," in 2020 International Conference on System Science and Engineering (ICSSE), Aug. 2020, pp. 1–5.

[27] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," 2019.

[28] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," CoRR, vol. abs/1705.06640, 2017. [Online]. Available: http://arxiv.org/abs/1705.06640

[29] F. Ayaz, I. Zakariyya, J. Cano, S. L. Keoh, J. Singer, D. Pau, and M. Kharbouche-Harrari, "Improving Robustness Against Adversarial Attacks with Deeply Quantized Neural Networks," in 2023 International Joint Conference on Neural Networks (IJCNN), 2023, pp. 1–8.

[30] C. Shorten and T. M. Khoshgoftaar, "A Survey on Image Data Augmentation for Deep Learning," Journal of big data, vol. 6, no. 1, pp. 1–48, 2019.

[31] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking State-of-the-Art Deep Learning Software Tools," CoRR, vol. abs/1608.07249, 2016. [Online]. Available: http://arxiv.org/abs/1608.07249

[32] L. Liu, Y. Wu, W. Wei, W. Cao, S. Sahin, and Q. Zhang, "Benchmarking Deep Learning Frameworks: Design Considerations, Metrics and Beyond," in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1258–1269.

[33] B. Collie, P. Ginsbach, J. Woodruff, A. Rajan, and M. F. O'Boyle, "M3: Semantic API Migrations," in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 90–102.

[34] P. Stratis and A. Rajan, "Speeding Up Test Execution with Increased Cache Locality," Software Testing, Verification and Reliability, vol. 28, no. 5, p. e1671, 2018.

[35] N. Mahmoud, Y. Essam, R. Elshawi, and S. Sakr, "DLBench: An Experimental Evaluation of Deep Learning Frameworks," in 2019 IEEE International Congress on Big Data, 2019, pp. 149–156.

[36] Y. Wu, L. Liu, C. Pu, W. Cao, S. Sahin, W. Wei, and Q. Zhang, "A Comparative Measurement Study of Deep Learning as a Service Framework," IEEE Transactions on Services Computing, vol. 15, no. 1, pp. 551–566, 2022.

[37] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 1027–1038.

[38] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep Learning Library Testing via Effective Model Generation," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, p. 788–799.

[39] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated Testing for Deep Learning Frameworks," in 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020, pp. 486–498.

[40] M. Openja, A. Nikanjam, A. H. Yahmed, F. Khomh, and Z. M. J. Jiang, "An Empirical Study of Challenges in Converting Deep Learning Models," in 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022, pp. 13–23.

[41] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A Comprehensive Study of Deep Learning Compiler Bugs," in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, p. 968–980.

[42] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1110–1121.

[43] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "Exploring Effects of Computational Parameter Changes to Image Recognition Systems," arXiv preprint arXiv:2211.00471, 2022.

[44] ——, "Assessing Robustness of Image Recognition Models to Changes in the Computational Environment," in NeurIPS ML Safety Workshop, 2022. [Online]. Available: https://openreview.net/forum?id=-7DjNGvdpx

[45] ——, "Fault Localization for Buggy Deep Learning Framework Conversions in Image Recognition," in 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Sep. 2023.

[46] J. Roesch, S. Lyubomirsky et al., "Relay: A High-Level Compiler for Deep Learning," 2019.

[47] "IEEE Standard for Floating-Point Arithmetic," IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84, 2019.

[48] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to Optimize Tensor Programs, booktitle = Advances in Neural Information Processing Systems 31," 2018, pp. 3393–3404.

[49] L. Zheng, C. Jia et al., "Ansor: Generating High-Performance Tensor Programs for Deep Learning," in 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 863–879.

[50] P. Gibson and J. Cano, "Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation," in Proceedings of the International Conference on Parallel Architectures and CCompilation Techniques (PACT), 2023, p. 28–39.

[51] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A Large-Scale Hierarchical Image Database," in 2009 IEEE conference on computer vision and pattern recognition, 2009, pp. 248–255.

[52] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," in 2011 International Conference on Parallel Processing, 2011, pp. 216–225.

[53] O. R. developers, "ONNX Runtime," https://onnxruntime.ai/, 2021, [Accessed 15-Feb-2023].

[54] "Convert TensorFlow Models," https://www.tensorflow.org/lite/models/convert, 2023, [Accessed 26-June-2023].

[55] J. Camacho-Collados and M. T. Pilehvar, "On the Role of Text Preprocessing in Neural Network Architectures: An Evaluation Study on Text Categorization and Sentiment Analysis," CoRR, vol. abs/1707.01780, 2017. [Online]. Available: http://arxiv.org/abs/1707.01780