Fault Localization for Buggy Deep Learning Framework Conversions in Image Recognition

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*—When deploying Deep Neural Networks (DNNs), developers often convert models from one deep learning framework to another (e.g., TensorFlow to PyTorch). However, this process is error-prone and can impact target model accuracy. To identify the extent of such impact, we perform and briefly present a differential analysis against three DNNs widely used for image recognition (MobileNetV2, ResNet101, and InceptionV3) converted across four well-known deep learning frameworks (PyTorch, Keras, TensorFlow (TF), and TFLite), which revealed numerous model crashes and output label discrepancies of up to 72%. To mitigate such errors, we present a novel approach towards fault localization and repair of buggy deep learning framework conversions, focusing on pre-trained image recognition models. Our technique consists of four stages of analysis: 1) conversion tools, 2) model parameters, 3) model hyperparameters, and 4) graph representation. In addition, we propose various strategies towards fault repair of the faults detected. We implement our technique on top of the Apache TVM deep learning compiler, and we test it by conducting a preliminary fault localization analysis for the conversion of InceptionV3 from TF to TFLite. Our approach detected a fault in a common DNN converter tool, which introduced precision errors in weights, reducing model accuracy. After our fault localization, we repaired the issue, reducing our conversion error to zero.**

I. Introduction

Deep Neural Network (DNN) models, trained using a given deep learning (DL) framework (such as PyTorch [1], TensorFlow (TF) [2]), can be converted to a different DL framework (such as Keras [3]). Common reasons for this conversion include 1) deployment on resource-constrained environments such as IoT devices, which may require lightweight DL frameworks (e.g., TFLite), and 2) support for a wider set of features, that allow more in-depth model modification and optimization, such as explicitly defining forward propagation implementation. Conversion of DNN models between DL frameworks is facilitated by automated conversion processes using tools such as `tf2onnx` [4], `onnx2keras` [5], `onnx2torch` [6], and `MMdnn` [7]. However, this conversion process can introduce faults [8]–[11], which can make the converted models undeployable or reduce performance on their target task [12], [13].

In order to mitigate this problem, we propose an automated approach for fault localization and repair of faults introduced by the DL framework conversion process. We focus on DL framework conversion used in deployment of pre-trained image recognition models, utilized for image classification tasks. Note that our methodology is agnostic to the DNN architecture and can be applied to other tasks such as image segmentation. Our approach detects faults introduced in model parameters, hyperparameters, and the model graph, as the primary coefficients that define DNN model behaviour. The proposed approach performs analysis and comparison against source and target model parameters and hyperparameters, as well as comparison of layer activations for inputs resulting in output label discrepancies against the source and the target model. Additionally, we explore potential discrepancies introduced by graph transformations between the source and the target model during the conversion process. Then, we propose a set of strategies to mitigate conversion faults such as the replacement of model parameters of the target model with those from source, and applying graph transformations that eliminate the error from the converted model. Finally, we present an evaluation example of the conversion process for the InceptionV3 model converted from TensorFlow to TFLite. Our technique is able to detect precision errors in weights related to convolutional layers introduced by the TFLiteConverter tool, with value deviations of up to 0.01 between *Source* and *Target*, which, although small, affected the model performance.

Overall, the main contributions of this paper are: 1) A novel method to systematically localize faults in DL framework conversion processes, and 2) repair strategies for said faults.

Authors, Ajitha Rajan and Nikolaos Louloudakis, would like to acknowledge 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.

II. Related Work

A number of studies have been conducted related to faults introduced in the deployment process of DNNs. For instance, a study of 3023 Stack Overflow posts built a taxonomy of faults and highlighted the difficulty of DNN deployment [12]. Another study explores the effect of DNN faults on mobile devices by identifying 304 faults from GitHub and Stack Overflow [13], while other studies provide surveys on existing contributions towards machine learning testing components, workflow and application scenarios [14]. In addition, there are works related to exploring the test oracle problem in the context of machine learning [15], [16]. In terms of fault localization, DeepCover [17] attempts to apply a statistical fault localization approach, focusing on the extraction of heatmap explanations from DNN inputs. DeepFault [18] focuses on a suspiciousness-oriented spectrum analysis algorithm in order to detect parts of the DNN that can be responsible for faults, while it also proposes a method for adversarial input generation. DeepLocalize [19] attempts to detect faults in DNNs by converting them to an imperative representation

and then performing dynamic analysis on top of its execution traces. Regarding fault localization in DL framework specifically, CRADLE [20] tries to detect faults introduced by DL frameworks by performing model execution graph analysis. LEMON [21] leverages the metrics used by CRADLE for its analysis to apply mutation testing.

Although the above works attempt to overcome fault localization challenges for DNNs, none of them considers model conversions as a factor of fault introduction in DNNs and, therefore, no previous work explores this problem. However, several tools exist to ease the DL model conversion process, including MMdnn [7], tf2onnx [4], onnx2keras [5], onnx2torch [6], and tflite2onnx [22]. There are also some native APIs for DL framework conversion to ONNX found within PyTorch [1] and TFLite [2]. These tools are extensively used, as they all have more than 100 stars on their GitHub repositories. In addition, a recent study by Openja et al. [23] highlights the challenges of the conversion process, while our preliminary work [8], [24] explores the robustness of DNNs against different computational environment aspects, including DL framework conversions. However, the impact of DL framework conversions on DNN model correctness is not explored in-depth in the literature.

To the best of our knowledge, this paper is the first attempt focusing on the error proneness, fault localization, and repair of DL framework conversions for DNN models. We focus on image recognition models as a a starting point, but our work is applicable to DNNs used in other domains.

III. Motivation

To observe the potential impact of DNN model conversions, we conducted an initial evaluation using three widely used image recognition models of varying size and architectural complexity: MobileNetV2 [25], ResNet101 [26], and InceptionV3 [27]. For each model, we used pre-trained versions from official repositories of four different DL frameworks: TensorFlow [2], TFLite [2], Keras [28], and PyTorch [1]. We refer to the pre-trained model of each DL framework as the *Source* model. As a result, we have 4 *Source* versions for each of our 3 models. We then convert each *Source* model to use a different DL framework; we refer to the converted model as *Target*. To implement the conversion, we use tools that convert the *Source* model either directly to *Target*, or to the ONNX [29] format, a popular model representation format that is designed to act as a common interchange format between frameworks. 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 [4]. We then convert from ONNX to *Target* using a number of widely used libraries, such as onnx2keras [5] and onnx2torch [6]. Following the conversion process, we perform pairwise comparison between *Source* and *Target* model inferences using the ILSVRC 2017 object detection test dataset [30], in order to detect discrepancies in classification introduced by the conversion process.

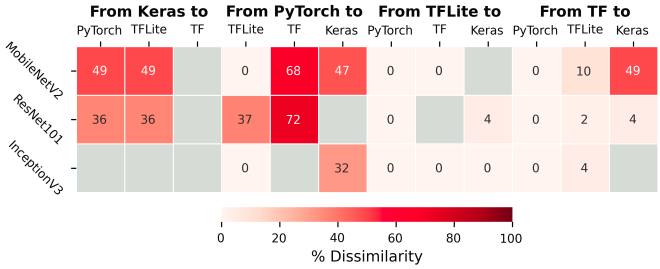

Fig. 1. Pairwise comparison of output labels between *Source* and converted *Target* models.

For each image of the dataset, we compare the output labels of *Target* against *Source* to check if any errors were introduced by the model conversion. The proportion of output label dissimilarities between *Source, Target* pairs across all images in the dataset is shown in Figure 1. As can be seen from the empty grey cells, 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. Additionally, we observe a further 10 cases where the conversion succeeded without crashing, but the *Target* model gave considerable label discrepancies in comparison to the *Source* model (over 35%), with a maximum observed discrepancy of 72% in the output labels when converting the ResNet101 model from PyTorch to TF. The conversion of TensorFlow models to Keras gives varying results across models, with MobileNetV2 having a considerable amount of dissimilarity (49%), ResNet101 having 4% dissimilarity, and InceptionV3 leading to a crash. This points to weaknesses in the conversion tool with certain model architectures. Finally, for conversions between TF or TFLite to PyTorch no conversion errors were observed, while when converting TF to TFLite across all models we see relatively small discrepancies, 0-10%, demonstrating a more reliable conversion. However, even small discrepancies may have non-negligible impact when these models are used in safety critical applications.

From Figure 1, it is clear that the conversion process is error-prone and there is a need for a technique to localize and fix faults introduced by DL frameworks converters. We discuss our approach for fault localization and repair in Section IV.

IV. Methodology

The stages in our proposed approach for fault localization and repair are shown in Figure 2. It starts by converting a given model from *Source* to *Target* DL framework, then performs inference over both models over an input dataset, compares output labels to identify parts of the dataset which led to different outputs, and finally performs a fault localization and repair process where possible/appropriate. We describe the fault localization and repair steps below.

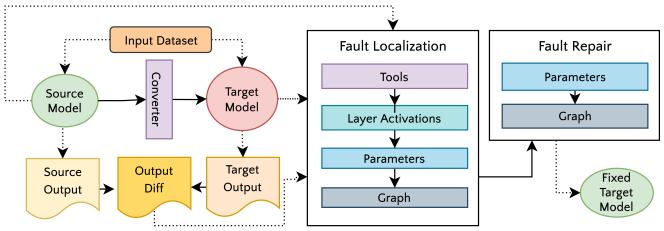

Fig. 2. Fault Localization & Repair Pipeline.

A. Fault Localization & Repair

We start by examining the tools involved in the DNN model conversion to identify if the fault is introduced during conversion from *Source* to the ONNX format, or from ONNX to *Target*. We then complement this analysis by examining differences for three key DNN model architecture aspects: 1) Hyperparameters (such as kernel and batch size), 2) parameters (such as weights and biases), and 3) the model's graph structure (such as operations and their connections). We describe these steps below.

1) Conversion Tools Analysis: Following the DNN model conversion process, when discrepancies are observed between the *Source* and the *Target* model, it is important to identify which part of the conversion process was responsible. The conversion process typically uses more than one tool, e.g., one for conversion to ONNX format from *Source*, and another to convert from ONNX to *Target*. We explore this over a subset of the images that presented discrepancies between *Source* and *Target* while also considering the intermediate ONNX representation. In particular, we consider a subset of the dataset inputs that presented different outputs between *Source* and *Target* models. In addition, we perform inference using the ONNX intermediate representation from the conversion process, and compare the outputs against the *Source* and *Target*. If the conversion process involves multiple steps, we repeat the process for all intermediate steps, so that we can better localize where the fault is introduced.

2) Parameter Analysis: A correct DNN model conversion should result in a target model having the same parameters, and producing the same output as the source model. However, if for some reason the parameters are altered (e.g., due to a precision error in the conversion process), this could potentially affect the model's correctness.

To detect this fault, we take the *Source* and *Target* model variants, and extract their parameters (e.g., weights and biases). We then compare the parameters between model variants across layers of the same type (e.g., convolutions, bias additions, etc), by computing $\text{mean}(\text{abs}(P_{source} - P_{target}))$, where P_{source}, P_{target} are the parameters of the source and converted target models, respectively. The value of the mean difference is expected to be zero when the model parameters are unaffected, and any other value indicates that there is a difference across the parameters in a specific layer, which is a potential cause for bugs.

3) Hyperparameter Analysis: Much like parameters, incorrectly converted hyperparameters are another potential source of error. For example, we would expect for a convolutional layer, the padding, strides, dilation, and other configurations would remain unchanged during a conversion. However, a difference could indicate a potential source of error and is marked for further evaluation in our fault localization approach.

4) Layer Analysis: To detect faults that occur on a specific layer, we propose a layer-based dynamic analysis approach. Using a small, indicative subset of 5 images from the dataset that present output label discrepancies between *Source* and *Target* models, we perform inference and compare per-layer activations between the models. For each input, we compute the mean of differences found across activations for each layer. We then further examine the layers affected sequentially, starting from the first layer and moving forward. We focus on errors in the graph representation of that layer, as well as on implementation details. In particular, we examine if a layer or its graph neighbors are implemented in a different manner or are using different but equivalent operations (e.g., *reshape* and *flatten*) between the *Source* and *Target* model.

5) Fault Repair: Once a difference is detected, we attempt one of the following options based on the location of the difference for fault repair.

(a) For differences in model hyperparameters, weights, and biases, the respective values from source can be replaced by the target model. Since the conversion process should preserve those values, then the replacement in the target model should resolve the observed differences.

(b) For differences detected in layer activations, there are a number of measures that can be applied. First, a set of mappings can be applied in order to perform in-place replacement of parts of the graph that should behave similarly, but differences in implementation (such as the selection of a different layer type, or the addition of extra redundant layers) could cause differences in layer outputs. For example, we observed cases (e.g., MobileNetV2, PyTorch-to-Keras conversion) where the *flatten* layer was replaced by a *reshape* layer by the converter tools. We instruct a layer replacement to the target based on the layer of the source model, while adjusting tensor inputs and outputs to preserve model validity. In addition, if there are extra nodes added close to the layer affected, they could be modified and removed as an attempt to eliminate errors. For instance, we observed the addition of some padding layers to the target model for a number of conversions (e.g., MobileNetV2, TF-to-PyTorch conversion). A potential fix is to simply remove this node.

Our current approach has limitations for cases where whole sub-graphs in the *Target* model have completely different structure than the *Source*. A replacement in this scenario is non-trivial and is subject to consideration for future work. Once a fix is applied, inference is performed with the target model against the inputs causing discrepancies, and the behavior is monitored. If an improved result is detected for some or all of the images, then the fix is considered successful.

B. Implementation Details

Our methodology is implemented using Apache TVM [31], a cross-platform machine learning compiler framework. We use TVM in order to build and perform inference for the *Source* and *Target* models, while we extract the graph parameters, graph structure provided by each model for weights, biases, and hyperparameters utilizing the model static parameters and graph description metadata generated across the build process. We also use ONNXRuntime [32] to perform intermediate representation inference. In addition, we utilize the TVM Debugger to extract layer activations upon inference, as well as set specific inputs and extracting targeted outputs from hidden layers. The TVM debugger was also used in order to achieve model repair strategies, such as replacing weights, biases, and hyperparameters. For the graph modification part, we utilized the ONNX [29] API in combination with ONNX-Modifier [33] in order to apply graph modifications. We also used Netron [34] for DNN graph observation purposes.

C. Preliminary Evaluation

As an initial case study, we consider the conversion of InceptionV3 using TensorFlow (TF) as *Source* and converting it to TFLite as *Target*. The conversion involved two utilities, the native API of *TFLite* (*TFLiteConverter*), and *tf2onnx*. We observed label differences between *Source* and *Target* models for 4% of the input images (240 out of 5500 images). We were interested in this particular case study because the conversion error is low but still present in a small number of images. This "subtle" failure is of particular interest for safety critical applications, where edge case behavior is more important.

For fault localization, we start by performing an analysis of the conversion tools on the images showing label differences. As seen in Table I, for three sample images label differences occur when converting models from TF to TFLite, but not in the conversion to ONNX. As a result, we find that the TFLiteConverter is the problematic part of this particular conversion process. We perform further investigation of this tool in the next steps.

TABLE I
TOP-1 INFERENCE OF IMAGES FOR INCEPTIONV3 USING TF, INTERMEDIATE ONNX AND TFLITE CONVERTED FROM TF.

Image ID	TF	TFLite (TF)	ONNX
Image 1	drum	drum	drum
Image 2	wallet	purse	purse
Image 3	wallaby	It. greyhound	It. greyhound

We then proceed with *parameters* and *layers* analysis between *Source* and *Target* to further examine the effects and the potential reasons for the problem. We consider an image presenting no discrepancies and two images presenting minor and major label discrepancies (by calculating and comparing Kendall's Tau coefficient [35] for the top-5 inference labels). We present the results in Figure 3, where the *Parameters* line indicates the mean of differences per-layer (x-axis) in parameters for two types of layers, convolutions and bias additions. The remaining lines depict the differences in activations (mean of tensor values comparison) for each layer between *Source* and *Target*. Image 1 presented no discrepancies, Image 2 presented small discrepancies, and Image 3 presented major discrepancies, measured using Kendall's Tau coefficient. We observe layer 2 started presenting discrepancies between *Source* and *Target* for all images under test, affecting the model early in the process. Additionally, there is a spike in the difference observed in layers 170 onwards for Image 3 (which presented large discrepancies between *Source* and *Target*). We examined if the cause of the discrepancy was errors introduced in the model weights while using TFLiteConverter in the conversion process. In particular, we performed a manual *Source* and *Target* model parameters inspection using Netron [34], which confirmed the fault localization finding, as we observed precision errors in the generated ONNX graph from *Source*.

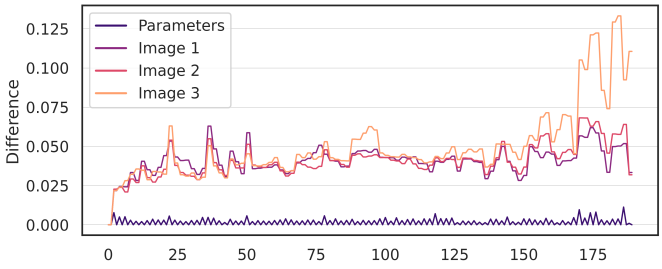

Fig. 3. Layer-wise evaluation of the differences between InceptionV3 model sourced from TensorFlow, and converted to TFLite. *Parameters* shows the mean difference between their weights and biases for convolutional and bias addition layers. *Image 1*, *Image 2*, and *Image 3* show models' differences in activations for two inputs across *Source* and *Target* models.

In order to fix the error, we replaced the model weights of the *Target* model with those from the *Source*, and performed inference against the subset of images presenting discrepancies between the models. The outputs of the updated *Target* were identical to the original *Source*, resolving the issue, and proving its cause.

V. CONCLUSIONS AND FUTURE WORK

We presented a novel fault localization approach for errors encountered during DL framework conversion in image recognition models. It focuses on key DNN model elements such as parameters, hyperparameters, and graph architecture. We also propose strategies to repair the detected errors, such as correcting corrupted model weights. As an example, we examined InceptionV3 when converted from TF to TFLite, which resulted in discrepancies for a small fraction of the input images. We used our approach to localize the conversion bug and fix it. As future work, we aim to evaluate our approach against all conversion tools in Figure 1 and other image recognition models. We will also apply it to other DL tasks such as object detection. Finally, we plan to expand our fault repair strategies to address conversion errors that cause significant changes in *Source* and *Target* model graphs.

REFERENCES

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. 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," CoRR, vol. abs/1912.01703, 2019. [Online]. Available: http://arxiv.org/abs/1912.01703

[2] 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, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[3] F. Chollet, "Keras," 2015. [Online]. Available: https://keras.io/

[4] "tf2onnx," https://github.com/onnx/tensorflow-onnx, 2019, [Accessed 6-June-2023]. [Online]. Available: https://github.com/onnx/tensorflow-onnx

[5] "onnx2keras," https://github.com/gmalivenko/onnx2keras, 2019. [Online]. Available: https://github.com/gmalivenko/onnx2keras

[6] "onnx2torch," https://github.com/ENOT-AutoDL/onnx2torch, 2021, [Accessed 6-June-2023]. [Online]. Available: https://github.com/ENOT-AutoDL/onnx2torch

[7] 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.

[8] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "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

[9] 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.

[10] 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.

[11] "TensorFlow - Frequently Asked Questions," 2021. [Online]. Available: https://www.tensorflow.org/lite/guide/faq

[12] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A Comprehensive Study on Challenges in Deploying Deep Learning Based Software," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 750–762.

[13] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, "An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 674–685.

[14] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 1–36, 2022.

[15] A. Fontes and G. Gay, "Using Machine Learning to Generate Test Oracles: A Systematic Literature Review," in Proceedings of the 1st International Workshop on Test Oracles, 2021, p. 1–10.

[16] F. Tsimpourlas, A. Rajan, and M. Allamanis, "Supervised Learning Over Test Executions as a Test Oracle," in Proceedings of the 36th Annual ACM Symposium on Applied Computing, 2021, pp. 1521–1531.

[17] Y. Sun, H. Chockler, X. Huang, and D. Kroening, "Explaining Deep Neural Networks Using Spectrum-Based Fault Localization," CoRR, vol. abs/1908.02374, 2019. [Online]. Available: http://arxiv.org/abs/1908.02374

[18] H. F. Eniser, S. Gerasimou, and A. Sen, "DeepFault: Fault Localization for Deep Neural Networks," in Fundamental Approaches to Software Engineering, 2019, pp. 171–191.

[19] M. Wardat, W. Le, and H. Rajan, "DeepLocalize: Fault Localization for Deep Neural Networks," in Proceedings of the 43rd International Conference on Software Engineering, 2021, p. 251–262.

[20] 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.

[21] 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.

[22] "tflite2onnx," https://github.com/zhenhuaw-me/tflite2onnx, 2020, [Accessed 6-June-2023]. [Online]. Available: https://github.com/zhenhuaw-me/tflite2onnx

[23] 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.

[24] 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.

[25] 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

[26] 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

[27] 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

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

[29] "Open Neural Network Exchange," https://onnx.ai/, 2019. [Online]. Available: https://onnx.ai/

[30] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," International Journal of Computer Vision (IJCV), vol. 115, no. 3, pp. 211–252, 2015.

[31] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "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.

[32] O. R. developers, "ONNX Runtime," https://onnxruntime.ai/, 2018, [Accessed 6-June-2023].

[33] "ONNX Modifier," 2022. [Online]. Available: https://github.com/ZhangGe6/onnx-modifier

[34] "Netron App," https://github.com/lutzroeder/netron, 2018, [Accessed 6-June-2023]. [Online]. Available: https://github.com/lutzroeder/netron

[35] M. G. Kendall, "A New Measure of Rank Correlation," Biometrika, vol. 30, no. 1/2, pp. 81–93, 1938. [Online]. Available: http://www.jstor.org/stable/2332226