Automatic Fault Detection for Deep Learning Programs Using Graph Transformations

Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, Foutse Khomh

Automatic Fault Detection for Deep Learning Programs Using Graph Transformations

AMIN NIKANJAM[*], K. N. Toosi University of Technology, Iran and SWAT Lab., Polytechnique Montreal, Canada
HOUSSEM BEN BRAIEK[*], SWAT Lab., Polytechnique Montreal, Canada
MOHAMMAD MEHDI MOROVATI, SWAT Lab., Polytechnique Montreal, Canada
FOUTSE KHOMH, SWAT Lab., Polytechnique Montreal, Canada

Nowadays, we are witnessing an increasing demand in both corporates and academia for exploiting Deep Learning (DL) to solve complex real-world problems. A DL program encodes the network structure of a desirable DL model and the process by which the model learns from the training dataset. Like any software, a DL program can be faulty, which implies substantial challenges of software quality assurance, especially in safety-critical domains. It is therefore crucial to equip DL development teams with efficient fault detection techniques and tools. In this paper, we propose *NeuraLint*, a model-based fault detection approach for DL programs, using meta-modelling and graph transformations. First, we design a meta-model for DL programs that includes their base skeleton and fundamental properties. Then, we construct a graph-based verification process that covers 23 rules defined on top of the meta-model and implemented as graph transformations to detect faults and design inefficiencies in the generated models (i.e., instances of the meta-model). First, the proposed approach is evaluated by finding faults and design inefficiencies in 28 synthesized examples built from common problems reported in the literature. Then *NeuraLint* successfully finds 64 faults and design inefficiencies in 34 real-world DL programs extracted from Stack Overflow posts and GitHub repositories. The results show that *NeuraLint* effectively detects faults and design issues in both synthesized and real-world examples with a recall of 70.5 % and a precision of 100 %. Although the proposed meta-model is designed for feedforward neural networks, it can be extended to support other neural network architectures such as recurrent neural networks. Researchers can also expand our set of verification rules to cover more types of issues in DL programs.

CCS Concepts: • **Software and its engineering → Model-driven software engineering**; **Software testing and debugging**.

Additional Key Words and Phrases: Graph transformations, Model-based verification, Deep learning, Fault Detection.

[*]Both authors contributed equally to this research.

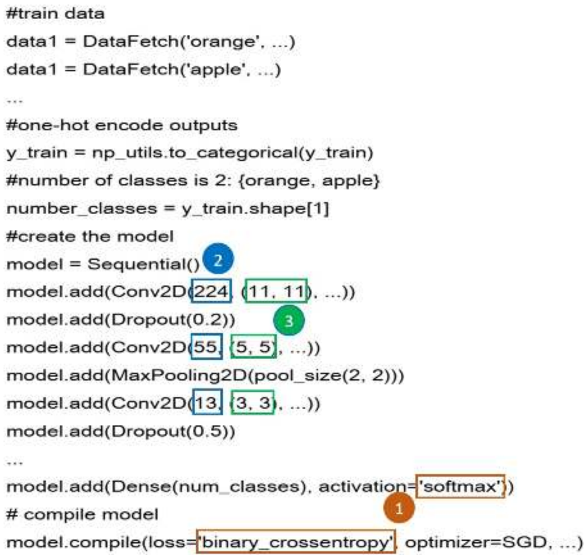
Authors' addresses: Amin Nikanjam, K. N. Toosi University of Technology, Tehran, Iran, SWAT Lab., Polytechnique Montreal, Montreal, Canada, nikanjam@kntu.ac.ir,amin.nikanjam@polymtl.ca; Houssem Ben Braiek, SWAT Lab., Polytechnique Montreal, Montreal, Canada, houssem.ben-braiek@polymtl.ca; Mohammad Mehdi Morovati, SWAT Lab., Polytechnique Montreal, Montreal, Canada, mehdi.morovati@polymtl.ca; Foutse Khomh, SWAT Lab., Polytechnique Montreal, Montreal, Quebec, Canada, foutse.khomh@polymtl.ca.

XXXX-XXXX/2021/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. 2021. Automatic Fault Detection for Deep Learning Programs Using Graph Transformations. 1, 1 (June 2021), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many developers, entrepreneurs, and researchers are showing an increasing enthusiasm in developing and using Deep Learning (DL) in a variety of domains. The core of DL programs is deep neural networks. After building a desirable DL model, it should be trained by executing a learning algorithm on a dataset. Easy-to-use libraries such as TensorFlow or Keras have been developed to simplify the development process. However, leveraging these libraries to implement a training program is still challenging, in particular for developers who are not experts in machine learning and neural networks. Like any software, a DL program often contains design issues and bugs. For example, let us consider the program in Figure 1 which is extracted from Stack Overflow (SO) post #44322611 and is reported to have a low accuracy.



```
#train data
data1 = DataFetch('orange', ...)
data1 = DataFetch('apple', ...)
...
#one-hot encode outputs
y_train = np_utils.to_categorical(y_train)
#number of classes is 2: {orange, apple}
number_classes = y_train.shape[1]
#create the model
model = Sequential()
model.add(Conv2D(224, (11, 11), ...))
model.add(Dropout(0.2))
model.add(Conv2D(55, (5, 5), ...))
model.add(MaxPooling2D(pool_size(2, 2)))
model.add(Conv2D(13, (3, 3), ...))
model.add(Dropout(0.5))
...
model.add(Dense(num_classes), activation='softmax'))
# compile model
model.compile(loss='binary_crossentropy', optimizer=SGD, ...)
```

Fig. 1. Simplified example DL program from SO_44322611.

This program which implements a Convolutional Neural Network (CNN) has three issues. The first issue (i.e., ①) is a bug due to the incompatibility between the *softmax* as output activation and *binary_crossentropy* as loss function. In fact, developers should have chosen *categorical_crossentropy* because it works with one-hot encoding labels and softmax activation to solve multi-label classification problems (including binary labels problems)[1]. This kind of API misuse error can be identified by verifying the consistency of the involved mathematical operations. For instance, error ① induces an inconsistency between the loss function and the last layer activation. The other two errors are ② decreasing filters count $224 > 55 > 13$ and ③ decreasing filtering spatial size $(11, 11) > (5, 5) > (3, 3)$, which represent poor structural CNN choices that violate the common design patterns of effective and optimal CNN architectures [26, 53]. These structural errors are often detected through manual code reviews which is time consuming. A static code analysis

[1] We refer the reader to [59] for more practical use cases about how to choose the last layer activation and loss function when using the Keras Library.

using automated tools can significantly speed up this process. In this paper, we examine common structural errors and design inefficiencies occurring in DL programs and propose *NeuraLint*, a model-based verification approach for their detection. To design *NeuraLint*, we first propose a meta-model for DL programs that includes their base skeleton and fundamental properties. This meta-model captures their essential properties independent of available DL libraries. Considering the proposed meta-model, we specify for each fault or design issue, a verification rule that can be used to detect its occurrence. Finally, we propose a checking process to verify models of DL programs that are conforming to the meta-model. We employ graph transformations to implement *NeuraLint*. We present a type graph for the meta-model and graph transformations for the verification rules.

We evaluate our approach *NeuraLint* by finding various types of faults and design issues in 28 synthesized examples built from common problems reported in the literature [31] and 34 real-world DL programs extracted from GitHub repositories and SO posts. The results show that *NeuraLint* effectively detects faults and design issues in both synthesized and real-world examples. This paper makes the following contributions.

- We propose a meta-model for DL programs;
- We describe 23 common errors and poor design practices of DL training programs and provide rules for their detection;
- We propose a model-based verification approach for DL programs, using meta-modelling and graph transformation rules.
- We provide a concrete implementation of the approach as a tool that DL developers can use to detect errors and design issues in their DL training programs.

The remainder of this paper is organised as follows. Section 2 provides background information about graph transformation systems, and deep neural networks. Section 3 presents our meta-modeling of DL programs. Section 4 introduces the studied issues and rules for their detection. Section 5 presents our proposed approach *NeuraLint*. Section 6 reports the empirical evaluation of *NeuraLint*. Section 7 discusses the related literature. Finally, Section 8 concludes the paper and discusses future work.

2 BACKGROUND

In this section, we introduce background knowledge about graph transformation systems, and deep neural networks.

2.1 Graph Transformation Systems

Graph transformation system (GTS) [30] (also called graph grammar) is a formal language for the specification of software systems, in particular those with dynamic structures. The definition of an attributed GTS consists of a triplet *(TG, HG, R)* in which *TG* is a type graph, *HG* is a host graph, and *R* is a set of rules for graph transformation. *TG* is defined by four components, $TG = (TG_N, TG_E, src, trg)$. TG_N and TG_E includes all node types and edge types respectively. *src* and *trg* are two functions $src : TG_E \rightarrow TG_N$ and $trg : TG_E \rightarrow TG_N$, that determine the source/destination nodes of an edge, respectively. The initial configuration of a system specified by GTS is presented by the host graph which is an instance of the type graph. Therefore, each component of the host graph, node or edge, must have a component type in the type graph. A host graph *HG* may instantiate from a type graph *TG* using a graph morphism function $type_G : HG \rightarrow TG$, in which the components of *HG* are instantiated from *TG*. Other configurations or states of a system are generated by successive applications of transformation rules on the host graph. A transformation rule *r* in *R* is defined by a triplet (LHS_r, RHS_r, NAC_r) in which LHS_r (left-hand side) represents the preconditions of the rule whereas RHS_r (right-hand side) describes the postconditions. Moreover, there may be a Negative Application Condition (NAC) for the rule r, meaning that the rule r can be applied only when NAC_r does not exist in the host graph. By applying the rule r to the host graph HG, which is an instance model of the meta-model or type graph, a matching of the LHS_r in HG is replaced by RHS_r. Formally, a graph morphism exists between LHS_r and the instance model HG. The application of a rule is performed in four steps: (1) find a matching of LHS_r in HG, (2) check NAC_r that forbid the presence of certain nodes and edges, (3) remove a part of the host graph that can be mapped to LHS_r but not to RHS_r, and (4) add new nodes and edges that can be mapped to the RHS_r but not to the LHS_r.

2.2 Deep Learning Software Development

In this section, we first introduce the architectures of Deep Neural Networks (DNNs) supported by our meta-model and verification approach. Then, we present the state-of-practice regarding the software implementation of these DNNs.

2.2.1 Feedforward Neural Network (FNN). FNN [23] is the principal neural network architecture used for solving classification and regression problems, where the task is to learn a mapping function capable of converting input data to a target output. FNN consists of many, and sometimes diverse, sequences of layers of computational units. These computational layers are trained to extract features hierarchically (i.e., starting from low-level to high-level features), then, detect discriminative and informative patterns, which serve the FNN to derive either the class label (classification problems) or continuous outcome (regression problems). It is called feedforward because the information flows in a forward manner from the input layer, through the hidden layers (if any) and to the output layer, e.g., a class probability or a predicted real value.

Computational Layers. FNN layers are constructions of neurons, where each performs:*(i) A linear calculation* that consists of a weighted sum of all the signals from the previous layer with addition of a constant (i.e., bias); *(ii) A non-linear activation* that consists of a gate function, filtering out the computed quantities, to derive the input signals for the neurons of the next layer. The initialization of layers' parameters (i.e., weights and biases) and the choice of activation functions represent essential parts of the FNN design that have a crucial impact on the performance of the training procedure [27].

Parameters Training. Starting from a random initialization, the training process consists in updating iteratively the model parameters, towards minimizing the loss of DNN's predictions as regards to the training labeled data. Indeed, a loss/cost function is defined to estimate the average distance between predicted and actual outcomes. The training pass over FNN's parameters is performed by backpropagation that makes the loss gradients flow over the neurons defined in the network but in the opposite direction, in order to update the parameters of each layer. The update quantities are mostly approximated using a first-order optimization algorithm such as SGD, Momentum, and Adam. Thus, the training procedure resides in fulfilling several training passes on *i.i.d* batches sampled from training data (independent and identically distributed) until reaching a local or the global minimum of the loss. Commonly, the best-fitted FNN is found after multiple epochs (i.e., passes over all the training data).

Regularization Methods. The regularization is required to improve the convergence and generalizability of the above-mentioned parameters training procedure. For DNNs, many regularization techniques have been proposed and the most used ones are dropout and batch-normalisation (batchnorm). Dropout [56] masks at every training iteration a random subset of units (i.e., nullify them). The stochasticity injected into the inference calculation, only during the training, prevents the co-adaptation of feature detectors and encourages the DNN to learn robust patterns against partially-hidden information. Batchnorm [34] acts differently on activations by normalizing their values using statistics (i.e., mean and variance) of the current batch of data during the training. During the testing, it updates internally, the population statistics of all batches for each level of activations in order to switch to normalizing against population, rather than batch, statistics. This normalization of intermediary input data has shown its effectiveness in smoothing the loss landscape, which ensures faster and safer training convergence with high potential to escape weak local minima.

Dense and Convolutional Architectures. The basic FNN architecture consists of stacking dense layers, where all the neurons of two consecutive layers are fully-connected. In this paper, we consider convolutional architectures which represent a particular type of FNN designed for multi-dimensional input data, such as 2D images and audio spectrograms, or 3D videos. The benefit of CNNs lies in their ability to take into account the spatial information in their feature extraction process. To do that, CNNs stack, earlier, two specialized layers: convolutional layer and pooling layer. *(1) Convolutional layer* applies spatial filters over the input data and each filter's weights are learned to detect relevant features supporting the network's task. Thus, it yields a feature map for each learned filter, where each unit is connected to a local region (i.e., size of spatial filtering window) in its previous layer's feature maps and *(2) pooling layer* performs spatial pooling over the computed feature map to reduce its dimensionality and retain the most relevant information. The spatial pooling can be either average or max aggregation that computes, respectively, the average or max of all the units in the specified spatial window. Conventionally, a CNN architecture incorporates a bundle of convolutional layers with increasing filter count and separated by pooling layers to shrink gradually the feature map area. Hence, the extracted feature space tends to become deeper and narrower throughout the network until it becomes ready to be flattened and fed to the dense layers in charge of mapping the features into the target output.

2.2.2 DL Software Development. DL libraries encode the DNN as a Directed Acyclic Graph (DAG), where nodes and edges represent, respectively, operations and data paths. This computational graph represents a high-level of abstractions to organize the components and their relations. Thus, the internal calculations can be decentralized where each component performs its local computation and communicates the result via data paths to its connected components. Besides, the DAG abstraction allows the DL library's graph compiler to optimize the computations' execution for a target hardware architecture.

Leveraging DL libraries to implement a training program for a designed DNN is not straightforward and it can be error-prone. DL libraries often have to trade off between the coverage of novel DL functionalities and the ease of rapid implementation and extension of DNN software prototypes. As a compromise solution, they uniformly include, for each newly-implemented DL functionality, a bundle of automated steps and default settings following its common usage trends in the community. This enables quick prototyping of regular DNNs while keeping the flexibility to try other configurations with the tweakable setting options available for every provided DL routine. As a consequence, DL developers should be aware of the intricacies of these DL libraries to choose the appropriate configurations and avoid breaking their implicit assumptions in regard to the usage of their built-in routines.

3 META-MODELING DL PROGRAMS

With the proliferation of libraries supporting the development of DL programs, a fundamental question emerges: is there any generic representation of DL programs that is independent from these libraries? In other terms, can we define a meta-model of DL programs and how can we model

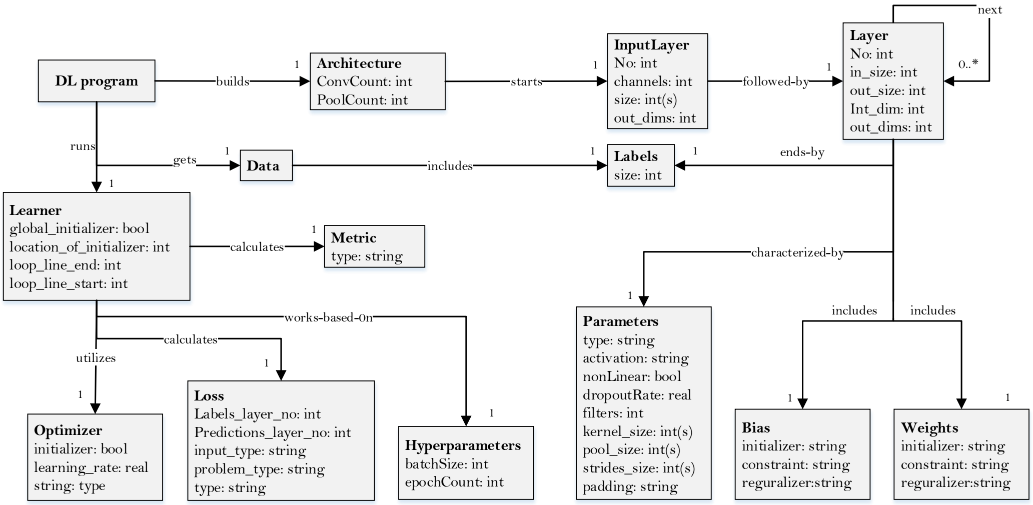


Fig. 2. The proposed meta-model (type graph) for DL programs.

a DL program? Answering this question would pave the way for the application of model-driven engineering techniques to the detection of errors in DL programs. In [25], researchers proposed a meta-model for meta-learning. They presented an overview of the meta-learning concepts –on a meta-modelling level– with possible variabilities and discussed how their meta-model could be integrated into existing modelling frameworks and tools. However, while their meta-model includes "Learning Block", "Learning Algorithm", "Optimizer" and "Hyperparameters", no further details like specifications of learning algorithms or blocks are presented and they did not explore the possibility of identifying errors in machine learning models. In this section, we present a particular meta-model for DL programs and our approach for meta-modeling of such programs to perform static analysis of DL programs. We describe possible variabilities of the meta-model and how concrete DL programs can be generated from it. We think that a generic meta-model for DL programs can significantly facilitate the use of DL in various applications and would be helpful for understanding DL programs written by developers using third-party DL libraries. In fact, model-driven engineering is a perfect tool to make this idea come to life and ease the process of developing and debugging DL programs.

3.1 A Meta-Model for Deep Learning Programs

A DL program has different components. The core of each DL program is a DNN. For the sake of simplicity, we only consider the feedforward multilayer perceptron (MLP) architecture. Like other computational models, DNN attempts to find a mathematical mapping from the input into the output during a learning phase. Usually, a set of inputs and desired outputs (or targets) is provided for learning, which is called Dataset. Therefore, our meta-model includes three main parts: Architecture of DNN, Learner, and Data. Since we have used GTS for modeling DL programs, our proposed meta-model is represented by a type graph. The proposed type graph is illustrated in Figure 2. The node representing the **DL program** has three edges to **Architecture**, **Learner** and **Data** nodes indicating its main components. In the following, we describe the meta-model in detail. It should be noted that our aim of meta-modeling is the detection of faults in DL programs; therefore the most relevant components have been incorporated into the meta-model.

3.1.1 Architecture of deep neural network. An architecture starts with the input layer, continues with some hidden layers and ends with the output layer. We have considered a distinctive node for the **InputLayer** because of its importance but all other successive layers are modelled as **Layer**. Each layer has a **size** indicating the number of neurons in that layer. There are specific properties among nodes that are modelled as edges. For example, **Architecture** starts by **Input Layer**, **Input Layer** is followed by other **Layer**s, each **Layer** may have next layers and each **Layer** has a **Type** as an attribute. There are different types for a layer in DL, e.g., dense, 1D and 2D convolution, pooling or data processing layers like flatten. There may be other attributes for a layer like **Bias**, **Weights**. An architecture ends with **Labels**, the desired outputs of DNN that are used to calculate the error of the network in **Loss** function. Actually, **Labels** is a part of **Data** associated with the DL program.

3.1.2 Learning algorithm. A DL program normally employs a learning algorithm, **Learner** in our meta-model, to learn the mapping from inputs to outputs. A **Loss** function is used to calculate the error of a neural network in matching the target (desired) and output value during training. The goal of learning is minimizing the loss by modification of the network's parameters (weights) by an **Optimizer**, e.g., Adam or stochastic gradient descent (SGD). The overall performance of the network is measured using a **Metric**. Moreover, there are some **Hyperparameters** like the number of epochs or batch size.

3.1.3 Data. This node contains **Labels**, meta-data, features, and related information about the data set, like shuffling and batching.

3.2 DL programs modeling

A model of a DL program includes components that form its source code. There are two ways to build a model of DL programs: configure an arbitrary model directly or transform a DL program to a model. One may design a model for a DL program that conforms to the proposed meta-model by configuring each component of the meta-model. Starting from an empty model, layers are added one by one to **Architecture**; making a chain of layers that starts by **InputLayer** node, follows by other **Layer** and ends with **output**. Each layer is configured separately to set **Parameters**, **Weights**, etc, and once a layer is configured completely, the model will proceed to the next layer. Other components of DL programs like **Learner** and **Data** are configured respectively. This process is similar to what a developer does when developing a DL program using popular DL frameworks. Therefore, the meta-model and resulting models would be realistic from a practitioner point-of-view and sufficiently flexible in representing plenty of DL programs.

On the other hand, a model could be configured according to a DL program that has already been developed by a programmer. The source code of a DL program is converted to a model, which is an attributed graph. Dedicated convertors are programmed in *NeuraLint* to convert a DL program written by different DL libraries to its model. The source code of a program is parsed to extract relevant information that is necessary to configure the model. The meta-model is generic enough to be independent of any specific DL library. Hence, we can have a model of a DL program that conforms to the meta-model; making possible further investigations on the model, such as verification. Apart from the work and analysis that are presented in the rest of this paper, we believe that this meta-model can be very useful to understand DL programs written by third parties. It will be helpful in understanding the development activities of DL practitioners; the way they write DL programs and the type of faults that they experience.

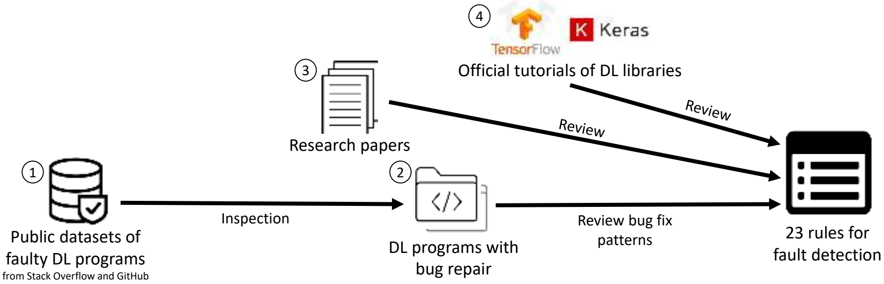

4
TensorFlow
Keras
Official tutorials of DL libraries
3
Research papers
Review
Review
1
Public datasets of faulty DL programs
from Stack Overflow and GitHub
Inspection
2
DL programs with bug repair
Review bug fix patterns
23 rules for fault detection

Fig. 3. The adopted methodology for extracting rules from different sources.

4 MODEL-BASED VERIFICATION RULES

In this section, we present the proposed rules for detecting faults in DL programs. First, we report the adopted methodology for extracting rules. The rules are then described in Subsection 4.2. Afterward, we present a discussion on the application scope of rules. Finally, we describe the approach adopted to implement the rules using graph transformations.

4.1 Methodology

Figure 3 illustrates the adopted methodology for extracting the rules. We have explored three main sources: datasets of buggy DL programs (including bug repairs), relevant research papers and official DL libraries' tutorials. In the first step, we manually inspected the labeled bugs in the public datasets of buggy DL programs released by former research studies [31, 35, 61] collected from SO and GitHub with the objective of discovering finer root causes of bugs. Zhang et al. [61] published the first empirical study on real-world DL bugs occurring in Tensorflow-based software systems including their high-level root causes and symptoms. Then, Islam et al. [35] extended the investigated cases to include DL software systems written using other competitive DL libraries such as Pytorch and Caffe, and studied the relationship and the evolution of different DL bug types. Last, Humbatova et al. [31] refined the former bug investigation [35, 61] into a taxonomy of real faults that occur in DL software systems. In the second step, we have inspected bug fixes suggested in accepted answers of SO posts and fix patterns adopted in GitHub samples to identify patterns followed to fix the reported bugs.

We reviewed research studies on DL bugs [31, 35, 61] and fundamental DL design principles [22, 26, 46, 53] in the third step. Regarding the former, the aim was finding rules for validating the correctness of model structure and configuration choices through the DL program's model drawn from the code. We build on these previous works to specify rules that can be used to detect occurrences of different types of issues in DL programs and validate the conformity of the DNN design to common patterns through static code inspection. However, most of the DNN design patterns and principles have been deduced from state-of-the-art CNN architectures [16, 45] that have shown their effectiveness on public computer vision datasets and competitions such as ImageNet classification [18] or COCO object detection [44]. Thus, we aim to report warnings to the user whenever a poor design choice is spotted with respect to these empirical research studies on DNN design principles. This would likely steer the user to redesign his model in order to avoid the performance degradation either at the training or at the inference mode. Finally, to support practitioners in their DL program debugging, we also proceed with a dual analysis over the commonly reported APIM bugs and the official DL libraries' tutorials.

In the end, we have come up with 23 rules for detecting bugs and issues in DL programs. The rules are organized into different high-level root causes as initially introduced in [61], namely Incorrect Model Parameter or Structure (IPS), Unaligned Tensor (UT), API Misuse (APIM), and Structure Inefficiency (SI).

4.2 Rules

4.2.1 Incorrect Model Parameter or Structure (IPS). IPS bugs are related to modeling faults that arise from either an inadequate model parameter like learning rate or an incorrect model structure like missing or redundant layers [61]. The major symptom of IPS bugs is anomalous training behaviors leading to low effectiveness such as low precision and a huge loss.

Rule 1: Asymmetric Units Initialization. *The initialization of weights should not be constant to break the symmetry between neurons* [22]. For instance, a common mistake is to start with null weights, which eliminates asymmetry between the neurons (i.e., all the neurons would output the same value, and then, would receive the same gradients).
Rule 2: Null Biases Initialization. *The initialization of biases is preferred to be zeros* [22]. It is a common practice to expect that the outputs could be totally explained by the input features. Indeed, no custom initial bias provided a consistent improvement, but it may weaken the learning.
Rule 3: Non-Linear Activation Requirement. *Activations for learning layers (i.e., convolution and fully-connected layer) should be a non-linear function.* This key attribute is needed to enhance the ability of DNN to model highly nonlinear mappings and draw complex shape decision boundaries [46].
Rule 4: Unnecessary Activation Removal. *Multiple and redundant connected activations are not allowed.* Since all activation functions are designed to transform real values into a restricted interval [46], successive activations applied to the same features can make their last activation unable to produce its full output range.
Rule 5: Class Probability Conversion. *A last layer activation is required to transform the logits into probabilities for classification problems.* In detail, sigmoid ($\sigma(z) = \frac{1}{1+e^{-z}}$) and softmax ($\sigma(z)_i = \frac{e^{-z_i}}{\sum_{j=1}^{L} e^{-z_j}}$, for $i = 1, ..., L$ and $z \in \mathbb{R}^L$) are, respectively, needed to normalize outputs with single unit ($z \in \mathbb{R}$) and multiple units ($z \in \mathbb{R}^L$). A common mistake is to use softmax instead of sigmoid for binary classification without one-hot encoding beforehand, which totally obstructs the learning because the outcome is always equal to $1 = e^{-z_1}/e^{-z_1}$, $z \in \mathbb{R}^1$.

Indeed, the above-mentioned rules show that static DL code analysis can help detect earlier structural bugs and misconfigurations, however, incorrect parameters like learning rate or neural network size (width and depth) could be identified through empirical evaluation of the model on the underlying data.

4.2.2 Unaligned Tensor (UT). The computational units in a DNN graph are mostly tensor-based operations, where each one receives and returns tensors (i.e., multi-dimensional arrays). Their connections can hide issues related to the compatibility of tensors' shapes. DL developers often fail to express and manipulate the shapes of tensors properly [39] because DL libraries mask all the algebra computations and dynamic shapes' inference details. A bug triggered during the DNN graph construction when the shapes of one operation's tensors do not match is called an Unaligned Tensor (UT) [61]. The major symptom of UT bugs is runtime errors because the underlying tensor-based operation could not run on two incompatible tensors. However, the dynamic shape inference included in most DL libraries often makes the exception of incompatible shapes triggering far from its localization in the DL code; so the error message can be misleading. In the following, we describe various DNN layers' connectivity and configuration rules that can be checked on the DL model to identify the UT bug type and localization.

Rule 6: Consecutive Layers Compatibility. *A processing layer that operates on a N-dimensional tensor, should receive a valid input tensor with exactly N-dimensional shape.* For instance, a Conv2D layer works on 4-D tensors, i.e., $[samples, height, width, channels]$, but a Dense layer works on 2-D, i.e., $[samples, units]$, which means a reshape layer is needed to flatten the convolutional feature space before starting the dense layers' inference.
Rule 7: Spatial Size Agreement. *A processing layer should receive sufficient-sized feature space to perform its spatial filtering or pooling.* For instance, 2-D processing layers like Conv2D and MaxPooling2D require a size of feature space greater or equal to their local window size, i.e., $(window\_height \leq input\_height)$ and $(window\_width \leq input\_width)$.
Rule 8: Reshaped Data Retention. *A reshape layer should preserve the total data elements.* More specifically, we verify that the product of original tensor dimensions equals the product of reshaped tensor dimensions.
Rule 9: Separate Item Preservation. *A reshape layer should never alter the size of elements (i.e., first dimension).* Otherwise, the reshape would provoke an overlap between data items (i.e., points in the feature space), and as a consequence, invalidate the following layers designed to process each data item, independently.

4.2.3 API Misuse (APIM). APIM bugs are the ones introduced by practitioners who misunderstand some essential assumptions made by the used DL APIs [61]. Indeed, most DL libraries encode the DNN as an acyclic computational graph where the edges are tensors and the nodes correspond to operations. The operations include all the supported computational units that form the linear computations, activations, gradient estimations, etc. Programmatically, practitioners describe their designed DL program by inserting and configuring built-in DL routines, and connecting them by putting the outputs of one operation as inputs to another. When these routines are added without fulfilling their usage conditions or without context alignment, the DL program would not reflect the designed DL model or cannot be successfully executed by the DL core framework, which leads, respectively, to low effectiveness or runtime exceptions. Below, we detail the verification rules that should be executed on the generated static analysis-based graph model to confirm the existence of essential DL program's components and their consistency with API assumptions and recognized application context.

Rule 10: Valid Loss Linkage. *The loss should be correctly defined and connected to the last layer activation in accordance with its input conditions (i.e., shape and type).* For instance, the input type for cross-entropy based losses could be either logits or probabilities. Indeed, numerically stable implementations regarding the cross-entropy based losses require merging both loss and last activation functions together to rewrite the join formula carefully without any risk of $\log(0)$ or $\exp(\infty)$. However, ignoring this difference between theoretical loss functions and their numerically-stable implementations gives rise to a common mistake in the development of DNN programs, as passing activated output to this logit-based loss would cause redundant activations.
Rule 11: Valid Optimizer Linkage. *The optimizer should be correctly defined and connected to the computational graph.* Depending on the DL library, it could be either connected to the loss (e.g., TensorFlow) or the learnable parameters (e.g., Pytorch).
Rule 12: Single Global Initialization. *The learnable parameters should be totally initialized once at the beginning of the training.* For some DL libraries (e.g., TensorFlow), this mandatory condition should be carried out by the developer.

Rule 13: Zero Gradients Reset. *The gradients should be re-initialized after each training iteration.* This clears old gradients from the last step; otherwise accumulating the gradients hinders the optimization process. Some DL libraries (e.g., Pytorch) delegate this necessary reset step to their users.
Rule 14: Iterative Training Procedure. *The loss minimization problem should be solved iteratively with continuous update of parameters.* Depending on the granularity level of the API used, it could be a native loop of optimization routine calls or a single call of a configurable fit function.

4.2.4 Structure Inefficiency (SI). SI issues reflect a misconfiguration in the DNN design and its structure that leads likely to performance problems, contrary to IPS bugs that leads to functional incorrectness [61]. SI issues may result in performance inefficiencies (like long time of model training/inference) or poor predictions (like low classification accuracy). As an example, large feature-maps, especially in the early layers, provide more valuable information for the CNN to utilize and improve its discriminative power. Therefore, it is crucial to avoid prematurely down-sampling and excessive appliances of pooling. Otherwise, the model will lose some information extracted in early layers resulting in poor performance. Since the best trained model cannot guarantee 100% of accuracy, it is challenging to detect design issues by assessing the performance of the obtained models. Indeed, some misconfigurations and poor design choices may definitely introduce inefficiencies on the internal functioning of the DNN or one of its components, which can hinder the expressiveness of mapping functions, memory and compute consumption. For example, when increasing the depth of a DNN, it is important to control both the model size and the computational cost (regarding the specific task); otherwise, stacking a high number of layers can worsen the performance.

Rule 15: Effective Neurons Suspension. *The dropout layer must be placed after the maximum pooling layer to be more effective.* Considering the case studies with max-pooling layers [56], the dropout has been applied on the pooled feature maps, which becomes a heuristic followed by the state-of-the-art CNN architectures [16, 45]. The intuitive explanation is that dropping out the activation before the pooling could have no effect except in cases where the masked units correspond to maximums within input pooling windows because the max-pooling would keep only these maximums as inputs for next layers.
Rule 16: Useless Bias Removal. *A learning layer should no longer include a bias when it is followed by batchnorm.* Batchnorm applies, after the normalization, a linear transformation to scale and shift the normalized activations $\hat{a_i} = \alpha a_i + \beta$, where α and β are learnable parameters. This allows DNN to compensate for any loss of information by the value distortions in order to preserve its expressive power. Since, batchnorm already adds a β term fulfilling the same role of bias, so "its effect will be canceled" [34] in the presence of a bias.
Rule 17: Representative Statistics Estimation. *Batchnorm layer should be before the dropout.* Otherwise, batchnorm computes non-representative global statistics (i.e., moving average and moving variance) on the dropped outputs of the layer. Li et al. [43] discussed the reason behind this disharmony between dropout and batchnorm and showed experimental results reinforcing their explanation.
Rule 18: Pyramid-shaped Construction. *The area of feature maps and the width of fully-connected units should be progressively decreasing over the layers.* It has been shown [24] that the progressive size reduction of activations implicitly forces the neural network to find and learn more robust features. Hence, it significantly improves its predictions, since the network decisions are based on more discriminative and less noisy features.
Rule 19: Maximum Pooling Domination. *Max-pooling is the preferred down-sampling strategy.*

In fact, down-sampling [55] can be done by max- or average-pooling or strided convolution (strides greater than 1). Nevertheless, max-pooling operation has been shown [50] to be extremely superior for capturing invariances in data with spatial information, compared to other downsampling operations.

Rule 20: Gradual Feature Expansion. *The number of feature maps should be gradually expanded while the feature map area is retracted.* The growth of feature maps count is recommended [42] to compensate for the loss of representational expressiveness caused by the continuous decreasing of the spatial resolution of the learned feature maps. Throughout the layers, the feature space becomes synchronously narrower and deeper until it gets ready to be flatten and fed as input vector to the dense layers.

Rule 21: Local Correlation Preservation. *The local window size for spatial filtering should generally increase or stay the same throughout the convolutional layers.* It makes sense that by using CNNs, the locality of information is crucial for performing the task. Thus, it is important to preserve locality throughout CNN to guarantee its success in detecting various features and relations between them [41]. Furthermore, early convolutional layers learn lower level features while deeper ones learn more high-level and domain specific concepts. It is recommended [52, 57] to start with small spatial filtering to collect much local information and then gradually increase it to represent more compound information.

Rule 22: Maximum Information Utilization. *Deep CNN should not apply pooling after every convolution.* For instance, we use, as approximations, the minimum of 10 layers to consider a CNN deep and $1/3$ as threshold for the proportion of pooling layers with respect to the total of convolutional layers (convolution + pooling) to pinpoint a high amount of pooling. In fact, it has been shown [28, 33, 57] that larger feature-maps, especially in the early layers, provide more valuable information for the CNN to utilize and improve its discriminative power. Therefore, it is crucial to avoid prematurely down-sampling and excessive appliance of pooling.

Rule 23: Strive for Symmetry and Homogeneity. *Deep CNN should favor blocks of* 2, 3 *or even* 4 *homogeneous convolutional layers with similar characteristics.* Indeed, going deeper does not refer to simply maintaining stacking a series of convolution and pooling layers. Advanced CNN architectures [29, 32, 38] have shown the benefit of having several homogeneous groups of layers, where each one is specialized to achieve a particular goal. Indeed, building blocks of convolutional layers with similar characteristics (i.e., the same number of feature maps and feature map sizes) increases the homogeneity and the structure symmetry within the CNN. Hence, larger kernels can be replaced into a cascade of smaller ones, which enhances the nonlinearity and yields better accuracy [52]. For instance, one 5×5 can be replaced by two 3×3 or four 2×2 kernels. Moreover, spatial filtering with reduced size decreases massively the computation power requirement because recent NVIDIA cuDNN library (version 5.x or higher) is not optimized for larger kernels such as 5×5 and 7×7, whereas CNN [52] with entirely 3×3 filters achieved a substantial boost in cuDNN performance.

4.3 Application scope

The rules are defined to support the debugging of DL programs through static analysis-based graph models. On the first hand, we have been limited to the information that could be parsed from the source code of a DL program. For instance, there are some model parameters that should be experimentally tested to assess their adequacy for the underlying problem, particularly for IPS bugs. Thus, the bugs related to data (type, format, and preprocessing steps) and hardware issues (GPU configuration and required memory) are excluded from the rules and debugging scope because the information needed to diagnose the issue and identify those bugs are mostly out of the static DL code scope. In fact, these types of bugs could be better detected using Python and GPU firmware native debugging tools that help inspect step by step the executed statements at runtime. On the other hand, we have defined a high-level meta-model that could be instantiated to represent any DL program; so, bugs and intricacies that are related to specific DL libraries or APIs are discarded from the verification routines. Referring to the identified high-level root causes of DL bugs [61], we did not consider API Change (APIC), which reflects anomalies by a DL program upon a new release of the used library and Confusion with Computation Model (CCM), which includes bugs arising from misunderstanding the DL library computation model such as DAG and deferred execution of Tensorflow, and bugs which are related to regular programming mistakes like for any traditional software.

According to the categories in the most recent taxonomy of DL faults [31], we mention the type of faults that could be covered by the proposed rules: *Wrong Tensor Shape, Wrong Shape of Input Data, Model Properties, Layers, and Loss Function.* Based on the count of manually-analyzed real-world buggy programs in [31], we found that the covered DL bugs/issues in *NeuraLint* represents 51.7%, of all reported DL buggy samples in the taxonomy. In the following, we report a prevalence ratio for each type of bugs, i.e., the number of buggy DL programs assigned to the underlying category divided by the total number of buggy DL programs in [31]:

- *Wrong Tensor Shape (*14.1%*).* It refers to errors leading to unexpected tensor shape and mismatch between operations' shapes of tensors.
- *Wrong Shape of Input Data (*14.8%*).* It assembles the bugs caused by invalid shapes of input data for a computational layer including input layer, hidden layers, and output layer, as well as the shape of math function's inputs.
- *Model Properties (*2.7%*).* It comprises improper modeling choices that can dramatically degrade the DNN's performance such as missing, wrong model initialization, or sub-optimal model structure.
- *Layers (*15.4%*).* It contains the bugs related to layers including missing, redundant, misconfigured and wrong neural network layers.
- *Loss Function (*4.7%*).* It covers different issues in relation with the loss component such as missing, inadequate and wrong loss function.

4.4 Representing Rules as Graph Transformations

In this paper, the meta-model is presented as a type graph and each model is a graph, instantiating the type graph. Each DL program is converted to a graph, as well. As a straightforward approach, graph transformations are chosen to implement the verification rules. Each verification rule is implemented as one or some graph transformations or graph processing operators. In fact, graph transformations are used to detect possible faults in a model, faults that are caused by violating the verification rules. Consequently, a transformation is applicable where the conditions of the corresponding rule are violated. In other words, if conditions of a verification rule are violated representing a fault in a model then the graph operation(s) of that rule will be applicable. Graph transformations are very flexible to find violation of some conditions in a graph. Recalling that a graph transformation r is defined by a triplet of (LHS_r, RHS_r, NAC_r), a specific condition would be checked by finding a match of LHS_r in the graph and/or the absence of NAC_r. Once a graph operation is applied, i.e., detecting a fault in a part of the graph, a specific fault code is added to the node or edge in which the violation occurred. This action is represented by the right hand-side of the rule RHS_r.

Figure 4 illustrates one of verification rules implemented as a graph transformation, showing LHS, RHS, and NAC. The transformation is an implementation of Rule 4 which asserts that: *Multiple and redundant connected activations are not allowed.* Developers usually add activations after learning

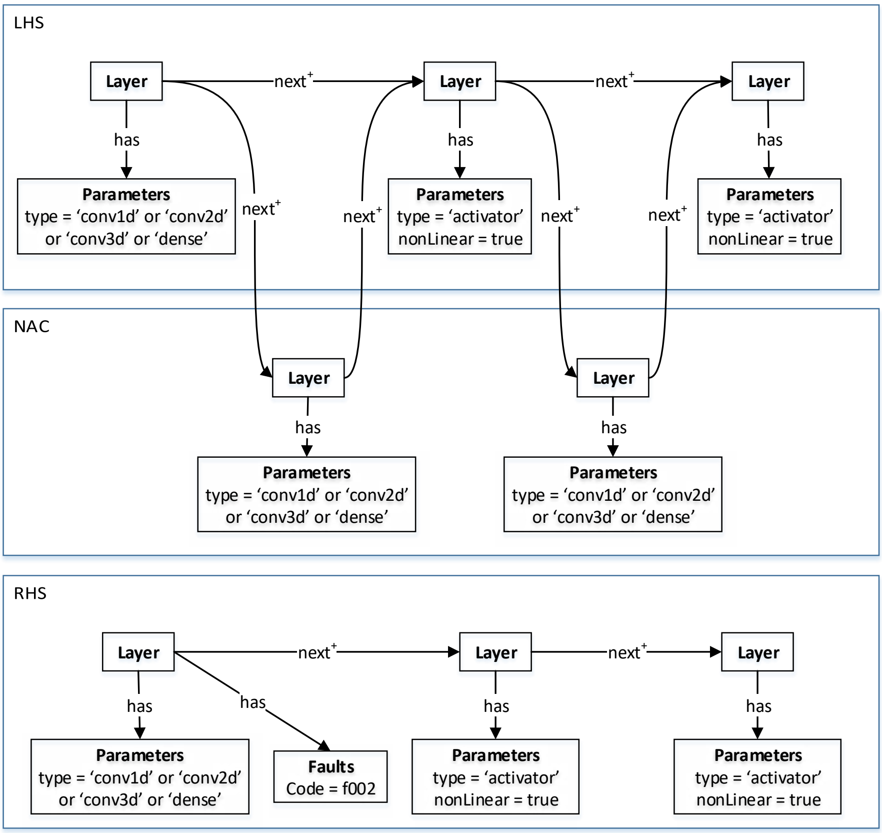


Fig. 4. An Example of Graph Transformation Rules: Implementation of Rule 4.

layers (like convolution and dense layers) to produce proper output signals. LHS shows a learning layer with the **type** of 'dense', 'conv1d', 'conv2d' or 'conv3d' in its **Parameters** node, followed by two consecutive layers containing the **type** of 'activator' and the **nonLinear** as *True* in their parameters. A positive closure is used on the label of incoming edges to activation layers (*next+*). This states that activations may appear in any **Layer** node on the path beginning from the learning layer and including multiple *next* edges (≥ 1). To be sure that another learning layer would not appear on this path, e.g., false detection of the next learning layer followed by its only activation, NAC forbids the existence of any learning **Layer** node on the subpath leading to activation nodes. If such a match is found in a model (graph of DL program), Rule 8 is violated. Therefore, RHS just adds a **Faults** node with relevant fault code to the faulty component, i.e., learning layer. Because of space limitation, we cannot present in the paper all the graph transformations implemented for our model verification. We refer interested readers to the source code of *NeuraLint* which is available online [11].

5 A MODEL-BASED VERIFICATION APPROACH FOR DL PROGRAMS

In this section, we describe our approach, *NeuraLint* for detecting faults in DL programs. *NeuraLint* is a model-based automated approach that performs a static analysis of a DL program to detect faults and design inefficiencies. Algorithm 1 shows the pseudocode of *NeuraLint*. The inputs are a DL program and a graph grammar, i.e., a set of graph transformations rules. As presented in Algorithm

1, *NeuraLint* has three main steps: extract a graph from the DL program, perform graph checking and generate a report from the resultant graph. At first, the DL program is modeled as a graph that conforms to the proposed meta-model, i.e., type graph. Then, a checking process runs to find bugs/issues in the model. This process attempts to apply rules to the graph and stops when further rule application becomes impossible. Then, *NeuraLint* traverses this graph to generate a report for the user, containing a description of the faults and design issues found for each component. Except graph checking and graph transformations, all other parts of *NeuraLint* are implemented in Python. We discuss details of each step in the rest of this section. The source code, developer's guide and some examples are available online [11].

5.1 Modeling DL Program as Graph

In our graph-based approach, a DL program is modeled by a graph instance conformed to the type graph, i.e., meta-model. To fulfill this primary step, we implement the graph generation relying on static code analysis that examines the source code and extracts the valuable code units and segments the information needed to instantiate the type graph's components. This analysis is performed without executing programs to extract the structure of the DL model from the code. Based on the OMG (Object Management Group) taxonomy of software analysis types [54], our graph generation process belongs to the category of technology-level software analysis because the code inspection routines are further customized to consider the interactions between identified units and connect them to the internals of the used technology (DL library). This provides a more holistic and semantic view of the analyzed DL program, that allows detecting faults related to either DL library's API misuse or DL algorithm implementation requirements. Hence, a specific graph generator should be implemented for each supported DL library. Without loss of its generality, *NeuraLint* currently supports DL programs written using TensorFlow and Keras as two well-known and popular libraries. It should be noted that *NeuraLint* can be extended to detect bugs/issues in DL programs developed by other DL libraries (like PyTorch), as well. The only necessary step is extending the parser to cover specific APIs of each DL framework. Moreover, by employing static analysis we are limited to the information available prior to the runtime, in contrast to dynamic analysis performed on programs while they are executing. Hence, we cannot detect bugs/issues depending on information introduced in the runtime environment, like dynamic types or dynamic constructions. However, we believe that the current approach can detect a significant amount of bugs/issues since DL programming is usually simple, and much useful information on DL models can be extracted by static analysis. In the following, we describe steps of modeling of DL programs as attributed graphs.

Our approach consists in parsing the DL routines called in the DL program line by line to extract the components related to both the DNN model and the DL training algorithm, as well as their configurations. The identified components are independent of the context of the parsed program. We used Abstract Syntax Tree (AST) to parse the DNN program script. AST represents the abstract synthetic structure of the scripts as a tree. This tree represents the abstract syntactic structure of the source code of the DL program. Each node of the tree denotes a construct or statement in the source code of a DL program. Arguments of function calls or assignment statements are extracted and stored in subtrees of the node. As we process the code line by line according to AST, the graph is constructed gradually by appending nodes and edges. In each DL library, there are specific built-in routines for defining various layers, configuring them (e.g., adding dropout and activations), connecting layers to each other, feeding input, calculating output, and training the network. In this way, the most important parts in the DL program for constructing its model like layers, dimensions, loss and optimizer functions could be identified by the parser. Algorithm 2 illustrates this process. Based on the information extracted by AST, for any line in the code of a DL

```
Algorithm 1: NeuraLint
  Input: A DL program, program, and rules as a graph grammar
  Output: List of bugs or warnings to improve the program
  graph ← extractGraphFromProgram(program) (Algorithm 2)
  final ← graphChecker(graph, rules) :
      (1) starting by graph, apply enabled rules.
      (2) apply enabled rules recursively.
      (3) terminate when further application of rules becomes impossible.
      (4) return final.
  report ← extractReportFromGraph(final)
  return report
```

program that indicates an assignment statement, we build a dictionary to store its value for further usage. For API calls that add or configure a layer, we firstly extract the related properties like type and number of units (neurons) from AST or dictionary of variables. Since the current version of *NeuraLint* is designed to handle DL programs developed using TensorFlow or Keras libraries, we have covered the API of these frameworks. Sometimes, a computation phase is required for each layer to process its attributes and attach it to other layers in the graph correctly. For instance, the dynamic shapes of the layer's tensors (i.e., input and output data layer) should be computed. In such cases, the required properties are computed before adding the node to the graph. If an API call relates to compiling or training the model (building the DNN, adding loss and optimizer functions), the optimizer and loss nodes are added to the graph. At the end, we will check all the interconnected layers to verify the coherence of the datashapes flowing throughout the DNN's computational layers and apply the required corrections. Afterward, the generated graph, including all relevant components (nodes/edges and their properties) based on the extracted and computed information is returned.

5.2 Model-based Verification using Graph Transformations

The verification rules are implemented as graph transformations to process and verify the graph. Each graph transformation applies to the graph if conditions of the rule are violated. Once the DL source code is modeled as a graph, the violations of rules can be detected with a graph transformation tool that executes the sequence of rules over the model of the DL program. In this paper, we have used the GROOVE toolset [49] to perform graph operations. GROOVE is a tool for implementing, simulating, and analysis of graph transformation systems. It is capable of exploring recursively and collecting all possible rule applications over a host (start) graph. This is referred to as the exploration of the state space of a graph grammar. GROOVE explores the state space by applying a slightly modified version of standard graph traversal algorithms, like depth-first search (DFS) or breadth-first search (BFS). Furthermore, it has a graphical interface for editing graphs and rules, and for exploring and visualising the GTS which could be called via command line, as well. The output of GROOVE is called the final graph on which no further rule application is possible. For more information about GROOVE's internal mechanism and its capabilities for modeling and simulating GTS, the interested reader may refer to [21].

In order to find which rules are violated, the graph transformation system must be simulated. The simulation performed by GROOVE automatically applies the matching transformation rules over the graph of the DL program. Actually, this process generates a state space, in which the model of the DL source code (*graph*) is the start state and the transitions are the applied transformation rules.

Algorithm 2: Extracting graph from DL program

```
Input: A DL program in Python developed by TensorFlow or Keras
Output: A graph indicating program's DL model with respect to the meta-model
graph ← empty
dictionary ← empty
for each line of DL program do
    if line encodes an assignment statement then
        extract left-hand (variable) and right-hand side (value) of the statement
        add the statement to the dictionary
    end
    if line encodes a layer then
        extract properties of layer (type, size, kernel, padding, ...)
        look up values of variables in dictionary
        add the corresponding node(s) and edge(s) to the graph
    end
    if line encodes compilation of the model then
        extract properties of compilation (loss and optimizer function)
        look up values of variables in dictionary
        add the corresponding node(s) and edge(s) to the graph
    end
end
return graph
```

It explores the state space of all graphs that are reachable from *graph*. In certain states, no more transformation rules can be applied; these states are called final states. A path starting from the start state and leading to a final state, consists of applied transformations indicating the detected faults or violation of verification rules. Moreover, this path indicates the type and location of detected faults. A specific code for each type of fault has been associated with the faulty component when the rule has rewritten the graph.

The rules are implemented in such a way that starts from the first layer and proceeds to the next layers one by one. At first, the general structure and connectivity of deep neural networks is tested assuring that input, hidden and output layers are well-formed and connected. These transformation rules mark the graph components (nodes and edges) with relevant flags to indicate the performed tests. Then, each graph operation checks specific conditions that are asserted in its rule using the information provided in the graph. A transformation should be fired if a rule violation is observed in the model of a DL program. If there are multiple rule violations or various instances of a violation in the considered model, all of them will be detected by applying multiple enabled rules. At last, a parser is developed to process the final graph and extract information about detected issues/bugs to generate a report for the user.

6 EMPIRICAL EVALUATION

In this section, we report an empirical evaluation that aimed to assess the effectiveness of *NeuraLint*.

6.1 Studied Programs

We have evaluated the effectiveness and efficiency of *NeuraLint* in detecting bugs/issues on a set of synthetic and real-world faulty DL programs. To create realistic synthetic examples, we also need some real-world DL programs to imitate the faults occurring in them. To find a proper set of real-world faulty DL programs, we have used two main sources: 1) samples found by directly searching over SO with keywords related to the categories of bugs covered by *NeuraLint*, and 2) public datasets of buggy DL programs (from SO and GitHub) released by previous research studies. For the former, we chose SO because it is the most popular Q&A forum for software development. As of May 2020, it has collected more than 19 million questions and 29 million answers. It has been also leveraged by previous studies on DL software systems [35, 60, 61]. Since *NeuraLint* currently supports both TensorFlow and Keras, we searched SO posts tagged by one of these libraries with the objective of collecting buggy DL code including multi-granularity levels such as a single function call, a snippet (few lines) of code or a whole DL program. In fact, we found that SO assembles 57,104 and 27,008 questions, tagged respectively with TensorFlow and Keras, that comprise diverse issues encountered by DL practitioners when dealing with these libraries. Hence, we refined our search queries with keywords related to the categories of bugs covered by *NeuraLint* that are described in Section 4 resulting in 255 posts. We manually inspected, for each type of bug, the top-10 relevant SO posts (i.e., according to built-in SO relevance criterion) mentioning one or more of its associated keywords. We consider SO posts, containing full code script or code snippets that are related to one or multiple bugs belonging to the above-mentioned categories. This process left us with 18 faulty DL programs.

Regarding public datasets of buggy DL programs (from SO and GitHub), we consider three publicly available datasets/replication packages [31, 35, 61]. Also we consider another dataset from a recent research on bug fix patterns in DL programs developed by five popular DL libraries including Tensorflow and Keras [36]. They have studied several repair patterns in DL programs. All these studies investigated various faulty DL programs from SO and GitHub. We have manually inspected all artifacts they have used in their study to find relevant faulty examples to evaluate *NeuraLint*. Actually, finding proper samples for evaluating our tool is not an easy task. We explain the methodology followed and encountered difficulties in the rest of this section. Some DL programs were developed by libraries other than Tensorflow/Keras which are out of scope of the current version of *NeuraLint*. In total, we had 733 SO posts and 682 samples from GitHub from all these sources where 622 programs were developed by TensorFlow and 793 by Keras. Among DL programs developed by Tensorflow/Keras, we have excluded programs containing types of faults that are not covered by the current version of *NeuraLint*, for example those related to recurrent neural networks. So, we were left with 566 faulty DL programs. In the next round, programs developed with older versions of Tensorflow/Keras were discarded if the API related to the fault was not supported in later versions. Many of the remaining samples (89 from SO and 126 from GitHub), especially those from GitHub, were actually libraries (not DL programs) that have been developed on the top of DL libraries for particular problems or domains, e.g. image/speech processing, reinforcement learning, or natural language processing. We also discarded these libraries which were 86. Although their implementation contained bugs that led to buggy DL models, when the libraries were used to build DL models, we discarded them because they do not build a model explicitly using Tensorflow/Keras APIs. For example, they get a specific configuration file or a code written by their own high-level APIs as input, and use it to construct a DL model. Therefore, it is impossible to use those examples since the scope of *NeuraLint* is defined to cover DL programs developed directly by employing Tensorflow/Keras built-in APIs. It should be noted that customized parsers can be developed to extract DL models from any configuration file or high-level code that is not currently covered by *NeuraLint* and then use our tool to find bugs/issues in them. After processing all these artifacts, we ended up with 26 buggy DL programs shared on SO (18 from our direct search and 8 from public datasets) and 8 from GitHub.

Since *NeuraLint* requires a full DL program to construct a graph on which the model verification is performed, we decided to prepare synthetic examples by a mix of synthetic code and reproduced real DL programs for the evaluation of *NeuraLint*. The reproduction of buggy DL programs from the SO posts is quite difficult when a major part of the code is not provided in the post. Anyway, to reproduce real buggy DL programs, we proceed as follows: (1) we first implement two well-known CNN applications, LeNet [40] on MNIST data [39] and VGG-16 [52] on Imagenet data [18], as base programs. To enhance diversity at technology level as well, we use both of our supported DL libraries, Tensorflow for LeNet and Keras for VGG-16 following, respectively, the official implementations [10] and [9] published on GitHub; (2) regarding implementation-related bugs, we inject each fault found, to one of the base DL programs; (3) regarding design-related issues, we poorly re-designed the structure of the base program's model to include inefficiencies violating the common patterns and best practices mentioned in Section 4. Finally, we constructed a total of 28 buggy synthetic programs which corresponds to one or two examples per detection rule. We constructed two faulty examples for a rule when there are two contexts in which the rule can be triggered, one example for each of these contexts. For instance, Rule 10, which validates the loss linkage, has been evaluated against both contexts of binary cross-entropy (used for binary class problem) and categorical cross-entropy (used for multi-class problem). For injecting bugs, we followed fault patterns observed in real buggy samples during our rule extraction process (illustrated on Figure 3). Hence, the injected bugs are realistic reproductions of faults. Our goal for evaluating *NeuraLint* using synthetic examples is debugging, i.e., making sure of its accuracy and effectiveness prior to evaluating it on real-world examples. For more details, please see our replication package containing all samples and implemented synthetic code [11].

Based on DL bug symptoms defined in [35], we found three bug symptoms in our studied DL programs: *i) Bad performance.* Bad or low performance is a common effect of conceptual issues related to design structure inefficiency or poor choices, misconfiguration of DL components; *ii) Incorrect Functionality.* This symptom refers to situations where the DL program behaves in an unexpected way without any runtime or compilation error. For instance, the DNN outputs only one label among class labels; *iii) Program Crash:* This bug effect is common for all software programs, it means that the program stopped running and raised an exception. Regarding the recommended fixes, we examined the accepted or endorsed answers of SO users to determine the bug-fixing repair (i.e., accepted) or recommendations provided to guide the user who asked the question towards finding the root cause of the error (i.e., open question).

6.2 Results

First, we have evaluated *NeuraLint* using 28 synthesized examples to investigate the correctness and preliminary effectiveness of the proposed approach. *NeuraLint* has successfully detected the bugs and issues in all synthetic examples. It should be noted that *NeuraLint* extracts the DL model from a DL program and employs graph transformations to apply the proposed rules on the model, not the code, to detect possible bugs/issues. For extracting the model from the code, *NeuraLint* relies on TensorFlow/Keras APIs and not on any particular patterns in the code. Moreover, we have not limited our experiments to these synthetic samples and have tested *NeuraLint* on real-world faulty DL programs. In other words, the tool is evaluated on faulty DL programs with bug patterns that were not considered when creating the tool or synthetic examples.

To evaluate practical effectiveness and accuracy of *NeuraLint*, 34 real-world DL programs from SO posts and GitHub repositories are used. Results are presented in Table 1 and Table 2. Table 1 reports results over DL programs extracted from SO posts. For each DL program, we report the ID of the post over SO, reported symptoms of buggy programs by the developer, fixes recommended by other users, output of *NeuraLint* (violated rules), number of true positive cases and false negative cases respectively. True positives are reported as a+b where a is the number of bugs/issues reported by

SO users that are detected by *NeuraLint*, and b is the number of bugs/issues detected by *NeuraLint* that are not mentioned by SO users. For b, two of the authors independently have checked each program and the output of *NeuraLint* manually to ensure that the output is correct. The total number of false positive cases is zero, so we do not report them. It is well-known that the best practice is analyzing SO posts with accepted answers (No. 1 to 20 in Table 1) ensuring the proposed solution is a real fix and addresses the mentioned problem. In our searching process for faulty samples, however, we have encountered 6 posts in SO without accepted answers (No. 21 to 26 in Table 1) containing relevant DL buggy programs or code snippets. Although none of the provided answers in these posts were accepted by the user who asked the question, we found at least one helpful and correct answer in the posts after a careful analysis. Specifically, one of the authors has manually inspected answers to make sure that SO users pointed out a right solution to the problem according to our verification rules. This process has been verified by another author assuring that we have a correct assessment and that the output of *NeuraLint* is accurate. Regarding samples from GitHub, the results are reported in Table 2. True positives are again reported as a+b where a is the number of bugs/issues that are successfully detected by *NeuraLint* according to reported problems in GitHub or a previous research study as mentioned in Subsection 6.1. On the other hand, b is the number of bugs/issues detected by *NeuraLint* but not reported in GitHub or a previous study. Similar to what we have done for SO posts, we have checked each program and the output of *NeuraLint* manually to ensure that the output is correct. In all tables, the rules which detect the bug/issue as reported by the developer are highlighted in bold letters. Detailed information of each sample including the link to GitHub repositories are available in our replication package [11].

In total, 31 out of 44 bugs/issues are detected correctly by *NeuraLint*, so the recall is evaluated as 70.5 %. All of these bugs/issues were identified by users/developers or previous research studies. The recall for SO posts with accepted answers is 76.9 % (20 out of 26), for SO posts without accepted answers is 75 % (6 out of 8), and for GitHub samples is 50 % (5 out of 10). The precision is 100 % meaning that we do not observe any false positive case in our evaluation. Moreover, *NeuraLint* correctly detected 33 additional bugs/issues that were not reported by users/developers who commented on the SO posts or GitHub repositories. Most of them, 29 out of 33, are design issues. *NeuraLint* has successfully detected 64 bugs/issues in 34 real DL programs in overall.

While fewer bugs are detected by *NeuraLint* in GitHub samples compared to SO posts, more design inefficiencies are detected by *NeuraLint* in GitHub samples (14 in 8 samples). Also, since *NeuraLint* is based on a static analysis, being able to detect half of the faults contained in the studied Github projects is already an interesting feat, since it allows developers to catch them early on, before they have to run their programs. Based on these results, we can report that the performance of *NeuraLint* is noteworthy; about three-quarters of known bugs are successfully detected (recall) and a significant number of hidden bugs and design inefficiencies of DL programs. Moreover, its precision is 100 % meaning that while the tool may miss some faults in the evaluated DL programs (overall recall is 70.5 %), it never detects bugs/issues wrongly. The reason is that our detection process is based on proposed verification rules and we report their violations in DL models extracted from DL programs.

We have performed the experiments using a machine with Intel i7-9750H CPU and 16GB of main memory running Windows 10. The average execution time of *NeuraLint* for the studied TensorFlow and Keras samples are 1.800 and 2.049 seconds, respectively. It should be noted that graph checking (performed by GROOVE) consumes the main portion of the execution time, about 99.7 %. Our preliminary analysis revealed that the running time mainly depends on the number of layers of the DL model. Details of the execution time of *NeuraLint* for five real DL programs with different sizes are reported in Table 3. According to these results, the execution time of extracting and checking the graph increases as the number of layers grows. Extracting the graph is accomplished by single

Table 1. Results of validating *NeuraLint* using real DL programs selected from StackOverflow.

No.	SO #	Symptom	Recommended Fix	Violated Rules	TP	FN
1	44399299	Program Crash	Change the shape of the input layer	**7**	1+0	0
2	43464835	Program Crash	Change the shape of the input layer	-	0+0	1
3	42913869	Program Crash	Change the number of units for the output layer	3	0+1	1
4	48518434	Program Crash	Reduce spatial size of both Conv. filtering and pooling widows	**7**	1+0	0
5	40857445	Program Crash	Adding a flatten layer	**6**	1+0	0
6	50555434	Bad Performance	Use softmax activation instead of sigmoid and categorical_crossentropy loss instead MAE	**10**	1+0	0
7	46177505	Program Crash	Change spatial size of Conv. filtering and pooling widows	5, 10	0+2	1
8	50426349	Program Crash	Change the shape of the input layer	19, 20	0+2	1
9	38584268	Program Crash	Adding a flatten layer	**6**, 21	1+1	0
10	45120429	Program crash	Change the number of units for the output layer, Adding a flatten layer	**6, 19**, 10	2+1	0
11	45378493	Incorrect Functionality	Use a sigmoid for last layer activation	**5, 10**, 16, 19, 20	2+3	0
12	45711636	Program Crash	Use channels_last format for input data	**7**	1+0	0
13	34311586	Bad Performance	Remove the last layer activation	**5, 10**, 19	2+1	0
14	50079585_1	Bad Performance	Use softmax activation instead of sigmoid and categorical_crossentropy loss instead binary_crossentropy	-	0+0	1
15	50079585_2	Incorrect Functionality	Change the number of units for the output layer	**10**	1+0	1
16	51749207	Bad Performance	Use of sigmoid activation instead of softmax	**5, 10**, 19	2+1	0
17	53119432	Program Crash	Adding a flatten layer	**6**, 19	1+1	0
18	55731589	Program Crash	Use of 'same' instead of 'valid' for layer padding type	**7**	1+0	0
19	58844149	Bad Performance	Use of sigmoid as last layer activation	**5, 10**, 21	2+1	0
20	61030068	Program Crash	Adding a flatten layer	**6**	1+0	0
21	33969059	Bad Performance	Change the number of units for the output layer	**10**	1+0	0
22	44184091	Program Crash	Fix the limit size for input sequence data	15	0+1	1
23	44322611	Bad Performance	Prune the DNN, use RMSprop instead SGD	10, 20, 21	0+2	1
24	49117607	Program Crash	Reduce spatial size of both Conv. filtering and pooling widows	16	0+1	0
25	55776436	Bad Performance	Try Data augmentation, Regularization, filtering spatial size reduction, and DNN Depth Increase	**7, 16, 17, 20**	4+0	0
26	60566498	Bad Performance	Try Data augmentation and Hyperparameters Tuning	**15**, 16	1+1	0

or multiple passes through the code. Hence, the execution time grows linearly by the number of layers as adding/configuring each layer needs a few API calls in TensorFlow/Keras. On the other hand, GROOVE supports priority-based rule application as well as various search strategies to explore the full state space, i.e., checking and applying all applicable rules in each state [21]. We have used BFS and priority-based rule application to improve the efficiency. However, the execution

Table 2. Results of validating *NeuraLint* using real DL programs selected from GitHub.

No.	Symptom	Recommended Fix	Violated Rules	TP	FN
1	Bad Performance [4]	Changing the last layer activation	19, 20, 21	0+3	1
2	Bad Performance [2]	Changing layer dimensions	19, 20, 21	0+3	1
3	Bad Performance [5]	Changing layer dimensions (padding)	**4**, 19, 21	1+1	1
4	Bad Performance [6]	Adding a pooling layer	**5**, 16, 20, 21	1+3	1
5	Bad Performance [3]	Changing layer dimensions	**19**, 16, 20, 21	1+3	0
6	Bad Performance [7]	Adding ReLU activation to the last layer	**3**	1+0	0
7	Bad Performance [8]	Adding ReLU activation to the last layer	**3**	1+0	0
8	Program Crash [1]	Changing layer dimensions	19	0+1	1

time of graph checking grows faster than extracting the graph as the number of layers of the DL model increases. The running time of *NeuraLint* can be improved which is left for future work.

6.3 Limitations and Discussion

GitHub samples were developed by advanced developers and are more complex than SO posts. While the scope of *NeuraLint* is defined to cover convolutional architecture as a particular type of FNN designed mainly for classification of 2D images, audio spectrograms, or 3D videos, some of studied GitHub samples have used convolutional architecture for data generation (e.g., extraction of structural lines of images[1]) or text classification[2]. In another sample[3], developers concatenated outputs of multiple convolutional architecture, each layer taking all preceding feature-maps as input, which is not frequent in popular CNNs. Using popular convolutional architectures such as VGG, ResNet, or MobileNet as a part of a DL model or modifying them for particular tasks is also observed in studied samples from GitHub[4]. Although the proposed meta-model is capable of representing these models as FNNs, particular rules must be proposed to find faults and improve the accuracy of *NeuraLint* on these samples. The developer added a dropout layer after dense layers to fix the problem in one of GitHub samples[5]. Although, as mentioned in Subsection 2.2, regularization methods are required to improve the convergence and generalizability of DL models, we need more investigations for proposing a rule to detect lack of enough or proper regularizations.

The focus of the design and implementation of *NeuraLint*, is on faults that relate to structural (architectural) properties of DL programs rather than their dynamic properties that need the programs to be executed. In other words, there are some frequent types of bugs/issues in DL programs that could be detected without dynamic analysis of the DL program [31]. However, the lack of dynamic analysis of DL programs is a limitation of our approach. Such analysis would allow for the detection of runtime bugs and bugs/issues in training/inference of DL models. For example, in program No. 22 in Table 1, the mismatch shapes is caused by the size of loaded input size during the execution, it is a runtime bug and could not be detected by the current version of *NeuraLint*. Some other bugs in DL programs need in-depth runtime analysis. For example, using dropout before batchnorm makes the behavior of DNN different during training and evaluation phases. This is the case for program No. 15 in Table 1. In another sample[6], batch size has been modified to improve the performance of the learning phase which cannot be investigated without analyzing the learning performance during runtime. Detecting these faults is currently out of scope of *NeuraLint*

[1] https://github.com/hepesu/LineDistiller

[2] https://github.com/bwallace/rationale-CNN

[3] https://github.com/cmasch/densenet

[4] https://github.com/mateusz93/Car-recognition/commit/94b36ea

[5] https://github.com/dishen12/keras_frcnn/commit/38413c6

[6] https://github.com/taashi-s/UNet_Keras/commit/b1b6d93

Table 3. Execution time of *NeuraLint* for five real DL programs with different sizes (times are in seconds).

No.	Number of layers	Running time of Graph extraction	Running time of Graph checking
1	6	0.003	1.757
2	8	0.003	1.787
3	12	0.002	1.836
4	13	0.003	1.906
5	38	0.004	3.111

and to cover them, DL programs must be experimentally tested to assess their performance for the underlying problem and then detect it.

Another challenge that we faced is related to the multiple releases of TensorFlow library that significantly changed the API functions; which makes the graph generator mainly designed and implemented in regards to the 1.15 version, incapable of detecting some of the required components for versions other than 1.15.

Lack or limited access to real DL programs annotated with possible bugs, design inefficiencies and recommended fixes to evaluate DL testing approaches accurately and effectively could be regarded as a barrier in this line of research. Finally, the current version of *NeuraLint* could find problems in FNNs, particularly CNNs. Other neural network architectures, like recurrent neural networks are out of the scope of this version. *NeuraLint* could be applied to detect bugs/issues in such neural networks by extending the meta-model to capture their properties, i.e., possibility of connections between the nodes that form a cycle. Also, new rules could be defined to detect specific problems in each architecture according to frequent observed bugs/issues or best practices.

7 RELATED WORKS

The current work on model verification of DL programs using meta-modeling and graph transformations is related to a number of approaches in the literature.

Meta-modeling and deep learning: Perez has introduced the idea of meta-model for deep learning in [48]. After sketching a model for deep learning by noting UML meta-modeling, he argued that entities like Layers, Objectives, Activations, Optimizers, Metrics in high-level DL libraries are meta-models for deep learning. However, the paper keeps its interesting ideas in a high-level of abstraction. A meta-model for meta-learning is presented in [25], alongside a discussion of its possible integration into state-of-the-art modelling frameworks. Meta-learning or automatic machine learning is a recent approach that applies learning algorithms on metadata about machine learning experiments to automatically improve the learning process. The concept of meta-learning is viewed from a meta-modeling point of view and its variabilities are described. Although the presented meta-model has similarities with our meta-model, the scope is different: while they focused on meta-learning, in this paper, we present a meta-model of DL programs. Moreover, [25] did not explore the possibility of identifying bugs in learning programs through model verification.

Graph representation and transformations for program verification: Ciraci et al. have proposed a graph-based automatic process for verification of static program constraints on elements like macros and comments [17]. Their main contribution is introducing a meta-model which is called Source Code Modeling Language (SCML) to express constraints over program elements. *NeuraLint* also employs meta-modelling and a graph-based approach while it focuses on detecting bugs and bad practices in DL programs. Hoppity is a tool to detect and fix bugs in Javascript programs [19]. The researchers formulate the problem as learning some consecutive graph transformations. The buggy program is modelled by a graph structure, and then Hoppity predicts the location and type of bugs and some graph edits to fix the detected bugs. Hoppity employs the abstract syntax tree

(AST) to model the buggy program as a graph while we have used a meta-model and then extract relevant information from AST to build the model. Moreover, we propose the graph transformations for localization and detection of bugs in *NeuraLint* without adoption of a learning algorithm. Researchers have proposed other approaches [12, 37] that have learned graphs from the source code of programs (as proper representations of program semantics) for bug detection. However, in *NeuraLint* we do not use learning algorithms to find or configure graphs/graph transformations.

Testing Different Aspects of DL Software Systems: Over the last few years, researchers have developed multiple testing approaches for DL software systems [15]. While the larger part of these approaches test the exactness of the model predictions against instances from the input data distribution [13, 47, 58], a few other approaches [14, 20, 51] focused on the validation that DL training programs are bug-free. Dwarakanath et al. [20] have proposed high-level metamorphic relations that can detect bugs altering substantially the program behavior, but they do not provide guidance to identify the root causes and have not studied real buggy DL software. In contrast, *NeuraLint* is able to detect common DL software bugs and to steer the developer towards identifying and fixing the root cause. Selsam et al. [51] leverage machine-checkable proofs to validate the math consistency of the DNN training program. However, a proof-based approach could be difficult to adopt in practice because it requires code written using low-level mathematical libraries, and nowadays, DL practitioners mostly use third-party DL libraries to construct reliable and scalable DL software systems. On the contrary, *NeuraLint* is quite extensible and can cover any API by constructing a new graph generator dedicated to the target library. Braiek and Khomh [14] proposed, TFCheck, a property-based testing method for DNN training programs, which continuously collects metrics on training dynamics and performs rule-based assertions to validate that a DNN program behaves properly. Nevertheless, it requires a running DNN training program, which is unfortunately not always available. For instance, an error of mismatched shapes leads the graph construction to fail and cause the crash of the DNN program. To fill-in the gap, *NeuraLint* detects issues in the DL software program through its written code without even a single run.

8 CONCLUSION

In this paper, we have introduced *NeuraLint*, a model-based verification approach for DL programs. As a model-driven approach, a meta-model has been proposed for DL programs at first. We have defined a set of verification rules for DL programs based on the meta-model. A model of each DL program is configured by parsing its code to extract relevant information. Afterward, a graph checking process is performed to verify the model and detect potential bugs or design inefficiencies. Graph transformation systems are used to implement the verification rules and modeling approach. The meta-model is represented by a type graph, DL programs are modeled as host graphs, and graph transformations execute the verification rules. *NeuraLint* has been evaluated using synthetic and real DL programs. The results show that *NeuraLint* effectively detects faults and design issues in both synthetic and real-world DL programs with a recall of 70.5 % and a precision of 100 %. Refining (or even redesigning) the meta-model and graph transformations are required to improve the accuracy and detect false negative cases. The proposed meta-model is designed for FNNs but can be extended to support other neural network architectures such as recurrent neural networks. Another direction of research for the future is expanding our set of verification rules to cover more types of issues occurring in DL programs. While *NeuraLint* currently supports DL programs developed by TensorFlow/Keras, it can be extended to detect bugs/issues for other DL libraries (like PyTorch) by extending its parser to cover specific APIs of each DL library.

ACKNOWLEDGMENTS

This work is partly funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) via the DEpendable Explainable Learning (DEEL) project, the Fonds de Recherche du Québec (FRQ), and Institut de VAlorisation des DOnnées (IVADO).

REFERENCES

[1] 2016. https://github.com/katyprogrammer/regularization-experiment/commit/b93dd636. Accessed: 2021-02-05.

[2] 2017. https://github.com/dishen12/keras_frcnn/commit/38413c6. Accessed: 2021-02-05.

[3] 2017. https://github.com/yumatsuoka/comp_DNNfw/commit/30e0973. Accessed: 2021-02-05.

[4] 2018. https://github.com/taashi-s/UNet_Keras/commit/b1b6d93. Accessed: 2021-02-05.

[5] 2018. https://github.com/keras-team/keras-applications/commit/05ff470. Accessed: 2021-02-05.

[6] 2018. https://github.com/mateusz93/Car-recognition/commit/94b36ea. Accessed: 2021-02-05.

[7] 2018. https://github.com/tf-encrypted/tf-encrypted/issues/248 (network B). Accessed: 2021-02-05.

[8] 2018. https://github.com/tf-encrypted/tf-encrypted/issues/248 (network C). Accessed: 2021-02-05.

[9] 2020. *Keras-based VGG16.* https://github.com/keras-team/keras-applications/blob/master/keras_applications/vgg16.py.

[10] 2020. *Tensorflow-based LeNet.* https://github.com/tensorflow/models/blob/master/research/slim/nets/lenet.py.

[11] 2021. *Replication package and source code of NeuraLint.* https://github.com/neuralint/neuralint.

[12] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations.* https://openreview.net/forum?id=BJOFETxR-

[13] Houssem Ben Braiek and Foutse Khomh. 2019. DeepEvolution: A Search-Based Testing Approach for Deep Neural Networks. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 454–458.

[14] Houssem Ben Braiek and Foutse Khomh. 2019. TFCheck: A TensorFlow Library for Detecting Training Issues in Neural Network Programs. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS).* IEEE, 426–433.

[15] Houssem Ben Braiek and Foutse Khomh. 2020. On testing machine learning programs. *Journal of Systems and Software* 164 (2020), 110542.

[16] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2016. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678* (2016).

[17] Selim Ciraci, Pim van den Broek, and Mehmet Aksit. 2010. Graph-based verification of static program constraints. In *Proceedings of the 2010 ACM Symposium on Applied Computing.* 2265–2272.

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition.* Ieee, 248–255.

[19] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations. h ps://openreview. net/forum.*

[20] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 118–128.

[21] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. 2012. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer* 14, 1 (2012), 15–40.

[22] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics.* 249–256.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

[24] Dongyoon Han, Jiwhan Kim, and Junmo Kim. 2017. Deep pyramidal residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 5927–5935.

[25] Thomas Hartmann, Assaad Moawad, Cedric Schockaert, Francois Fouquet, and Yves Le Traon. 2019. Meta-Modelling Meta-Learning. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS).* IEEE, 300–305.

[26] Seyyed Hossein Hasanpour, Mohammad Rouhani, Mohsen Fayyaz, Mohammad Sabokrou, and Ehsan Adeli. 2018. Towards principled design of deep convolutional networks: introducing SimpNet. *arXiv preprint arXiv:1802.06205* (2018).

[27] Soufiane Hayou, Arnaud Doucet, and Judith Rousseau. 2019. On the impact of the activation function on deep neural networks training. In *International Conference on Machine Learning.* PMLR, 2672–2680.

[28] Kaiming He and Jian Sun. 2015. Convolutional neural networks at constrained time cost. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5353–5360.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[30] Reiko Heckel. 2006. Graph transformation in a nutshell. *Electronic notes in theoretical computer science* 148, 1 (2006), 187–198.

[31] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1110–1121.

[32] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. 2014. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869* (2014).

[33] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[34] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. PMLR, 448–456.

[35] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.

[36] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 1135–1146.

[37] Roshni G Iyer, Yizhou Sun, Wei Wang, and Justin Gottschlich. 2020. Software Language Comprehension using a Program-Derived Semantic Graph. *arXiv preprint arXiv:2004.00768* (2020).

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[39] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[40] Yann LeCun et al. 2015. LeNet-5, convolutional neural networks. *URL: http://yann.lecun.com/exdb/lenet* 20, 5 (2015), 14.

[41] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[42] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[43] Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. 2019. Understanding the disharmony between dropout and batch normalization by variance shift. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2682–2690.

[44] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 740–755.

[45] Dmytro Mishkin, Nikolay Sergievskiy, and Jiri Matas. 2017. Systematic evaluation of convolution neural network advances on the imagenet. *Computer Vision and Image Understanding* 161 (2017), 11–19.

[46] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. 2018. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378* (2018).

[47] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

[48] Carlos E. Perez. 2016. *The meta model and meta meta-model of deep learning*. https://medium.com/intuitionmachine/the-meta-model-and-meta-meta-model-of-deep-learning-10062f0bf74c

[49] Arend Rensink. 2003. The GROOVE simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*. Springer, 479–485.

[50] Dominik Scherer, Andreas Müller, and Sven Behnke. 2010. Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*. Springer, 92–101.

[51] Daniel Selsam, Percy Liang, and David L Dill. 2017. Developing bug-free machine learning systems with formal mathematics. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 3047–3056.

[52] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[53] Leslie N Smith and Nicholay Topin. 2016. Deep convolutional neural network design patterns. *arXiv preprint arXiv:1611.00847* (2016).

[54] Richard Mark Soley. 2013. *How to Deliver Resilient, Secure, Efficient, and Easily Changed IT Systems in Line with CISQ Recommendations*. https://www.omg.org/news/whitepapers/CISQ_compliant_IT_Systemsv.4-3.pdf

[55] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* (2014).

[56] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[57] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[58] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 303–314.

[59] Chengwei Zhang. 2018. *How to choose Last-layer activation and loss function.* https://www.dlology.com/blog/how-to-choose-last-layer-activation-and-loss-function/

[60] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael R Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *The 30th IEEE International Symposium on Software Reliability Engineering (ISSRE)*.

[61] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.