Design by Contract for Deep Learning APIs

Shibbir Ahmed
Dept. of Computer Science, Iowa State University
Ames, IA, USA
shibbir@iastate.edu

Sayem Mohammad Imtiaz
Dept. of Computer Science, Iowa State University
Ames, IA, USA
sayem@iastate.edu

Samantha Syeda Khairunnesa
Dept. of Computer Science and Information Systems, Bradley University
Peoria, IL, USA
skhairunnesa@fsmail.bradley.edu

Breno Dantas Cruz
Dept. of Computer Science, Iowa State University
Ames, IA, USA
bdantasc@iastate.edu

Hridesh Rajan
Dept. of Computer Science, Iowa State University
Ames, IA, USA
hridesh@iastate.edu

ABSTRACT

Deep Learning (DL) techniques are increasingly being incorporated in critical software systems today. DL software is buggy too. Recent work in SE has characterized these bugs, studied fix patterns, and proposed detection and localization strategies. In this work, we introduce a preventative measure. We propose design by contract for DL libraries, *DL Contract* for short, to document the properties of DL libraries and provide developers with a mechanism to identify bugs during development. While *DL Contract* builds on the traditional design by contract techniques, we need to address unique challenges. In particular, we need to document properties of the training process that are not visible at the functional interface of the DL libraries. To solve these problems, we have introduced mechanisms that allow developers to specify properties of the model architecture, data, and training process. We have designed and implemented *DL Contract* for Python-based DL libraries and used it to document the properties of *Keras*, a well-known DL library. We evaluate *DL Contract* in terms of effectiveness, runtime overhead, and usability. To evaluate the utility of *DL Contract*, we have developed 15 sample contracts specifically for training problems and structural bugs. We have adopted four well-vetted benchmarks from prior works on DL bug detection and repair. For the effectiveness, *DL Contract* correctly detects 259 bugs in 272 real-world buggy programs, from well-vetted benchmarks provided in prior work on DL bug detection and repair. We found that the *DL Contract* overhead is fairly minimal for the used benchmarks. Lastly, to evaluate the usability, we conducted a survey of twenty participants who have used *DL Contract* to find and fix bugs. The results reveal that *DL Contract* can be very helpful to DL application developers when debugging their code.

ESEC/FSE 2023, 3 - 9 December, 2023, San Francisco, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. 2023. Design by Contract for Deep Learning APIs. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep learning is a popular tool for solving complex software development problems such as NLP and vision, but research has shown that deep learning models also have unique bugs [28, 30, 31, 55]. To address this, SE researchers have focused on detecting and localizing these bugs [39, 46, 51]. In this work, we explore an alternative approach to improve the reliability of deep learning software, design by contract (DbC). Traditional DbC [17, 34, 36, 41] provides support for writing preconditions and postconditions at APIs. However, prior work does not provide mechanisms for documenting properties of the model architecture, data, and training process, which are crucial for applying DbC to deep learning APIs. Recent research has proposed techniques for inferring these properties, but DbC aims to provide specification mechanisms for programmers.

We propose a DbC methodology for deep learning libraries, called *DL Contract*. It exposes meta-level properties of the DL training process and model structure as variables, called *ML variable*, for use in writing contracts. Unlike grey-box contracts [18] that expose part of the program, *ML variable* provide a higher level abstraction of the training process and model structure. They are similar to specification-only fields [22] in object-oriented programs [35, 42], but abstract away from the details of the DL model.

We have developed *DL Contract* for Python and a runtime assertion checking framework for *DL Contract*. We have applied contracts to key API methods of the *Keras* library and evaluated them using four benchmarks for deep learning bug detection from prior works [43, 46, 51, 54], comprising 272 *Keras* codes. Our results show that the *Keras* library with contracts can identify 95% of such bugs during runtime checking. Additionally, we have evaluated the annotation overhead of *DL Contract* and found it to be zero for users of DL libraries. This means that users do not need to add any contract annotations to their code in order to benefit from our approach. We have also added 15 contracts to the model compilation and training

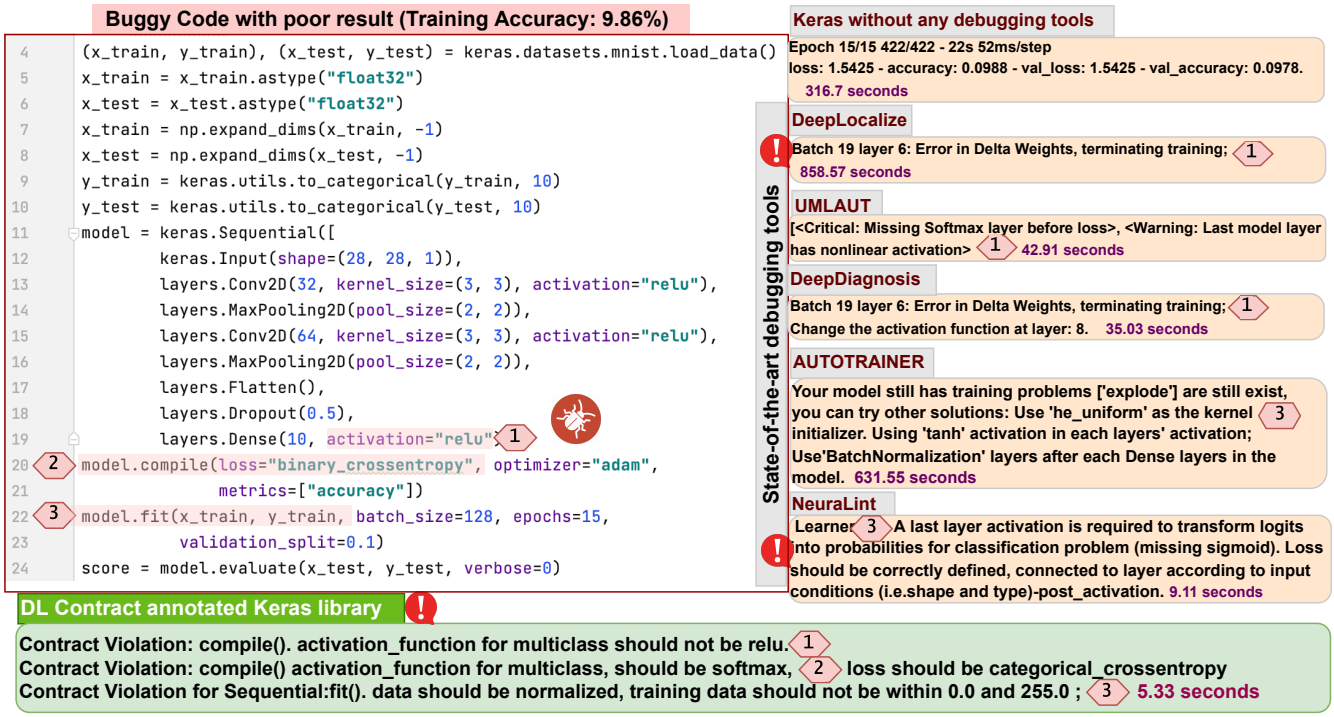

Figure 1: Buggy code [2, 3, 6, 7, 9] achieves 9.78% training accuracy. Similar correct code [10] achieves ~ 99% training accuracy.

methods of the *Keras* API and evaluated 257 correct programs, finding 18 false positives due to the randomness effect during training. To evaluate the usability of the contract-enabled *Keras* library, we conducted a user study with 20 participants with varying levels of expertise in DL application development. Our evaluation also shows that the runtime overhead of checking contracts is fairly minimal. We found that the runtime overhead increases by around 15% compared to the baseline. *DL Contract* can be disabled during production to result in zero overhead.

Our contributions are as follows:

- A novel methodology for writing and checking contracts for deep learning libraries by specifying DL APIs with preconditions and postconditions.
- A framework which is extensible and generalized to different class of DL bugs and maps contract violation as bug, symptoms as constraints to check and contract violation message as suggestion to fix those bugs.
- The notion of specifying DL-specific contracts by abstracting the DL model architecture, its data properties, and training behavior.
- A collection of 15 contracts that prevents prevalent training problems and structural bugs in DL programs.
- An annotated version of *Keras* with the *DL Contract* as a virtual environment (*@Keras*). Developers can use this *@Keras* environment for debugging without any annotation overhead and minimal runtime overhead (≈15%).

2 MOTIVATION

To highlight the difficulty in specifying deep learning APIs and the need for *DL Contract*, consider a simple Convolutional Neural Network (CNN) code shown in Fig. 1. This code is intended for digit classification and when implemented correctly, as outlined in the *Keras* documentation [10], it achieves 99% training accuracy on the MNIST dataset. In the correct version, images are normalized to the range [0,1] before being processed by a Sequential model with a specific layer architecture. The model is configured using the `Compile` API and trained using the `Fit` API, and the `evaluate` API is used to calculate the loss and accuracy. However, as shown in Fig. 1, the code snippet contains three bugs (on lines 19, 20, and 22) which result in low accuracy and high training time. These bugs are specific to DL programs [51] and may not cause crashes. For example, on line 19, the incorrect activation function, 'relu' is used in the last layer of the `Dense` API [2, 5, 6]. Additionally, on line 20, the incorrect loss function of 'binary_crossentropy' is applied in the `Compile` API [2, 3, 9]. Lastly, on lines 5 and 6, the data is not normalized before being fed into the `Fit` API [6, 7].

This example also illustrates another challenge for specifying DL APIs. All DL APIs work on a shared DL model, where early APIs construct the model and later APIs, such as `fit`, `compile`, and `evaluate`, make use of it. To write pre/postconditions for DL APIs, having access to only the formal parameters and return values of the APIs is not sufficient. Correct usage depends on the state of the model at the point of the API call. *DL Contract* addresses these challenges and can help prevent such bugs by providing a clear specification of the intended behavior of deep learning APIs.

3 DEEP LEARNING CONTRACTS

In the *DL Contract* approach, we abstract the data properties, expected output, model architecture, and training behavior of a DNN model and specify the properties of DL APIs connected via a computation graph. We gather and inspect necessary conditions from three sources (details in §4.1). We filter out the obligations from the DL app developer as *preconditions* and expectations from DL software in as *postconditions*. Here, we use a novel runtime assertion check in DL computation. In the contract checker modules first parse those contracts and translate them into templates. Those templates are validated to handle the exception if it occurs. If a contract is violated, the user receives a contract violation message Otherwise, the API returns the normal execution output. Thus, our proposed solution generalizes to other bugs and model categories in this way. It would be easy for library developers to specify the contracts for other types of bugs following these procedures of *DL Contract*.

Next, we present the design and usage of *DL Contract*, including examples and our approach for abstracting DL related properties.

3.1 Writing Deep Learning Contract

DL Contract uses an annotation-based approach [16, 23] to add contracts to DL APIs, which allows library developers to add contracts without modifying compilers and build tools. This means that software using DL APIs does not need to be modified. DL library developers can add *preconditions* that must be satisfied before the API is called and *postconditions* that the API guarantees to be true upon completion.

3.1.1 Syntax. To use contracts in a deep learning library, it is necessary to annotate the API with `@contract` and `@new_contract`. This allows library developers to create expressions for checking specified contracts. *DL Contract* can check types such as tensors and model objects, as well as simple data types like strings, floats, numbers, arrays, and booleans. It utilizes logical operators like AND(,) and OR(|) and allows for arithmetic and comparison expressions. Additionally, *DL Contract* can be used to check constraints of various model properties during training and abstraction.

3.1.2 Illustrative Example. To create a contract, a library developer annotates a DL API using `@contract` and `@new_contract`. Inside `@contract`, the developer defines types and functions for checking contracts. Using `@new_contract`, the developer writes functions for performing computations necessary for a contract and for checking preconditions and postconditions. For example, in Listing 3.1, a contract is imposed as a precondition on the *Keras* training function `fit` to ensure that data is within a specified range before training. To prevent this type of bug, a function `data_normalization` is declared as a contract definition using the `@contract` annotation (line 8) using the parameter `x`. Inside the `@contract` annotation, in the `data_normalization` function (line 2), the developer further computes to get the range of training data, declared as `normalization_interval` as a *ML variable* (line 3). The developer can specify the appropriate range of the *ML variable* within the contract checker function. The condition is checked on line 4 and if the contract is violated, a suggestion to fix the issue is raised on line 7.

```
1 | @new_contract
2 | def data_normalization(x):
3 |   normalization_interval = np.max(x) - np.min(x)
4 |   if(normalization_interval>2.0):
5 |       msg = "Data should be normalized before training,train and
6 |       test data should be divided by value " + str(np.max(x))
7 |       raise ContractException(msg)
8 | @contract(x='data_normalization')
9 | def fit(self, x=None, y=None,...):
```

Example 3.1: A contract on `Fit` API inside *Keras* library

When a buggy DL program makes use of this annotated API, *DL Contract* will throw the following error.

```
| ContractViolated: Data should be normalized before training, train and test
  data should be divided by value 255.0.
```

Example 3.2 illustrates the use of *DL Contract* to prevent overfitting bugs [39], in which a model has high training accuracy but low test accuracy. A contract is specified on the validation loss and training loss to check for increasing difference in validation loss and decreasing difference in training loss [46], which is a common cause of overfitting. This expectation is encoded as a postcondition.

```
1 | @new_contract
2 | def overfitting(history):
3 |   i=0
4 |   while i<=(len(history.epoch)-2):
5 |       epochNo = i + 2
6 |       diff_loss = history['loss'][i + 1] - history['loss'][i]
7 |       diff_val_loss = history['val_loss'][i + 1] -
8 |       history['val_loss'][i]
9 |       i += 1
10|       if(diff_val_loss>0.0):
11|           if(diff_loss<=0.0):
12|               msg = "After Epoch"+str(epochNo)+",diff_val_loss="
13|               +str('%.4f' % diff_val_loss)+"and diff_loss="
14|               +str('%.4f' % diff_loss) + "causes overfitting"
15|               raise ContractException(msg)
16| @contract(returns='overfitting')
17| def fit(self, x=None, y=None,...): return self.history
```

Example 3.2: Overfitting Contract on `Fit` API

To prevent overfitting, a contract can be added to the output of the `Fit` method in *Keras* using `@contract` and a postcondition can be checked using the `overfitting` function specified with `returns` (line 16). In this function, the contract writer uses the obtained history object to compute `diff_loss` and `diff_val_loss` (line 6-7) and checks if the difference between validation loss of consecutive epochs tends to increase while the difference between training loss continues to decrease. If this condition is not met, a contract violation message is thrown and when a buggy DL program uses this annotated API, *DL Contract* will throw an error.

```
| ContractViolated: After Epoch: 11, diff_val_loss = 0.34 and diff_loss = -0.12
  causes overfitting.
```

3.2 DL Contract Approach

Next, we present our approach and describe technical challenges in DL contract checking, such as the need for context-aware *ML variable* (§3.2.1), assertion techniques (§3.2.2), and support for contracts across multiple APIs in the ML pipeline (§3.2.3). We also discuss our technique's support for post-training contract checking (§3.2.4).

3.2.1 ***Abstraction of DL specific properties to contracts.*** To enforce DbC technique for deep learning APIs, a mechanism is needed to capture model abstraction, data properties, and training behavior beyond just the formal parameters and return values of the DL APIs. Standard contracts only enforce constraints on the values of formal parameters and return values of an API method

or attributes of an API class. Additionally, machine learning APIs are not isolated, but connected through a computational graph [15]. Therefore, specifying contracts on one API with its formal parameters alone is not sufficient in the DL specific settings.

Fig. 2 describes a scenario in which the developer wants to add a contract to the method `dense` to ensure that the activation function for the last layer is not `relu` [8]. Additionally, the developer wants to check the appropriate loss function parameter for the `Compile` API Fig. 2. The problem with this scenario is that the conventional Design by Contract (DbC) technique cannot specify this contract on a model's API without causing false alarms in correct codes because it only allows for checking contracts on each API of a model.

To solve such problem, we design a way to write *DL Contract* using functions that allows to compute subset of meta-information with *ML variable* abstracting model architecture, data properties, training behavior. Fig. 2 shows one way to solve this challenge using *DL Contract*. In this solution, `activation`, and `loss_func` are computed in specified `@new_contract contract_checker` functions where `activation` is the parameter of last layer `Dense` API and `loss_func` is the parameter of `Compile` API. This is how *DL Contract* mechanism enables specifying and checking contract with abstracted model properties which works on any stage of computation graph pipeline.

3.2.2 ***DL Contract runtime assertion technique.*** A model is more than what the configuration script defines. Many properties of the model only become tractable during training. As a result, a *DL Contract* must enable a runtime assertion technique that allows enforcing contracts beyond formal parameters, unlike traditional contract checkers. Furthermore, it must be possible to impose contracts on different pipeline stages of the modeling, i.e., data preprocessing, during model building, and training, etc. To that end, we propose a *DL Contract* checker with such capabilities by enabling library developers to annotate APIs. Eventually, *DL Contract* annotations benefit end-users to check their model, data properties, and training behavior at different stages in the DL pipeline.

Our method outlined in Algorithm 1 shows the steps involved in parsing and checking contracts in a library. It consists of two steps: registering new contracts defined by the library developer and parsing and validating newly defined contracts applied to the functions defined by the library developer. The framework inspects the library code base to find custom user-defined contracts defined as functions with the `@new_contract` annotation. The usage of `@new_contract` on a function invokes the `register_new_contract` method, which stores a reference to the function in a dictionary. This way of annotating contracts allows writing contracts using abstracted DL properties as discussed in section 3.2.1. For instance, if a library developer writes a contract with any of the properties of *model* object and checks as a precondition before model compilation or before model training, our technique allows doing that in this way (more details in Example 3.3) which is different than the traditional way of writing contract. The `contract_checker` method is used to intercept and validate such contracts applied to user-defined functions with the `@contract` annotation before the function is executed. The method parses the annotation reference, obtains a dictionary of conditions applied to the function's arguments, and validates the conditions using the visitor design pattern.

Algorithm 1 DL Contract Checker

```
1:  procedure CONTRACT_CHECKER (USERFREF, ANNOTEREF)
2:      fArgs ← formal_arguments(userFRef)
3:      argContrDict ← parse_contract(annotateRef)
4:      for each (fArg, cond) in argContrDict do
5:          aArg ← actual_arguments(fArg, userFRef)
6:          template ← parse_template(cond)
7:          template.check_contract(aArg)
8:      returnCondition ← parse_contract(annoteRef)
9:      aArgs ← actual_arguments(userFRef)
10:     result ← userFRef(aArgs)
11:     returnTemplate ← parse_template(returnCondition)
12:     returnTemplate.check_contract(result)
13:     return result
14: procedure REGISTER_NEW_CONTRACT (FUNCREF)        ▷ @new_contract
15:     identifier ← getFuncName(funcRef)
16:     newContRegister[identifier] ← funcRef
17: procedure PARSE_TEMPLATE (COND)
18:     if len(cond) > 1 then                        ▷ multiple conditions
19:         subclauses ←[ ]
20:         for  c ∈ cond do
21:             subclauses ← parse_template(c)
22:         return And(subclauses)
23:     if istype(cond) then                         ▷ if it is cond type
24:         return CheckType(cond)
25:     if cond ϵ newContRegister then               ▷ if it is callable
26:         return CheckCallable(newContRegister[cond])
```

Consider a contract, `@contract(loss='str,contract_func')`. It validates the loss function and the validation takes place inside a user-defined contract, `contract_func`. The contract body is stored in `argContrDict` as `<loss,(str,contract_func)>`. Then, it obtains the value for the argument `loss`. The method `parse_template` is used to obtain a validation tree for the conditions by composing validation classes (in Algorithm 2). In the example of `loss` contract, an `And` class is obtained, with each condition as a subclause. If the first condition, `str`, is satisfied, a `CheckType` validation class is returned. If the second condition is a user-defined function, a `CheckCallable` validation class is returned. The composed validation tree is returned in a template variable. Each validation class implements the method `check_contract`. To validate the template, `check_contract` is invoked on the root validation class, which is `And`. If validation fails for any subclause, `And` raises an exception. The argument on which a contract is imposed is validated. If preconditions are satisfied, the postconditions are validated. The returned result of the user function is validated as per written contracts.

3.2.3 ***Contextualized Inter-API Call Contracts.*** The next challenge is to ensure that *DL Contract* can be written involving multiple APIs at different stages of the DL pipeline. To solve this problem, *DL Contract* is designed to write multiple functions using `@new_contract` annotations that takes formal parameters across multiple DL APIs. For example, when the number of the target class is 2 (i.e., binary classification), the activation function of the last layer should not be `softmax` or `relu` [3, 5, 9] (which is a type of contract within the same `Dense` API) and loss function should be 'binary_cross-entropy' [2, 3] (which is an inter-argument contract with different APIs, i.e., between last layer and compile API). Although the best activation function for hidden layers is ReLu [25], if ReLu is used on the last layer, it will set all the negative output to zero, thus leading to accuracy problem. To prevent such kinds of problems in model architecture, library developers can write *DL*

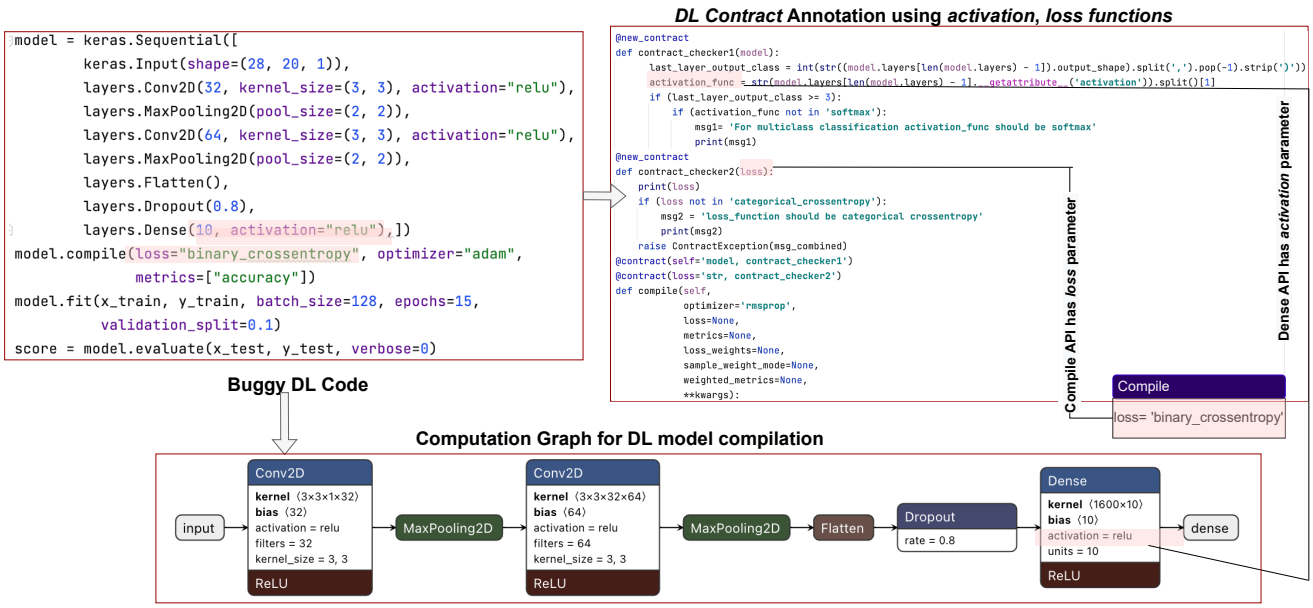

Figure 2: *DL Contract* approach using *activation* and *loss* functions involving multiple APIs in DL computation graph

Algorithm 2 Check Contract

```
1: class CheckContract
2:     procedure abstract check_contract (value)
3: end class
4: class CheckCallable(CheckContract)
5:     procedure init (funcRef)
6:         callable ← funcRef
7:     procedure check_contract (value)
8:         if callable(value) raised exception then
9:             raise contractException()
10: end class
11: class And(CheckContract)
12:     procedure init (subclauses)
13:         subclauses ← subclauses
14:     procedure check_contract (value)
15:         for sc ∈ subclauses do
16:             if sc.check_contract(value) raised exception then
17:                 raise contractException()
18: end class
19: class CheckType(CheckContract)
20:     procedure init (type)
21:         expected_type ← type
22:     procedure check_contract (value)
23:         actual_type ← getActualType(value)
24:         if actual_type ∉ expected_type then
25:             raise contractException()
26: end class
```

Contract using the activation and loss function for the binary and multi-class classification according to the experts' suggestion [2, 3]. Our insight is that such types of contracts can be added to deep learning model-compilation API, i.e., *Keras* `Compile`, exposing objects capturing the entire model properties.

```
1 | @new_contract
2 | def contract_checkerfunc1(model):
3 |     last_layer_output = int(str((model.layers[len(model.layers)
4 |     - 1]).output_shape).split(',').pop(-1).strip(')'))
5 |     activation_func = str(model.layers[len(model.layers) - 1].
6 |     __getattribute__('activation')).split()[1]
7 |     if (last_layer_output >= 3):
8 |         if (activation_func not in 'softmax'):
9 |             msg1= 'For multiclass classification activation_func
10|             should be softmax'
11|             raise ContractException(msg1)
12| @new_contract
13| def contract_checkerfunc2(loss):
14|   if (loss not in 'categorical_crossentropy'):
15|       msg2 = 'loss should be categorical crossentropy'
16|       raise ContractException(msg2)
17| @contract(self='model, contract_checkerfunc1')
18| @contract(loss='str, contract_checkerfunc2')
19| def compile(self,optimizer='rmsprop',loss=None,metrics=None,...):
```

Example 3.3: Last layer activation and loss function contract on *Keras* `Compile` API

Example 3.3 shows last layer activation and loss function contract applied to *Keras* `Compile` API which asserts before compile API execution. Here, `contract_checker1` has been annotated with `model` object type on line 17 and `contract_checker2` has been annotated using `loss` parameter with string type on line 18. Here, `last_layer_output` and `activation_func` is computed on line 3 and line 5 from `model` object. The loss function has been a formal parameter of `Compile` API, and `contract_checkerfunc2` checks the condition on line 14 and shows a message with suggestions to fix if a contract violation occurs for both `Dense` and `Compile` APIs. Here, 'contract_checkerfunc2' is only executed if 'contract_checkerfunc1' is executed, as those specified contacts are ANDed one after another for one contract (last layer activation and loss function). Since 'contract_checkerfunc1' checks whether the number of classes $\geq$ 3, then 'checkerfunc2' would also know if the program runs a multiclass classification. In Example III.3, on lines 17–18, 'contract_checkerfunc1' and 'contract_checkerfunc2' have been enforced together. A case of violation of that contract is shown below,

```
| ContractViolated: For multiclass classification activation_func should be
  softmax, loss should be categorical crossentropy.
```

3.2.4 ***Post-training Contracts.*** The challenge of capturing DNN training behavior at different stages of the DL pipeline can be addressed with our proposed *DL Contract*. Library developers can

specify desired training behavior for their DL software by adding training-related contracts on properties such as, gradients rate, gradients percentage etc. Training behavior-related properties indicate the expected output from the DL model, so this is a postcondition. The root cause behind a training problem could be client obligation in hidden layers APIs such as activation function which is a parameter of `Dense` API (this is a precondition) We might encounter such types of preconditions and postconditions in DL-specific settings and contracts can be specified using `@new_contract` and `@contract` annotations in our proposed approach. To handle such cases, *DL Contract* advocates specifying contracts as postconditions on DL training APIs, e.g., *Keras* `Fit` API, which provides detailed training history. Based on supplied contract checking function in `@new_contract`, we compute relevant training properties from the history object such as *validation accuracy*, *loss value*, *gradient rate* etc. Algorithm 1 (lines 8–13) describes how we check and validate postconditions in our framework. Example 3.2 demonstrates this type of postcondition contract.

4 EVALUATION

In this section, we aim to answer the following research questions:

- **RQ1 (Effectiveness)**: How effective is *DL Contract* in real world programs?
- **RQ2 (Applicability)**: Is *DL Contract* enabled *Keras* applicable to find performance (i.e., low accuracy, high training time) bugs?
- **RQ3 (Efficiency)**: How efficient is *DL Contract* for detecting DL performance bugs in terms of precision and recall?
- **RQ4 (Overhead)**: What is the overhead of the *DL Contract* compared to related works in terms of runtime?
- **RQ5 (Usability)**: How useful is the *DL Contract* enabled Keras in developing DL Apps?

First, in order to evaluate our approach, we collect contracts by following the procedure described in §4.1. We implemented *DL Contract* (in §4.2) using our proposed approach (in §3.2). Then we conducted experiments using the setups (in §4.3). Finally, we report results and analysis (in §4.4).

4.1 Deep Learning Contracts Collection

In this section, we describe the process of contract collection used in the evaluation. We have identified contracts related to the model, data, and training properties. These contracts prevent structure and training bugs, which lead to performance issues (i.e., low accuracy, high training time). DL libraries like *Keras* does not provide error messages for such types of bugs yet. Fig. 3 shows how we collected the conditions of *DL Contract*. In 1, we abstract the data properties, expected output, model architecture, training behavior of a DNN model. In 2, we gather and inspect necessary conditions from three sources. We used the official *Keras* library documentation [4]. In particular, we followed the selection criterion from DL bugs from prior works [29–31] while focusing on the APIs used for model compilation and training. Again, we collected a list of state-of-the-art research articles and their benchmarks of buggy and correct DL programs [29–31]. The selection criterion for these articles is that if the work in question solves DL performance bugs and renders the conditions that lead to these bugs. We filter out the obligations from

Table 1: Collected contracts targeting DNN structural and logical bugs, improper data, and well known training problems

Class of bugs		*DL Contract*
Improper data	Data normalization problem	Precondition: normalization_interval≤2, Postcondition: True
Structural and logic bugs	Incorrect activation and loss function: regression	Precondition: activation='linear|tanh', loss_func='mse', Postcondition: True
	Incorrect activation and loss function: binary classification	Precondition: activation='sigmoid', loss_func='binary_crossentropy', Postcondition: True
	Incorrect activation and loss function: multiclass classification	Precondition: activation='softmax', loss_func='categorical_crossentropy', Postcondition: True
	Incorrect activation and loss function: multilabel multiclass classification	Precondition: activation='sigmoid', loss_func='binary_crossentropy', Postcondition: True
	Incorrect activation in hidden layers	Precondition: activation !=linear, Postcondition: True
	Incorrect hyperparameter	Precondition: learn_rate >0.0000007,<0.01, Postcondition: True
Training problem	Overfitting	Precondition: True, Postcondition: diff_val_loss<0, diff_loss≤0
	High validation accuracy	Precondition: True, Postcondition: val_acc_threshold<0.95, diff_val_acc_train_acc<0.05
	High dropout rate	Precondition: dropout_rate>0.5
	Dying relu	Precondition: activation!='tanh|exponential|relu|sigmoid' Postcondition: zero_gradients_percentage ≤ λ
	Vanishing gradient	Precondition: activation!='tanh|exponential|relu|sigmoid' Postcondition: gradients_rate > β_1, norm_kernel>ζ
	Exploding gradient	Precondition: activation!='tanh|exponential|relu|sigmoid' Postcondition: gradients_rate_EG<β_2, gradient_value!=nan
	Oscillating loss	Precondition: True, Postcondition: accuracy_fluctuation_rate ≤ η, val_acc_diff >= δ
	Slow convergence	Precondition: True, Postcondition: acc_diff >= δ

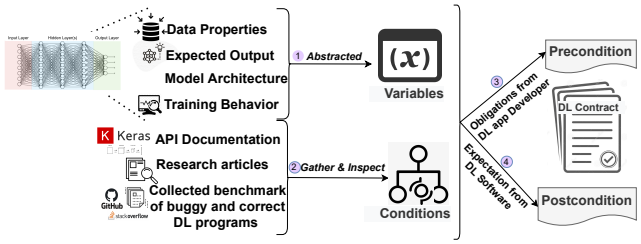

Figure 3: Methodology to collect deep learning contracts

DL app developer as *preconditions* (in 3) and expectation from DL software as *postconditions* (in 4). This process resulted in the collection of 15 contracts. A detailed table(Table 1) with collected contracts with corresponding bugs are shared in the supplementary material [11].

4.2 Implementation

To implement *DL Contract*, we extended the open-source package *PyContracts* [27]. *PyContracts* allows developers to declare constraints on method parameters and return values. We have extended *PyContracts* to support tensor, model types, as existing DL APIs require additional preconditions and postconditions [32]. We have addressed all the technical challenges described in §3.2.

4.3 Experimental Setup

To evaluate *DL Contract* on *Keras*, we modify the library by importing the extended *PyContracts* package in library codes. We also decorate respective *Keras* APIs with relevant implemented contracts that prevent performance bugs (in §4.1). We have conducted all the experiments on a machine with a 2 GHz Quad-Core Intel Core i7 and 32 GB 1867 MHz DDR3 RAM running the macOS 11.14.

Table 2: Effectiveness of *DL Contract* in real world programs targeting different class of bugs using collected benchmarks

DL Contracts targeting class of bugs		DeepLocalize		UMLAUT		AUTOTRAINER				NeuraLint		#Contract
		SO	GH	CIF-10	FMNIST	Blob	Circle	MNIST	CIF-10	SO	GH	Violation
Improper Data	Data normalization problem	5	2	1	1	-	-	-	-	1	1	11
Structural bugs	Incorrect activation & loss function	17	5	1	1	-	-	-	-	6	5	35
	Incorrect activation in hidden layers	3	1	1	1	-	-	-	-	-	-	6
	Incorrect learning rate	1	1	1	1	-	-	-	-	-	-	4
Training problem	Overfitting	1	-	-	-	-	-	-	-	-	-	1
	High validation accuracy	2	1	1	1	-	-	-	-	-	-	5
	High dropout rate	1	1	1	1	-	-	-	-	1	-	5
	Dying relu	1	-	-	-	4	9	23	36	-	-	73
	Vanishing gradient	-	-	-	-	16	36	34	35	-	-	121
	Exploding gradient	-	-	-	-	11	18	21	20	-	-	70
	Oscillating loss	1	-	-	-	1	3	1	-	-	-	6
	Slow convergence	5	2	-	-	28	41	19	42	-	-	137

* Numbers represented total contract violations in real world buggy programs from *DeepLocalize*, *UMLAUT*, *AUTOTRAINER*, *NeuraLint* benchmarks; SO, GH, CIF-10 indicates benchmark from *Stack Overflow*, GitHub, CIFAR-10 respectively, **"-"** indicates contracts are satisfied and did not trigger a violation in buggy programs.

Table 3: Applicability of *DL Contract* comparing against *Keras* Callbacks, Deeplocalize [51] and *DL Contract* (full table [14])

DeepLocalize Benchmark		Buggy Code													Correct Code		
		Original	TOnNaN		ES('loss')		ES('accuracy')		Union (TOnNaN, ES)		DeepLocalize		DL Contract		Original	DL Contract	RT
Source	#	RT	RT	Bug#	RT	Bug#	RT	Bug#	RT	Bug #	RT	Bug#	RT	Bug#	RT	RT	Overhead
StackOverflow	30	42.20	30.15	2	24.92	19	24.76	23	18.53	27	447.05	27	6.41	29	34.60	37.93	0.22
GitHub	11	352.90	439.88	0	299.18	6	269.70	7	160.77	7	2772.77	7	9.23	10	345.04	27.37	0.23

* Total detected bugs in buggy and correct codes (#), Keras debugging TerminateOnNan (TOnNan), EarlyStopping(monitor='loss') (ES(loss), EarlyStopping(monitor='accuracy' (ES(accuracy))

Table 4: Applicability of *DL Contract*, Runtime comparison between *UMLAUT* callback [46] and *DL Contract*

Benchmark	Buggy Code					Correct Code				
	Original Runtime	UMLAUT		DL Contract		Original Runtime	UMLAUT Runtime	DL Contract Runtime	UMLAUT Runtime Overhead	DL Contract Runtime Overhead
		Bug	Runtime	Bug	Runtime					
A1 (CIFAR-10)	1318.99	Y	8.85	Y	28.52	1376.04	1455.99	1353.81	1.06	0.02
A2 (CIFAR-10)	1483.26	Y	8.93	Y	24.80	1459.21	1384.75	1478.52	0.95	0.01
A3 (CIFAR-10)	1455.93	Y	140.98	Y	23.56	1483.29	1251.69	1497.94	0.84	0.01
B1 CIFAR-10	1493.12	Y	152.57	Y	16.28	1420.40	1049.76	1438.59	0.74	0.01
B2 (CIFAR-10)	1319.18	Y	8.85	Y	26.58	1448.27	792.97	1440.55	0.55	0.01
B3 (CIFAR-10)	1692.83	Y	664.60	Y	669.83	1463.87	795.15	1499.39	0.54	0.02
A1 (F-MNIST)	17.09	Y	7.02	Y	23.25	16.84	15.55	22.76	0.92	0.35
A2 (F-MNIST)	17.04	Y	9.61	Y	18.62	16.36	15.59	24.57	0.95	0.50
A3 (F-MNIST)	15.69	Y	9.61	Y	18.30	16.36	14.34	23.52	0.88	0.44
B1 (F-MNIST)	17.90	Y	9.87	Y	21.06	15.93	14.96	22.92	0.94	0.44
B2 (F-MNIST)	15.96	Y	7.16	Y	18.94	16.96	14.48	23.93	0.85	0.41
B3 (F-MNIST)	17.62	Y	12.31	Y	31.85	15.39	14.85	24.06	0.96	0.56

Benchmark selection: To answer the RQs, we compare and contrast *DL Contract* against four recently-published DL performance bug localization benchmarks [43, 46, 51, 54]. The *DeepLocalize*'s benchmark proposed by Wardat *et al.* [51] consists of 41 executable *Keras* codes with buggy and correct versions of DL programs from *Stack Overflow* (30) and GitHub (11). For the *UMLAUT* benchmark [46], we followed their procedure. AUTOTRAINER [54] reported their tool's results on 495 DL programs where 262 have training problems Here, we have utilized 4 out of 6 datasets which are comprised of sequential models. NeuraLint [43] utilized a total of 63 buggy programs of crash and performance bugs. We have used 16 buggy programs from the benchmark which does not yield crash bugs. We have considered all 4 of these benchmarks as "unseen" because we have not seen their buggy and correct programs before writing and implementing contracts.

Metrics: We recorded the total execution time utilization for all techniques when analyzing buggy and correct programs from the benchmarks and computed overhead. Also, we recorded how many bugs were detected by each approach. For computing the efficiency of *DL Contract*, we utilize performance metrics as precision, recall following prior work [39, 54]. We consider the benchmarks as ground truth for buggy and correct programs. Here, a false positive indicates that a bug was detected in the correct program. True positive represents if a bug is detected in a buggy program. A false negative indicates that there is no bug detected in a buggy program. Lastly, if there is no bug detected in a correct program, we consider that as a true negative.

We collected the real-world time elapsed between the program entry and program exit using the python time module. We collected this information for both correct and buggy programs five times to reduce randomness, following [50, 54]. To isolate the other process and void interference in this experiment, we executed only one program under analysis in a standalone environment inside the IDE. We start recording the time from the beginning of a DL program until the first contract violation has been thrown, and the rest of the execution is halted in the buggy program. For the correct program and if there is no contract violation, we obtained the elapsed time until the complete execution of the program.

4.4 Results and Analysis

4.4.1 RQ1 (Effectiveness). To demonstrate the effectiveness of *DL Contract* in real-world programs, we have utilized 4 benchmarks of DL performance bugs. Table 2 shows the results of *DL Contract* targeting different class of bugs. We have developed total 15 contracts and annotated on model compilation and training *Keras* APIs using *DL Contract* approach targeting different classes of bugs related to improper data, structural bugs and training problems. In particular, each row represents the number of contract violations in buggy programs where *DL Contract* successfully detected bugs and terminated the program execution. We observe that in the last 'Contract Violation' column, those 15 contracts triggers total 474 contract violation messages in 272 buggy programs. In Table 2, "-" indicates contracts were used but did not trigger a violation for that class of bugs. For example, *AUTOTRAINER* mainly focuses on training problems, which is why there is no contract violation involving structural and improper data-related bugs. Those contracts (postcondition) violations have been triggered by *DL Contract* using abstracted training properties. *DeepLocalize*, *UMLAUT*, *NeuraLint* benchmarks consist of structural and data bugs, that precondition violation triggers using *ML variable* related to model abstraction. *DL Contract* did not detect bugs in 13 out of 272 programs. We have investigated these undetected bugs and discussed in §4.4.2. We also evaluated that same 15 contracts were used in 257 correct programs from those benchmark and found 18 contract violations as false positives, mainly due to randomness factor [44, 56] during training. *In summary, DL Contract is efficient in real-world DL programs.*

4.4.2 RQ2 (Applicability). Table 3, 4, 5, and 6 show the applicability of *DL Contract* on real-world benchmarks comprising of performance bugs in DL software. Each table highlights and summarizes the results of Buggy and Correct programs.

Table 3 shows the summary of the results [14] of deploying the *DeepLocalize* benchmark. Please refer to supplementary material [12] for more details. Table 3 shows that *DL Contract* can detect **39 out of 41 buggy programs** with precise contract violation message. Out of these results, 29 are from *Stack Overflow* and 9 out from GitHub. Also, when compared with *Keras* and *DeepLocalize* callbacks. *Keras* debugging techniques *TerminateOnNan*, *EarlyStopping(monitor='loss')*, *EarlyStopping(monitor='accuracy')* and *DeepLocalize* can detect 2, 24, 28, 32, and 34 respectively [51]. Again, 2 out of 41 were not detected from DeepLocalize [51] benchmark. SO52800582 and GH[2] were missed because generalized contracts cannot be applied on weight initialization and optimizer. Finally, regarding bug detection speed, *DL Contract* is 200 times faster than *DeepLocalize* and 11 times faster than *Keras* callbacks.

Table 4 shows that *DL Contract* applies to all **12 buggy programs** from the *UMLAUT* benchmark. In terms of computation overhead, we observed *DL Contract* has lower runtime than *UMLAUT* (in §4.4.4). Lastly, we have manually verified the contract breaches reported by *DL Contract* and found no false alarm for buggy programs.

Table 5 shows that *DL Contract* have detected **195 bugs in 203 buggy programs** in the AUTOTRAINER benchmark. While *AUTOTRAINER* reports the symptoms of 5 training problems, *DL Contract* detects bug as postcondition violations. We observed that both approaches detect the Slow Change in accuracy (SC) more often than the other four symptoms. 8 out of 203 buggy programs in AUTOTRAINER [54] benchmark were not detected due to the randomness in DNN training. In terms of runtime, *DL Contract* is slightly faster than *AUTOTRAINER*. In particular, *DL Contract* takes on average 241.19 seconds, while *AUTOTRAINER* 248.43 seconds. Lastly, **out of 188 correct programs, *DL Contract* misdetected 3 programs**. Further investigation revealed that those misdetections were due to data normalization issues, unsupported by *AUTOTRAINER*.

Table 6 shows how *DL Contract* performs on **16 bugs** compared to the *NeuraLint* tool. *DL Contract* detected **13 out of 16 bugs** in the *NeuraLint* benchmark. 3 out of 16 from the *NeuraLint* benchmark [43] were missed because we investigated that we had no layer properties related contracts written. *NeuraLint* detects 14 out of 16 bugs but *DL Contract* requires less time than *NeuraLint*. In particular *DL Contract* on average required 5.10 seconds while NeuraLint 9.80 seconds. These buggy programs use common API methods such as compile and fit, which were annotated with 15 DL Contracts. These 272 buggy programs have common root causes and symptoms. For instance, AUTOTRAINER [54] benchmark consists of 203 buggy programs, with 5 different training problems. By writing 5 contracts on the fit method targeting those problems, DL Contract detects 195 out of 203 bugs.

In summary, DL Contract is applicable to detect performance bugs in real world buggy programs with good accuracy.

4.4.3 RQ3 (Efficiency). We have measured the efficiency of *DL Contract* using 4 benchmarks DeepLocalize [51], UMLAUT [46], AUTOTRAINER [54], NeuraLint [43] (in Table 7). We have evaluated 257 correct (clean) real-world programs and found 18 false positives. We have found 10 FPs in *DeepLocalize*, 0 in *UMLAUT*, 3 in *AUTOTRAINER* and 5 in *NeuraLint* benchmark. In terms of efficiency, our evaluation results show that *DL Contract* has similar accuracy to *UMLAUT* (12 TPs and no FPs) but has lower time consumption (in Fig. 4). Regarding the *AUTOTRAINER* benchmark, *DL Contract* could not detect bugs due to the accuracy threshold [54] (0.6) due to randomness factor during training. Regarding the *NeuraLint* benchmark, we observed 3 FN. As *DL Contract* does not have contracts on layer properties yet. Compared to other tools using *DeepLocalize* benchmark, we found *DeepLocalize*, *AUTOTRAINER*, *UMLAUT*, *NeuraLint*, *DeepDiagnosis* [50] resulted in 19, 14, 14, 35 TP and 22, 27, 23, 6 FN respectively [13]. *DeepDiagnosis* reported 70 FP and 67 FN in correct codes from *AUTOTRAINER* benchmark. *In summary, DL Contract efficiently detects performance bugs in real-world buggy programs.*

Superiority of DL Contract: Prior work specifically DeepLocalize [51], UMLAUT [46], AUTOTRAINER [54], *DeepDiagnosis* [50] and NeuraLint [43] are not comprehensive enough to detect different classes of structural and training bugs. Furthermore, these approaches depend on specific implementations such as model format (.h5), semantic change in model architecture, and rely upon additional debugging or verification facilities, e.g., Keras callbacks (*DeepLocalize*, *UMLAUT*, *AUTOTRAINER*), and Groove model checker (*NeuraLint*). Also, *DeepLocalize*, *UMLAUT*, *NeuraLint* did not compute FP and FN. *AUTOTRAINER* computed FP, FN only with *AUTOTRAINER* benchmark. All 4 baseline techniques did not compare against any other benchmarks except their own benchmarks. *DeepLocalize* [55] invokes callbacks after each epoch and computes

Table 5: Applicability of *DL Contract*, Runtime (RT) comparison with AUTOTRAINER [54] (AT) and *DL Contract* (DLC)

Benchmark	Buggy													Correct							
		AT						DLC												Overhead	Overhead
		Detected Symptoms #						Postcondition Violation #							Original		AT		DLC	AT	DLC
Dataset	#	VG	EG	DR	SC	OL	RT	VG	EG	DR	SC	OL	RT	#	RT	Sym #	RT	Viol #	RT	Runtime	Runtime
Blob	48	12	10	8	29	4	13.83	6	5	4	30	1	11.95	39	8.62	0	9.22	0	11.79	0.070	0.368
Circle	71	10	10	9	43	7	16.50	10	8	9	41	3	11.54	36	16.06	0	12.28	2	16.27	0.235	0.013
CIFAR-10	46	5	8	3	28	2	1302.06	5	16	37	43	0	487.43	35	1186.50	0	1898.54	0	1528.16	0.600	0.288
MNIST	38	8	3	4	21	8	688.02	8	13	23	19	1	423.55	78	466.22	0	535.90	2	535.51	0.149	0.149
Total/ Avg	203	35	31	24	121	21	505.10	29	42	73	133	5	233.62	188	419.35	0	613.99	4	522.93	0.464	0.247

* Count (#), Vanishing Gradient (VG), Explode Gradient (EG), Dying Relu (DR), Slow Change in Accuracy (SC), Oscillating Loss (OL), Symptom (Sym), Contract Violation(Viol)

Table 6: Applicability of *DL Contract*, Runtime overhead comparison with NeuraLint [43] and *DL Contract*

Benchmark	Buggy Code					Correct Code				
	Original	NeuraLint		DL Contract		Original	NeuraLint	DL Contract	Runtime Overhead	Runtime Overhead
	Runtime	Bug	Runtime	Bug	Runtime	Runtime	Runtime	Runtime	NeuraLint	DL Contract
50555434	2.73	Y	18.72	Y	4.77	2.66	6.29	4.78	1.37	0.80
34311586	3.17	Y	9.62	Y	4.81	3.09	3.18	5.12	0.03	0.66
50079585_1	2.88	Y	18.02	Y	5.11	2.75	6.52	4.92	1.37	0.79
51749207	2.80	Y	17.89	Y	4.87	2.68	5.89	4.84	1.20	0.81
58844149	3.03	Y	8.46	Y	5.06	2.89	6.28	5.01	1.17	0.73
33969059	5.20	Y	5.20	Y	6.53	2.62	2.64	4.63	0.01	0.77
44322611	2.93	Y	8.80	N	4.94	2.62	2.69	4.62	0.02	0.76
55776436	3.11	Y	10.78	Y	5.27	3.04	3.06	5.15	0.01	0.69
60566498	2.87	Y	16.57	Y	4.74	2.74	7.38	4.86	1.69	0.77
GH 1	2.90	Y	5.13	Y	5.04	2.96	7.71	4.97	1.60	0.68
GH 2	2.83	Y	5.52	N	4.91	2.80	6.94	4.86	1.48	0.74
GH 3	3.05	Y	5.51	Y	4.96	3.00	6.70	4.89	1.23	0.63
GH 4	4.43	Y	8.50	N	6.44	4.04	137.13	5.99	32.94	0.48
GH 5	2.78	Y	7.42	Y	4.88	2.78	6.38	4.82	1.30	0.74
GH 6	2.74	N	5.20	Y	4.69	2.68	6.14	4.66	1.29	0.74
GH 7	2.72	N	5.39	Y	4.59	2.71	5.97	4.61	1.20	0.70
Total/Average	**3.14**	**14**	**9.80**	**13**	**5.10**	**2.88**	**13.81**	**4.92**	**2.99**	**0.72**

Table 7: *DL Contract* efficiency on different buggy and correct benchmarks

Benchmark	DL Contract					
	FP	TP	FN	TN	Precision	Recall
DeepLocalize	10	39	2	31	0.80	0.95
UMLAUT	0	12	0	12	1.00	1.00
AUTOTRAINER	3	195	8	185	0.98	0.96
NeuraLint	5	13	3	11	0.72	0.81

metrics to detect numeric bugs which take lots of time. *AUTOTRAINER* [58] requires the model in a specific format and needs to finish the training to detect bugs and then provide solutions as fixes. In the case of UMLAUT [50], without semantic change of model, the tool will report a false alarm. *NeuraLint* [48] requires graph computation from the model and performs static checking with some specified rules which yields longer runtime.

4.4.4 RQ4 (Overhead). We have computed the runtime overhead of *DL Contract* using *UMLAUT*, *DeepLocalize*, *NeuraLint*, and *AUTOTRAINER* benchmark. Figure 4 shows the runtime overhead of *DL Contract*. *DL Contract* (DLC) runtime overhead is lower than the one of all approaches. In particular, *DL Contract* is 4.31, 3.69, 1.85, and 4.15 times more efficient in terms of runtime overhead than *DeepLocalize*, *UMLAUT*, *AUTOTRAINER*, and *NeuraLint*. The runtime overhead of DL Contract is minimal because the technique only checks model structure-related preconditions before model compilation API and training-related postconditions before training. Unlike techniques *DeepLocalize*, *UMLAUT*, *AUTOTRAINER*, that rely on *Keras* callbacks, *DL Contract* does not invoke model

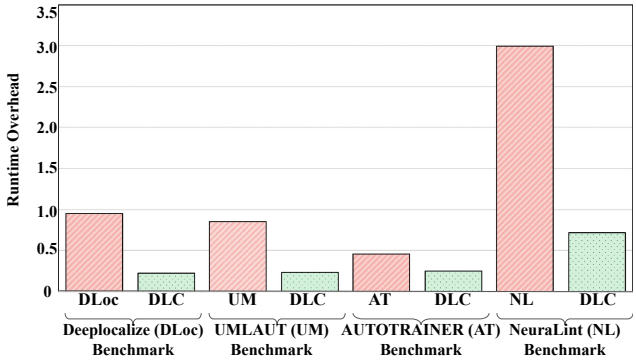

Figure 4: Comparison of runtime overhead

compilation or training APIs multiple times to monitor metrics periodically during or after training. Specifically, *NeuraLint* requires graph computation from model and performs static checking with some specified rules, yielding longer runtime. Also, We measured that the runtime overhead increases by around 15% compared to the baseline *Keras*. *In summary, DL Contract incurs less runtime overhead compared to existing deep learning debugging tools.*

4.4.5 RQ5 (Usability). We have evaluated the usability of *DL Contract* annotated *Keras* in terms of its usefulness to find and fix bugs while developing DL programs. Also, we evaluate separately the efforts of API designers to write and integrate *DL Contract*. To that end, we perform a user study following IRB guidelines and collected feedback on using *DL Contract* annotated *Keras*.

RQ5.1 (Usefulness): How useful is the *DL Contract* enabled Keras in developing DL Apps?

RQ5.2 (Easiness) : How easy is to write *DL Contract* and integrate it with DL library APIs?

Participants: After following similar procedure [19, 46] from prior work we recruited 20 participants from university mailing lists for our study (17 Ph.D., 2 MS students, and a Post-doc). Participants were asked to self classify their level of expertise from 1 - beginner to 5 - expert and we obtained their expertise level: programming ($\mu = 3.3, \sigma = 1.0$), debugging ($\mu = 2.9, \sigma = 1.4$), using existing neural networks ($\mu = 2.9, \sigma = 1.1$), and developing new DNNs ($\mu = 2.5, \sigma = 1.2$), and developing other ML algorithms ($\mu = 2.7, \sigma = 1.3$). So, the average/mean (μ) of the expertise levels is more than 2.5 in all of the 20 selected participants.

Study Design, Procedure and Tasks Participants completed an hour-long online study on their machines. Each participant completed two sessions with corresponding tasks. After each session, participants completed survey questions online via Qualtrics. For RQ5, in session 1, we provided the necessary environment to execute buggy programs in regular Keras (baseline condition) and *DL Contract* enabled Keras. We provided 3 buggy versions of randomly chosen real-world programs with 3 different performance bugs related to model architecture, data properties, and training behavior. The buggy programs have low accuracy and high training time issues. We asked the participants to execute the buggy programs using both regular *Keras* and *DL Contract* enabled *Keras*. Then, we asked participants to detect and fix the buggy programs by using the outputs from both regular *Keras* and DL Contract-enabled *Keras*. Finally, we asked participants the survey questions regarding their experience using *DL Contract*.

For RQ5, in session 2, we first provide tutorial to participants on how to write contracts on Keras API. Then, we asked them to write 3 similar contracts with instructions. After completing the sessions, participants filled up a survey indicating their experience while using *DL Contract* enabled *Keras* to detect and fix bugs as a DL application developer. In that survey, participants also shared their experience about the writing process of *DL Contract* as a library developer. The details of the survey questions for session 1 and session 2 are provided in the supplementary material [12].

Results and Discussion: *RQ5.1 (Usefulness):* For all 3 buggy programs in session 1, none of the participants was able to find any of the bugs in the baseline condition (regular *Keras*). That is because *Keras* does not inform users about such types of performance bugs. However, participants were able to detect and fix the bugs by following *DL Contract* enabled *Keras*'s contract violation messages. Furthermore, survey responses indicate that *DL Contract* enabled Keras helps participants to detect and fix bugs efficiently. In particular, on a 5-point Likert scale questions (1 = Not helpful to 5 = Very Helpful), participants rated their experience on questions. Participants indicated that, *DL Contract* enabled Keras was very helpful to 65% ($\mu = 4.55, \sigma = 0.67$) in detecting bugs in deep learning programs that yield unexpected performance (low accuracy, high training time). 25% rated helpful (rating 4), and 10% of participants rated reasonably helpful (rating 3). Therefore, 90% of the participants responded positively (rating > 3) regarding this criteria. Likewise, 95% of participants rated positively (rating > 3) about the message from DL Contract fixing those bugs ($\mu = 4.75, \sigma = 0.54$).

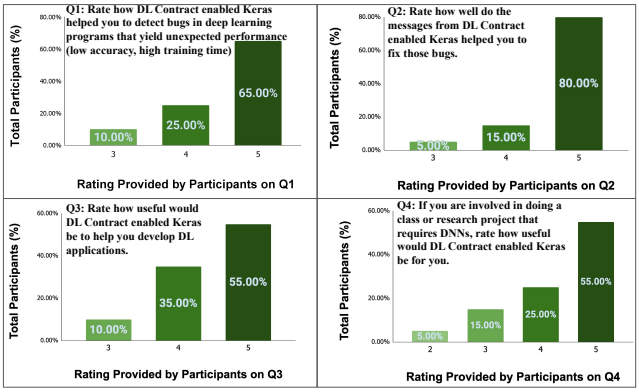

Figure 5: Survey results with participants ratings on how useful is *DL Contract* enabled *Keras* in developing DL Apps

Again, 90% of the participants rated positively (rating > 3) specifically, 55% of the participants indicates that it would be very useful to develop DL applications ($\mu = 4.45, \sigma = 0.67$). If participants are involved in doing a class or research project that requires DNNs, 80% rated positively especially, 55% of the participants rated *DL Contract* enabled Keras as very helpful ($\mu = 4.30, \sigma = 0.90$).

RQ5.2 (Easiness): Regarding how easy is to write *DL Contract* on top of *Keras* APIs, we have obtained that 65% of the participants rates the writing process of a contract to *Keras* positively (Rating > 3). Regarding the rating of the writing process of a contract to Keras, the participants' rating ($\mu = 3.8, \sigma = 0.67$) is moderate (35%), easy (50%), very easy(15%) as illustrated in Fig. 6. About the integration of the written contract with Keras library, 60% of the participants rated positively ($\mu = 3.75, \sigma = 0.69$). The detailed breakdown rating of integration of the written contract with Keras library, the participants' ratings is moderate 40%), easy (45%), very easy(15%) as shown in Fig. 6. *In summary, we have evaluated that DL Contract enabled Keras is very helpful to developers in debugging DL software and easy to writing and integrating DL Contract is very easy to API designers.*

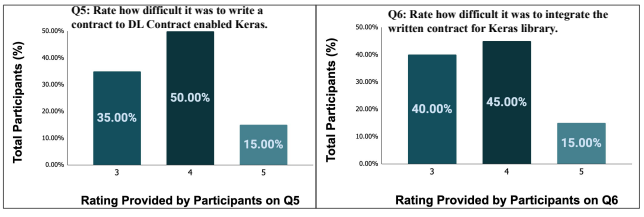

Figure 6: Survey results with participants ratings on how easy is to write *DL Contract* on DL library APIs

4.5 Limitations

Our proposed *DL Contract* approach has been tested primarily on problems related to multilabel, multiclass, binary classification, and regression with various structural and logical bugs in the sequential DNN model architecture and common training issues. Further research is needed to apply and evaluate our approach on other types of bugs and model categories. Despite this, the concept of using contracts in deep learning is not limited to *Keras* and can

be extended to other DL libraries. While our paper illustrates the idea of deep learning contracts for Keras, our contribution can be generalized to other DL libraries. For example, for TensorFlow, 7 out of 18 the ML variables (gradient_value, gradients_rate, zero_-gradients_percentage, etc.) are directly applicable. The rest of the ML variables representing model architecture, and training metrics need minimal implementation. However, all other major contributions of DL Contract runtime assertion, contextualized inter-API Call contracts, post-training contracts can be applied to other libraries. We decided to focus on Keras to keep implementation effort manageable and to leverage the large body of benchmarks for this library.

4.6 Threats to Validity

Our proposed approach may be affected by external threats such as imprecise precondition and postcondition definitions obtained from library documentation, *Stack Overflow* posts, and GitHub commits. However, we have adopted definitions from recent research studies [31, 39, 55] to mitigate this. Threshold parameters may also cause false positives in some new real-world programs. Additionally, implementation using *PyContracts* may have unforeseen internal threats, but our general open-source framework can be extended using reproducible package [12] with detailed results.

5 RELATED WORK

Specification of Deep Neural Networks: The closest related ideas in specification of DNNs include [26, 47, 48]. While [47] provides an overview of the opportunities and challenges of formalizing and reasoning about DNN properties, it does not propose any methodology for writing and checking specifications for deep learning libraries. In contrast, [26] presents a technique for computing input and layer properties from a feed-forward network using input-output characterizations as formal contracts. Additionally, [48] introduces a method for repairing neural network classifiers by inferring correct specifications. Both [26] and [48] propose inference techniques, while our technique proposes a specification and checking technique that enables the specification of DL libraries and checks those contracts in client code using those libraries, thus preventing bugs and providing fix suggestions.

Deep Learning Testing, Debugging, and Repairing: Prior work on DL testing, debugging, and repairing includes DeepLocalize [51], MODE [39], *AUTOTRAINER* [54], *DeepDiagnosis*[50], *DeepFD*[20], Ariadne [24], Lagouvardos [33], Nikanjam *et al.* [43], SHAPETRACER [37], and Tensfa [52]. These approaches focus on detecting and localizing bugs, but *DL Contract* supports documentation of expected behavior. While *DL Contract* checker can also double as a bug detection tool, in the longterm developers would also benefit from the documentation and write more correct DL programs. Empirical studies [21, 29–31, 45, 53, 55] have motivated the need for DL bug repair, but none propose a DbC methodology like *DL Contract*.

Existing DbC Methodology: Existing DbC frameworks for Python, such as PyContracts [27], Pylint [1], and PyTA [38], do not have the capability to check contracts for properties of models and data, or monitor training behavior of DL models. These frameworks do not address the technical challenges of checking contracts beyond API parameters, contracts involving multiple APIs at different stages of the ML pipeline, and contracts on intermediate properties to specify desired training behavior. Additionally, *DL Contract*'s use of runtime assertions is distinct from checking runtime properties, such as interpreting statecharts [40]. To the best of our knowledge, the concept of applying DbC over the DL computational graph and specifying DL-specific contracts is novel.

API Misuse Detection: There have been some API misuse detection techniques such as, [49] which examines the usage of machine learning (ML) cloud APIs in open-source applications and finds that a significant portion of these applications contain API misuses that degrade their functional, performance, or economical quality, leading to the development of automated checkers for identifying such misuses. [**?**] tackles API Misuse (APIM) bugs staitically by some rules that occur when practitioners misunderstand the underlying assumptions of deep learning APIs, leading to inconsistencies between the designed DL program and the API's usage conditions, potentially resulting in reduced effectiveness or runtime exceptions. Existing API misuse detection methods lack the capability to check contracts that capture properties of models, data, and training behavior at various program points. To address this limitation, the authors propose abstractions of models, data, and training behavior using machine learning (ML) variables, enabling library developers to write contracts based on them for APIs related to model architecture and DL model training. This approach overcomes technical challenges associated with checking contracts beyond formal API parameters, handling contracts involving multiple APIs at different stages of the ML pipeline, and specifying intermediate properties for desired training behavior.

6 CONCLUSIONS AND FUTURE WORK

In this work, we proposed a novel method for checking contracts for deep learning libraries by specifying DL APIs with preconditions and postconditions. Our approach is extensible and generalizable, allowing for the abstraction of model architecture, data properties, and training behavior. We developed 15 sample DL contracts targeting common bugs and found they effectively prevented structural bugs and training problems. Additionally, our user study showed the usability of *DL Contract* when applied to the Keras library. We have submitted an API design proposal for its incorporation in future releases of Keras. Possible future work includes static validation, unit testing, and inferring contracts for additional libraries.

7 DATA AVAILABILITY

The replication packages and results are available in this repository [12] that can be leveraged by further research.

REFERENCES

[1] 2016. *Pylint.* https://pylint.pycqa.org/en/latest/
[2] 2021. CNN with keras, accuracy not improving. https://stackoverflow.com/questions/50079585/. [Online; accessed Feb-2023].
[3] 2021. How to train and tune an artificial multilayer perceptron neural network using Keras? https://stackoverflow.com/questions/34673164/. [Online; accessed Feb-2023].
[4] 2021. Keras API reference. https://keras.io/api/. [Online; accessed Feb-2023].
[5] 2021. Keras unreasonnably slower than TensorFlow. https://stackoverflow.com/questions/47352366/. [Online; accessed Feb-2023].

[6] 2021. Loss becomes NaN. https://stackoverflow.com/questions/55328966/. [Online; accessed Feb-2023].

[7] 2021. Low accuracy after training a CNN. https://stackoverflow.com/questions/59325381/. [Online; accessed Feb-2023].

[8] 2021. Multilabel Text Classification using TensorFlow. https://stackoverflow.com/questions/35400065/multilabel-text-classification-using-tensorflow. [Online; accessed Feb-2023].

[9] 2021. Sigmoid layer in Keras. https://stackoverflow.com/questions/45442843/. [Online; accessed Feb-2023].

[10] 2021. Simple MNIST convnet example from Keras documentation. https://keras.io/examples/vision/mnist$_c$onvnet/. [Online; accessed Feb-2023].

[11] 2022. Collected Contracts Table in Anonymous Repo. https://github.com/researcher97/ESEC$_F$SE23DLC/tree/main/Additional%20Table. [Online; accessed Feb-2023].

[12] 2023. Anonymous Reposioty of DL Contract. https://github.com/researcher97/ESEC$_F$SE23DLC. [Online; accessed Feb-2023].

[13] 2023. Comparison of DeepDiagnosis with DL Contract in Anonymous Repo. https://github.com/researcher97/ESEC$_F$SE23DLC/tree/main/Results. [Online; accessed Feb-2023].

[14] 2023. DeepLocalize Full Results in Anonymous Reposioty. https://github.com/researcher97/ESEC$_F$SE23DLC/blob/main/Results/DeepLocalize$_R$esults.pdf. [Online; accessed Feb-2023].

[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[16] Mike Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K Rustan M Leino, Wolfram Schulte, and Herman Venter. 2005. The Spec# programming system: Challenges and directions. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 144–152.

[17] Mike Barnett, Manuel Fähndrich, K Rustan M Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and verification: the Spec# experience. *Commun. ACM* 54, 6 (2011), 81–91.

[18] Martin Büchi and Wolfgang Weck. 1999. *The Greybox Approach: When Blackbox Specifications Hide Too Much.* Technical Report 297. Turku Center for Computer Science. http://tinyurl.com/ywmuzy.

[19] Margaret Burnett, Robin Counts, Ronette Lawrence, and Hannah Hanson. 2017. Gender HCl and microsoft: Highlights from a longitudinal study. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 139–143.

[20] Jialun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung. 2022. DeepFD: Automated Fault Diagnosis and Localization for Deep Learning Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 573–585. https://doi.org/10.1145/3510003.3510099

[21] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 750–762.

[22] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. 2005. Model variables: Cleanly supporting abstraction in design by contract. *Software: Practice and Experience* 35, 6 (2005), 583–599.

[23] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *International conference on software engineering and formal methods*. Springer, 233–247.

[24] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for Machine Learning Programs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3211346.3211349

[25] Konstantin Eckle and Johannes Schmidt-Hieber. 2019. A comparison of deep networks with ReLU activation function and linear spline-type methods. *Neural Networks* 110 (2019), 232–242.

[26] Divya Gopinath, Hayes Converse, Corina Pasareanu, and Ankur Taly. 2019. Property Inference for Deep Neural Networks. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 797–809. https://doi.org/10.1109/ASE.2019.00079

[27] Brett Graham, William Furr, Karol Kuczmarski, Bernhard Biskup, and Adam Palay. 2010. *PyContracts*. https://andreacensi.github.io/contracts//

[28] 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[29] 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.

[30] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *ESEC/FSE'19: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2019)*.

[31] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: Fix patterns and challenges. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1135–1146.

[32] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[33] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[34] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes* 31, 3 (May 2006), 1–38. https://doi.org/10.1145/1127878.1127884

[35] K. Rustan M. Leino. 1995. *Toward Reliable Modular Programs.* Ph. D. Dissertation. California Institute of Technology. Available as Technical Report Caltech-CS-TR-95-03..

[36] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

[37] Chen Liu, JieLu, Guangwei Li, Ting Yuan, Lian Li, Feng Tan, Jun Yang, Liang You, and Jingling Xue. 2021. Detecting TensorFlow Program Bugs in Real-World Industrial Environment. In *ASE 2021: The 36th IEEE/ACM International Conference on Automated Software Engineering*.

[38] Nigel Fong Lorena Buciu, Simon Chen and et al. 2016. *PyTA*. https://www.cs.toronto.edu/~david/pyta/index.html/

[39] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 175–186.

[40] Tom Mens, Alexandre Decan, and Nikolaos I Spanoudakis. 2019. A method for testing and validating executable statechart models. *Software & Systems Modeling* 18, 2 (2019), 837–863.

[41] B. Meyer. 1992. Applying 'design by contract'. *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[42] Peter Müller. 2001. *Modular Specification and Verification of Object-Oriented programs.* Ph. D. Dissertation. FernUniversität Hagen, Germany. http://tinyurl.com/jtwot

[43] Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. 2021. Automatic Fault Detection for Deep Learning Programs Using Graph Transformations. *ACM Trans. Softw. Eng. Methodol.* 30, 5 (2021), 26 pages.

[44] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 771–783.

[45] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering* 25, 6 (2020), 5193–5254.

[46] Eldon Schoop, Forrest Huang, and Björn Hartmann. May 8–13, 2021. UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*.

[47] Sanjit A Seshia, Ankush Desai, Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte, and Xiangyu Yue. 2018. Formal specification for deep neural networks. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 20–34.

[48] Muhammad Usman, Divya Gopinath, Youcheng Sun, Yannic Noller, and Corina Pasareanu. 2021. NNrepair: Constraint-based Repair of Neural Network Classifiers. *arXiv preprint arXiv:2103.12535* (2021).

[49] Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2021. Are machine learning cloud apis used correctly?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 125–137.

[50] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs. In *ICSE'22: The 44rd International Conference on Software Engineering*.

[51] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault Localization for Deep Neural Networks. In *ICSE'21: The 43rd International Conference on Software Engineering*.

[52] Dangwei Wu, Beijun Shen, Yuting Chen, He Jiang, and Lei Qiao. 2021. Tensfa: Detecting and Repairing Tensor Shape Faults in Deep Learning Systems. In *ISSRE 2021: The 32nd International Symposium on Software*.

[53] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1159–1170.

[54] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *ICSE'21: The 43rd International Conference on Software Engineering*.

[55] 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.

[56] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. 2022. Randomness in neural network training: Characterizing the impact of tooling. *Proceedings of Machine Learning and Systems* 4 (2022), 316–336.