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  
Iowa State University  
Ames, IA, 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-verified 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-verified 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.

CCS CONCEPTS


KEYWORDS

Deep learning, API contracts, specification language

ACM Reference Format:


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 [33, 36, 37, 66]. To address this, SE researchers have focused on detecting and localizing these bugs [46, 57, 62]. In this work, we explore an alternative approach to improve the reliability of deep learning software, design by contract (DbC). Traditional DbC [19, 41, 43, 48] 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 [23] that expose part of the program, ML variable provides a higher-level abstraction of the training process and model structure. They are similar to specification-only fields [27] in object-oriented programs [42, 49], 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
Figure 1: Buggy code [2, 6, 7, 9] achieves 9.78% training accuracy. Similar correct code [10] achieves ~99% training accuracy.

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 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. This correct configuration is crucial for achieving high accuracy.

Our improvements 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 [15] that is extensible and generalized to different classes of DL bugs and maps contract violation as a bug, symptoms as the constraint to check, and contract violation messages as suggestions to fix 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) [11]. Developers can use this @Keras environment for debugging without any annotation overhead and minimal runtime overhead (~15%).
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 [18, 28] 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 instance, in Example 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.

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 [46], 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 differences in validation loss and decreasing differences in training loss [57], which is a common cause of overfitting. This expectation is encoded as a postcondition.

```
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.

```
```
```
```
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 [16]. 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 problems, 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.

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 sub-clause. 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 take 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 [30],

---

**Algorithm 1 DL Contract Checker**

```
1: procedure contract_checker (userRef, annotateRef)
2:    [Args ← formal_arguments(userRef)]
3:    argContrDict ← parse_contract(annotateRef)
4:    for each ([Arg, cond]) in argContrDict do
5:        aArg ← actual_arguments(Arg, userRef)
6:        template ← parse_template(cond)
7:        template.check_contract(aArg)
8:        returnCondition ← parse_contract(annotateRef)
9:        aArgs ← actual_arguments(annotateRef)
10:       result ← userRef(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:    newContractRegister[identifier] ← funcRef
17: procedure parse_template (cond) > multiple conditions
18:    if len (cond) > 1 then
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 e newContractRegister then > if it is callable
26:        return CheckCallable (newContractRegister[cond])
```

---
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 Anti(CallCheckContract)
12.     procedure init (subclauses)
13.         subclauses ← subclauses
14.     procedure check_contract (value)
15.         for sc in 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.     procedure check_contract (value)
22.         actual_type ← GetActualType (value)
23.         if actual_type # expected_type then
24.             raise ContractException()
25. end class
26. end class

if ReLu is used on the last layer, it will set all the negative output to zero, thus leading to an accuracy problem. To prevent such kinds of problems in model architecture, library developers can write DL 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-configuration API, i.e., Keras Compile, exposing objects capturing the entire model properties.

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 are computed on line 3 and line 5 from model object. The loss function has been a formal parameter of Compile API, and contract_checker2 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. As those specified contacts are ANDed one after another for one contract (last layer activation and loss function), 'contract_checkerfunc2' is only executed if 'contract_checkerfunc1' is executed. Since 'contract_checkerfunc1' checks whether the number of classes ≥ 3, then 'contract_checkerfunc2' would also know if the program runs a multiclass classification. In Example 3.3, on lines 17–18, 'contract_checkerfunc1' and 'contract_checkerfunc2' have been enforced together. A case of that contract violation is shown below,

```
| ContractViolated: For multiclass classification activation_function 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 the 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 [34, 36, 37] 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.

<table>
<thead>
<tr>
<th>Class of bugs</th>
<th>DL Contract</th>
</tr>
</thead>
</table>
| Data normalization problem | @new_contract(normalization, normal interruption);
| | Postcondition: True |
| Incorrect activation and loss function regression | @new_contract(activation, loss function, regression);
| | Postcondition: True |
| Incorrect activation and loss function: binary classification | @new_contract(activation, loss function: binary classification);
| | Postcondition: True |
| Incorrect activation and loss function: multiclass classification | @new_contract(activation, loss function: multiclass classification);
| | Postcondition: True |
| Incorrect activation and loss function: multilabel/multiclass classification | @new_contract(activation, loss function: multilabel/multiclass classification);
| | Postcondition: True |
| Activation rate | @new_contract(activation);
| | Postcondition: True |
| Hyperparameter | @new_contract(hyperparameter);
| | Postcondition: True |
| Dying relu | @new_contract(dying relu);
| | Postcondition: True |
| Vanishing gradient | @new_contract(vanishing gradient);
| | Postcondition: True |
| Exploding gradient | @new_contract(exploding gradient);
| | Postcondition: True |

Figure 3: Methodology to collect deep learning contracts

4.2 Implementation
To implement DL Contract, we extended the open-source package PyContracts [32]. 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 [39]. 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.
Benchmark selection: To answer the RQs, we compare and contrast DL Contract against four recently-published DL performance bug localization benchmarks [52, 57, 62, 65]. The DeepLocalize's benchmark proposed by Wardat et al. [62] 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 [57], we followed their procedure. AUTOTRAINER [65] 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 [52] 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 [46, 65]. 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 [61, 65]. 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 until the complete execution of the program.

Lastly, if there is no bug detected in a correct program, we consider that as a true negative.

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 [46, 65]. 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 [61, 65]. 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 until the complete execution of the program.

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

<table>
<thead>
<tr>
<th>DL Contracts targeting class of bugs</th>
<th>DeepLocalize SO</th>
<th>UMLAUT CIF-10</th>
<th>NeuraLint SO</th>
<th>#Contract Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improper Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data normalization problem</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Structural bugs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incorrect activation &amp; loss function</td>
<td>17</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Incorrect activation in hidden layers</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Incorrect learning rate</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Training problem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overfitting</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>High validation accuracy</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>High dropout rate</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Dying relu</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Vanishing gradient</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Exploding gradient</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Oscillating loss</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Slow convergence</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

* 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, Deeplocate [62] and DL Contract (full table [14])

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DeepLocalize</th>
<th>UMLAUT</th>
<th>DL Contract</th>
<th>Correct Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source: StackOverflow</td>
<td>SO</td>
<td>CIF-10</td>
<td>CIF-10</td>
<td>CIF-10</td>
</tr>
<tr>
<td>Source: GitHub</td>
<td>GH</td>
<td>CIF-10</td>
<td>CIF-10</td>
<td>CIF-10</td>
</tr>
<tr>
<td>Source: CIFAR-10</td>
<td>CIF-10</td>
<td>CIF-10</td>
<td>CIF-10</td>
<td>CIF-10</td>
</tr>
<tr>
<td>Source: CIFAR-10</td>
<td>CIF-10</td>
<td>CIF-10</td>
<td>CIF-10</td>
<td>CIF-10</td>
</tr>
</tbody>
</table>

* 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 [57] and DL Contract

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Runtime</th>
<th>UMLAUT Runtime</th>
<th>DL Contract Runtime</th>
<th>Correct Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 (CIFAR-10)</td>
<td>1318.99</td>
<td>8.85</td>
<td>28.52</td>
<td>1376.04</td>
</tr>
<tr>
<td>A2 (CIFAR-10)</td>
<td>1483.26</td>
<td>8.95</td>
<td>24.80</td>
<td>1459.21</td>
</tr>
<tr>
<td>A3 (CIFAR-10)</td>
<td>1455.93</td>
<td>140.98</td>
<td>23.56</td>
<td>1453.29</td>
</tr>
<tr>
<td>B1 CIFAR-10</td>
<td>1493.12</td>
<td>152.57</td>
<td>16.28</td>
<td>1420.40</td>
</tr>
<tr>
<td>B2 (CIFAR-10)</td>
<td>1319.18</td>
<td>8.85</td>
<td>26.58</td>
<td>1448.27</td>
</tr>
<tr>
<td>B3 (CIFAR-10)</td>
<td>1692.83</td>
<td>664.60</td>
<td>669.83</td>
<td>1463.87</td>
</tr>
<tr>
<td>A1 (F-MNIST)</td>
<td>17.09</td>
<td>7.02</td>
<td>23.11</td>
<td>16.84</td>
</tr>
<tr>
<td>A2 (F-MNIST)</td>
<td>17.04</td>
<td>9.61</td>
<td>18.62</td>
<td>16.36</td>
</tr>
<tr>
<td>A3 (F-MNIST)</td>
<td>15.69</td>
<td>9.61</td>
<td>18.30</td>
<td>16.36</td>
</tr>
<tr>
<td>B1 (F-MNIST)</td>
<td>17.90</td>
<td>9.87</td>
<td>21.06</td>
<td>15.93</td>
</tr>
<tr>
<td>B2 (F-MNIST)</td>
<td>15.96</td>
<td>7.16</td>
<td>18.94</td>
<td>16.96</td>
</tr>
<tr>
<td>B3 (F-MNIST)</td>
<td>17.62</td>
<td>12.31</td>
<td>51.85</td>
<td>15.39</td>
</tr>
</tbody>
</table>

We recorded the total execution time utilization for all techniques when analyzing buggy and correct programs from the benchmarks and computed overhead.
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 a total of 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 trigger a total of 474 contract violation messages in 272 buggy programs. In Table 2, "-" indicates contract violations were detected 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 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 the same 15 contracts were used in 257 correct programs in benchmarks. We found 18 contract violations as false positives, mainly due to randomness factor [55, 67] 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 [15] for more details. Table 3 shows that DL Contract can detect 39 out of 41 buggy programs with precise contract violation messages. 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 [62]. Again, 2 out of 41 were not detected from DeepLocalize [62] 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 alarms for buggy programs.

Table 5 shows that DL Contract has detected 195 bugs in 203 buggy programs in the AUTOTRAINER benchmark. While AUTOTRAINER reports the symptoms of 5 training problems, DL Contract detects bugs 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 AUTO-TRAINER [65] 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 [52] 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 Compil and Fit, which were annotated with 15 DL Contracts. These 272 buggy programs have common root causes and symptoms. For instance, AUTOTRAINER [65] 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 [62], UMLAUT [57], AUTOTRAINER [65], NeurAlint [52] (in Table 7). We have evaluated 257 correct (clean) real-world programs and found 18 false positives. We have found 10 FP’s 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 TP’s and no FP’s) but has lower time consumption (in Fig. 4). Regarding the AUTOTRAINER benchmark, DL Contract could not detect bugs due to the accuracy threshold [65] (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 [61] 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 [62], UMLAUT [57], AUTOTRAINER [65], DeepDiagnosis [61] and NeurAlint [52] 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 [65] (AT) and DL Contract (DLC)

Table 6: Applicability of DL Contract, Runtime overhead comparison with NeuraLint [52] and DL Contract

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

Figure 4: Comparison of runtime overhead
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 [24, 57] from prior work we recruited 20 participants from university mailing lists for our study (17 Ph.D., 2 M.S. 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 ($\mu$) 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 [15].

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 evaluated 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 for 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 like TensorFlow, PyTorch. We focused on Keras to keep the implementation effort manageable and leverage this library’s large body of benchmarks.

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 [37, 46, 66] 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 [15] with detailed results.

5 RELATED WORK

Specification of Deep Neural Networks: The closest related ideas in the specification of DNNs include [31, 58, 59]. While [58] 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, [31] presents a technique for computing input and layer properties from a feed-forward network using input-output characterizations as formal contracts. Additionally, [59] introduces a method for repairing neural network classifiers by inferring the correct specifications. Both [31] and [59] 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. Recently, an empirical study [38] reports categories of required ML contracts, which may help designers of contract languages.

Deep Learning Testing, Debugging, and Repairing: Prior work on DL testing, debugging, and repairing includes DeepLocalize [62], MODE [46], AUTOTRAINER [65], DeepDiagnosis [61], DeepFD [25], Ariadne [29], Lagouvardos [40], Nikanjam et al. [52], SHAPETRACER [44], and Tensfa [63]. 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 long term, developers would also benefit from the documentation and write more correct DL programs. Empirical studies [26, 34, 36, 37, 56, 64, 66] 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 [32], Pylint [1], and PyTA [45], 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 [47]. 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, [60], which examines the usage of machine learning (ML) cloud APIs in open-source applications. This work finds that many of these applications contain API misuses that degrade their functionality and performance, leading to the development of automated checkers for identifying such misuses. [52] tackles API Misuse (APIM) bugs statically by some rules that occur when practitioners misunderstand the usage of deep learning APIs. Such misuse leads 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 may not be suitable for checking contracts written by library API designers that capture properties of models, data, and training behavior at various program points during runtime. To address this limitation, our 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. With ongoing research on decomposing DNN into modules [35, 53, 54], we intend to write contracts for the expected behavior of a DNN module effectively. We want to explore writing contracts to prevent nonfunctional bugs such as fairness bugs [20, 21]. We would also like to extend our approach to prevent additional types of bugs in different stages of the ML pipeline [22]. We can adapt techniques [50, 51] for collecting contracts from mined models with improved performance in terms of accuracy and training time.

7 DATA AVAILABILITY

The replication packages and results are available in this repository [17] that can be leveraged by further research.

ACKNOWLEDGMENTS

We acknowledge the reviewers for their insightful comments. We also thank the user study participants. This material is based upon work supported by the National Science Foundation under Grant CCF-15-18897, CNS-15-13263, CNS-21-20448, CCF-19-34884, and CCF-22-23812. All opinions are of the authors and do not reflect the view of sponsors.