Singapore Management University

Institutional Knowledge at Singapore Management University

Semantic-based neural network repair

Richard SCHUMI
Singapore Management University, rschumi@smu.edu.sg

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Semantic-Based Neural Network Repair

Richard Schumi
Singapore Management University
Singapore
rschumi@smu.edu.sg

Jun Sun
Singapore Management University
Singapore
junsun@smu.edu.sg

ABSTRACT

Recently, neural networks have spread into numerous fields including many safety-critical systems. Neural networks are built (and trained) by programming in frameworks such as TensorFlow and PyTorch. Developers apply a rich set of pre-defined layers to manually program neural networks or to automatically generate them (e.g., through AutoML). Composing neural networks with different layers is error-prone due to the non-trivial constraints that must be satisfied in order to use those layers. In this work, we propose an approach to automatically repair erroneous neural networks. The challenge is in identifying a minimal modification to the network so that it becomes valid. Modifying a layer might have cascading effects on subsequent layers and thus our approach must search recursively to identify a "globally" minimal modification. Our approach is based on an executable semantics of deep learning layers and focuses on four kinds of errors which are common in practice. We evaluate our approach for two usage scenarios, i.e., repairing automatically generated neural networks and manually written ones suffering from common model bugs. The results show that we are able to repair 100% of a set of randomly generated neural networks (which are produced with an existing AI framework testing approach) effectively and efficiently (with an average repair time of 21.08s) and 93.75% of a collection of real neural network bugs (with an average time of 3min 40s).

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Computing methodologies** → **Artificial intelligence**; • **Theory of computation** → *Constraint and logic programming*.

KEYWORDS

automatic AI model repair, deep learning models, semantics, specification, AI model generation, neural network generation, TensorFlow, Prolog

1 INTRODUCTION

Artificial intelligence (AI) is based on imitating natural intelligence or learning behaviour with a machine. The method called deep learning (DL) employs artificial neural networks to simulate the neurons of brains [43]. In recent years, DL has advanced into numerous fields, like language processing, face and speech recognition, due to improvements in computer hardware [42].

Although AI systems have been successfully used in many domains, they are not failure-free. Especially in safety critical applications, like autonomous driving, or medical systems, even minor bugs can have severe consequences. Studies of bugs in AI systems [32, 53, 64] have analysed various posted bug reports (or issues), and the large number of identified posts illustrates the high frequency of such bugs. These studies evaluated the time it took from posting a bug until it was resolved and the time ranges from weeks to months depending on the type of the issue.

Moreover, there are other factors that make it especially difficult and time-consuming to debug AI systems. Neural networks can have a long training time of days or weeks, which makes it cumbersome to evaluate potential ways of fixing a neural network. Often the error messages that occur during AI development can be unrelated to the actual issue that needs to be fixed [32], the error messages can be inconsistent, or there can even be hidden issues that do not produce error messages [45]. Another difficulty comes from the underlying AI frameworks such as TensorFlow or PyTorch. These frameworks are still being rapidly developed, and a new release may not be backward compatible and can break existing code [32]. Furthermore, the provided documentation can sometimes be vague or buggy [35], which makes it hard for developers to use unfamiliar AI components.

In order to make it easier for developers to fix and prevent such bugs, it is important to be able to systematically (and automatically) test and repair the underlying neural networks or deep learning models. Thus, in this work, we present a novel semantic-based neural network repair approach that helps AI developers fix common errors in the architecture and with the parameters of AI models.

Designing AI or deep learning models is a cumbersome task that requires a lot of expertise, domain knowledge, and effort. AI frameworks provide dozens of deep learning layers that can perform simple mathematical functions or complex operations, like different convolutions or recurrent layers. Most of these layers have preconditions, e.g., for the input data or for the parameters. Building a valid model requires the fulfilment of all the preconditions from all layers of the model, which can be challenging since layers can not only be combined in sequences, but in general can be in the form of directed acyclic graphs. Given such a graph, every connection can cause potential precondition violations, and identifying them can be cumbersome. Moreover, the debug information that is provided by AI frameworks can be imprecise or inconsistent [45]. Due to

that, the process of developing a new model can be time-consuming and frustrating. With our approach, an AI developer is provided with automatically generated changes that can repair the model. The changes are presented in a list, which is ordered based on a change indicator value that reflects the number (and magnitude) of changes that are required. Our approach facilitates and speeds up the time-consuming task of finding a valid model architecture and consistent model parameters. This can be especially helpful for new AI developers, who still lack the necessary skills to fix such issues on their own. Moreover, our approach can support other AI techniques, like AutoML [29] where, e.g., a model architecture can be automatically derived for a given problem, or automatic AI model generation [55], which can be used in various scenarios, like AI framework testing. For both these techniques, it can be hard to find valid model architectures, especially if randomness is involved. To avoid generating invalid models, existing AutoML or AI framework testing approach often rely on a limited set of models defined by templates. By repairing generated models, unnecessary pruning of the search space can be avoided, which might benefit both these techniques.

Our approach relies on an existing semantics called ExAIS [45] which defines the functionality of almost all TensorFlow layers in the logical programming language Prolog. ExAIS contains a number of preconditions that produce debug messages and enable the identification of model bugs. Based on these messages, we apply multiple algorithms to fix different types of errors. For example, for a dimension error (which occurs when the input does not have the required number of dimensions), a simple fix strategy could be to introduce a reshape layer.

However, it is usually not as easy to repair a neural network with just a single change, because in most cases there need to be further modifications in the following layers as a consequence, i.e., structural changes like that can also cause invalid weight shapes deeper in the network. Moreover, there are usually multiple potential fixes for a bug, and it can be challenging to find the best repair in different scenarios. There is already an existing tool called Tensfa [57, 58] that can automatically repair dimension errors or more generally tensor shape faults, which are also bugs that we are targeting with our method, but in contrast our approach can also repair different types of bugs and it works more straight forward as we will explain later.

To sum up, the main contributions of our work are:

- We present a novel semantic-based AI model repair approach that is able to suggest multiple model changes that can fix an invalid AI model.
- We collect a set of real bugs from AI developers in order to evaluate our approach in a realistic setting.
- Additionally, we performed an evaluation with randomly generated models in order to test our approach for a diverse set of models and bugs.

Structure. The rest of the paper is structured as follows. In Sect. 2, we introduce necessary background information, e.g., about the semantics that we apply. In Sect. 3, we present the underlying algorithms of our repair approach in detail. In Sect. 4, we show an evaluation with two usage scenarios. Lastly, we review the related work in Sect. 5 and conclude in Sect. 6.

2 BACKGROUND

In this section, we describe the technologies that will be supported by our repair approach as well as the underlying semantics.

2.1 AI Framework Testing

While our primary goal is to support programmers when they are developing AI models manually, another interesting application that can be supported by our approach is AI framework (such as TensorFlow and PyTorch) testing. There is an active line of research on automatic testing of AI frameworks [24, 25, 35, 40, 46, 49, 55]. These works are motivated by the fact that bugs in AI frameworks might potentially affect all AI applications that are built with such frameworks. Due to that and also due to the fact that AI frameworks can still suffer from severe bugs [45], it is important to thoroughly and systematically test them. Existing AI framework testing techniques can be categorized into the following groups: differential testing [40, 49, 55] and metamorphic testing [24, 25, 46]. Differential testing is a technique that compares multiple frameworks (or implementations) against each other, and bugs are discovered when there is an inconsistency. Metamorphic testing for AI frameworks usually introduces changes in the input data for the AI model that should not affect the output, and a different output (prediction result) would suggest an issue in the AI framework. For both these techniques, it is important to have a wide variety of valid AI models in order to test different aspects of the AI frameworks.

Our repair approach can support the generation of such models by enabling effective random model generation that would normally produce a high percentage of invalid models or by allowing other approaches to explore more complicated scenarios that might often be dismissed due to difficulties in finding a working model.

2.2 AutoML

Another application that can be supported by our repair approach is Automatic machine learning (AutoML) [29, 62]. AutoML, which is also called neural architecture search (NAS) in the deep learning context, is a term for methods that try to reduce the need for manual model building for various learning tasks, like image/object recognition, or language modelling. Developing AI systems usually requires a lot of expertise and effort, e.g., for data pre-processing tasks, like feature engineering and to find a deep learning model, i.e., to choose a model architecture and to tune the model in order to achieve an acceptable accuracy. There has been a lot of progress with AutoML techniques, which can potentially take over such tasks and sometimes produce models with higher accuracies compared to hand-crafted models from experienced AI developers [65]. There are many AutoML approaches that try to find model architectures and tune parameters with different strategies.

Depending on the search strategy, some AutoML approaches can produce invalid models that have to be discarded [65] which hinders the performance of AutoML. To avoid generating a large number of invalid models, AutoML approaches usually apply only a limited set of predefined layers or groups of layers (a.k.a. cells) to build neural networks based on knowledge from manually developed networks, which might hinder the discovery of architectures that outperform existing ones [26]. With our approach, we can support a variety of layers and neural network architectures. Hence, we

believe that our semantic-based repair method can help to enable AutoML approaches to explore an extended search space.

2.3 ExAIS

Our technique utilises an executable AI semantics called ExAIS [45] that is written in the logical programming language Prolog [38]. Prolog is a declarative language that relies on first order logic. Programs in Prolog are built with rules and facts, which are usually concerned with mathematical relations. The declarative nature of the language facilitates the creation of high level semantics. Moreover, it supports various list operations and mathematical expressions, which makes it convenient for specifying deep learning layer behaviour that is often concerned with high dimensional input and mathematical operations.

Listing 1 shows the Prolog semantics of a Dense layer [2] of ExAIS. It is a standard densely connected layer, which contains a number of nodes, each of which is connected with all inputs. The output is computed as the sum of all inputs (multiplied with the weights) at each node with an added bias value. The specification works as follows. The rule starting in Line 1 has multiple parameters: a list [I|Is], a weight array IWs, and a bias array Bs (which can be intuitively regarded as inputs) and a list Os (which can be regarded as the output). The list notation [I|Is] enables access to the first element I and the remaining elements Is of a list. Line 2 constrains the depth of the nested input to two. We handle higher dimensional input separately. Line 3 applies a predicate that is true when O can be unified as the expected output for an input I, and Line 4 is a recursive constraint, which intuitively continues with the next inputs. The rule in Lines 6–9 is similar, except that it handles (layer) inputs with a higher dimension, which is checked in Line 7, and recursively uses the initial predicate from Line 1 since the dense layer only performs computations in the innermost list even when it receives high dimensional input data. Line 11 (and Line 18) are the base cases for the recursion, i.e., when only an empty list remains.

The predicate in Line 13 encodes the main layer functionality and becomes true when the Res variable is the expected output for the input [I|Is]. It has the same arguments as the first rule and an additional temporary variable Res0 for the result. It consists of clauses for multiplying the weight arrays IW with each input I and for adding the results in Line 16. The predicates multiply_list_with and add_lists are straightforward and are therefore omitted. With this Prolog semantics, we can now answer a variety of queries, e.g., to compute the expected output of a Dense layer.

More relevantly, ExAIS contains preconditions that reflect layer requirements, like a specific input shape, or dependencies between the arguments. An example precondition to check if layer input data has a minimum number of dimensions is illustrated in Listing 2. The predicate check_min_dimensions takes the input data and a minimum dimension value as arguments. Line 2 shows a predicate that becomes true, when D1 can be unified to the dimension number of Is. Next, there is a condition to check if the dimensions of the input are smaller than the given minimum value. If it is smaller, then an error message is produced. Otherwise, the predicate becomes true. Most layers of ExAIS contain preconditions in this form. The preconditions are part of the ExAIS semantics and were created manually by the ExAIS developers according to the TensorFlow

```
1   dense_layer([I|Is], IWs, Bs, [O|Os]) :-
2       depth([I|Is],2),
3       dense_node_comp(I, IWs, Bs, O),
4       dense_layer(Is, IWs, Bs, Os).
5
6   dense_layer([I|Is], IWs, Bs, [O|Os]) :-
7       depth([I|Is],D), D > 2,
8       dense_layer(I, IWs, Bs, O),
9       dense_layer(Is, IWs, Bs, Os).
10
11  dense_layer([], _, _, []).
12
13  dense_node_comp([I|Is],[IW|IWs],Res0,Res) :-
14      multiply_list_with(IW,I,Res1),
15      add_lists(Res0,Res1,Res2),
16      dense_node_comp(Is,IWs,Res2,Res).
17
18  dense_node_comp([],[],Res,Res).
```

Listing 1: Prolog semantics of the Dense layer [45].

```
1   check_min_dimensions(Is, D) :-
2     depth(Is,D1),
3     (D1 < D ->(write("Invalid Model, Badness Value: "),
4         BV is D1-D,BV1 is BV*100000000000000000,
5         writeln(BV1),
6         S1 = "Dimension error, Input Shape ",
7         shape(Is,Shape),
8         term_string(Shape,S2),
9         string_concat(S1,S2,RS),
10        S3 = ", Expected Min Dimensions ",
11        string_concat(S3,D,RS1),
12        string_concat(RS,RS1,S),
13        throw(S));true).
```

Listing 2: Precondition to check if the input data has a minimum number of dimensions.

documentation and according to other publications that describe the layer functionality and requirements. When an AI model is executed with the semantics, then all the preconditions of the individual layers are checked. Any violation of the preconditions would make the model invalid. Hence, the preconditions can help identify problematic model aspects. In this work, we utilize this feature to enable our automated AI model repair approach. The execution of the Prolog predicates works similarly to the execution of functions. There is a predicate for each layer, which contains precondition calls and calls to subpredicates, which enables an automatic precondition check during the layer execution. There might be layer preconditions that are not specified in ExAIS and their violations would not be repairable by our approach, but since we tested thousands of random models with our approach, we are confident that we can resolve precondition violations in most cases.

The semantics consists of 65 deterministic layers and seven non-deterministic layers. We focus on repairing deterministic layers in this work. Only six of the 65 layers have no preconditions (i.e., the Flatten layer, ReLU, ThresholdedReLU, LeakyReLU, Masking, TimeDistributed). Seven of the layers have simple preconditions (UpSampling1D-3D, ZeroPadding1D-3D, Embedding), i.e., they only require input with a certain number of dimensions. The remaining layers have non-trivial preconditions that mainly fall in the following categories. First, there are consistency requirements between the layer arguments (or inputs) and the weight shape, e.g., for a

dense layer the first dimension of the weight needs to have the same size as the last dimension of the input. Second, we may have inconsistencies among the shapes of multiple inputs of a layer, i.e., some layers perform mathematical operations, like addition or multiplication of multiple layer inputs. These layers may require that the shape of the inputs must be the same. Third, there can be consistency requirements between the layer arguments or with the layer arguments and the inputs. For example, for some convolutional layers setting a dilation_rate value not equal to one is incompatible with specifying any stride value not equal to one.

During our investigations, we noticed that there are various studies that evaluate AI faults (based on bug reports) and that some of them are related to violations of the layer preconditions of ExAIS and can thus be detected by its precondition checks. For example, one study [32] investigated bug reports and it illustrates data and layer dimension errors, which make up close to 30% of the findings. Moreover, the study showed a broad categorisation of the manual repairs that were suggested in the bug reports. Another study [31] discusses Tensor shape errors that are related to input shape violations that can be captured with ExAIS's preconditions. The source of most bugs of those studies was not directly related to a misuse of layers or a wrong network architecture, which made them inapplicable for our approach, but their frequency highlights the significance of such bugs.

Additionally, we performed experiments with randomly generated models, and collected bug reports from stackoverflow as we will explain in Section 4 and in both cases we observed bugs that are related to precondition violations of ExAIS. Hence, we believe that bugs based on the precondition violations are relevant and that it is important to provide better ways to fix such bugs. Based on the three categories of preconditions and the simple dimension preconditions, we developed repair algorithms for the issues that are identified with the preconditions. We explain these algorithms in the following section.

3 METHOD

In this section, we describe how we fix specific bugs and how our repair algorithm for AI models works in detail.

We consider a repair to be valid if it removes TensorFlow errors (that occur during the model execution) with minimal adjustments. We created our repair suggestions with the intention to preserve the original model as much as possible, i.e., we were looking for minimal model adjustments that maintain the layers and most of the structure of the model. The motivation behind this is the fact that developers usually make small mistakes [50]. For simple cases, like dimension errors, our repairs follow the best practice, as suggested by the accepted solutions from bug reports. For more complicated cases, like argument or shape issues, we developed repairs that only utilise standard layers [3] (or argument modifications). There would be various other repair options if more deep learning operations, like NumPy functions [28], would be considered. We believe that fixes with standard layers are well suited since they are straightforward, easier to understand, and many AI developers stick to these layers. Generally, it is hard to justify the quality of more complicated repairs without an additional training and model validation step. We intend to further explore these steps in future

work. Overall, we believe that our repairs are reasonable, especially since they often were equivalent to accepted solutions for real bugs as we will explain later in Section 4.

To illustrate the repair of specific model bugs, we present our approaches for fixing common errors that are related to the misuse of standard deep learning layers, i.e., dimension, input shape, and argument errors. Bugs that originate from other AI development tasks, like the data preprocessing, training, or model validation, are out of the scope of this work. Table 1 gives a simplified overview of our repair approaches for these errors and shows example fixes.

For dimension errors, there are two cases: the expected number of dimensions is larger than the actual number of dimensions, or the opposite. For the first case, we check if the problematic layer can be replaced with a higher dimensional version (e.g., Conv2D with Conv3D). Alternatively, this case can be repaired by adding a reshape layer with dimension size one before the layer with the bug. A reshape layer can modify the shape of the input data, while it still keeps the same number of values, i.e., the product of the dimension sizes will be the same after a reshape. The second case, i.e., the expected number of dimensions is smaller than the actual dimension number, can be handled similarly. A layer can be replaced with a smaller dimensional version, but in contrast to the previous case, the reshape works differently. In order to obtain a smaller dimension number, a new shape is computed by combining the last two dimensions of the given input (by multiplying them). Both these repair options are considered for our algorithm. The final minimal suggested fix is determined based on which of the two fixes leads to a smaller overall change.

Next, for input shape bugs, there are also two potential repairs. Inconsistent input shapes can occur when a layer that takes multiple inputs (e.g., Add) has incompatible inputs with different shapes or dimension sizes. One way to resolve this bug, is to add padding around one of the inputs, i.e., with a ZeroPadding layer that increases the input space and adds zeros around the given data. An example repair with such a layer is illustrated in the second row of Table 1. It shows a graph model with an Add layer that has two ReLU layers with different shapes as input. The fixed model has an additional ZeroPadding1D layer that is in-between the ReLU layer with the smaller input and the Add layer. A padding layer can resolve most of such bugs, but not all, since there are only padding layers that take three to five-dimensional inputs. Alternatively, if the dimension is outside this range, the shape mismatch can be fixed with a Concatenate layer. This layer can adjust the input shape by combining a certain dimension of the input with additional values or arrays of values, which enables our repair approach to change the input shape according to the requirements of the invalid layer.

Thirdly, for argument errors, there are also a number of possible repairs. Most of these errors can be repaired by regenerating the layer arguments, i.e., by randomly exchanging the arguments of the layer with random values. This procedure works step by step, first a replacement of a single argument is tried and if this is not successful then more arguments are exchanged. A special case of an argument error is a weight shape bug that occurs when the weight data does not match with other arguments/input data of the layer or when it has a wrong dimension number. This case can be repaired by regenerating the weights of the layer while considering the other layer arguments and the input shape.

Table 1: Repair approaches for different types of bugs.

	Repair Approaches	Examples
| **Dimension Error** | **Input data needs more dimensions**<br><br>• replacement of the layer that causes the error with a lower dimension version if it is available<br>• insertion of a reshape layer that adds a dimension with size one to increase the number of dimensions of the data<br><br>**Input data needs fewer dimensions**<br><br>• replacement of the layer that causes the error with a higher dimension version if it is available<br>• insertion of a reshape layer that reduces the number of dimensions by combing (multiplying) the last two dimensions into one dimension (e.g., a shape (2,3,2) would become (2,6)) | **Model with a bug**<br><br>```
model = tf.keras.Sequential([
  Dense(10, input_shape=(10,10,10,)),
  Conv2D(10, kernel_size=(2,2)),
  MaxPooling3D(pool_size=(3,3,3))])
```<br>**Fixed model**<br><br>```
model = tf.keras.Sequential([
  Dense(16, input_shape=(10,10,10,)),
  Conv2D(16, kernel_size=(2,2)),
  MaxPooling2D(pool_size=(3,3))])
``` |
| **Input Shape Bug** | **Multiple inputs of a layer need to have matching shape or dimension sizes**<br><br>• addition of a ZeroPadding1D-3D layer to adjust one of the input shapes according to the expected one from the error message (if the dimensions of the input are between three and five)<br>• insertion of Concatenate layer that can adjust the input shape by combining a specified shape with additional values | **Model with a bug**<br><br>```
R1 = ReLU(input_shape=(2,2,))
R2 = ReLU(input_shape=(5,2,))
A  = Add()([R1, R2])
```<br>**Fixed model**<br><br>```
R1 = ReLU(input_shape=(2,2,))
Z  = ZeroPadding1D(padding=(2,1))(R1)
R2 = ReLU(input_shape=(5,2,))
A  = Add()([Z, R2])
``` |
| **Argument Error** | **Layer arguments have consistency requirement violations among each other or with the input**<br><br>• step-wise (random) regeneration of the layer arguments by increasing the number of arguments that are changed<br><br>**Weight shape bugs due to weight data that does not match with other arguments/input data or has wrong dimensions**<br><br>• regeneration of the layer weights while considering the other layer arguments and input shape<br><br>**Pool or kernel shapes that are too large for the input shape**<br><br>• regeneration of the pool/kernel shape<br>• regeneration of layer arguments, like padding | **Layer with a bug**<br><br>```
Cropping1D(input_shape=(4,4,),cropping=(2,2))
```<br>**Fixed layer**<br><br>```
Cropping1D(input_shape=(4,4,),cropping=(2,1))
```<br><br>**Layer with a bug**<br><br>```
MaxPooling2D(input_shape=(8,8,8,),pool_size=(9,9))
```<br>**Fixed layer**<br><br>```
MaxPooling2D(input_shape=(8,8,8,),pool_size=(4,3))
``` |

Another case that needs to be handled separately are pool (or kernel) shapes that are inconsistent with the input shape, because they are too large. These bugs can be fixed by adopting the pool (or kernel) shapes or by regenerating other layer arguments, like padding. There are more special cases like this that have their own preconditions and need some specific approaches to be repaired, which were omitted for brevity. There are two example layers given for this bug type as illustrated in the last row of Table 1. First, there is a Cropping1D layer, that applies too much cropping, which would result in an empty output. The bug is fixed by reducing the cropping size values.

Another example shows a pool shape bug in a MaxPooling2D layer that has a pool shape that is too large for the specified input shape. A fix for this bug is produced by replacing the pool sizes with smaller values. It should be noticed that in reality such bugs are much harder to find since the models are larger, have many arguments, and the input and output shapes of layers are often not easy to see. Moreover, a model can usually not just be repaired with a singular change of a layer since in many cases there need to be further adjustments deeper in the neural network. For example, argument regenerations often change the output shape of a layer, which can cause inconsistencies with the weight or shape requirements in the next layers. The following simple example model illustrates this cascading behaviour.

```
C1 = Conv1D(2,input_shape=(8,1,),kernel_size=2,dilation_rate=3)
C2 = Conv1D(2,input_shape=(16,1,),kernel_size=2,dilation_rate=3,
        strides=3)
S  = Subtract()([C1, C2])
```

Algorithm 1 Pseudo code of the repair algorithm.

Input: *SemanticHelper* helper class for the execution of the semantics

1: **function** FINDFIXWITHMINIMALCHANGE(rand,model, error, maxFixes) ▷ find single fix with minimal change in the AI model
2:     *workingFixes* ← {}
3:     *fixingCount* ← 0
4:     **do**
5:         *fixes* ← *errorSpecificFixes*(*rand, model, error, maxFixes*) ▷ produces fixes based on the approaches of Table 1
6:         **for each** $fix \in Fixes$ **do**
7:             (*success, newError*) ← *SemanticHelper.run*(*fix*) ▷ run semantics, get success/error
8:             **if** $\neg success \wedge$ (*newError.getBadness*() < *error.getBadness*() ∨ *newError.getLocation*() ≥ *error.getLocation*()) **then**
9:                 *fix* ← *findFixWithMinimalChange*(*rand, fix, newError, maxFixes*)
10:             *workingFixes.add*(*fix*)
11:         *fixingCount* ← *fixingCount* + 1
12:     **while** *workingFixes.isEmpty*() ∧ *fixingCount* < *maxFixes*
13:     **return** *getFixWithSmallestChangeValue*(*workingFixes*) ▷ returns the fix with the smallest change value

14: **function** FINDFIXES(rand, model, error, maxFixes)
15:     *workingFixes* ← {}
16:     *fixes* ← *errorSpecificFixes*(*rand, model, error, maxFixes*) ▷ produces fixes based on the approaches of Table 1
17:     **for each** $fix \in Fixes$ **do**
18:         (*success, error*) ← *SemanticHelper.run*(*fix*) ▷ run the semantics, get success/error
19:         **if** $\neg success$ **then**
20:             *fix* ← *findFixWithMinimalChange*(*rand, fix, error, maxFixes*) ▷ uses the first function
21:         *workingFixes.add*(*fix*)
22:     **return** *sortByChangeValue*(*workingFixes*) ▷ sorts the fixes accoring to their change value

---

It shows a Subtract layer that has two Conv1D layers as inputs. The model seems to be valid at a first glance. Both Conv1D layers would produce the same output shape, since the stride argument (that specifies the step size with which the kernel is moved over input data) offsets the larger input shape of the second convolution. However, a stride value greater than one is incompatible with a dilation_rate value greater than one. (A dilation_rate can be specified to expand a kernel with zero values.) A regeneration of the arguments to fix this violation, e.g., by replacing the stride or dilation_rate value, will always change the output shape. As a consequence, there will be an inconsistent input shape error at the Subtract layer, which needs to be fixed as shown in Table 1.

The overall repair approach that incorporates the specific fixes (from Table 1) is outlined in Algorithm 1. It consists of two major functions. A function that tries to find a singular working fix with a minimal change compared to the original model, and a function that returns a number of potential working fixes by applying the first function. The first function *findFixWithMinimalChange* takes a random object, a model to be fixed, an error object, and a maximum number of fixes that should be considered as input. In Lines 1–2, we initialise a set *workingFixes* and a counter *fixingCount*. Then, there is a do-while loop that continues until there is a working fix, or until the maximum number of allowed fixing attempts is reached. Within the loop, the *errorSpecificFixes* function (Line 5) is called to receive the potential fixes for a given error. For each of the fixes, we apply our *SemanticHelper* to check if it produced a valid model. The *SemanticHelper* is a wrapper class that helps with the execution of ExAIS to make a prediction for a given model, by returning a success message or an error object. If it returns an error, then we check if the error helped to improve the model, i.e., with the help of a badness value (Line 8), and try to further repair the model by

recursively calling the function. The badness value is calculated by preconditions of ExAIS and returned with an error message, and it works similar as fitness in search-based software testing [27]. It is a distance metric that is larger when the layer arguments and inputs are far from being valid, i.e., the layer preconditions are ranked based on severity and a value is calculated by taking into account the difference of an observed and an expected argument value and by multiplying it with a severity factor [45].

Finally, in Line 13, a helper function is used that sorts our working fixes based on a change value that indicates the similarity to the original model, and return the fix with the smallest change. The change value is another distance metric that is calculated by considering the number of layer arguments, the layer replacements and additions that are required to repair the original model. We consider an argument modification the smallest change (change value 1), followed by a layer replacement (change value 5), and the largest changes are layer additions (change value 10).

The second function *findFixes* has the same arguments, and also initializes a set for the fixes. Line 16 applies the *errorSpecificFixes* function to obtain potential fixes for the specified error. Then, for each of these fixes, we check if it is working with the *SemanticHelper*. If it is not, then we apply the *findFixWithMinimalChange* to recursively find a working fix. The fix is added to the fix set, which is sorted and returned at the end of the function. The difference of this function to the first one is that it produces a number of potential fixes instead of a single one.

Listing 3 illustrates a real neural network in the form of a TensorFlow Python program that was posted on stackoverflow [4] and contains a bug. The model is invalid, because the Conv1D layer requires input data with three dimensions, but it receives a two-dimensional input. In order to run such a neural network with

```
1    import tensorflow as tf
2    from tensorflow.keras import layers, models
3    model_valid = tf.keras.Sequential([
4    layers.Flatten(input_shape=(10,)),
5    layers.Dense(16,  activation='relu'),
6    layers.Conv1D(16, kernel_size=(2), activation='relu
         ', padding='same'),
7    layers.MaxPooling1D(pool_size=(4), strides=3,
         padding='valid'),
8    layers.Flatten(),
9    layers.Dense(1, activation='softmax')
10   ])
```

Listing 3: Real model with a bug from stackoverflow [4].

ExAIS, it needs to be converted into a Prolog model, which is shown in Listing 4. Fortunately, this conversion can be done automatically with our repair tool. Our tool supports models in an JSON format which is based on a format introduced by SOCRATES [41]. AI models can be exported into this format in a straightforward way. We automatically convert these JSON models into Prolog models (in the form of queries). The model consists of predicate calls for the layers that are assigned to variables (Lines 1–6). For the execution of the model (i.e., making a prediction) there is an exec_layers predicate in Line 7. This predicate takes a list of layer variables, executes them, checks the preconditions and helps to identify the name of a layer that violates a precondition. Since the model is invalid, it cannot be used for a prediction. The execution returns the following error message to show the precondition violation.

```
Invalid Model, Badness Value: -100000000000000000
Aborted at Con60545: Dimension Error, Input Shape [1,16],
    Expected Dimensions 3!!!
```

This message gives us the necessary information to repair the bug, i.e., it shows the problematic layer and what needs to be changed. A fix that was produced for the error is shown in Listing 5. This fix was computed by the *findFixWithMinimalChange* function and it has minimal changes, i.e, it is the most similar to the original model, since the only change is the insertion of a reshape layer. The reshape adds a dimension with size one to the input data to satisfy the dimension requirements for the Conv1D layer. Alternatively, such an error could be resolved by looking for a lower dimensional version of the Conv1D layer, but since it is already the version with the lowest number of dimensions, this is not feasible in this case.

4 EVALUATION

In this section, we evaluate the effectiveness and performance of our approach. We implemented our approach that can support various types of neural networks and tested it with a variety of AI models. To demonstrate its usefulness, we show two example applications, i.e., repairing automatically generated neural networks, and repairing manually designed neural networks with bugs. We perform multiple experiments to answer the following research questions (RQ).

- *RQ1: How effective is our approach in repairing automatically generated models?* There are various issues, like inconsistencies with layer arguments, or a faulty network structure, that can result in invalid neural networks. It is important to evaluate to what extent we can repair such bugs.

- *RQ2: How effective is our approach in repairing manually written models?* Developing an AI model is an error-prone task. To highlight the usefulness of our repair approach in practice, we demonstrate the applicability of our approach for a number of real modelling bugs, and investigate to what extent our suggested repairs are relevant for fixing these bugs.

- *RQ3: What kind of AI models can we support with our approach, and how efficient is our repair method?* To clarify the scope of our supported models, it is important to explain what kind of layers and model types we support. Moreover, it also makes sense to investigate up to what network size our approach still runs in a reasonable time.

The experiments were performed on a 7th Gen. Lenovo X1 Carbon ThinkPad with an 8th Gen i7 CPU with four 1.80 GHz cores and 16 GB RAM. For executing the Prolog semantics, we used SWI-Prolog 8.2.1, and our repair tool was built in Java 13.0.7. It consists of about 10,000 lines of code and includes some of the functionality of the ExAIS test case generator in order to produce Prolog models. Moreover, it uses the JSON.simple library and can load and save models in a JSON format that is based on the format from SOCRATES [41]. Additionally, it can store models as TensorFlow programs, and visualize them with Dot from Graphviz. The tool together with our experiment data and results are available in our repository [22]. In the following, we present our answers to the research questions.

RQ1: How effective is our approach in repairing automatically generated models? There are a number of approaches that automatically generate AI models for the purpose of AI framework testing. Wang et al. [55] show a differential testing approach called LEMON that uses mutations to generate AI models. The approach produces 100% valid models since the mutations are designed to generate only working models, but LEMON is limited to 24 types of layers.

A fuzzing method that extracts information from AI library documentations was illustrated by Li [35]. The approach generates AI models based on learned input requirements of deep learning layers, but it only produces about 25% valid models that are limited to singular layer models.

Another fuzzing approach that generates AI models was presented by Schumi and Sun [45]. The method is semantic-based and produces AI models with an optimisation algorithm with some rudimentary fix components that work more random and focus more on replacing problematic layers. The approach focuses more on generating valid models, but incorporates rudimentary repairs, for dimension errors and input shape bugs. The algorithm works by adding, replacing or deleting layers and it is guided by a badness value that indicates how far the model is away from being valid. However, it restarts the generation if it cannot find a working model or it just deletes problematic model components. Hence, in many cases it does not perform a fix. It is able to produce 99.1% valid models for differential testing. We use this approach as a baseline to compare it to our method.

In order to evaluate the effectiveness of our approach, we performed an experiment with 1,000 randomly generated invalid AI models. The models were produced with an adopted generation approach from [45] by randomly connecting various types of layers.

```
1    LFla71197 = flatten_layer([[1.2079020849786344, 1.449234775683509, ...]], Fla71197),
2    LDen79959 = dense_layer(Fla71197, [[0.6039479770827674, 0.410518536377564, ...]], Den79959),
3    LCon60545 = conv1D_layer(Den79959, 2,[[[0.6711,...]]],[...], 1, true, 1, Con60545),
4    LMax46787 = max_pool1D_layer(Con60545, 4, 3, false, Max46787),
5    LFla21960 = flatten_layer(Max46787, Fla21960),
6    LDen25740 = dense_layer(Fla21960, [[0.6182226038831956, ...]],[...], Den25740),
7    exec_layers([LFla71197,LDen79959,LCon60545,LMax46787,LFla21960,LDen25740],["Fla71197","Den79959","Con60545","
         Max46787","Fla21960","Den25740"],Den25740,"Den25740").
```

Listing 4: ExAIS Prolog code of a neural network with a bug.

```
1    import tensorflow as tf
2    from tensorflow.keras import layers, models
3    model_valid = tf.keras.Sequential([
4    layers.Flatten(input_shape=(10,)),
5    layers.Dense(16,  activation='relu'),
6    layers.Reshape((16,1)),
7    layers.Conv1D(16, kernel_size=(2), activation='relu
         ', padding='same'),
8    layers.MaxPooling1D(pool_size=(4), strides=3,
         padding='valid'),
9    layers.Flatten(),
10   layers.Dense(1, activation='softmax')
11   ])
```

Listing 5: Potential repair for the buggy model.

Many layers are connected sequentially by feeding the output as input to the next layer in the sequence. However, there are also layers that take multiple inputs, e.g., to perform mathematical operations, like addition. For these layers, multiple input layers are connected. It should be noted that a random model that is produced in such a way is nearly always invalid due to various layer preconditions that must be satisfied.

The resulting neural networks had an average size of 7.4 layers (excluding input layers). We set the parameters to use small inputs with a range of one to four values per dimension, since bigger sizes vastly increased the overall size of the highly dimensional input data (with up to 7 dimensions) and slowed down the execution of ExAIS. On average there were 1.8 bugs per model and in total there were 1797 bugs that needed to be fixed. An overview of the types of bugs is shown in Table 2. It can be seen that argument errors are the most common, followed by input shape bugs and dimension errors. Our approach successfully repaired all the bugs, i.e., we were able to repair 100% of the random bugs that occurred within the randomly generated neural networks, and it took on average only 21.08s per model. Being able to repair such a wide set of models with various different layer types shows that our approach is highly effective. In order to compare our approach to the baseline, we executed the same 1,000 invalid random tests with the optimisation algorithm from [45]. The approach was able to find a valid fix that was similar to our solution in 87.9% of the models, i.e., the layers of the model were kept and similar adoptions as with our algorithm were made. It took on average 8.96s, which is about twice as fast as our approach, because it discards models (or model parts) early when they are not working, and since it does not include a comprehensive search algorithm that tries to find minimal working fixes. For 121 cases, it did not produce a model that we would consider a fix since the algorithm was randomly replacing or removing layers or regenerating the whole model,

Table 2: Overview of the produced model bugs.

Type of Bug	Number of Bugs
Dimension Error	219
Weight Shape Issue	45
Input Shape Bug	705
Argument Error	828
Total	**1797**

i.e., the fixed models were altered in a way that made them no longer recognisable compared to the original model. Hence, we believe that our method showed a significant improvement of this state-of-the-art approach.

RQ2: How effective is our approach in repairing manually written models? In order to answer this research question, we collected a set of real modelling bugs from AI developers that were reported on stackoverflow. We searched issues for which people were struggling to find a solution for an error message from TensorFlow that was caused by a misuse of one of the standard layers or a wrong combination of these layers. To find these issues, we used search queries based on common error messages from TensorFlow that we obtained with the previous random test models. For example, we searched with error specific queries, like "TensorFlow shape must be rank", "TensorFlow inconsistent input shapes", "Operands could not be broadcast together with shapes", and with general queries, like "TensorFlow layer argument error" or "TensorFlow layer dimension error". We found hundreds of bugs reports that we had to manually inspect since the vast majority of them had causes unrelated to the model or layer definition, i.e., they were not relevant to our approach.

Some of the bugs are outside the scope of our method, since the cause of these bugs is not related to the model structure or layers. AI development has several steps, like the data preprocessing, training, or model validation, where bugs can occur, but our approach is only concerned with bugs within the models, e.g., by a wrong use of standard layers. Moreover, many models contain hand-crafted layers or enhanced features that are not supported by the semantics.

We found 16 bugs that matched our criteria, five with dimension errors [4, 6, 9, 16, 17], three with invalid kernel or pool shapes [12, 20, 21], five weight shape issues [7, 10, 13, 18, 19], and three bugs with inconsistent arguments or input shapes [8, 14, 15]. An overview of these bugs, their type, model sizes and repair times is shown in Table 3. For 15 of the models, our method was able to find potential repairs. One model [9] was too large for the execution with the semantics since the vocabulary data (of an Embedding

layer) was too much (25001x200) for Prolog to handle, but after reducing the vocabulary size (to 2501x200), which was irrelevant to the bug in the model, our method was able to produce a repair. On average the repair time was 3min 40s, the maximum time was 23min 21s. The model sizes ranged from about 1KB up to 43MB.

The results show the effectiveness of our method in repairing real modelling bugs, which we believe is important because it seems to be common that AI developers have difficulties to correctly design an AI model or to understand error messages from AI frameworks.

In order to further evaluate the practical usefulness of our approach, we evaluated our produced repairs for these manual bugs by comparing them to the accepted solutions from stackoverflow. Table 4 gives an overview of this comparison. Out of the 16 models, eight of our repairs were equivalent or very close (in terms of the type and magnitude of the modification) to the suggested solutions on stackoverflow [4, 6, 7, 9, 10, 17–19]. For example, for dimension issues, our solutions would be to introduce resize layers as it was also suggested on stackoverflow. Five of our repairs were of similar quality compared to the posted solutions [8, 12–14, 16], e.g., for inconsistent weight shapes we usually regenerate the layer weights based on the other layer arguments. Alternatively, such issues can be addressed by adjusting the layer arguments to correct the weight shape. Our remaining three repairs were not related to the suggested stackoverflow solutions [15, 20, 21], but they still produced working models that might be helpful as an alternative solution for the bugs. For example, a wrong pool/kernel shape can be corrected in multiple ways. Our solution to randomly regenerate the layer arguments step by step usually produced a valid repair, but there can still be other better solutions. Based on these results, we believe that our repairs are reasonably useful in practice and can be helpful for various real modelling bugs.

RQ3: What kind of AI models can we support with our approach, and how efficient is our repair method? Since our approach is based on the existing semantics ExAIS, we support neural networks that are supported by the semantics. ExAIS supports 65 deterministic layers of various types, like convolutional, pooling, recurrent, activation, normalisation, mathematical, and cropping. It was developed for TensorFlow and support almost all of its layers, i.e., the developers built ExAIS for TensorFlow 2.4 and identified 72 unique non-abstract and non-wrapper layers. Only 7 of these layers were not included. A full list of the supported layers is available in the repository of ExAIS [22]. Hence, our repair approach is able to repair a variety of neural networks, and nearly all types of layers.

In order to evaluate to which extent our approach works with different model sizes and how fast it can repair these models, we set up two experiments. (1) We evaluate the repair time of randomly generated neural networks of different types with an increasing number of layers. For this generation, we restricted the types of layers since the execution time of the semantics is very different depending on the type of layers. (2) We use models for benchmark datasets with more realistic input and weight data sizes in order to evaluate the performance in a practicable setting.

The results of the first experiment are shown in Fig. 1 and Fig. 2, which illustrate the repair time of our approach for different types of models and for increasing layer numbers. Fig. 1 shows sequential models that include activation layers and (1) dense layers, (2)
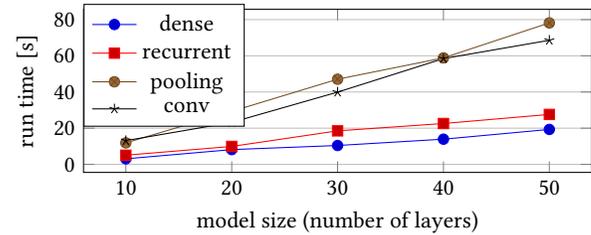

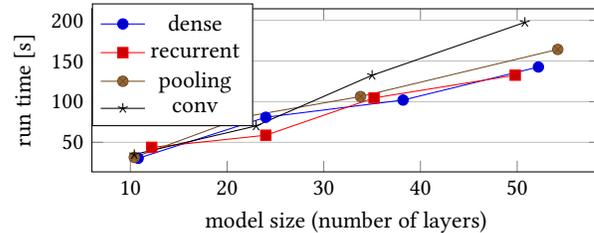**Figure 1: Run time of our repair algorithm.**

Figure 2: Run time of our repair algorithm for graph models.

recurrent layers, (3) pooling layers, and (4) convolutional layers. Fig. 2 shows the same, but also includes graph models that include forks as a result of mathematical layers with multiple inputs. It can be seen that even relatively large models can be repaired in a reasonable time. Sequential models with up to 50 layers can be repaired in 80s, even if they contain mostly complex layers like convolutional or pooling. Graph models with about 50 layers take longer, but can still be repaired in max 3min 40s. We believe that compared to the manual effort of inspecting and repairing such a large model, our repairing time is reasonably practical.

For the second experiment, we applied three models for the well-known CIFAR [34] and MNIST datasets [11] that were provided by SOCRATES [41]. The models were called `cifar_conv_small_relu`, `mnist_conv_small_relu_diffai` and `mnist_conv_9_200`. Additionally, we used a model [5] for the Fashion-MNIST dataset [59] and another model [23] for the Street View House Numbers (SVHN) dataset [37]. In contrast to the previously generated models, these neural networks had input data of up to 64KB and a model size of up to 10MB due to bigger weight size. In order to evaluate our approach for these models, we manually added inconsistent arguments, dimension errors, and weight shape issues. On average, it took 3min 12s to repair these models, and the maximum repair time was 5min 54s. This demonstrates that we are also able to apply our approach to practical neural networks and not just randomly generated ones. Moreover, it shows that the repair time of these larger neural networks was still reasonable, especially since the experiments were only performed on a laptop with rather limited computing resources and without parallelisation.

Discussion. A potential threat to the validity of our evaluation might be that we focused too much on models with small input sizes. Generally, AI models can handle huge data like voice recordings, or large images. An evaluation with larger test inputs and models might be more realistic, but a large input size (in the MB range)

Table 3: Overview of the real model bugs and their types and model sizes.

	References	Number of Bugs	Min Model Size	Max Model Size	Min Repair Time[s]	Max Repair Time[s]	Avg. Repair Time[s]
Dimension Error	[4, 6, 9, 16, 17]	5	4KB	10MB	5	301	69
Pool/Kernel Shape Bug	[12, 20, 21]	3	35KB	6MB	5	501	181
Weight Shape Issue	[7, 10, 13, 18, 19]	5	1KB	43MB	5	1402	388
Input Shape Bug	[8, 14]	2	2MB	5MB	238	254	246
Argument Error	[15]	1	186KB	186KB	217	217	217
Total	-	16	1KB	43MB	5	1402	220

Table 4: Comparison of the repairs of the real model bugs to the solutions from stackoverflow.

	Equivalent or Closely Related	Comparable Quality	Not Comparable but Still Valid	Total
Dimension Error	4	1	-	5
Pool/Kernel Shape Bug	-	1	2	3
Weight Shape Issue	4	1	-	5
Input Shape Bug	-	2	-	2
Argument Error	-	-	1	1

will produce even bigger models, which would soon cause memory overflows in the semantics. However, usually bugs in large models can be broken down to smaller version that are easier to process. We believe that our test models with rather small inputs were still reasonable and did not represent a big limitation, and it is well-known that small test cases can reveal various bugs [33]. Moreover, we tested reasonably sized models from several benchmark datasets, and we evaluated real model bugs.

One might argue that the performance of our approach for larger models is limited, since it can take a couple of minutes to execute our approach for large models. We believe that the performance is still reasonable especially since it can avoid a lot of effort that would be required to manually inspect a model for bugs, or when the long resolution time for AI bug reports is considered that can range from weeks to months [53].

Another threat to the validity of our evaluation might be a potential bias when we selected the real model bugs. It is true that we had quite restricted criteria when we were looking for these bugs. There are numerous bug reports from AI developers that are outside the scope of this work since they are not related to the misuse or wrong composition of standard layers. Moreover, there are still many bugs that come from AI models that include non-standard features, like custom layers, which make them unusable with our approach. Hence, it was a cumbersome task to look for relevant bugs for our approach. We believe that we still managed to find a representative set of model bugs that was suited for a good evaluation of our method. There might be rare bugs in the same category that are not supported by our method, but our evaluation showed that we are able to repair common modelling bugs in a reasonable time.

Another question that might come up is why we do not use error messages from an AI framework instead of utilizing a semantics to support our repair approach? It is true that our algorithms would in principle also work with error messages from AI frameworks and

it would even be faster, but there are a number of problems with these messages [45]. The error messages are not always consistent, i.e., even for the same type of bug there can be various different messages, which might be caused by independent implementations in different layers. The source of an error is not always clear and the necessary debug information to repair an error can be hard to extract since the messages have no clear and consistent form. In contrast, ExAIS provides error messages that are well-structured and consistent. They can easily be automatically parsed and provide clear debug information and the source of a bug. Moreover, it is easy to extend ExAIS with additional preconditions, which can be helpful for checking custom model properties or it enables the identification of problematic behaviour that might not lead to error messages in an AI framework. Hence, we believe that it was a good choice to apply ExAIS for our repair approach.

5 RELATED WORK

Most of the related work in neural network repair focuses on other aspects of the neural networks, like improving the accuracy or ensuring certain properties.

For example, Sohn et al. [47] introduce a repair technique called Arachne that can improve pre-trained models by adjusting the weights of layers. The method applies differential evolution to optimise the weights and to correct misbehaviour, like misclassifications. Moreover, they demonstrate how to resolve fairness issues, i.e., by repairing a bias in a gender classifier.

Another approach that deals with fairness properties was presented by Sun et al. [52]. The work illustrates a causality-based repair technique called CARE that can identify problematic neurons that are responsible for undesired neural network behaviour. It is able to ensure that a neural network satisfies various fairness and safety properties, e.g., it can remove backdoors caused by malicious training data.

Yang et al. [61] shows a repair framework that is able to ensure safety and robustness with regard to input-output safety specifications. They illustrate a depth-first-search reachability analysis algorithm to find unsafe input regions and examples that represent these regions. The approach is evaluated with an aircraft collision avoidance and a rocket landing system. Similarly, Sotoudeh and Thakur [48] present a provable point repair algorithm that is able to deal with misclassifications and that can ensure safety properties.

Usman et al. [54] illustrates a constraint-based repair method called NNREPAIR for neural network classifiers. The technique applies fault localization to find problematic network parameters, and it can improve the accuracy of a model and fix safety properties.

Xie et al. [60] introduced a model-based repair approach for recurrent neural networks (RNN) called RNNRepair. It is based on an influence model that relates the behaviour of a network to the training data. The method can help to understand the behaviours of an RNN, as well as increase the accuracy and repair safety properties.

Zhang et al. [63] introduced a DNN training monitoring and automatic repairing tool called AUTOTRAINER that can fix common training problems, like a slow convergence or fluctuating accuracies. Similarly, a tool called DeepDiagnosis that further improved the repair performance for such bugs, was introduced by Wardat et al. [56] It applies dynamic analysis to monitoring and detect errors according to various symptoms. It can fix eight different training problems and can do this more efficient and with a better performance than other tools. In contrast to both these methods, our approach is only concerned with the model and layer definitions and not with the training phase.

Related work is also in the field of AI model debugging which includes a number of approaches and tools [30, 36, 39, 44, 51] that offer features like fuzzing, faulty neuron (or feature) localisation, visualisation, or auditing, to enable finding bugs that lead to inaccuracy or property violations. However, such approaches are usually limited to specific types of models, e.g., classifiers.

In contrast to all these approaches, our work is not concerned with the accuracy or with certain properties about the predictions of a neural network. We focus on repairing modelling bugs that cause invalid neural networks with structural problems, wrongly used layers, or inconsistent layer arguments.

The closest related work is the tool called Tensfa from Wu et al. [57, 58]. The tool can automatically detect and repair tensor shape faults, which are comparable to our dimension and input shape bugs. For these bugs, the approach works for even broader scope of AI models and programs since it supports more general operations with Tensors, like array adjustments with NumPy [28], which are not standard layers. The approach uses decision tree model to detect bugs based on crash messages from TensorFlow. It applies static data dependence analysis and a dynamic shape tracking techniques to locate faults, and multiple mutation strategies that perform repairs by considering the frequency and time cost of a number of repair patterns. Some of the repair patterns, like the introduction of reshape layers, are similar to our fixes, but since Tensfa uses more operations than just standard layers, it has more repair options for fixing shape issues. Additionally, Tensfa can check if the input and output shapes of a model are causing a bug, which is out of the scope of our method.

In contrast, our repair method with ExAIS works more straight forward. To identify and localize faults, we just run a model with ExAIS and the precondition checks will identify and localize the bugs. Moreover, our approach is not just limited to Tensor shape bugs, since we support other issues, like argument or weight shape errors, and the focus of our repairs is on finding minimal changes with standard layers. Lastly, Tensfa can only fix issues when there is an error message. ExAIS can also identify rare bugs that do not produce error messages from TensorFlow [1].

To the best of our knowledge, our work is the first that shows an automatic semantic-based repair approach for AI model bugs.

6 CONCLUSION

We have introduced a novel neural network repair approach that is based on an executable semantics and demonstrated two applications. Our method is able to repair invalid AI models that suffer from structural problems, wrongly used layers, or wrongly connected layers. It works by executing a given AI model with an existing AI framework semantics called ExAIS that has built-in preconditions that are able to provide error messages with debug information that helps to localise and characterise a bug. Based on these error messages that are produced when there is a precondition violation, we are able to identify and repair invalid model aspects with a number of algorithms for different types of bugs.

One major application of our approach is the repair of automatically generated neural networks that can, e.g, be used for AI framework testing. In order to evaluate the effectiveness of our approach for this application, we generated 1,000 random models with bugs and repaired them. Our approach was able to repair 100% of these neural networks, and it took on average only 21.08s to completely repair the models. Moreover, we evaluated the approach by repairing a set of larger test models (for well-known benchmark datasets) which required about 192s on average.

Another application that we presented is the repair of real modelling bugs from AI developers. For this use case, we collected a set of neural network bugs that developers were struggling with from stackoverflow. Out of 16 faulty AI models, we could directly repair 15 models, and also the one remaining model could be repaired after a minor size adjustment of the model. The average repair time for the bugs was only 3min 40s. Inspecting the quality of our repairs, showed that 13 of our produced repairs were equally as good or were comparable to the solutions on stackoverflow.

To sum up, our approach was able to effectively and automatically produce practical repairs for real world bugs within minutes. This shows that it can be valuable for AI developers since it can reduce a lot of debugging effort.

We believe that these two approaches highlight the usefulness and applicability of our approach, which has potential to enable further applications. In the future, we aim to explore semantic-based repair techniques for other AI model aspects.

REFERENCES

[1] 2021. *cropping layer additional error message.* https://github.com/tensorflow/tensorflow/issues/50612
[2] 2021. *Dense Layer.* https://tensorflow.org/api_docs/python/tf/keras/layers/Dense
[3] 2021. *TensorFlow layer documentation.* https://www.tensorflow.org/versions/r2.4/api_docs/python/tf/keras/layers
[4] 2022. *1D Convolution Layer dimension mismatch issue.* https://stackoverflow.com/questions/69305003/1d-convolution-layer-dimension-mismatch-issue
[5] 2022. *Fashion Classification Using CNN.* https://github.com/heisenbuug/Fashion-Classification-Using-CNN
[6] 2022. *How to resize input size for Conv1DTranspose layer in Keras?* https://stackoverflow.com/questions/64312226/how-to-resize-input-size-for-conv1dtranspose-layer-in-keras
[7] 2022. *Incomparable weight shape between caffe and tensorflow / keras.* https://stackoverflow.com/questions/71054511/incomparable-weight-shape-between-caffe-and-tensorflow-keras
[8] 2022. *Keras Maxpooling2d layer gives ValueError.* https://stackoverflow.com/questions/39815518/keras-maxpooling2d-layer-gives-valueerror
[9] 2022. *Keras ValueError: Input 0 is incompatible with layer conv2d_2.* https://stackoverflow.com/questions/45329416/keras-valueerror-input-0-is-incompatible-with-layer-conv2d-2
[10] 2022. *Layer Weight Shape (6,6) Not Compatible With Provided Weight Shape (6,).* https://stackoverflow.com/questions/72079459/layer-weight-shape-6-6-not-compatible-with-provided-weight-shape-6
[11] 2022. *MNIST Adversarial Examples Challenge.* https://github.com/MadryLab/mnist_challenge
[12] 2022. *Negative dimension size caused by subtracting 3 from 1 for conv2d_2/convolution.* https://stackoverflow.com/questions/45645276/negative-dimension-size-caused-by-subtracting-3-from-1-for-conv2d-2-convolution
[13] 2022. *Tensorflow Keras Embedding Layer Error: Layer weight shape not compatible.* https://stackoverflow.com/questions/58548520/tensorflow-keras-embedding-layer-error-layer-weight-shape-not-compatible
[14] 2022. *Tensorflow ValueError: Operands could not be broadcast together with shapes (5, 5, 160) (19, 19, 80).* https://stackoverflow.com/questions/60575399/tensorflow-valueerror-operands-could-not-be-broadcast-together-with-shapes-5
[15] 2022. *TypeError and ValueError while training model with keras.* https://stackoverflow.com/questions/56624408/typeerror-and-valueerror-while-training-model-with-keras
[16] 2022. *ValueError: Input 0 of layer lstm is incompatible with the layer: expected ndim=3, found ndim=2 in a LSTM model.* https://stackoverflow.com/questions/71734426/valueerror-input-0-of-layer-lstm-is-incompatible-with-the-layer-expected-ndi
[17] 2022. *ValueError: Input 0 of layer lstm_cell_1 is incompatible with the layer: expected ndim=3, found ndim=2.* https://stackoverflow.com/questions/70611262/valueerror-input-0-of-layer-lstm-cell-1-is-incompatible-with-the-layer-expec
[18] 2022. *ValueError: Layer weight shape (3, 3, 3, 64) not compatible with provided weight shape (64, 3, 3, 3).* https://stackoverflow.com/questions/48283625/valueerror-layer-weight-shape-3-3-3-64-not-compatible-with-provided-weight
[19] 2022. *ValueError: Layer weight shape (43, 100) not compatible with provided weight shape (412457, 400).* https://stackoverflow.com/questions/59840678/valueerror-layer-weight-shape-43-100-not-compatible-with-provided-weight-sha
[20] 2022. *ValueError: Negative dimension size caused by subtracting 22 from 1 for conv3d_3/convolution (op: Conv3D).* https://stackoverflow.com/questions/60959631/valueerror-negative-dimension-size-caused-by-subtracting-22-from-1-for-conv3d
[21] 2022. *ValueError: Negative dimension size caused by subtracting 5 from 4.* https://stackoverflow.com/questions/61868973/valueerror-negative-dimension-size-caused-by-subtracting-5-from-4
[22] 2023. *ExAIS: Executable AI Semantics Repository with AI Framework Testing and AI Model Repair Tools.* https://github.com/rschumi0/ExAIS
[23] Laxmi Chaudhary. 2022. *SVHN Deep Neural Network Image Classification.* https://github.com/LaxmiChaudhary/SVHN-Deep-Neural-Network
[24] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. 2017. Validating a Deep Learning Framework by Metamorphic Testing. In *2nd IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2017, Buenos Aires, Argentina, May 22, 2017.* IEEE Computer Society, 28–34. https://doi.org/10.1109/MET.2017.2
[25] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 118–128. https://doi.org/10.1145/3213846.3213858
[26] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *J. Mach. Learn. Res.* 20 (2019), 55:1–55:21. http://jmlr.org/papers/v20/18-598.html

[27] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2010. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Empirical Software Engineering and Verification - International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures (Lecture Notes in Computer Science, Vol. 7007)*, Bertrand Meyer and Martin Nordio (Eds.). Springer, 1–59. https://doi.org/10.1007/978-3-642-25231-0_1
[28] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nat.* 585 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2
[29] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowl. Based Syst.* 212 (2021), 106622. https://doi.org/10.1016/j.knosys.2020.106622
[30] Fred Hohman, Minsuk Kahng, Robert Pienta, and Duen Horng Chau. 2019. Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers. *IEEE Trans. Vis. Comput. Graph.* 25, 8 (2019), 2674–2693. https://doi.org/10.1109/TVCG.2018.2843369
[31] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1110–1121. https://doi.org/10.1145/3377811.3380395
[32] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: fix patterns and challenges. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1135–1146. https://doi.org/10.1145/3377811.3380378
[33] Daniel Jackson and Craig Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996.* ACM, 239–249. https://doi.org/10.1145/229000.226322
[34] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
[35] Yitong Li. 2020. *Documentation-Guided Fuzzing for Testing Deep Learning API Functions.* Master's thesis. University of Waterloo.
[36] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 175–186. https://doi.org/10.1145/3236024.3236082
[37] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading digits in natural images with unsupervised feature learning. (2011).
[38] Ulf Nilsson and Jan Małuszyński. 1990. *Logic, programming and Prolog.* Wiley Chichester.
[39] Augustus Odena, Catherine Olsson, David G. Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4901–4911. http://proceedings.mlr.press/v97/odena19a.html
[40] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107
[41] Long H. Pham, Jiaying Li, and Jun Sun. 2020. SOCRATES: Towards a Unified Platform for Neural Network Verification. *CoRR* abs/2007.11206 (2020). arXiv:2007.11206 https://arxiv.org/abs/2007.11206
[42] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria E. Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. 2019. A Survey on Deep Learning: Algorithms, Techniques, and Applications. *ACM Comput. Surv.* 51, 5 (2019), 92:1–92:36. https://doi.org/10.1145/3234150
[43] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117. https://doi.org/10.1016/j.neunet.2014.09.003
[44] Eldon Schoop, Forrest Huang, and Bjoern Hartmann. 2021. UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*, Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, Takeo Igarashi, Pernille Bjørn, and Steven Mark Drucker (Eds.). ACM, 310:1–310:16. https://doi.org/10.1145/3411764.3445538

[45] Richard Schumi and Jun Sun. 2022. ExAIS: Executable AI Semantics. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 859–870. https://doi.org/10.1145/3510003.3510112

[46] Arnab Sharma and Heike Wehrheim. 2019. Testing Machine Learning Algorithms for Balanced Data Usage. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 125–135. https://doi.org/10.1109/ICST.2019.00022

[47] Jeongju Sohn, Sungmin Kang, and Shin Yoo. 2019. Search Based Repair of Deep Neural Networks. *CoRR* abs/1912.12463 (2019). arXiv:1912.12463 http://arxiv.org/abs/1912.12463

[48] Matthew Sotoudeh and Aditya V. Thakur. 2021. Provable repair of deep neural networks. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 588–603. https://doi.org/10.1145/3453483.3454064

[49] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-Implementation Testing of Supervised Learning Software. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018 (AAAI Workshops, Vol. WS-18)*. AAAI Press, 384–391. https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/17301

[50] Eike Stein, Steffen Herbold, Fabian Trautsch, and Jens Grabowski. 2021. A new perspective on the competent programmer hypothesis through the reproduction of bugs with repeated mutations. *CoRR* abs/2104.02517 (2021). arXiv:2104.02517 https://arxiv.org/abs/2104.02517

[51] Hendrik Strobelt, Sebastian Gehrmann, Michael Behrisch, Adam Perer, Hanspeter Pfister, and Alexander M. Rush. 2019. Seq2seq-Vis: A Visual Debugging Tool for Sequence-to-Sequence Models. *IEEE Trans. Vis. Comput. Graph.* 25, 1 (2019), 353–363. https://doi.org/10.1109/TVCG.2018.2865044

[52] Bing Sun, Jun Sun, Hong Long Pham, and Jie Shi. 2022. Causality-based Neural Network Repair. In *44th International Conference on Software Engineering (ICSE 2022)*. IEEE.

[53] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*. IEEE Computer Society, 271–280. https://doi.org/10.1109/ISSRE.2012.22

[54] Muhammad Usman, Divya Gopinath, Youcheng Sun, Yannic Noller, and Corina S. Pasareanu. 2021. NNrepair: Constraint-Based Repair of Neural Network Classifiers. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 3–25. https://doi.org/10.1007/978-3-030-81685-8_1

[55] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 788–799. https://doi.org/10.1145/3368089.3409761

[56] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 561–572. https://doi.org/10.1145/3510003.3510071

[57] Dangwei Wu, Beijun Shen, Yuting Chen, He Jiang, and Lei Qiao. 2021. Tensfa: Detecting and Repairing Tensor Shape Faults in Deep Learning Systems. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 11–21. https://doi.org/10.1109/ISSRE52982.2021.00014

[58] Dangwei Wu, Beijun Shen, Yuting Chen, He Jiang, and Lei Qiao. 2022. Automatically repairing tensor shape faults in deep learning programs. *Information and Software Technology* 151 (2022), 107027. https://doi.org/10.1016/j.infsof.2022.107027

[59] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017). arXiv:1708.07747 http://arxiv.org/abs/1708.07747

[60] Xiaofei Xie, Wenbo Guo, Lei Ma, Wei Le, Jian Wang, Lingjun Zhou, Yang Liu, and Xinyu Xing. 2021. RNNRepair: Automatic RNN Repair via Model-based Analysis. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 11383–11392. http://proceedings.mlr.press/v139/xie21b.html

[61] Xiaodong Yang, Tom Yamaguchi, Hoang-Dung Tran, Bardh Hoxha, Taylor T. Johnson, and Danil V. Prokhorov. 2022. Neural Network Repair with Reachability Analysis. In *Formal Modeling and Analysis of Timed Systems - 20th International Conference, FORMATS 2022, Warsaw, Poland, September 13-15, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13465)*, Sergiy Bogomolov and David Parker (Eds.). Springer, 221–236. https://doi.org/10.1007/978-3-031-15839-1_13

[62] Quanming Yao, Mengshuo Wang, Hugo Jair Escalante, Isabelle Guyon, Yi-Qi Hu, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. 2018. Taking Human out of Learning Applications: A Survey on Automated Machine Learning. *CoRR* abs/1810.13306 (2018). http://arxiv.org/abs/1810.13306

[63] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 359–371. https://doi.org/10.1109/ICSE43902.2021.00043

[64] 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, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 129–140. https://doi.org/10.1145/3213846.3213866

[65] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=r1Ue8Hcxg