NNrepair: Constraint-based Repair of Neural Network Classifiers

Muhammad Usman[1], Divya Gopinath[2], Youcheng Sun[3], Yannic Noller[4], and Corina S. Păsăreanu[2]

[1] University of Texas at Austin, USA
muhammadusman@utexas.edu
[2] KBR Inc., Nasa Ames
{divya.gopinath,corina.s.pasareanu}@nasa.gov
[3] Queen's University Belfast, UK
youcheng.sun@qub.ac.uk
[4] National University of Singapore
yannic.noller@acm.org

Abstract. We present NNrepair, a constraint-based technique for repairing neural network classifiers. The technique aims to fix the logic of the network at an *intermediate layer* or at the *last layer*. NNrepair first uses *fault localization* to find potentially faulty network parameters (such as the *weights*) and then performs *repair* using *constraint solving* to apply small modifications to the parameters to remedy the defects. We present novel strategies to enable precise yet efficient repair such as inferring correctness specifications to act as oracles for intermediate layer repair, and generation of *experts* for each class. We demonstrate the technique in the context of three different scenarios: (1) Improving the *overall accuracy* of a model, (2) Fixing security vulnerabilities caused by *poisoning* of training data and (3) Improving the *robustness* of the network against *adversarial* attacks. Our evaluation on MNIST and CIFAR-10 models shows that NNrepair can improve the accuracy by 45.56 percentage points on poisoned data and 10.40 percentage points on adversarial data. NNrepair also provides small improvement in the overall accuracy of models, without requiring new data or re-training.

1 Introduction

Neural networks have many applications, being used for example in pattern analysis, image classification, or sentiment analysis for textual data, and also in medical diagnosis or perception and control in autonomous driving, which bring safety and security concerns [10]. These systems learn the network parameters (weights and biases) through *training* on a set of labeled examples. The performance of the trained networks is independently validated by computing the *accuracy* on a held-out labeled test set.

Just like other software systems, trained neural networks can have *defects* that need *repair*. For example, a trained neural network may have low accuracy

which may be due to limited training data. One would like to repair the network by modifying its parameters (or a subset of them) to improve its overall accuracy, even in the absence of additional training data. In another scenario, the training data for a neural network has been *poisoned* by an adversary leading to high accuracy on normal data but poor accuracy on poisoned data [7,6,11]. In this case, one would like to repair the network to remedy the defect while still maintaining a high accuracy on non-poisoned data. In yet another scenario, a trained network may have high accuracy on the test set but may be vulnerable to adversarial perturbations, i.e., small modifications to the inputs that lead to unexpected outputs. Recent studies [20,15,8] show that this defect is very common even for highly trained, highly accurate networks. In this case, one would like to repair the network to make it *robust* against adversarial perturbations while at the same time retaining its accuracy on the normal, unperturbed test set.

Retraining could be used to alter the neural network parameters and repair for faults, but it can be very difficult and expensive subject to uncertainties, and may result in a network that is quite different from the original one, thus wasting the effort of the original training.

We present a novel constraint-solving based approach, NNREPAIR, to repair neural networks trained for the task of classification, with respect to all three scenarios described above. Similar to traditional program repair [22,5,13], NNREPAIR first uses *fault localization* to identify the network parameters that are the likely source of defects, followed by *repair*, which uses *constraint solving* to apply small modifications to the network parameters to remedy the defects.

Given a trained neural network model, the potentially faulty components could be the architecture of the model (which is fixed in the design stage) or the learn-able parameters such as the weights and the biases (which are determined during training). In this work, we focus on the learn-able parameters of a neural network model, specifically the weights on the edges connecting neurons. As observed in [9], changing the weights is a common fix for neural networks.

We leverage the organization of a neural network into layers and the natural decomposition of computation that each layer provides, and scope our work to focus the repair on a single layer of the network. Repairs across multiple layers are possible, but they would be less scalable and involve more complex modifications. We propose two types of repairs: *intermediate-layer repair* and *last-layer repair*. Intermediate-layer repair attempts to fix failures by modifying the behavior of neurons at an inner layer of the network. Last-layer repair, on the other hand, attempts to modify the decision constraints at the last layer.

Fault localization is used to mark one or more neurons at a layer as 'suspicious' and to find a sub-set of incoming edges to the suspicious neurons, whose weights will be the target for repair. The repair process involves solving constraints collected from the network, via a simple form of concolic execution [17]. For last-layer repair, the oracle of the repair is the desired label for every failing input and the repair constraints encode this decision. For intermediate-layer repair, we propose a novel use of activation patterns representing specifications

of correct behavior at the layer [4] as oracles for repair. This enables us to keep the repair local to the layer and therefore efficient.

Furthermore, to make the *constraint solving* scalable, instead of solving for constraints for all classes at once, we propose to *decompose* the repair into a set of sub-tasks, one for each output class. Specifically, we set-up the constraint solving to correct a subset of the weights with the goal of improving accuracy of the model wrt a specific output class. The result of this repair is a set of *experts*, which are neural networks that improve accuracy of the network wrt specific output classes. We then combine the experts to obtain the final repaired model.

There are a few recent related techniques that propose to use constraint solving for neural network repair. We summarize them in Section 6. These techniques tend to focus on last layer repair while we also propose repair at an intermediate layer. Furthermore, we evaluate our initial prototype in three scenarios: improving accuracy, robustness and resilience towards poisoned data. None of the related techniques address all three (albeit potentially possible).

We summarize our contributions as follows.

- We propose and implement a repair technique that applies fault localization and constraint solving to neural networks. Our approach can perform both *last* and *intermediate* layer repair.
- To achieve scalability, our approach decomposes the repair into a set of *experts* which display superior accuracy for specific labels. These are then combined using a set of *strategies* that we propose to obtain the final repair.
- We present a novel technique to make it more efficient to repair inner layers of a neural network by *inferring specifications of correct behavior* (in the terms of the activation patterns) at the output of inner layers, and using them as oracles for repair.
- While previous neural network repair techniques (see Section 6) tend to focus solely on improving accuracy, we demonstrate our technique in the context of three different scenarios: (1) Improving the overall accuracy of a model, (2) Fixing security vulnerabilities caused by *poisoning* of training data and (3) Improving the *robustness* of the network against *adversarial* attacks.
- We evaluate the techniques in the context of image classifiers for the MNIST and CIFAR-10 data sets. The results indicate that NNrepair can improve the performance of the network by 45.56 percentage points on poisoned data and 10.40 percentage points on adversarial data. NNrepair also provides small improvement (+0.20 percentage points), in the overall accuracy of models, without requiring new data or re-training.

2 Background

Neural Networks In this work we focus on neural network classifiers. These networks take in an input, such as an image, and output a class (or label) specific to the problem they have been trained to solve. Networks are organized in *layers* of different types, including convolutional, activation, and pooling, each of which has a number of nodes. For this paper, we focus on activation layers.

Each node from the previous layer will output into the associated node in the activation layer, which will apply an *activation function.* Common activation functions include linear rectification (a.k.a. ReLU) and sigmoid. For simplicity we discuss here ReLU activations but our work applies to arbitrary activations as discussed below. Let $N(X)$ denote the value of a neuron as a function of the input. $N(X) = \sum_i w_i \cdot N_i(X) + b$ where N_i's denote the values of the neurons in the previous layer of the network and the coefficients w_i and the constant b are referred to as *weights* and *bias*, respectively. If this function evaluates to a non-negative value, the node is *activated* and outputs that value, otherwise it outputs 0. A final decision (logits) layer produces the network decisions based on the real values computed by the network, by applying e.g., a softmax function; in our work we use the max function instead. For a comprehensive introduction to neural networks, see [3].

Activation Patterns We leverage previous work [4] to infer network properties based on the *activation patterns* of neurons in the network. We will use these activation patterns as oracles for the intermediate layer repair. An activation pattern σ specifies an activation status (*on* or *off*) for some subset of neurons at a layer in the network. All other neurons do not matter. We write $on(\sigma)$ for the set of neurons marked *on*, and $off(\sigma)$ for the set of neurons marked *off* in the pattern σ. Each activation pattern σ defines a predicate $\sigma(X)$ that is satisfied by all inputs X whose evaluation achieves the same activation status for all neurons as prescribed by the pattern.

$$\sigma(X) ::= \bigwedge_{N \in on(\sigma)} N(X) > 0 \ \wedge \bigwedge_{N \in off(\sigma)} N(X) \leq 0 \tag{1}$$

A decision pattern σ is a property wrt network F and postcondition P if:

$$\forall X : \sigma(X) \Rightarrow P(F(X)). \tag{2}$$

A postcondition for a classification network is that the top predicted class is C, i.e., $P(Y) := argmax(Y) = C$.

The previous work [4] also describes how to compute activation patterns. The idea is to observe the activation signatures of a large number of inputs and apply decision tree learning over them to infer activation patterns that are thus empirically valid. We adopt the same approach here. The *support* of a pattern is formed by all the inputs that satisfy the pattern. We are interested in computing high-support patterns as they are the most likely to reflect valid properties of the network.

3 Example

This section demonstrates *Intermediate-layer* and *Last-layer* repair on a simple example. Figure 1 shows a simple two-input network with two hidden layers; each containing two ReLU nodes (ReLU(x) = x (on) if x > 0, 0 (off) otherwise), and

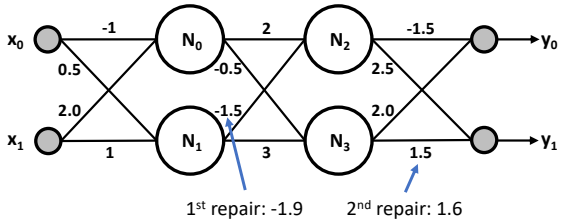

Fig. 1. Example

Table 1. Data for Example

	x_0	x_1	N_0	N_1	N_2	N_3	y_0	y_1	class	ideal
X_0	1	1	1	1	0	1	8	6	0	0
X_1	0	1	1	1	1	1	0.25	9.25	1	1
X_2	1	0	0	1	0	1	3	2.25	0	0
X_3	-1	1	1	1	1	0	-7.87	13.12	1	1
X_4	1.5	2	1	1	1	1	12.68	12.68	1	0
after repair:	1.5	2	1	1	0	1	13.3	10.5	0	0
X_5	0.6	1	1	1	1	1	5.91	5.62	0	1
after repair:	0.6	1	1	1	1	1	5.91	5.95	1	1

two outputs, y_0 and y_1. The weights are depicted on the edges between nodes. For simplicity we assume biases are 0. The input X, which is a two-element array denoted $[x_0, x_1]$, is assigned class 0 if $y_0 > y_1$ and 1 otherwise. Let us assume the model behaves correctly on the first four inputs shown in Table 1. The table also shows the decisions of the ReLU activations for nodes N_0, N_1, N_2, N_3, respectively. Whenever a ReLU node is on, the decision is indicated as a 1 and if it is off, then the decision value is shown as 0.

Consider now the input $X_4 = [1.5, 2.0]$. Assume this input is mis-classified; the output class is 1 but the ideal class is 0. The inaccuracy of the model could be a result of insufficient training. We then aim to build a repair, which in our case focuses on a single layer of the network and modifies the weights feeding into the neurons at that layer.

We keep the repair local to the layer by using activation patterns [4] in lieu of the decision constraints, The insight in [4] is that the logic that every layer implements could be captured as rules in terms of the activation patterns of the neurons. We can observe in the example, that for all inputs correctly classified with label 0, the neuron pair (N_2, N_3) in the second layer has the activation pattern (off, on). For the failing input, this pattern is not satisfied; in fact the activation for (N_2, N_3) for the failing input is (on, on). We use the above observation to fix the failure by performing *intermediate layer* repair. We aim to modify the neuron activations of the second layer on the failing input to satisfy the correct-label pattern for class 0 at the layer.

We aim to perform the repair by making minimal changes to the model. We identify the weights to be modified using an attribution-based approach and use

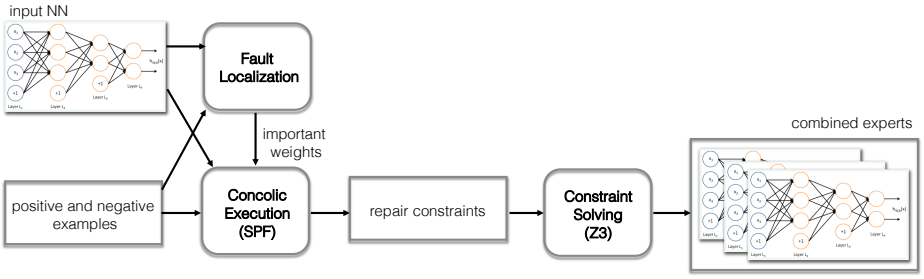

Fig. 2. Overview of the Approach

constraint solving to compute the values of the new weights (see Section 4 for details). Changing the weight of a single edge, connecting N_1 and N_2 from -1.5 to -1.9 changes the activation pattern for (N_2, N_3) to (off, on) on the failing input, while preserving the behavior of the neurons (in terms of their activation pattern) and the output of the model on the passing inputs.

Consider now another input for the above-corrected network, $X_5 = [0.6, 1.0]$. This input is very close to $X_1 = [0.0, 1.0]$ (correctly classified to 1) with a small change to x_0 that makes the model mis-classify the input to 0. This represents a typical adversarial scenario where a correctly classified input is perturbed slightly to create an input that 'jumps' the decision boundary of the network leading to a mis-classification. It can be observed that the activation patterns of the internal layer neurons for X_5 are the same as for the correctly classified input X_1, thus an intermediate-layer repair would not work for this input. Therefore we perform *last-layer repair*. We localize the weights of the edges in the last layer that need repair. Changing the weight on the edge between N_3 and y_1 (from 1.5 to 1.6) corrects the class for the failing test to 1, while retaining the same labels for the other inputs.

4 Approach

Figure 2 gives an overview of our approach. We aim to repair a faulty trained neural network classifier, which is given as input. As in other repair approaches, we consider both positive and negative examples for the repair. The negative examples are used to guide the repair towards correcting the faults while the positive examples are used to constrain the repair to not damage the existing good functionality of the network. We aim for a repair strategy that is scalable and applies small changes to the network. We therefore target the repair on a single layer of the network. Repairs across multiple layers of the network are possible, but they would be less scalable and involve more complex modifications.

Unlike all previous work, which tends to focus the repair at the last layer (see Section 6) we propose here techniques for both intermediate and last layer repair. Intuitively, a last layer repair is easier as it aims to modify the weights that impact directly the decisions, and can use the network's output as an *oracle*

to guide the repair. However the resulting repair may not generalize well and furthermore the network may be faulty at some intermediate layer. A repair at an intermediate layer can have a higher impact over the network's behavior but it is more difficult as it is not clear what *oracle* to use to guide the repair. One can use the output of the network as the oracle but this may result in an un-manageable large number of constraints to solve. In this work we propose a novel use of neuron activation patterns to act as oracles in intermediate layer repair.

As repairing for all the output classes at the same time can be very difficult, our proposed approach obtains instead a set of *expert networks*, one for each target class, which are easier to compute. These experts are combined to obtain a final repaired classifier. Specifically, our repair strategy has the following steps:

1 *Fault Localization*: The goal of this step is to identify a small set of suspicious neurons and incoming suspicious edges, whose weights we aim to correct.
2 *Concolic Execution*: For the weights of the suspicious edges, we add δ values that are set to 0 in concrete mode, but are designated as symbolic for the symbolic mode. The network is executed concolically along positive and negative examples, to collect the values of suspicious neurons in terms of symbolic expressions.
3 *Constraint Solving*: The symbolic expressions are assembled with a set of repair constraints which are then solved with an off-the-shelf solver. Essentially, the repair constraints need to encode the network decision for the positive examples and modify (i.e., correct) the network decision for the negative examples. For the last layer repair this amounts to adding constraints imposed by the decision layer. For intermediate-layer repair, we use activation patterns instead of decision constraints, allowing us to keep the repair local to the layer.
The solutions for the symbolic δ's obtained from the solver are used to update the weights of the network, thus obtaining an *expert* for a specific class.
4 *Combining Experts*: Finally the experts obtained for each class are combined to obtain the repaired classifier. This needs to be done carefully, to avoid redundant computations among experts and to not damage the overall accuracy and timing performance of the classification.

In the following we give more details about our approach.

4.1 Intermediate-layer repair

Fault localization We explore the usage of *activation patterns* of the network (Section 2) to act as oracles of correct behavior. We also use these patterns to guide the identification of potentially faulty neurons. Specifically, we use the decision-tree learning approach from [4] to extract *correct-label patterns* corresponding to every output class at an intermediate layer. Each pattern is satisfied by a group of inputs correctly classified to a certain label. Typically multiple correct-label patterns are generated. We select the ones with the highest support, which are mostly likely to hold true on the network for all inputs. Note

also that the work in [4] considers ReLU activations but it could be extended to consider arbitrary linear or non-linear activation functions, by comparing the values of neurons with a threshold.

A *correct-label* pattern with high support at a layer indicates that there is a high chance that any input satisfying the pattern at the layer would be classified by the network to the corresponding label. Furthermore, a mis-classified input will not satisfy the *correct-label* pattern for the respective ideal label. For every failing input, we compare the activations of the neurons with those in the respective *correct-label* pattern and consider those *neurons* whose activations differ as the potentially faulty ones. The repair then aims to change the outputs of the neurons for each of the failing inputs, such that they satisfy the correct-label pattern for their ideal labels.

In this work, we select a dense layer (i.e., a fully connected layer which receives input from every neuron in the previous layer) with ReLU activations. Typically such dense layers appear closer to the output and may impact the classification decision more than convolutional layers which process the input. Further, the number of neurons at fully connected layers is typically smaller than at other layers making the pattern-extraction process efficient.

Consider a mis-classified input, X_f with ideal label C. Let σ_C be the *correct-label* pattern with highest support for C. Let L be the layer for this pattern, and let N denote a neuron at layer L. Then the set of suspicious or faulty neurons $\mathcal{N}_{faulty}$ can be defined as follows;

$$N \in \mathcal{N}_{faulty} \iff (N \in on(\sigma_C) \wedge N(X_f) \leq 0) \vee (N \in \mathit{off}(\sigma_C) \wedge N(X_f) > 0) \quad (3)$$

Once the neurons whose outputs need to change are identified, we also need to identify the incoming edges to those neurons whose weights we aim to modify. We use a simple statistical method to identify the *important weights* which impact the respective neuron's output, more for the failing inputs as compared to the passing inputs.

Consider a set *Fail* of failing inputs with the same ideal label C and a set *Pass* of passing inputs. We use $\#(\cdot)$ to denote the cardinality of the sets. The defect score for each edge is determined as follows.

$$Score(E_i) ::= \frac{\sum_{X \in Fail} |N_i(X) \cdot w_i|}{\#Fail} - \frac{\sum_{X \in Pass} |N_i(X) \cdot w_i|}{\#Pass} \quad (4)$$

Here E_i denotes an incoming edge (for a faulty node N), N_i is the corresponding node in the preceding layer and w_i is the weight of the edge.

Thus, we take the average of the absolute values passing through the edge for all the negative examples for C and the average of the absolute values passing through the edge for all the positive examples and subtract them. The intuition is to identify the edges which have more influence on the incorrect decision of the network. We calculate the defect score for each incoming edge to each neuron (N) in $\mathcal{N}_{faulty}$. We then select the edges with top n% of the scores to create the set of faulty edges, for a small n.

Concolic Execution We perform a simplified form of concolic execution to form symbolic constraints for suspicious neurons. For the weights of the suspicious edges, we add δ values that are set to 0 in concrete mode, but are designated as symbolic in the symbolic mode. The network is executed concolically along both positive and negative examples, to collect the values of neurons as weighted sums in terms of both concrete values and the symbolic δ values. The value of a neuron is computed as constraints of the following form:

$$Sym_{N,X} = \sum_i (w_i + \delta_i) \cdot N_i(X) + \sum_j w_j \cdot N_j(X) + b \tag{5}$$

Here $Sym_{N,X}$ is a fresh symbolic variable introduced to encode the symbolic value of neuron N for input X, w_i's denote the weights of the suspicious edges (in the suspicious layer) while the w_j's denote the other weights, which do not need modification. Furthermore, $N_i(X)$, $N_j(X)$ represent the concrete values of the neurons coming from the previous layer. Note that no expensive constraint solving is needed in this step.

Repair Constraints and Constraint Solving For intermediate-layer repair, we add the activation patterns constraints (that imply the decision constraints, see Equation 1) to the set of constraints. Specifically, for each neuron N in $\mathcal{N}_{faulty}$, and for each (passing or failing) input X we add $Sym_{N,X} > 0$ if $N \in on(\sigma_C)$ and we add $Sym_{N,X} \leq 0$ if $N \in off(\sigma_C)$.

The solutions for the symbolic δ's obtained from the solver guarantee that all the inputs (both passing and failing) satisfy the pattern and are thus likely to be classified as C by the network. These solutions are then used to update the weights of the network, thus obtaining an *expert* for the class C.

Example Let us consider the example from Section 3, the case of the intermediate-layer repair. As already discussed in Section 3, let us suppose we consider the activation pattern for class 0 at layer 2. We select N_2 as the target for repair (since its activation along the failing test X_4 is on instead of off) and we want the input to satisfy the pattern {off, on} for $\{N_2, N_3\}$. We compute defect scores for the incoming edges to N_2 using the failing input and all passing inputs for classes 0 and 1. The score of the edge between N_0 and N_2 is 2.0 while the score of the edge between N_1 and N_2 is 2.81, we therefore select the second edge as a target for repair. We then build the following constraints from the failing test.
$Sym_{N_2,4} = 2.0 \cdot (-1.0 \cdot 1.5 + 2.0 \cdot 2.0) + (-1.5 + \delta) \cdot (0.5 \cdot 1.5 + 1.0 \cdot 2.0) \wedge Sym_{N_2,4} \leq 0.0$

Similarly, we build constraints from the passing tests that satisfy the pattern for label 0, X_0 and X_2:
$Sym_{N_2,0} = 2.0 \cdot (-1.0 \cdot 1.0 + 2.0 \cdot 1.0) + (-1.5 + \delta) \cdot (0.5 \cdot 1.0 + 1.0 \cdot 1.0) \wedge Sym_{N_2,0} \leq 0.0 \wedge$
$Sym_{N_2,2} = 2.0 \cdot (-1.0 \cdot 1.0 + 2.0 \cdot 0.0) + (-1.5 + \delta) \cdot (0.5 \cdot 1.0 + 1.0 \cdot 0.0) \wedge Sym_{N_2,2} \leq 0.0$

In practice we also add some constraints on δ to keep it small but we omit them here for simplicity. A solution for all the constraints is $\delta = -0.4$ which is used to update the weight for the target repair resulting in an expert for class 0.

4.2 Last-layer Repair

Fault localization In a classifier network the last layer typically contains as many neurons as the number of classes. An input is classified to label C, if the output of the respective neuron is greater than the values of all other output neurons. It is therefore natural to designate this neuron as suspicious for target class C. Let N_C denote the neuron at the last layer corresponding to a class C. We use the same technique as in intermediate layer repair (Equation 4) to *localize edges* and short-list the important weights which are the target for repair.

Concolic execution Similar to the intermediate layer repair, we add symbolic δ values to the important weights and perform concolic execution along passing and failing tests to create the symbolic expression for the node $Sym_{N_C,X}$ (following Equation 5).

Repair constraints and constraint solving We then add the decision constraints for the passing and failing inputs:

$$\bigwedge_{C \neq C'} \quad Sym_{N_C,X} > Sym_{N_{C'},X} \tag{6}$$

The obtained solutions guarantee that all the inputs that were used in the repair (both positive and negative) are classified to the correct class. The solutions are used to build the expert for each class. We then combine the experts using the combination strategies outlined in the next section.

Example Consider now the example from Section 3, the case of the last-layer repair. As we aim to repair for class 1 we select for repair the neuron named y_1 in the figure. The score for the edge between N_2 and y_1 is -2.75 and the score for the edge between N_3 and y_1 is 0.45 so we select the latter for repair. We then build the following constraints based on the failed test (note that the expression for the second variable simplifies to a concrete value):
$Sym_{y_1,5} = (2.5 \cdot (2 \cdot (-1 \cdot 0.6 + 2 \cdot 1.0) - 1.9 \cdot (0.5 \cdot 0.6 + 1 \cdot 1.0)) + (1.5 + \delta) \cdot (-0.5 \cdot (-1 \cdot 0.6 + 2 \cdot 1.0) + 3 \cdot (0.5 \cdot 0.6 + 1 \cdot 1.0)))\wedge$
$Sym_{y_0,5} = (-1.5 \cdot (2 \cdot (-1 \cdot 0.6 + 2 \cdot 1.0) - 1.9 \cdot (0.5 \cdot 0.6 + 1 \cdot 1.0)) + 2.0 \cdot (-0.5 \cdot (-1 \cdot 0.6 + 2 \cdot 1.0) + 3 \cdot (0.5 \cdot 0.6 + 1 \cdot 1.0)))\wedge$
$Sym_{y_1,5} > Sym_{y_0,5}$

Similar constraints are added for the positive inputs (we omit them here for brevity). Solving these constraints gives $\delta = 0.1$ which is added to the weight for the edge between N_3 and y_1 to obtain an expert for class 1.

4.3 Combining Experts

We create experts for each label in the dataset. For example, for a neural network trained on the MNIST data set (which is used for the classification of handwritten digits from 0 to 9), we generate 10 experts – one expert per label. We propose three variants of how to combine these experts:

(A) execute the model for all experts and combine the results afterwards,
(B) *merge* all experts into one combined expert before model execution, and
(C) filter strong experts first, then follow variant (A) or (B).

Variant (A) is an instance of *ensemble modeling* [1], which typically involves creating multiple models to predict an outcome. In our case, we start by executing all the experts for each input. This is done in a combined fashion, to avoid repeated execution of same code: before the repaired layer the model is executed with the original weights; starting from the repaired layer the execution is split up for the different experts. At the end of the execution, each expert classifies the input to a certain label. We need to combine the results from all the experts in order to classify the input to a single label.

Each expert can classify the input to any of the labels, however, each expert can be trusted to produce the correct result only for its own respective label. Therefore, we start by generating a set E including the experts that classify the inputs to their respective labels. Note that it could be that multiple experts report that the given input belongs to their respective class or it could be that no expert classifies the given input to the expert's class. If E is empty, then we select the label by the original model. If there is one expert in E, then we select this unique expert. If there are multiple experts in E, then we need to resolve the conflict between experts and choose one label, for which we propose three strategies:

Naive: This strategy simply falls back to the original model.
Confidence: This strategy selects the expert from E with the highest confidence for its own label, i.e, the absolute value of the output node corresponding to the label.
Voting: For the label corresponding to each expert in E, this strategy collects votes from the other experts for the respective label. It then selects the expert from E with the majority of the votes.

In variant (B), we propose to merge the experts before executing the model. For the intermediate-layer repair, for every weight that is considered faulty we update it with the one δ value, which is the *average* of the solutions from all the experts. This creates a single *merged* network. For the last-layer repair, we simply apply all the repairs at once; there is no need for an average as the nodes (and edges) that are targets for repair are disjoint.

In variant (C), instead of using all experts we select a subset of strong experts. Note that each expert is constructed from failing inputs only for the respective label. Therefore, when exposed to data which are supposed to be classified to the expert's label, the expert displays higher accuracy than the original model (higher recall). However, when exposed to data which can belong to different labels, the experts could display lower overall accuracy than the original model (lower precision) due to high false positives. Therefore, we determine which of the experts have both their precision and recall (*F1 score*), computed over all positive/negative inputs, higher than the original model and retain only those while filtering out the rest. The same combination strategies, variant (A), are used to obtain a single classification result for the input.

5 Evaluation

We implemented our approach in the NNREPAIR tool pipeline, which is based on NEUROSPF [21]. It first translates a trained Keras model into Java, uses Symbolic PathFinder (SPF) [16] for concolic execution and z3 [14] for constraint solving. In this section we evaluate NNREPAIR by considering its application to three highly common scenarios; *Scenario 1:* improving accuracy, *Scenario 2:* fixing backdoor attacks, and *Scenario 3:* enhancing adversarial robustness. Our experiments use two commonly used datasets for image classification networks, MNIST and CIFAR-10. We consider two architectures for MNIST with 10 and 7 layers respectively. They are convolutional neural networks (CNNs) and have the typical structure of modern neural networks such as convolutional/dense, max-pooling and softmax layers. The first MNIST model has an accuracy of 96.34% on the standard test set, while the second model has an accuracy of 98.89%. We refer to these models as MNIST-LQ (low-quality) and MNIST-HQ(high-quality) respectively. The CIFAR-10 model is a 15-layer CNN with 890k trainable parameters and has an accuracy of 81.04%. In order to validate our approach, we consider the following research questions:

RQ1 Is NNREPAIR successful in correcting the defects in all three scenarios?
RQ2 How do intermediate-layer repair and last-layer repair compare with each other?
RQ3 What is the inference time overhead introduced by NNREPAIR over the original model?

5.1 Scenarios

(1) The goal of repair in the first scenario is to improve the overall accuracy of a model. We measure the improvement in accuracy on the standard test set, henceforth denoted Test. We use positive and negative examples from the train set, henceforth denoted Train, to generate the repair.
(2) For this scenario, we apply the backdoor attack from [6]. Samples of poisoned data are shown in Figure 3. The poisoned models have good accuracy on the standard data, but poor accuracy on the poisoned data. The goal of the repair is to improve the accuracy on poisoned data, which we measure on a separate poisoned test set P-Test. At the same time, we expect the repair to retain the accuracy on standard, un-poisoned data, which we measure on Test. In this scenario, the first 600 inputs in Train are poisoned (P-Train). We draw from these particular inputs to get the negative examples to focus the repair on the defect. We draw the positive examples from Train.
(3) For the last scenario, we apply adversarial perturbations over Train and Test using FGSM[5], for $\epsilon = 0.05$. This results in four data sets: Train, Adv-Train, Test and Adv-Test. The models have good overall accuracy on Train and Test, but poor accuracy on Adv-Train and Adv-Test. The goal of the repair here is to

[5] `https://www.tensorflow.org/tutorials/generative/adversarial_fgsm`

Fig. 3. Example poisoned data for MNIST (left) and CIFAR-10 (right). The backdoor is embedded as the white square at the bottom right corner of each image. When the backdoor appears, the poisoned MNIST model will classify the input as '7' and the poisoned CIFAR-10 model will classify it as 'horse'.

improve the accuracy on the adversarial data (which we measure on Adv-Test) without damaging too much the accuracy on standard data (which we measure on Test). We draw the negative examples to be used in repair from Adv-Train, while we use positive examples from both Adv-Train and Train. Since we use two separate sets to generate experts, when computing the F1-score for selection of experts, we explored two different options: computing F1 score over Adv-Train only and computing harmonic mean of the F1 scores computed over Train and Adv-Train separately. However, in practice there was no difference as same experts were filtered in both cases.

5.2 Experiment Set-up

For each of the three scenarios, we experimented with both intermediate-layer and last-layer repairs. We evaluated all the combination strategies (Naive, Confidence, Voting, and Merged) with the F1-filtering option being OFF and ON. When F1-filtering is OFF, the experts for all labels are used in the combination strategies by default, while when it is ON, we only include those experts whose F1 score on Train is greater than the original model.

Intermediate-layer repair: We focused on the dense layer just before the output layer for both the MNIST and CIFAR models. The intuition for this selection is that dense layers appearing closer to the output potentially impact the classification decision more than convolutional layers closer to the input (which have the role of feature extraction). The MNIST models have 128 and 100 ReLU nodes and 576 and 400 incoming edges to each neuron at this layer respectively, while the CIFAR model has 512 ReLU neurons and 1,600 incoming edges at this layer. We extracted high support patterns for correct classification; the average support per label was within 1,013 - 2,502 across scenarios, out of around 6,000 inputs per label. The neurons short-listed in $\mathcal{N}_{faulty}$ using pattern-based localization varied between 1 and 10 in number. We focused on modifying the weights of the incoming edges having their scores within the top 10%.

We used the patterns extracted at the layer to select a subset of tests for the purpose of constraint solving. As explained in Section 4, we used decision-tree learning to extract patterns for correct classification for every label. We also extracted patterns for incorrect classification for each label, which represent neuron activations satisfied by inputs which should ideally be classified to the

given label but get mis-classified. From the set of all failing tests for a given label, we select all inputs that satisfy the pattern for incorrect classification for the label. From the set of all passing tests for a given label, we select the subset of inputs that satisfy the pattern for correct classification. We then randomly select # failing tests + 100 inputs from this set. The subset of failing and passing tests selected using the procedure above is used for constraint solving.

Last layer repair: At the last layer, the two MNIST models have 10 ReLU nodes and 128 and 100 incoming edges to each neuron respectively, while the CIFAR model has 10 ReLU neurons and 512 incoming edges. For each label, we selected 5 failing and 5 passing inputs randomly from the respective datasets. For the first scenario, both these failing and passing inputs come from the Train set. For the last layer repair top 5 suspicious weights were made symbolic for each expert. We determined empirically that a larger number for symbolic weights and/or passing/failing inputs leads often to unsat constraints while a smaller number may not improve the network.

The poisoned (2) and adversarial (3) scenarios differ from scenario 1, in that they seek to address two challenges. The repaired model needs to have better accuracy than the original model on poisoned and adversarial inputs respectively (evaluated on the P-Test and Adv-Test sets), as well as the accuracy on normal inputs should not be degraded much (evaluated on the normal Test set). For this reason, for the purpose of constraint solving in addition to including passing tests from the respective poisoned and adversarial train sets, we also include passing tests from Train. We performed experiments increasing the number of passing tests included from the normal train set from 0 to 10, 50, and 100.

5.3 Results

Table 2 presents a summary of our results (please refer to the Appendix[6] for more detailed results). The table displays the results for MNIST and CIFAR models for the three scenarios. For each scenario, the results for both intermediate layer and last layer repair are presented in terms of the improvement in accuracy obtained over the original model. This is the best result corresponding to the improvement in accuracy on the respective test sets (normal Test for the first scenario, P-Test for the second and Adv-Test for the third). The combination strategy and the F1-Filter setting (ON/OFF) used to obtain the best result are also displayed, along with the corresponding improvements in accuracy on the other train and test sets. For the repair, z3 was able to generate solutions for each expert within a minute. The constraint generation using SPF was the bottleneck and SPF generated constraints for each expert within 15 - 60 minutes, depending on the number of tests included. However, this could be improved since running SPF on all positive/negative inputs can be performed in parallel. Experiments were performed on a Windows 10.0 machine with Intel Core-i5 and

[6] `https://arxiv.org/abs/2103.12535`

Table 2. Summary of NNrepair performance on all models. Repair column shows the type of repair, i.e., intermediate or last layer. Increase/decrease in accuracy shown in terms of the difference between the accuracy of the repaired model and the original model on the respective datasets. Accuracy of the original model is shown in brackets (in bold) below each data set. The *Strategy* column shows the combination strategy which work best for each scenario. *ALL* means that all strategies performed equally. *F1-Filter* shows if best results were obtained by turning F1-Filter ON or OFF. The number of experts used are shown in brackets.

Model	**Repair**	**Increase/Decrease in Accuracy**				**Strategy**	**F1-Filter**
		Train (96.59%)		**Test (96.34%)**			
MNIST-LQ	Interm.	+0.22		+0.20		Votes	ON(3)
MNIST-LQ	Last	+0.00		+0.00		ALL	ON(0)
		Train (99.81%)		**Test (98.89%)**			
MNIST-HQ	Interm.	+0.01		+0.02		Merged	ON(3)
MNIST-HQ	Last	+0.00		+0.00		ALL	ON(0)
		P-Train (98.99%)		**Test (98.63%)**	**P-Test (10.38%)**		
MNIST-Pois.	Interm.	+0.00		-0.01	+1.81	Votes	ON(2)
MNIST-Pois.	Last	-2.60		-3.11	+45.56	Confidence	OFF
		Train (98.67%)	**Adv-Train (29.92%)**	**Test (97.87%)**	**Adv-Test (28.37%)**		
MNIST-Adv.	Interm.	-4.35	+2.75	-4.15	+3.87	Confidence	ON(9)
MNIST-Adv.	Last	-3.99	+11.15	-3.14	+10.40	Merged	ON(10)
		Train (87.25%)		**Test (81.04%)**			
CIFAR10	Interm.	+0.03		+0.03		Merged	ON(1)
CIFAR10	Last	+0.12		+0.16		ALL	ON(1)
		P-Train (96.97%)		**Test (72.26%)**	**P-Test (15.89%)**		
CIFAR10-Pois.	Interm.	+0.03		+0.02	+0.81	Merged	ON(4)
CIFAR10-Pois.	Last	-0.89		-0.61	+3.77	Merged	OFF
		Train (87.25%)	**Adv-Train (34.39%)**	**Test (81.04%)**	**Adv-Test (35.96%)**		
CIFAR10-Adv.	Interm.	+0.05	+0.22	-0.07	+0.34	Merged	ON(10)
CIFAR10-Adv.	Last	-0.25	+0.37	-0.27	+0.27	Merged	ON(10)

16GB RAM. The code, constraint files along with Z3 solution files are available at `https://github.com/muhammadusman93/nnrepair`.

RQ1: For this research question we seek to investigate if NNrepair is successful in correcting defects in all three scenarios. To measure success, we consider the improvement in accuracy provided by the repair in all three scenarios.

The effectiveness of NNrepair in improving accuracy (Scenario 1) can be analyzed by considering Table 2 (cases MNIST-LQ, MNIST-HQ and CIFAR10).

We observe that the best results provided by NNrepair for the MNIST-LQ model was +0.20, +0.02 for the MNIST-HQ model and +0.16 for the CIFAR10 model. This improvement (albeit small) was achieved without any new inputs or re-training. The quality of the improvement appears to degrade as the quality of

the original model increases. We note that achieving improvement in the overall accuracy of an already high-quality model without new data is very challenging. In fact this improvement appears to be in line or better than related repair techniques (see Section 6). Note also that the complexity and size of the models do not seem to have an impact on the effectiveness of the repair. The MNIST-HQ architecture is simpler than MNIST-LQ and the CIFAR10 architecture is much bigger and more complex than the MNIST models.

For Scenario 2, on the MNIST-Pois model, NNrepair increased the accuracy from 10.38% to 55.94% on poisoned inputs (P-Test). The repair causes a slight decrease (-3.11) in accuracy on non-poisoned inputs (Test) but the repaired model still has a high accuracy ($\geq$ 95.5%) on non-poisoned inputs. On the more challenging CIFAR10-Pois model, the best improvement provided by NNrepair is a +3.77 increase on poisoned inputs, and a small decrease in accuracy on non-poisoned inputs (-0.61). For Scenario 3, on the MNIST-Adv model, NNrepair increased the accuracy from 28.37% to 38.77% on adversarial inputs, while causing a small decrease (-3.14) in accuracy on non-adversarial inputs. For CIFAR10-Adv, the best result was an increase of +0.34 on adversarial inputs with a minor decrease of -0.07 on non-adversarial inputs.

For the last two scenarios, the primary goal is to improve accuracy on poisoned or adversarial data. Although ideally we would also want to preserve the original accuracy on normal data, this may not always be possible in practice. We experimented with varying number of passing tests from Train for scenarios 2 and 3. The results are presented in the first table in the Appendix. The accuracy of the resulting repair on the poisoned/adversarial test sets tends to decrease as the number of normal passing tests goes up. However, this also reduces the degradation in the accuracy on normal test set. Previous studies in adversarial robustness [23] indicate that one can obtain robust networks but the price to be paid is a significant decrease in accuracy on normal data. Similar considerations apply to the poisoning case. Therefore, we tolerate small decrease in the accuracy on normal Test in our work as well.

The last two columns in Table 2 list the combination strategies and the F1-filtering option which work best for each scenario. The Merged strategy seems to work well for the CIFAR10 model for all the three scenarios. However, there is no clear winner for the MNIST models. In fact, for the last layer repair on CIFAR10, all the strategies gave the same improvement in accuracy. In practice, the users would need to use a separate validation set and try all the strategies to pick the best one for their application domain.

Answer RQ1: NNrepair shows benefit in all three scenarios. It can repair a network to make it robust against adversarial perturbations/poisoned inputs while at the same time retain a good accuracy on the normal, unperturbed/non-poisoned test set. NNrepair can also improve the overall accuracy of the models, however the effectiveness of the repair tends to decrease when the original accuracy is already high.

RQ2: Table 2 can be used to compare the performance of intermediate-layer and last-layer repair on the different scenarios. For the MNIST models, last-layer repair did not help in improving the overall accuracy. Repairing the dense layer before the output layer using the pattern-based repair helps in increasing the accuracy albeit by a small amount. For the CIFAR10 model, on the other hand, repairing the output layer increases the overall accuracy of the model by 0.16, which is better than intermediate-layer repair (+0.03).

For the poisoned and adversarial scenarios, on the MNIST models, last-layer repair performed better than intermediate layer-repair on the targeted test sets. Intermediate-layer repair increased the accuracy by 1.81 on the poisoned model and 3.87 on the adversarial model while last-layer repair increased the accuracy by 45.56 on the poisoned model and 10.40 on the adversarial model. For CIFAR10-Pois, intermediate layer repair increases the accuracy by 0.81 while last layer repair improves it by 3.77. Note that intermediate-layer repair seems to help better in retaining the accuracy on the standard Test, albeit providing smaller improvements on the target sets (detailed results in the Appendix). Furthermore, for CIFAR10-Adv, intermediate layer repair gives better results than last-layer repair (0.34 vs 0.27 respectively).

To summarize, focusing only on an inner layer of the network or just the output layer may not suffice to correct errors in all models and scenarios. We plan to investigate application of repair at more than one layers. Fault localization approaches may help determine the layer/s to focus on for effective repair for a given application.

Answer RQ2: Intermediate-layer repair helped more in improving the overall accuracy of the models (except for CIFAR10) and last-layer repair was more effective in repairing specific failures such as vulnerabilities to poisoned or adversarial inputs (except on CIFAR10 adversarial model). The take away is that there is not a specific type of repair (last-layer or intermediate layer) that works well consistently and different models and failure scenarios may necessitate repair at different layers.

RQ3: To understand the overhead introduced by running multiple experts and the combination logic, we conducted experiments on one of the models, MNIST-LQ. We executed the original model on the test set and compared the inference time with the model produced by a repair at an intermediate layer (i.e., layer 6) and by a repair at the final layer (i.e., layer 8). Additionally, we measured the inference time for an intermediate layer repair with F1-Filtering (i.e., layer6-F1). We performed this comparison for all 10,000 inputs in the test set.

The *Merge* combination strategy does not require any expert combination after model execution because this strategy merges the repairs in advance. Therefore, there is no change with regard to the original model execution except the weight values used in the calculations, and we did not observe any difference in terms of the inference time. We focus the remaining discussion on the strategies

that require the execution of multiple experts. Our experiments show that the time for the expert combination after model execution (as necessary for *Naive*, *Confidence*, and *Voting* combination strategies) is negligible with around 0.0008 ms and also is similar for all these combination strategies. The main overhead is introduced by the additional calculations necessary to compute the multiple expert values at each layer. The box plot in Figure 4 shows the total time for the model execution for the experts inclusive the time for the *Naive* expert combination.

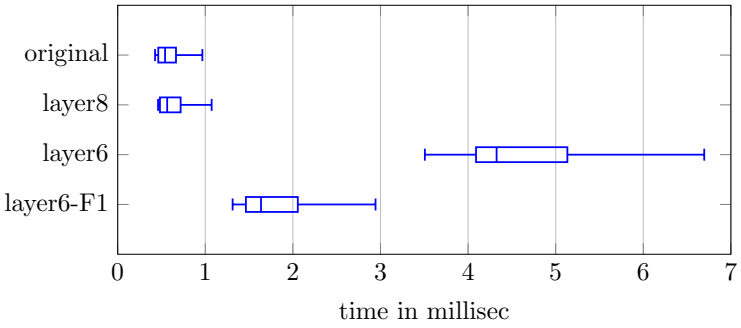

Fig. 4. Inference Time Comparison (*Naive* Combination Strategy)

The repair at the last layer produces an average slowdown (compared to the original model) of 1.0383x. In contrast, the repair at the intermediate layer produces an average slowdown of 7.7638x. Therefore, it makes sense to apply some filtering of experts, which do not show good performance on the training set (see F1-score filtering in Section 4.3). For this experiment we kept 3 experts (see the plot with *layer6-F1*). This reduced model produces an average slowdown of only 3.0742x.

Answer RQ3: The *Merge* combination strategy does not impact the inference time. All other combination strategy introduce a similar overhead. While the inference time for the last-layer repair is comparable with the original model, the inference time for an intermediate-layer repair is expensive. However, it can be significantly reduced with F1 filtering.

5.4 Discussion

The purpose of our evaluation was to showcase the versatility of NNrepair in different scenarios. The takeaway from the experiments is that there is not a specific type of repair (last-layer or intermediate layer) that works well consistently

and different models and failure scenarios may necessitate repair at different layers. In particular, we believe that the intermediate-layer repair holds the most promise for scaling to large networks and we plan to further experiment with the technique in the future.

Generally, the best repair results are obtained on the poisoning task, where the accuracy can be increased by up to 45% and 3.7% on MNIST and CIFAR10, without a need for retraining, which can be expensive in practice. Furthermore, note that we do not assume knowledge of the poison, as our techniques only use information about correct and incorrect classification. In the future, we plan to perform more experiments with different poisoning scenarios.

We were able to obtain modest accuracy improvements on the high-quality models, while for the low-quality models, re-training can achieve better results (see comparison with MODE in the next section). More experimental comparison with retraining and/or fine-tuning the models is needed to further assess the merits of our constraint-based repair.

The gains in the adversarial setting are not very significant for the larger models. In this work, our goal was to demonstrate the feasibility of using localized constraints solving as a generic technique for addressing a wide range of challenges in deep learning. Adversarial attack is only one potential application scenario that is considered. There is a large body of research work on adversarial attacks and we can not claim in any way that we can cover all attacks.

We also note that the efficacy of NNrepair is evaluated statistically (over the test set) as our method does not provide any formal guarantees. In general, it is difficult to guarantee an improvement of the overall accuracy with formalisms, as there are no formal specifications for the image classification domain. Thus, in practice one builds (trains) a model using a statistical measure of accuracy.

6 Related Work

The emphasis of this paper is on neural network repair, where the goal is to "correct" the neural network and improve its performance, robustness and security, by using a small number of labeled inputs. There have been relatively few attempts for repairing a neural network. These neural network repair works can be classified given if re-training is needed and/or if there is a first step to prioritize neuron weights to fix. A number of fix patterns and challenges for neural network repair were collected in [9].

In MODE [12], a neural network is said to be buggy for a specific output label if its test accuracy is lower than the expectation. This is fixed by selecting features that are critical for the misbehavior via differential analysis using a subset of training data and then retraining by selecting inputs from the remaining unused training inputs based on the differential heat map. We ran MODE on the MNIST models from our study. The results are as follows:

Model	Test Acc. (%)
MNIST-LQ	+0.37
MNIST-HQ	-0.40

NNrepair has similar performance, i.e., slightly better than MODE on MNIST-HighQuality and slightly worse on MNIST-LowQuality. Meanwhile, the re-training procedure in MODE led to varied performances for the repaired model. The results for MODE are the average outcome after 10 runs, none of which improved the accuracy of MNIST-HighQuality.

Unlike MODE that identifies ill-trained weights or buggy neurons, Apricot [24] first generates a set of models from the original neural network with a reduced set of training data and at each iteration of the training, Apricot adjusts each weight of the repaired model towards the average weight of these reduced models correctly classifying the input while away from the misclassifications. The approach from [19] uses constraint solving for repairing neural networks. It considers a two-dimension slice of the input space of ACAS Xu and uses SMT constraints to achieve weight changes for correct cases that are checked against the specification. We found it non-trivial to extend this approach to typically high-dimensional input space of the image classifiers that we study in this paper.

Typically, a software repair technique (including for neural networks) employs as a first step *fault localization* to determine the code entities that need to be fixed. DeepFault [2] is an approach to spectrum-based fault localization that aims to identify the neurons that are 'more' responsible to adversarial behaviours of a neural network. However, the aim of DeepFault is to generate more adversarial examples, which is the opposite to the repair purpose of our paper. Another related approach, Arachne [18], uses fault localization to identify neural weights (connected to the final output layer) to modify, using Particle Swarm Optimisation (PSO), for better weights to improve the model's accuracy on some particular label. As also noted in [18], increasing the prediction accuracy for a particular label often comes along with the decreasing prediction accuracy of the overall neural network model.

Our NNrepair work provides a general repair approach which can be applied for improving accuracy, enhancing robustness against adversarial attacks and fixing the backdoor security problems for neural networks. Although previous techniques could be presumably extended to these scenarios, in practice they were only demonstrated for improving the prediction accuracy of the neural network (in MODE and Apricot) or a particular label (in Arachne).

7 Conclusion and Future Work

We presented NNrepair, which uses constraint solving for intermediate-layer and last-layer repair of neural networks. We demonstrated NNrepair in three scenarios: improving the *overall accuracy*, fixing security vulnerabilities caused by *data poisoning* and improving the *adversarial robustness* of the networks.

In future work, we plan to experiment with different localization techniques and to evaluate our repair on larger networks and different architectures. Our method can also be applied to multiple layers but we restricted to single-layer for scalability. One avenue for research is to apply single-layer repair repeatedly or compositionally to handle correcting bugs across multiple layers.

References

1. Ensemble learning methods for deep learning neural networks https://machinelearningmastery.com/ensemble-methods-for-deep-learning-neural-networks
2. Eniser, H.F., Gerasimou, S., Sen, A.: DeepFault: Fault localization for deep neural networks. In: International Conference on Fundamental Approaches to Software Engineering. pp. 171–191. Springer (2019)
3. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
4. Gopinath, D., Converse, H., Pasareanu, C., Taly, A.: Property inference for deep neural networks. In: 34th International Conference on Automated Software Engineering (ASE). pp. 797–809. IEEE (2019)
5. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (Nov 2019). https://doi.org/10.1145/3318162, https://doi.org/10.1145/3318162
6. Gu, T., Liu, K., Dolan-Gavitt, B., Garg, S.: BadNets: Evaluating backdooring attacks on deep neural networks. IEEE Access **7**, 47230–47244 (2019). https://doi.org/10.1109/ACCESS.2019.2909068
7. Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I., Tygar, J.D.: Adversarial machine learning. In: 4th workshop on Security and artificial intelligence. pp. 43–58. ACM (2011)
8. Huang, X., Kroening, D., Ruan, W., Sharp, J., Sun, Y., Thamo, E., Wu, M., Yi, X.: A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. Computer Science Review **37**, 100270 (2020)
9. Islam, M.J., Pan, R., Nguyen, G., Rajan, H.: Repairing deep neural networks: Fix patterns and challenges. In: 42nd International Conference on Software Engineering (ICSE) (2020)
10. Jordan, M.I., Mitchell, T.M.: Machine learning: Trends, perspectives, and prospects. Science **349**(6245), 255–260 (2015)
11. Liu, Y., Ma, S., Aafer, Y., Lee, W..C., Zhai, J., Wang, W., Zhang, X.: Trojaning attack on neural networks. In: 25th Annual Network and Distributed System Security Symposium (NDSS) (2018)
12. Ma, S., Liu, Y., Lee, W.C., Zhang, X., Grama, A.: MODE: automated neural network model debugging via state differential analysis and input selection. In: 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 175–186. ACM (2018)
13. Monperrus, M.: Automatic software repair: A bibliography. ACM Comput. Surv. **51**(1) (Jan 2018). https://doi.org/10.1145/3105906, https://doi.org/10.1145/3105906
14. de Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
15. Papernot, N., McDaniel, P.D., Jha, S., Fredrikson, M., Celik, Z.B., Swami, A.: The limitations of deep learning in adversarial settings. In: EuroS&P (2016)
16. Păsăreanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. Autom. Softw. Eng. **20**(3), 391–425 (2013). https://doi.org/10.1007/s10515-013-0122-2, https://doi.org/10.1007/s10515-013-0122-2

17. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC/SIGSOFT FSE (2005)
18. Sohn, J., Kang, S., Yoo, S.: Search based repair of deep neural networks. arXiv preprint arXiv:1912.12463 (2019)
19. Sotoudeh, M., Thakur, A.V.: Correcting deep neural networks with small, generalizing patches. In: Workshop on Safety and Robustness in Decision Making (2019)
20. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks (2013), technical Report. `http://arxiv.org/abs/1312.6199`
21. Usman, M., Noller, Y., Păsăreanu, C.S., Sun, Y., Gopinath, D.: Neurospf: A tool for the symbolic analysis of neural networks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 25–28 (2021). https://doi.org/10.1109/ICSE-Companion52605.2021.00027
22. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Commun. ACM **53**, 109–116 (May 2010). https://doi.org/http://doi.acm.org/10.1145/1735223.1735249, `http://doi.acm.org/10.1145/1735223.1735249`
23. Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: 35th International Conference on Machine Learning (ICML), Stockholmsmässan, Stockholm, Sweden. pp. 5283–5292. PMLR (2018), `http://proceedings.mlr.press/v80/wong18a.html`
24. Zhang, H., Chan, W.: Apricot: a weight-adaptation approach to fixing deep learning models. In: 34th International Conference on Automated Software Engineering (ASE). pp. 376–387. IEEE (2019)