Symbolic Execution for Importance Analysis and Adversarial Generation in Neural Networks

Divya Gopinath*, Mengshi Zhang†, Kaiyuan Wang†, İsmet Burak Kadron‡, Corina S. Păsăreanu*, Sarfraz Khurshid†
* Carnegie Mellon University and NASA Ames
Email: divgml@gmail.com, corina.pasareanu@west.cmu.edu
† University of Texas at Austin
Email: mengshi0617@gmail.com, wangkaiyuanzz@gmail.com, khurshid@ece.utexas.edu
‡ University of California, Santa Barbara
Email: kadron@cs.ucsb.com

***Abstract*—Deep Neural Networks (DNN) are increasingly used in a variety of applications, many of them with serious safety and security concerns. This paper describes DeepCheck, a new approach for validating DNNs based on core ideas from program analysis, specifically from symbolic execution. DeepCheck implements novel techniques for lightweight symbolic analysis of DNNs and applies them to address two challenging problems in DNN analysis: 1) identification of important input features and 2) leveraging those features to create adversarial inputs. Experimental results with an MNIST image classification network and a sentiment network for textual data show that DeepCheck promises to be a valuable tool for DNN analysis.**

***Index Terms*—**

I. Introduction

Deep Neural Networks (DNN) are increasingly used in a variety of applications, many of them with substantial safety and security concerns [20]. Our focus in this paper is on *classifiers*: DNNs that take in complex, high dimensional input, pass it through multiple layers of transformations, and finally assign to it a specific output label. Such networks are now being integrated into the perception modules of autonomous or semi-autonomous vehicles, at major car companies such as Tesla, BMW, Ford, and others. It is expected that this trend will continue and intensify.

Owing to the increasing trend of employing neural networks in safety critical applications which require high assurance guarantees, the traditional emphasis on obtaining high accuracy for DNNs is being augmented with safety and security goals [16]. However, validating DNNs is complex and challenging, due to the nature of the learning techniques that create these models. For example, it is not well understood *why* a DNN, say an image classifier, gives a particular output. This inability to explain the DNN decisions hinders their application in safety critical domains, such as autonomy. Furthermore, evaluating the robustness of a network against conceptually simple yet effective attacks is a hard technical problem, due to the huge input space of such networks.

This paper presents an approach for the analysis of deep neural networks based on *symbolic execution* [7], [19]. Symbolic execution is a well-known program analysis technique that has seen many advances in recent years [3], [4], [12], [18], [27] and applications in various domains, such as security [6], [8], smartphone apps [1], operating systems [36], and databases [10]. The technique executes the program on symbolic inputs and systematically collects symbolic mathematical constraints based on the branching conditions in the code. These constraints are solved with off-the-shelf solvers to obtain new inputs that execute feasible program paths.

We note that neural networks that employ piecewise linear activation functions can be seen as imperative programs, which makes them amenable to program analysis techniques. A popular class of neural networks use rectified linear units (ReLUs) activation functions and max pooling operations (convolutional neural networks). Such networks can be naturally translated into a branching structure , whereby a path through the neural network can conceptually be viewed as a path through the translated program. This enables the application of *symbolic execution* to build path conditions for paths through the network. We apply symbolic execution to select program paths of interest, e.g., paths taken by specific inputs from the network's training dataset.

However, just building the path condition for even one path, using a straightforward application of concolic (or dynamic symbolic) execution [3], [4], [12], [27], can take considerable amount of time, and solving a path condition with just one symbolic variable can stress modern constraint solvers. We perform a lightweight analysis (without any constraint solving) to determine inputs that have the *most impact on the classification decision* and perform a directed symbolic execution approach for *generating adversarial inputs for the evaluation of network robustness* with *minimal* constraint solving.

This paper makes the following contributions:

- **Idea.** We propose a symbolic execution analysis framework for neural networks which focuses on identifying *important input features* and using them to *guide* the symbolic execution for *adversarial attack generation*.
- **Approach.** We describe the DeepCheck approach, that is embodied by two techniques: DeepCheck^{Imp}, which applies symbolic execution for identifying important input features that intuitively provide explanations for the

network's decisions; and DeepCheck^{attack}, which applies symbolic execution to create adversarial attacks.

- **Evaluation.** We present an experimental evaluation on an image classification network using the widely studied MNIST dataset, and on a sentiment network that operates on textual data. The results highlight the feasibility of using symbolic execution to identify important input features and to create attacks, and also show that the identified important features enable a more scalable method for generating adversarial examples. To the best of our knowledge, this is one of the first approaches to generate meaningful adversaries for textual analysis applications in an efficient manner.
- **Robustness guarantees.** Although the adversarial attacks generated for the image classification network are simple and have been studied before [30], we make the surprising observation that neural networks can be vulnerable to such attacks even along the paths that follow the *same activation* patterns as validly classified inputs. Such attacks went unnoticed with previous testing techniques [26], [31], [34], which focused on generating tests that increase the coverage of activated neurons, and hence did not check for attacks along the same path. Furthermore, if such attacks are not found, our tool then is able to provide formal *guarantees* that the network is behaving as expected.

II. Background:

A. Neural Networks

A neural network defines a function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ mapping an input vector of real values $X \in \mathbb{R}^n$ to an output vector $Y \in \mathbb{R}^m$. For a classification network, the output typically defines a score (or probability) across m classes, and the class with the highest score is typically the predicted class. A *feed forward* network is organized as a sequence of layers starting with the input layer. Each intermediate layer consists of computation units called *neurons*. Each neuron consumes a linear combination of the outputs of neurons in the previous layer, applies a non-linear activation function to it, and propagates the output to the next layer. The output vector Y is a linear combination of the outputs of neurons in the final layer. For instance, in a Rectified Linear Unit (ReLU) network, each neuron applies the activation function $ReLU(x) = max(0, x)$. Thus, the output of each neuron is of the form $ReLU(w_1 \cdot v_1 + \ldots + w_p \cdot v_p + b)$ where $v_1, \ldots v_p$ are the outputs of the neurons from the previous layer, $w_1, \ldots, w_p$ are the weight parameters, and b is the bias parameter of the neuron.[1]

B. Symbolic Execution

Traditional symbolic execution executes programs on *symbolic*, instead of concrete inputs and systematically explores the program paths (up to a given depth bound). For each path explored, it builds *path conditions*, i.e., constraints on program inputs that execute that path based on the conditional branches in the code. To illustrate, when a conditional statement, say "$if(c)$..." is executed, each of the two conditional branches is individually explored, and the path condition PC is updated to $PC \wedge c$ for the *then* branch and to $PC \wedge \neg c$ for the *else* branch. The feasibility of the path conditions is checked using off-the-shelf constraint solvers, such as satisfiability modulo theories (SMT) solvers [2], [9], as branch conditions are encountered during symbolic execution to detect and avoid infeasible paths (if possible) and to generate test inputs that execute feasible paths (as desired). Overall, the program effects are computed as *functions* over the symbolic inputs.

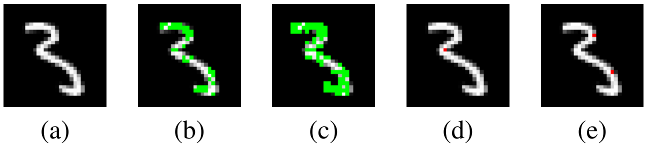


Fig. 1: (a) Example image with predicted label 3. (b) Top-5% important pixels (highlighted in green) identified by DeepCheck^{Imp}. (c) Top-10% important pixels (green) identified by DeepCheck^{Imp}. (d) 1-pixel attack (highlighted in red) identified by DeepCheck^{attack}; changing the red-pixel to black changes the predicted label to 8. (e) 2-pixel attack (red) that does include an attackable pixel for 1-pixel attack.

III. The DeepCheck Approach

This section gives an illustrative overview of our approach on an image classification network, trained on the MNIST dataset, which is one of our case-study subjects. Figure 1(a) shows an example image from the standard MNIST training data, which has the predicted label of 3 (which is the same as its true label). Our technique performs the following three steps:

```
$val := 0$ // Initilaization
for $j$ in $range(0,\ n^{h-1})$: // Linear layer
    $val := val + w_{j,i}^{h-1} \times s_j^{h-1}$
$val := val + b_i$
if $val > 0$: // ReLU layer
    $s_i^h := val$
else:
    $s_i^h := 0$
```

Fig. 2: $Node_i^h(S^{h-1})$ is an imperative function representing the branching transformation for neuron i at layer h.

A. Translation (DeepCheck$^{\tau}$)

We translate the trained model into an imperative program.A typical neural network structure does not have any branching. However, observe that in the case of rectified linear units (ReLU), the activation function $f(x) = max(0, x)$ can be naturally translated into a branching instruction, *if* $(x > 0)$ *then return* x; *else return* 0;. Thus, a path through a neural

[1] Most classification networks based on ReLUs typically apply a softmax function at the output layer to convert the output to a probability distribution. We express such networks as $F := softmax(G)$, where G is a pure ReLU network, and then focus our analysis on the network G.

network can be seen as a path through the translated program, where each executed branch corresponds to the respective ReLU node being activated or not. For simplicity, we discuss here only ReLU networks but our approach applies to other piecewise linear networks.

Consider a network $\mathcal{N}$ with input vector $X = \langle x_0, ..., x_{n-1} \rangle$, output vector $Y = \langle y_0, ..., y_{m-1} \rangle$ and l layers. Any layer h has n^h ReLU nodes and produces an output vector, $V^h = \langle v_0^h, ..., v_{n^h-1}^h \rangle$, which feeds as input to the subsequent layer. Each neuron consumes a linear combination of the outputs of neurons in the previous layer, applies a non-linear activation function to it (in our case max), and propagates the output to the next layer. The goal of DeepCheck$^{\mathcal{T}}$ is to convert this model into a semantically equivalent program $\mathcal{P}$, s.t. for any input x, $\mathcal{N}(x) \equiv \mathcal{P}(x)$.

Let us consider the input state of $\mathcal{P}$ to be S^{inp}, which is $\leftrightarrow X$, output state as S^{op} and the outputs of the hidden layer nodes represented as the intermediate states $S^h = \langle s_0^h, ..., s_{n^h-1}^h \rangle$. Any ReLU node i at layer h applies a function $h_{W,b}(V^{h-1})$ to produce output v_i^h. W is the weight matrix and b is the bias term ($w_{j,i}^{h-1}$ is the weight of the edge connecting node j of layer $h-1$ with node i of layer h and b_i^{h-1} is the bias term added for node i at layer h). The imperative code function corresponding to a ReLU node i at layer h ($Node_i^h(S^{h-1})$) is shown in Fig. 2. Invoking this function n^h times, produces the list of states, $S^h = \langle s_0^h, ..., s_{n^h-1}^h \rangle$ which is equivalent to the outputs of the intermediate layer h of the neural network, $V^h = \langle v_0^h, ..., v_{n^h-1}^h \rangle$. Application of the same process for every layer until softmax, produces the set of output states S^{op}, which is $\equiv Y$, the output of the network.

B. Important input identification (DeepCheck^{Imp})

This step aims at identifying the input variables that impact the decision of the network the most. We execute the program $\mathcal{P}$ on an input $\mathcal{I}$ and obtain the mathematical characterization of every output variable in terms of the input variables (784 pixels in the case of MNIST).

The output of any hidden neuron can be expressed in terms of the input variables. Let us consider the neuron i at the second hidden layer of a fully connected network. The output of the neuron before the application of the ReLU function can be expressed as follows: $w_{0,i}^1 \cdot (w_{0,0}^0 \cdot x_0 + w_{1,0}^0 \cdot x_1 + ... + w_{n-1,0}^0 \cdot x_{n-1} + b_0^0) + w_{1,i}^1 \cdot (w_{0,1}^0 \cdot x_0 + w_{1,1}^0 \cdot x_1 + ... + w_{n-1,1}^0 \cdot x_{n-1} + b_1^0) ... + w_{n^1-1,i}^1 \cdot (w_{0,n^1-1}^0 \cdot x_0 + ... + w_{n-1,n^1-1}^0 \cdot x_{n-1} + b_{n^1-1}^0) + b_i^1$. Therefore by induction each output element, y_i, could be expressed as

$$y_i = C_{i,0} \cdot x_0 + C_{i,1} \cdot x_1 + ... + C_{i,n-1} \cdot x_{n-1} + B_i$$

where B_i is a constant term, and $C_{i,j}$ is the coefficient (signed) of the linear polynomial, that can be calculated in terms of the weights of the non-zero edges from x_j to y_i as shown below;

$$C_{i,j} = \sum_{p \in Paths_{(i,j)}} \Big(\prod_{e \in Edges_p} w(e) \Big) \qquad (1)$$

where $Paths_{(i,j)}$ denotes the set of paths from input node x_j to output node y_i, $Edges_p$ denotes the set of edges on path p, and $w(e)$ denotes the weight of edge e.

Note that the coefficient term $C_{i,j}$ precisely corresponds to the respective partial derivative of the output variable y_i w.r.t the input variable x_j (dy_i/dx_j). Therefore, we use the value of the coefficient of the input variable to determine its impact on the output variable, akin to gradient based approaches that use the partial derivative to determine the impact of each input variable. Note that we can employ any of the existing gradient based approaches to identify the important input variables. The DeepCheck technique is built on a symbolic execution framework for attack generation and the coefficients are a by-product of the application of symbolic execution. Therefore we use them for importance analysis.

We use the coefficients of the input variables in the expression corresponding to the predicted label (3 in the example), to assign *importance scores* for every input variable. An input variable x_1 is considered more important than another x_2, if the classification decision is impacted more by x_1 than x_2. DeepCheck^{Imp} employs three importance metrics: *abs*(absolute value of coeff), *co*(actual signed value of coeff), *coi*(actual value of coeff × input value). The input variables (pixels) are then sorted in the descending order of their scores and those which are in the top threshold % of this ordered list are identified as being *important*. The rationale being that a small change to the image with respect to the important pixels, such as changing the value of just one important pixel can have a high impact on the classification decision, and may lead to the discovery of *adversarial* examples – the new image differs from the original image by the value of just one pixel but this makes the network incorrectly assign a different label to this image.

Figure 1(b) illustrates the top-5%, i.e., 39, important pixels highlighted in green. Note, how the important pixels trace the shape of the digit 3 and do not point to areas of the image irrelevant to the digit being identified as 3 such as the background or the edges. Figure 1(c) illustrates the top-10%, i.e., 78, important pixels highlighted in green. These important pixels form a denser pattern that traces the shape of the digit 3. This highlights that short-listing pixels based on their coefficient based importance scores can help *explain* the classification decision.

C. Adversarial attack generation (DeepCheck^{attack})

Our adversarial attack generation algorithm, DeepCheck^{attack}, aims to create a new input that differs from the original input at t input variables, and has (1) the same *activation pattern* as the original but (2) a different label. For a given input image $\mathcal{I}$, with t input variables symbolic, the path condition over program $\mathcal{P}$ till output layer Y, is a conjunction of inequalities (introduced by the ReLU function) of the form $PC = \bigwedge_{j=1}^{A} (B_j + \sum_{i=1}^{t} C_{j,i} \cdot X_i \ \gamma \ 0)$, where j represents the j^{th} activation function defined by the computation order, A is the total number of activation functions. B_j is the bias term of the output value of the j^{th} hidden neuron; $C_{j,i}$ and X_i are the i^{th} coefficient of the j^{th} hidden neuron and the i^{th} symbolic value respectively. $\gamma \in \{>, \leq\}$ and is determined by the activeness of the j^{th} hidden neuron. In practice, the number of conjunct clauses is smaller than A because sometimes all coefficients of the symbolic values ($C_{j,i}$) are 0 in which case the entire conjunct clause evaluates to true.

The output value of the j^{th} neuron in the output layer Y is a function of the symbolic variables X of the form $f_j(X) = B_j + \sum_{i=1}^{t} C_{j,i} \cdot X_i$. Assume that the network predicts label l (0 to n-1) for the input $\mathcal{I}$, then DeepCheck^{attack} add constraints $AC = \bigwedge_{j=1, j\neq l'}^{n} f_j(X) < f_{l'}(X)$ to require the network to predict a label l' where $l \neq l'$. DeepCheck^{attack} invokes a constraint solver with constraints $PC \wedge AC$ to solve for concrete values for all X_i. If a solution is found, DeepCheck^{attack} succeeds in an adversarial attack by setting X with the concrete values the solver returns and the network predicts label l' which is different from the original predicted label l with the same neuron activation pattern. If no solution is found, this is a proof of the robustness of the network to adversarial perturbations involving t input features or variables.

Figure 1(d) shows a 1-pixel attack identified by our approach for image $\mathcal{I}$; changing the red pixel to black changes the predicted label of the image to 8. This attackable pixel actually lies in the top-5% (top 39) important pixels for $\mathcal{I}$ identified by DeepCheck^{Imp}. The rank order of this attackable pixel in descending order of importance is 21. Hence, focusing the 1-pixel attack on important pixels can allow finding an attack much quicker than checking every pixel for attackability. In fact, this image only has one 1-pixel attack. A linear search that starts at the first image pixel (top-left corner) and scans left-to-right takes 346 attempts to find this attack pixel, which is over 16X the attackable pixel's rank-order (21). We believe *important pixels* can provide a practical heuristic for a *more scalable approach* to create attacks.

To create 2-pixel attacks, we focus DeepCheck^{attack} on the important pixels identified by DeepCheck^{Imp}, specifically on the top-5% important pixels. We make $\binom{39}{2}$ = 741 unordered pairs of the selected important pixels, and for each pair, we make the two corresponding variables symbolic, so each path condition created by symbolic execution contains exactly two symbolic variables. Applying DeepCheck^{attack} to the 741 pairs results in 93 unique potential 2-pixel attacks. 38 of the 2-pixel attack pairs contain as an element the pixel that was earlier identified for the 1-pixel attack, whereas 55 of the pairs contain only pixels that are not 1-pixel attackable; Figure 1(e) shows one such pair in red. The important pixels identified by DeepCheck^{Imp} play a key role in focusing DeepCheck^{attack} to find a 2-pixel attack. The first attack found by DeepCheck^{attack} for this example, includes the 2 of the 3 top-most important pixels. Thus, the search for a 2-pixel attack for this example requires checking no more than just $\binom{3}{2}$ = 3 pairs. These results illustrate the potential of using symbolic execution in identifying important pixels and creating 1-pixel and 2-pixel attacks, as well as the value of important pixels in finding attackable pixels and pixel-pairs.

IV. Evaluation

In this section we present two case-studies of applying DeepCheck on two networks; an image classification network for the MNIST dataset and on a sentiment network for textual data. DeepCheck has been implemented in Java (using the Z3 solver) and also has a Python version (using `pulp`) to facilitate easy interface with TensorFlow.

The image network is a fully connected network with the following configuration, 784×10×10×10×10. It was trained on 60,000 images of the MNIST dataset [22], and has an accuracy of 92%. The textual network consists of one convolutional layer (3×30, 64 filters) and one dense layer (1792×1), where the first layer uses ReLU activation and last layer uses a sigmoid activation. This network was trained on the IMDB movie reviews dataset with size 50000. Half of the dataset is used for training and the other half for testing. For the training, only top 10000 words are kept and each text is padded or pruned to 30 words long (for any input longer than 30 words, we use the last 30 words). We used embeddings with size 30 trained with word2vec algorithm, and it has an accuracy of 76%. Although the accuracy of the networks is below state-of-the-art, the simplicity of the networks make them amenable to analysis with our implementation.

We seek to address the following research questions as part of our evaluation.

1) **RQ1:** Are the input features identified by DeepCheck^{Imp} useful in explaining the classification decision?
2) **RQ2:** How effective is the attack generation technique, DeepCheck^{attack}, in generating adversarial inputs?
3) **RQ3:** Are the important input features identified by DeepCheck^{Imp} sensitive to adversarial perturbations and do they help in faster generation of attacks?
4) **RQ4:** How do the importance metrics (*abs*, *co*, *coi*) compare in terms of accurately identifying input features that impact the network decision?
5) **RQ5:** How does DeepCheck compare with another adversarial attack generation approach?

A. Image Classification Network

We first present the results of applying our technique on the MNIST network for important pixel identification and adversarial attack generation.

1) Important pixel identification: We used 10 images from the data set (covering all ten labels). For each image, we applied DeepCheck^{Imp} to compute a ranked list of pixels according to their relative importance based on the three metrics: *abs*, *co*, *coi*. Table I shows the results produced by DeepCheck^{Imp} for the three metrics for top-5%, top-10% and top-30% of important pixels. For each image (digit), the pixel values in the original image are shown in white and black, while the green color highlights the pixels that are identified important.

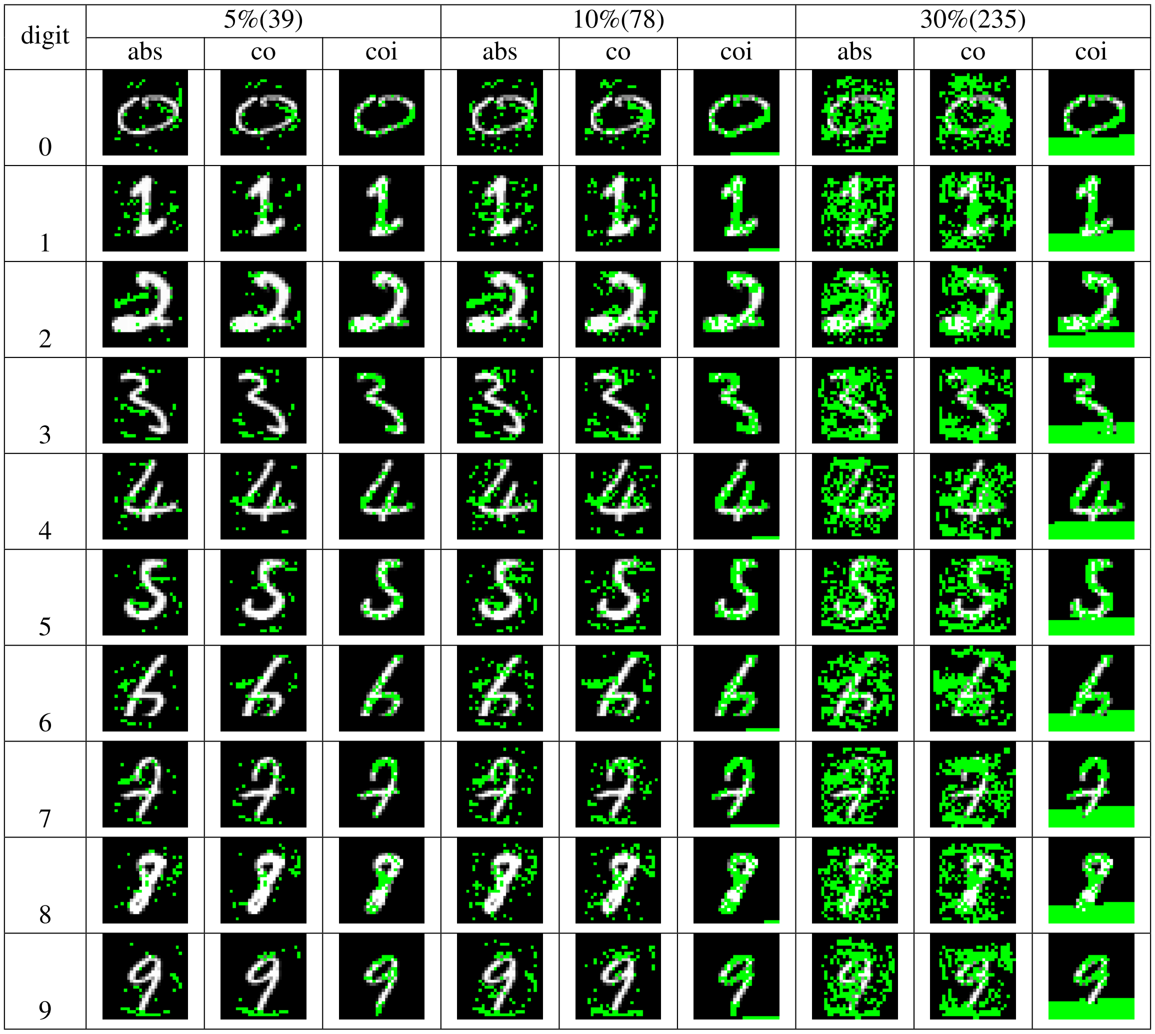

digit
5%(39)
10%(78)
30%(235)
abs
co
coi
abs
co
coi
abs
co
coi
0
1
2
3
4
5
6
7
8
9

TABLE I: Top-5%, top-10% and top-30% of important pixels (green) identified by DeepCheck^{Imp} for *abs*, *co*, and *coi*.

digit	1-pixel attack										2-pixel attack	
	# ap										# ap	# ap-new
	ordered baseline	shortlist-5%			shortlist-10%			shortlist-30%			shortlist-5%	
		abs	co	coi	abs	co	coi	abs	co	coi	coi	
0	25	3	7	16	7	11	17	20	16	17	548	60
1	4	0	1	3	1	2	4	3	4	4	198	87
2	1	0	0	1	0	1	1	1	1	1	48	10
3	1	0	0	1	0	0	1	0	1	1	93	55
4	6	4	3	4	4	3	4	6	4	4	260	114
5	36	1	2	11	3	2	18	14	11	19	463	100
6	1	1	1	1	1	1	1	1	1	1	287	186
7	47	8	7	18	14	11	22	22	18	22	651	75
8	2	0	0	2	0	0	2	2	2	2	111	36
9	3	0	0	2	0	0	2	2	2	2	171	96

TABLE II: # Attackable pixels detected by the different versions of DeepCheck^{attack} (baseline,ordered,shortlist5%,shortlist10%,shortlist30% for 1-pixel attacks, shortlist5% for 2-pixel attacks.)

For the top-5% and top-10% images, it can be observed that each metric marks pixels in the central part of the image as the most important, while the top-30% images seem to spread out towards the edges. The central part of an MNIST image houses the digit. This highlights that the importance metric does correctly point to the part of the image that aids the network to make the classification decision, with the precision decreasing as the threshold % increases. Further, the *abs* and

digit	1-pixel attack				2-pixel attack
	baseline	ordered			shortlist-5%
		abs	co	coi	coi
0	244	7	5	1	2
1	489	60	23	6	2
2	516	119	66	19	19
3	346	254	169	21	3
4	71	3	1	1	2
5	103	6	1	1	2
6	486	2	2	2	2
7	156	4	2	1	2
8	211	142	86	19	4
9	240	169	98	13	2

TABLE III: Number of pixels (out of 784) that had to be tried by the different versions of DeepCheck^{attack} before discovery of the first attack.

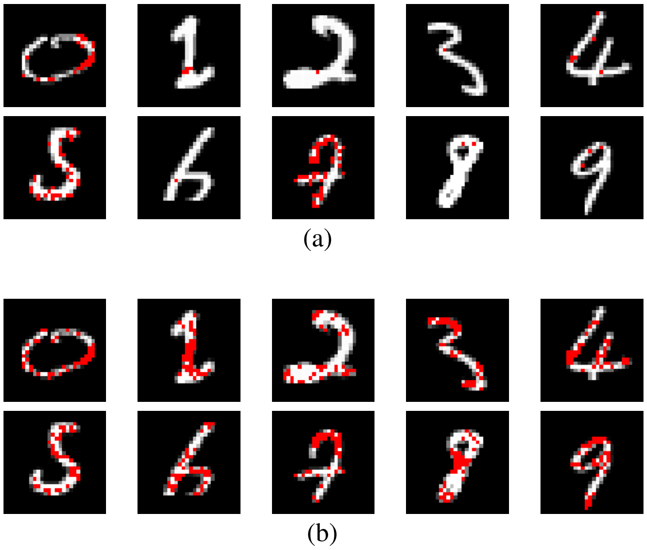
ID	Target input	L_{orig}	L_{NW}	Len_{Imp}	Len_{Rand}
1	“one on his plate he almost seemed to know this wasn’t going to work out and his performance was quite <UNK> so all you madison fans give this a miss"	NEG	POS	**4**	26
2	“some might even say bizarre this is worth the time br br unfortunately it’s very difficult to find in video stores you may have to buy it off the internet"	POS	POS	**13**	27
3	“any era that lets its guard down and is overwhelmed by <UNK> it’s a fascinating film even a charming one in its macabre way but its message is no joke"	POS	POS	**9**	21
4	“can hardly see what is being filmed as an audience we are <UNK> involved with the actions on the screen so then why the hell can’t we have night vision"	NEG	NEG	**6**	24
5	“shut about details but please try this game it’ll be worth it br br story 9 9 action 10 1 it’s that good <UNK> 10 attention <UNK> 10 average 10"	POS	POS	17	**9**
6	“should at least be put back on the channel this movie doesn’t deserve a cheap <UNK> it deserves the real thing i’m them now this movie will be on dvd"	POS	NEG	5	**3**
7	“words in each sentence and delivers them in an almost irritating manner its not funny ever but its meant to be bing and joan have done much better than this"	NEG	NEG	**2**	5
8	“in this genre few of them come up to alexander <UNK> original thief of <UNK> almost any other <UNK> nights film is superior to this one though it’s a loser"	NEG	POS	5	**2**
9	“good film i highly recommend watching this in <UNK> with the first and then <UNK> for how good the series could have been had it continued under burton and keaton"	POS	POS	**3**	14
10	“providing plenty of laughs and chuckles along the way as well as a good deal of suspense br br for <UNK> of black comedy this one is guaranteed to please"	POS	POS	7	**5**
11	“<UNK> series now they are <UNK> 1 and i don’t even think i will watch it oh who am i kidding i probably will and probably will be disappointed again"	NEG	NEG	**3**	22

TABLE IV: Attacks generated using DeepCheck^{attack}on the sentiment analysis network over a set of input sentences. Note that <UNK> is a placeholder word for rare words, which do not have a corresponding embedding vector in the vocabulary. L_{orig} and L_{NW} are the label of the input in the dataset and label that the trained network assigns. Len_{Imp} and Len_{Rand} represent the number of words changed from original sentence for a valid attack using importance selection and random selection respectively.

co metrics show similar patterns in the central region, while the *coi* metric most closely follows the digit’s shape. Specifically, the *coi* metric for top-10% pixels forms a dense pattern tracing the shape of the digit.

2) Adversarial generation: We evaluate the following versions of DeepCheck^{attack}.

- **Baseline:** An exhaustive search of the image is performed, one pixel at a time, starting from the top-left (0) to bottom-right (783). We apply DeepCheck^{attack} to check each pixel for attackability. An *attackable pixel* (ap) can be given a different value to change the image’s predicted label while preserving activation patterns of all the neurons in the path. All possible *attackable pixel*s (1-pixel adversarial attacks) are identified for every image.
- **Ordered:** The pixels are ordered based on the importance scores assigned by DeepCheck^{Imp}. We apply DeepCheck^{attack} one pixel at a time on this ordered list and identify all *attackable pixels* (1-pixel adversarial



(a)

(b)

Fig. 3: Attackable pixels for 1-pixel (a) and 2-pixel attacks (b) highlighted in red.

attacks).

- **Short-list-threshold%:** We apply DeepCheck^{attack} on a shorter version of the ordered list: for instance, short-list-5% applies DeepCheck^{attack} on the top 5% of the pixels ordered based on the importance scores assigned by DeepCheck^{Imp}. We used this technique to find all 1-pixel attacks and 2-pixel attacks within the short list.

a) ***1-pixel attacks:*** We applied the baseline, ordered, and short-list-threshold% (thresholds 5, 10, 30) versions of DeepCheck^{attack} to identify 1-pixel attacks in the 10 images. Figure 3(a) highlights, for each digit, each attackable pixel (in red) for a 1-pixel attack. The attackable pixels lie on or are very close to the shape of the corresponding digit. Some images, e.g., digit 2, contain one attackable pixel out of 784 pixels, whereas some others contain multiple, e.g., 47 for digit 7. All images except digit 6 when attacked get a unique incorrect label. Digit 6 has 2 attacks leading to incorrect labels 5 and 8 respectively, both using the same pixel.

Table II shows the number of attackable pixels discovered by all the techniques. The baseline and ordered versions of DeepCheck^{attack} detect all possible attackable pixels (# ap) for 1-pixel attacks. However, use of the ordered list helps discover attacks faster. Table III shows the number of pixels that needed to be tried before discovering the first attack. For all metrics and all images, no more than top one-third of the important pixels need to be checked by the ordered version, to find an attackable pixel. Moreover, for all metrics, less than 10 pixels needed to be checked to discover an attackable pixel for at least half of the images. The *coi* metric requires the least number of attempts (a maximum of 21 pixels to be checked across all the images), and for 4 images, the top most important pixel identified by *coi* is an attackable pixel. This highlights that ordering pixels based on their importance scores definitely helps reduce the time to find attacks.

In the short-list-threshold% versions of DeepCheck^{attack}, we consider only the top 39 (5%), 78 (10%), and 235 (30%) pixels respectively for the generation of attacks. Table II shows that for half of the images, use of the top 10% pixels suffices to cover all possible 1-pixel attacks. Even for the remaining images, short-list-10% discovers half of the total number of attacks. Digits 7 and 5 have are the most vulnerable to 1-pixel attacks. We find that DeepCheck^{Imp} helps in catching subtle adversarial attacks: i.e. on images that the network is mostly robust to adversaries (such as digits 3 and 6 with just 1 attackable pixel), the importance score helps identify the pixels that are sensitive to adversarial perturbations.

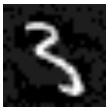


Fig. 4: Adversary generated using FGSM technique.

b) ***2-pixel attacks:*** It is not scalable to apply the baseline or ordered versions of DeepCheck^{attack} to check for all possible 2-pixel attacks. Therefore, we applied short-list-5% using a list ordered by the *coi* metric. For each image, we selected all $\binom{39}{2}$ = 741 unordered pairs that can be formed using the short-listed pixels. We then evaluated each pair to determine if they generated an attack. For each digit, Figure 3(b) shows the union of all pixels in any 2-pixel attack and displays their location. These pixels lie on or are very close to the shape of the corresponding digit. Table II shows the number of attackable pixels (# ap) comprising 2-pixel attacks identified by DeepCheck^{attack}, and of those attacks the number that does not include any attackable pixel for a 1-pixel attack (# ap-new). As expected, many 2-pixel attacks consist of a pixel that was 1-pixel attackable. However, several new attack pairs that do not include any pixel that is attackable for 1-pixel attack are found. 4 out of 10 digits can be attacked to create multiple incorrect labels, e.g., digit 8 can be attacked using 3 different 2-pixel attacks to make the network incorrectly classify it as 1, 2, or 3.

Table III presents the number of important pixels to explore to find the first 2-pixel attack for each image (digit). The worst case is for digit 2, where top-19 important pixels must be considered to find a 2-pixel attack. The best case happens for 7 out of 10 digits, where the top-2 important pixels allow DeepCheck^{attack}to create a 2-pixel attack.

B. Sentiment Analysis Network

The inputs to the sentiment analysis network are sequences of words rather than images. Each word is represented by a vector called word embedding that tries to capture the semantic relation between words. The words used in similar contexts in this vector space have similar embeddings and that can help with learning natural language processing (NLP) tasks [21]. This brings a challenge where identifying importance or getting attacks on single values does not mean much as the words are represented as vectors, therefore we need to identify the important vectors and get attacks on vectors.

To overcome this challenge we had to modify the importance calculation. We calculated the importance score for each element on the vector similar to the importance score calculation for pixels (Section III). However, we then sum these scores over the word embedding vector to get the importance score for a word. We then rank the words based on their importance scores, make all the elements of the embedding vectors for the top-k words symbolic, and apply the DeepCheck^{attack} algorithm to find a feasible attack.

We use the Python version of DeepCheck^{attack}, implemented using the Python LP solver library called puLP, for scalability. The solution returned by the solver is a set of values assigned to each of the elements made symbolic. However these values may not represent valid embeddings for words. Therefore, we find the nearest valid word embedding (from a vocabulary of word embeddings) for each vector in our solution (using the L_2 distance metric). We then replace the words corresponding to those vectors in the original sentence with the corresponding nearest words and test that this new sentence indeed represents a valid attack on the network by changing the sentiment.

We compare attack generation based on importance selection to attack generation based on random selection over example inputs to see whether selecting words based on importance helps us generate shorter attacks. Shorter attacks are preferable since they involve replacing fewer number of words in the original sentence such that some meaning of the original sentence is retained. Therefore, the sentence used for attack has semantical similarity to the original sentence and therefore should ideally be assigned the same sentiment by the network. Let us consider sentence 7 (Table IV), with a negative sentiment as an example. The importance analysis determined 'irritating' and 'this' to be the first two most important words. We made the embedding vectors for these two words symbolic and were able to obtain a solution that the network marks with a positive sentiment. The closest valid embeddings to this solution were the words 'stupid' and 'mode' for 'irritating' and 'this' respectively. We generated a new sentence replacing the two words, which still represented a negative sentiment, but was classified as being a positive sentiment by the network. If we select the words to be replaced randomly instead, we can find an attack by replacing 5 words on an average.

Let us consider another example, for sentence 9 (Table IV), we are able to obtain an attack by replacing the 3 most important words 'good' with 'breakdown', 'film' with 'confronts' and 'highly' with 'terrific'. The attack makes the first part of the sentence a bit nonsensical with 'breakdown confronts' but the positive sentiment is still in the text but the network classifies this sentence as negative instead. In comparison, without importance selection we are able to obtain an attack by changing 14 words, nearly half of the whole input text. This experiment highlights the fact that importance analysis can help us generate more focused attacks with less changes compared to random selection.

Note that we used only solutions that lead to attacks on the network after replacement of the sentence with the valid words closest to the respective solutions. For both approaches, there were cases where the solutions when replaced with the corresponding nearest valid words did not change the sentiment. We attribute this behavior to sparseness of words in this vector space which may affect the distance of the nearest word, and changes in activation patterns when the solution gets replaced by these word. The results of our experiments are displayed in Table IV.

C. Discussion:

In this section, we address the research questions based on our experimental results.

1) **RQ1:** For the image classification network, based on our observations (Table I) we can infer that DeepCheck^{Imp} is able to identify input pixels that define the shape of the input digit. The identified pixels can thus be considered responsible for the classification decision. We observe based on the results from the textual model, that DeepCheck^{Imp} helps identify important input features such as *words*, which impact the decision of the network the most.
2) **RQ2:** The results in Table II show that we were able to generate 126 1-pixel attacks and discovered 819 attackable pixels for 2-pixel attacks for the image classification network. We were also able to successfully generate attacks on the textual model (Table IV) which is more complex than merely modifying pixels in an image. Thus we were able to use DeepCheck^{attack} to generate attacks on networks of significant size and on a real datasets.
3) **RQ3:** The important pixels identified by DeepCheck^{Imp} do correspond to those that are sensitive to adversarial perturbations, as can be observed in Table II. Exploring just top 10% of the important pixels helps generate all possible 1-pixel attacks. Use of DeepCheck^{Imp} makes it feasible to generate 2-pixel attacks which would otherwise require considering $\binom{784}{2}$ = 306946 potentially attackable pairs. The experiment on the textual model highlights that choosing the important words identified by DeepCheck^{Imp} increases the chances to discovering semantically meaningful attacks more efficiently than a random selection of words. Overall, it can be inferred that the use of DeepCheck^{Imp} makes the attack generation process scalable and helps identify subtle adversaries.
4) **RQ4:** The importance metric *coi* seems to perform the best in terms of identifying pixels that impact the classification decision and are also vulnerable to attacks. For the MNIST images, this seems a little obvious since the background is always black (pixels have value zero). However, even on the textual model we observed that the *coi* metric consistently identified *attackable* words better than *co* and *abs*.
5) **RQ5:** The Fast Gradient Sign Method (FGSM) [13] is an existing popular approach to generate adversaries for MNIST images. In this technique, potential adversaries are generated by modifying the intensity of multiple pixels simultaneously such that the model's loss function

reduces. We used this method to generate an adversary (Figure 4) for the image of digit 3 in Figure 1(a)). We observed that in order to generate this adversary using FGSM, the value of almost every pixel on the shape of digit 3 had to be perturbed. On the other hand, we were able to generate attacks on the same image which involved the modification of just 1 pixel using DeepCheck (Figure 1(d)). Further, we also observed that the activation patterns or the path traced by the adversary generated using FGSM, differed a lot for the original validly labelled input. However, by construction the attack generated by DeepCheck preserves the activation patterns. The benefit of using semantic information from the model helps generate adversaries that the network views similarly as the original image but gives a different classification. Such an adversary could be useful in debugging the network and repairing the network. For example, a possible way to defend against such attacks would be a more focussed adversarial training; re-training the network with more inputs that follow the same path through the network.

V. Related Work

Recent independent work, developed concurrently with ours, proposes concolic testing for deep neural networks [31]. However their focus is on defining and achieving test coverage requirements, although their approach also produces adversarial images. In contrast we use symbolic execution for identifying important pixels and for specific 1-pixel and 2-pixel attacks, which target the same activation pattern as the original image; furthermore we use important pixels to focus the search for attackable pixels.

Other related recent techniques include formal methods [16], [17] and testing [26], [34] for deep neural networks. However none of previous work uses formal methods for important pixel identification, or more generally for explainability in neural networks. Furthermore, they do not check for attacks along the same activation patterns.

In our work, we used Z3 as the off-the-shelf constraint solver, which we inherited from a software analysis tool [25]. We note that other constraint solvers can be plugged in our analysis. For example, a good option is Reluplex [16], [17], which has been optimized specifically for the analysis of neural networks with ReLU activations. However, our methodology is general and can be in principle applied to other networks with linear units, such as Convolutional Neural Networks, which can not be handled by Reluplex. Another option is to use linear programming as in [31], since for a fixed activation pattern, the problem to be solved becomes linear. However linear solvers may behave unexpectedly when no solution exists, and can give unsound results due to overflow [35].

To our knowledge, existing approaches for testing, formal verification and attribution have not been applied to textual models. The rest of this section describes existing techniques related to attribution or explainability in neural networks and also existing techniques for adversarial example generation.

A. Techniques for Attribution

Despite the wide-spread adoption of neural networks, most deep neural network classifiers are black-boxes. It is crucial to understand the reasons behind the predictions of these classifiers in order to build trust in the model. Therefore, a number of techniques have been explored in the area of generating explanations for predictions. *Attribution* is a specific class of approaches, mostly applicable to image classification applications, where the technique attempts to assign "relevance", "contribution" to each input feature or pixel towards the classification decision. We describe below broad categories of attribution approaches.

Perturbation-based approaches alter the value of every input feature individually by a specific amount [37], re-run the network on the input and then measure the difference in the output value. However, these techniques tend to be slow and the computation time increases with the number of features. *Gradient-based approaches* [28]) compute the attributions of every feature in a single forward and backward pass of the network on a given input. They compute the signed partial derivative of the output w.r.t each input variable and multiple it by the input value to determine the impact of that variable on the output. *Integrated-gradients* [32]) proposed an approach that take an average of the attributions calculate along a linear path from a baseline (user-defined) until the given input. *Saliency maps* [29] consider the absolute value of the partial derivatives of the output w.r.t each input variable in order to identify pixels that can perturbed the least to observe a sizable change in the output value.

B. Techniques for adversarial attack generation

It has been observed that state-of-the-art networks are highly vulnerable to *adversarial perturbations*: given a correctly-classified input x, it is possible to find a new input x' that is very similar to x but is assigned a different label [33]. Goodfellow et al. [14] introduced the Fast Gradient Sign Method for crafting adversarial perturbations using the derivative of the model's loss function with respect to the input feature vector. They show that NNs trained for the MNIST and CIFAR-10 classification tasks can be fooled with a high success rate. An extension of this approach applies the technique in an iterative manner [11]. Jacobian-based Saliency Map Attack (JSMA) [24] proposed a method for targeted misclassification by exploiting the forward derivative of a NN to find an adversarial perturbation that will force the model to misclassify into a specific target class. Carlini et. al. [5] recently proposed an approach that could not be resisted by state-of-the-art networks such as those using defensive distillation. Their optimization algorithm uses better loss functions and parameters (empirically determined) and uses three different distance metrics.

The DeepFool [23] technique simplifies the domain by considering the network to be completely linear. They compute adversarial inputs on the tangent plane (orthogonal projection) of a point on the classifier function. They then introduce non-linearity to the model, and repeat this process until a true adversarial example is found. Deep Learning Verification (DLV) [16] is an approach that defines a region of safety around a known input and applies SMT solving for checking robustness. They consider the input space to be discretized and alter the input using manipulations until it is at a minimal distance from the original, to generate possibly-adversarial inputs. DeepSafe [15] is an approach that first applies a label-guided clustering algorithm on inputs with known labels to identify input regions that can be expected to be consistently labeled. It then employs the Reluplex solver [17] to verify that the all possible inputs within a given region are assigned the same label by the network. DeepRoad [38] introduces an unsupervised learning technique based on DNNs themselves for validating DNN-based autonomous drivers.

VI. Conclusion

We described a symbolic execution approach for the analysis of neural networks. Two analyses were presented: 1) to identify important inputs that can explain the classification decisions made by a neural network; and 2) to create attacks by constraint solving, guided by important inputs. The two analyses apply in synergy and provide a more scalable approach to finding attacks. An experimental evaluation using a MNIST model and a textual model demonstrates that the usefulness of the approach. For the future, we plan to evaluate our technique on larger networks that have higher accuracy; we are working on optimizing our tools to achieve this goal.

Acknowledgments

This work was partially supported by National Science Foundation NSF grant nos. CCF-1704790 and CCF-1718903.

References

[1] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *SIGSOFT FSE*. ACM, 2012, p. 59.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, Jul. 2011.

[3] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[4] C. Cadar and D. R. Engler, "Execution generated test cases: How to make systems code crash itself," in *SPIN*, 2005.

[5] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE S&P*, 2017.

[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *IEEE S&P*, 2012, pp. 380–394.

[7] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE TSE*, vol. 2, no. 3, 1976.

[8] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *USENIX Security*, 2013.

[9] L. de Moura and N. Bjorner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

[10] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *ISSTA*, 2007.

[11] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, "Adversarial machine learning at scale," 2016, technical Report. http://arxiv.org/abs/1611.01236.

[12] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*, 2005.

[13] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[14] ——, "Explaining and harnessing adversarial examples," 2014, technical Report. http://arxiv.org/abs/1412.6572.

[15] D. Gopinath, G. Katz, C. S. Pasareanu, and C. Barrett, "DeepSafe: A data-driven approach for checking adversarial robustness in neural networks," 2017, https://arxiv.org/abs/1710.00486.

[16] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *CAV*, 2017.

[17] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *CAV*, 2017.

[18] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS*, 2003.

[19] J. C. King, "Symbolic execution and program testing," *CACM*, vol. 19, no. 7, 1976.

[20] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[22] "The MNIST database of handwritten digits Home Page," http://yann.lecun.com/exdb/mnist/.

[23] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: A simple and accurate method to fool deep neural networks," in *CVPR*, 2016.

[24] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *EuroS&P*, 2016.

[25] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta, "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013. [Online]. Available: https://doi.org/10.1007/s10515-013-0122-2

[26] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *SOSP*, 2017.

[27] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/SIGSOFT FSE*, 2005.

[28] A. Shrikumar, P. Greenside, A. Shcherbina, and A. Kundaje, "Not just a black box: Learning important features through propagating activation differences," *CoRR*, 2016.

[29] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *CoRR*, 2013.

[30] J. Su, D. V. Vargas, and S. Kouichi, "One pixel attack for fooling deep neural networks," *CoRR*, vol. abs/1710.08864, 2017.

[31] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," *arXiv preprint arXiv:1805.00089*, 2018.

[32] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *ICML*, 2017.

[33] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013, technical Report. http://arxiv.org/abs/1312.6199.

[34] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 303–314.

[35] Y. Vizel, A. Nadel, and S. Malik, "Solving linear arithmetic with sat-based model checking," in *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, 2017, pp. 47–54.

[36] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically generating malicious disks using symbolic execution," in *IEEE S&P*, 2006.

[37] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *ECCV*, 2014.

[38] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems," in *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.