Inferring Data Preconditions from Deep Learning Models for Trustworthy Prediction in Deployment

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

Hongyang Gao
Dept. of Computer Science
Iowa State University
Ames, IA, USA
hygao@iastate.edu

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

ABSTRACT

Deep learning models are trained with certain assumptions about the data during the development stage and then used for prediction in the deployment stage. It is important to reason about the trustworthiness of the model's predictions with unseen data during deployment. Existing methods for specifying and verifying traditional software are insufficient for this task, as they cannot handle the complexity of DNN model architecture and expected outcomes. In this work, we propose a novel technique that uses rules derived from neural network computations to infer data preconditions for a DNN model to determine the trustworthiness of its predictions. Our approach, *DeepInfer* involves introducing a novel abstraction for a trained DNN model that enables weakest precondition reasoning using Dijkstra's Predicate Transformer Semantics. By deriving rules over the inductive type of neural network abstract representation, we can overcome the matrix dimensionality issues that arise from the backward non-linear computation from the output layer to the input layer. We utilize the weakest precondition computation using rules of each kind of activation function to compute layer-wise precondition from the given postcondition on the final output of a deep neural network. We extensively evaluated *DeepInfer* on 29 real-world DNN models using four different datasets collected from five different sources and demonstrated the utility, effectiveness, and performance improvement over closely related work. *DeepInfer* efficiently detects correct and incorrect predictions of high-accuracy models with high recall (0.98) and high F-1 score (0.84) and has significantly improved over the prior technique, *SelfChecker*. The average runtime overhead of *DeepInfer* is low, 0.22 sec for all the unseen datasets. We also compared runtime overhead using the same hardware settings and found that *DeepInfer* is 3.27 times faster than *SelfChecker*, the state-of-the-art in this area.

CCS CONCEPTS

• **Software and its engineering → Specification languages**; • **Computing methodologies → Machine learning**.

KEYWORDS

Deep neural networks, weakest precondition, trustworthiness

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623333

ACM Reference Format:
Shibbir Ahmed, Hongyang Gao, and Hridesh Rajan. 2024. Inferring Data Preconditions from Deep Learning Models for Trustworthy Prediction in Deployment. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623333

1 INTRODUCTION

Deep neural networks (DNN) are widely utilized nowadays, including in safety-critical systems. A DNN is trained on some data (training data), tested on possibly separate data (test data), and deployed in production, where they predict output for unseen data. A major challenge is: can we trust the output of a trained DNN on unseen data? Prior work has referred to these circumstances as data corruption bugs [37, 38] or conformal constraint violation [26, 28].

Prior research on the specification and verification of DNNs has focused on creating abstract representations for the verification of properties such as robustness and fairness [11, 14, 29, 35, 42, 43, 49, 60, 62, 65, 69]. However, these works have not addressed the questions of the trustworthiness of DNN outputs [72] on unseen data. Recent studies [26, 28] have explored techniques for discovering constraints, but they do not consider the DNN's structure in determining these constraints. In particular, the conformance constraints approach [26] uses the training dataset to establish a "safety envelope" that characterizes the inputs for which the model is expected to make trustworthy predictions. However, this work does not examine whether those conformation constraint violations of the safety envelope can determine correct or incorrect predictions with unseen data in the deployment stage. Our work fills this research gap. While many classifiers generate a confidence measure in addition to their class predictions, these measures are often unreliable due to inappropriate calibration [40, 48] and may not be sufficient to indicate trust in the classifier's prediction. In particular, the application of an activation function to raw numeric prediction values can lead to confidence measures that are not well-calibrated, making them difficult to determine whether the prediction with unseen data during deployment is correct or incorrect.

Recently, Xiao *et al.* proposed a technique, *SelfChecker* [72] that computes the similarity between layer features of test instances and the samples in the training set, using kernel density estimation (KDE) to detect misclassifications by the model in deployment. This technique has limitations, such as being restricted to the capability of computing density function from specific training and test data and the selected combination of layers with certain activation functions. Therefore, *SelfChecker* incurs a significant runtime overhead to compute KDEs for different combinations of layers for

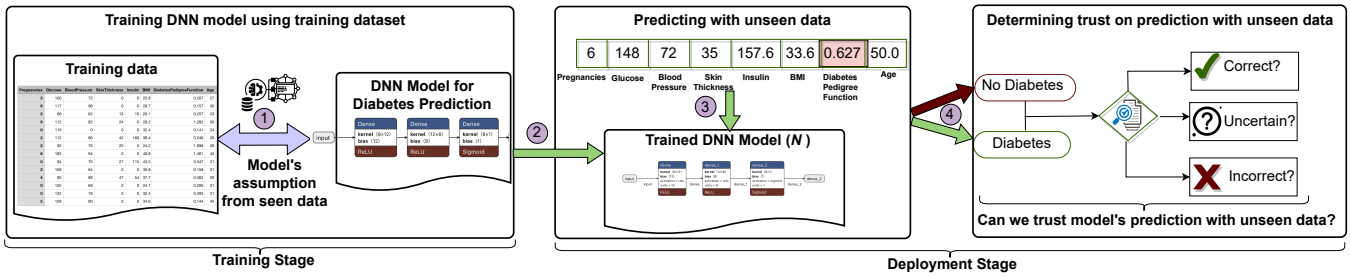

Figure 1: An example motivating how we can trust a model's prediction with unseen data in the deployment stage

each class in training and deployment modules for all training and test datasets. We address these shortcomings of the state-of-the-art techniques and aim to develop a technique that infers DNN model's assumption on training data and utilizes that inferred assumption during the deployment stage to determine correct or incorrect prediction, therefore implying trust in prediction with unseen data.

In this work, we provide a novel approach *DeepInfer* for reasoning about a DNN's prediction with unseen data by inferring data preconditions from the DNN model, i.e., structure of the DNN and trained parameters. The technical contributions of our approach include: a novel abstraction of DNN, including conditions, a weakest precondition (*wp*) calculus [34] for DNNs, and an algorithm that utilizes derived rules from the DNN abstraction and layer-wise computations to infer data preconditions and determine the model's correct or incorrect prediction. Starting with the conditions that should hold on the output of the DNN (*postconditions*), our *wp* rules provide mechanisms to compute conditions on the input of that layer (*preconditions*). Since the output of one layer (N) is fed to the input of the next layer ($N+1$) in a DNN, our approach then uses the preconditions of the $N+1$ layer as postconditions of the previous layer N. The precondition of the first layer, also called the input layer, in the DNN are *data preconditions*. The challenge in formulating *wp* rules lies in handling multiple layers with hidden non-linearities due to the architecture of the DNNs.

To evaluate our approach, we utilize 29 real-world models and 4 different datasets collected from prior research [9, 14, 64, 76] and Kaggle [41] to answer three research questions. We investigated whether data precondition violations determine incorrect model prediction. We also measure how effective *DeepInfer* is to imply trustworthiness in the model's prediction and compare against closely related work using their evaluation metrics [72]. We determine the performance, especially the runtime overhead of *DeepInfer* and compared it with the state-of-the-art using unseen data during deployment. Our key results are: ***DeepInfer* implies that data precondition violations and incorrect model prediction are highly correlated (**$pcc = 0.88$**)**, where *pcc* denotes Pearson correlation coefficient. Also, **the data precondition satisfaction and correct model prediction are strongly correlated (**$pcc = 0.98$**)**. ***DeepInfer* effectively implies the correct and incorrect prediction of higher accuracy models with recall (0.98) and F-1 score (0.84), compared to prior work *SelfChecker* with recall (0.59) and F-1 score (0.52)**. The average runtime overhead of *DeepInfer* is fairly minimal (0.22 sec for the entire test data). **Our proposed approach, *DeepInfer* is 3.27 times faster during deployment than *SelfChecker*, state-of-the-art in this area.**

In summary, this work makes the following contributions:

- a novel abstraction for trained DNN that incorporates pre and postconditions as predicate vectors for each layer;
- a weakest precondition calculus for the DNN abstraction that overcomes challenges due to non-linearities introduced by the DNN architecture;
- a novel technique for computing data preconditions from DNN models after training and utilizing those inferred preconditions for implying trust in the model's prediction during the deployment stage ;
- a detailed evaluation with publicly available datasets and models to demonstrate the utility, efficiency, and performance of *DeepInfer* with an open-source implementation [7] that can be leveraged by future research in explainable software engineering for machine learning.

2 MOTIVATION

We are aware that a DNN model's prediction could be correct or incorrect, but it is important to know how trustworthy the model's prediction is for unseen data during the deployment stage. To motivate our objectives, let us consider a deep neural network model in Fig. 1. The first layer, i.e., the DNN model's input layer, receives the input from training data, compiles it and produces the output (1). Then, the next layers receive the output from the previous ones as input. The model compiles the input data, evaluates it, predicts the output, and delivers it to the deployment stage (2). This model has been trained for the PIMA diabetes dataset with eight features for whether a patient has diabetes. Although the model's accuracy is 77%, when we get the output from the model, we do not really know how confident the model is for that output. In some cases, the model could be confidently incorrect. So, this model's prediction with an unseen data during the deployment stage might be correct or incorrect. For instance, during the deployment stage, unseen data is fed to the trained DNN model (3), which predicts whether the patient with that particular data point has diabetes or no diabetes (4). It is necessary to determine whether the model's prediction is correct and to trust this prediction or its prediction is incorrect and not to trust it with such unseen data points during the deployment stage. The growing prevalence of Deep Neural Networks (DNNs) in critical domains highlights the importance of ensuring the trustworthiness of their outputs. Despite their high accuracy, DNNs are still prone to prediction errors, and in applications such as autonomous vehicles and medical diagnosis, etc. It is reported that Uber's fatal self-driving crash was caused by software detecting objects on the

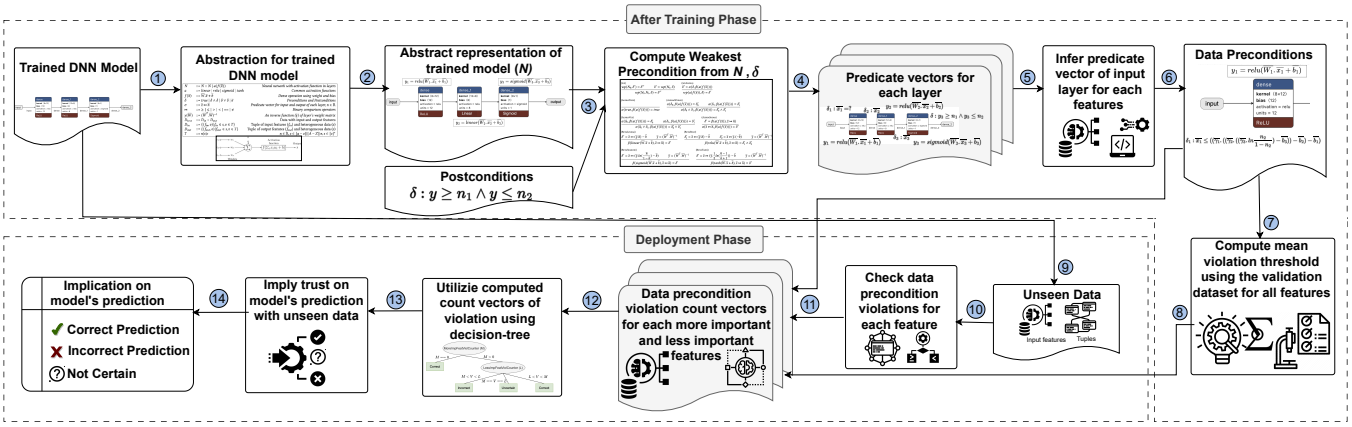

Figure 2: Overview diagram depicting the technique of data precondition inference from a trained DNN model after the training phase and how those are utilized in the deployment stage for implying trust in the model's prediction using unseen data

road [1] and AI models for health care that predict disease are not as accurate as suggested in reports [2]. Therefore, making the DNN black-box model explainable and determining correct, incorrect, or uncertain predictions during deployment is crucial.

Problem formulation: Given a trained DNN model and an unseen data instance, our goal is to derive preconditions from a trained DNN model's assumptions about the training data after the training stage, and leverage inferred data preconditions from the model to precisely determine whether a prediction by the DNN model with unseen data during deployment is correct or incorrect, or uncertain. By addressing the challenges posed by non-linear computation functions in DNN model and the variability of weights, biases, inputs, and outputs, our work aims to provide an efficient solution and significantly improved technique over state-of-art for ensuring trust in the DNN model's prediction in real-world applications.

3 *DEEPINFER* APPROACH

We present an overview diagram in Fig. 2 illustrating our proposed technique *DeepInfer*. The top portion of the diagram depicts how data preconditions are inferred from a trained DNN model after the training phase. In the bottom portion, we depict how the inferred data preconditions are utilized for determining the trustworthiness of the model's prediction using unseen data during deployment. At first, we utilize a trained DNN model for the novel abstraction with layers and activation functions incorporating preconditions and postconditions (1). Then, we represent a neural network with activation function operations inside layers (2). We compute the weakest preconditions from the abstract representation of the trained model (N) and postcondition (δ) (3). Then, we determine the predicate vectors for each layer utilizing the computed weakest preconditions from layer-wise operations (4). From (5), we infer the input layer's predicate vector for each feature. Therefore, we obtain the data preconditions using the trained model once after the training phase (6). Then, we compute mean data precondition violations for all features using the entire validation dataset, which serve as a threshold (7). In the deployment phase, *DeepInfer* utilizes the trained DNN model and obtained data preconditions for determining trust in the model's prediction with an unseen data point (9). Next, we check the data precondition violations (10) for each feature using the violation threshold (8) for that data point (11). Furthermore, we utilize the computed count vectors of the violation using a decision-tree-based approach (12). To that extent, we determine the trustworthiness of the model's prediction with unseen data (13). Finally, *DeepInfer* determines whether the model's prediction is correct and we can trust it or incorrect and not certain, and we can not rely on that prediction with unseen data during the deployment stage (14).

3.1 Abstract representation of a DNN model

We propose a novel abstraction for trained DNNs that incorporates pre and postconditions as predicate vectors for each layer. Let us consider the following grammar for representing DNN depicted in Fig. 3.

N	$::= N :: N \mid a(f(\overline{x}))$	*Neural network with activation function*
a	$::= linear \mid relu \mid sigmoid \mid tanh$	*Common activation functions*
$f(\overline{x})$	$::= \overline{W}.\overline{x} + \overline{b}$	*Dense operation using weight and bias*
δ	$::= true \mid \delta \wedge \delta \mid \delta \vee \delta \mid \overline{\sigma}$	*Preconditions and Postconditions*
σ	$::= \overline{z} \bowtie \overline{n}$	*Predicate vector of layer's input/output* $(\overline{z})$, $n \in \mathbb{R}$
$\bowtie$	$::= \geq \mid \leq \mid > \mid < \mid == \mid \neq$	*Binary comparison operators*
$\gamma(\overline{W})$	$::= (\overline{W}^T.\overline{W})^{-1}$	*Inverse function* $(\overline{\gamma})$ *of layer's weight matrix*
D_{test}	$::= D_{in} :: D_{out}$	Data with input and output features
D_{in}	$::= \{(f_{in}, v) \mid f_{in} \in s, v \in T\}$	Tuple of input features (f_{in}) and data (v)
D_{out}	$::= \{(f_{out}, v) \mid f_{out} \in s, v \in T\}$	Tuple of output features (f_{out}) and data (v)
T	$::= n \mid s \mid c$	$n \in \mathbb{R}, c \in [a-z] \mid [A-Z] \mid n, s \in [c]^*$

Figure 3: Grammar representing Neural network, preconditions, and postconditions

Let us consider the `Dense` layer computation denoting $f(\overline{x})$. In the grammar, we denote N as a neural network with activation function $a(f(\overline{x}))$ in layers. In this computation, the function is based on the neuron's weights, and bias where one weight is assigned to each component of the input $(\overline{x})$ with corresponding weight $(\overline{W})$ and bias $(\overline{b})$ in each layer. We consider some common activation functions [69] used in deep learning programs such as `linear, ReLU, sigmoid, tanh`. We consider each layer's output and input vector as $\overline{z}$ and the predicate as $\bowtie \overline{n}$, where $n \in R$ and $\bowtie$ represent the logical comparison operators. Here, $\overline{\gamma}$ denotes an

inverse function of a layer's weight matrix nonlinear computation. We represent the test dataset (D_{test}) as a tuple of features and data.

3.2 Computing weakest preconditions from abstract representation of a DNN model

To compute the weakest preconditions from the abstract representation of a DNN model (N), we consider a postcondition δ as the expected DNN model's output. As edges from one layer connect the neural network to another layer, we denote $\overline{z}$ as the input/output vector for the intermediate layers. We consider $\overline{y}$ the output of the last layer, and n is the prediction interval for a DNN model's prediction. Here, $\bar{x}$ is the input vector to the first layer. It is considered as data precondition. We follow Dijkstra's predicate transformer semantics rules in the form of $wp(N, \delta)$, where N is a program statement involving a DNN layer computation using activation functions as represented in the grammar, and δ is a postcondition on the program state. This transformer rule defines the weakest predicate, which holds the model statement before executing N to guarantee that the postcondition δ holds after N terminates. *DeepInfer* computes data precondition for a model using the defined rules in Fig. 4. We compute the data precondition, which is obtained recursively by following these rules from the last layer until the first layer of a DNN model. Therefore, the computation of the data precondition from a DNN model is done recursively for a given representation N from the DNN model and postcondition δ using the rules illustrated in Fig. 4. Here, the rules (WP), (WPALPHA) represent recursion over inductive type N by the function wp, eventually satisfying base cases of N. These base cases of wp use α to compute the precondition where α does recursion over the cases of the inductive type δ represented using rules (WPALPHATRUE), (WPALPHAWEDGE), (WPALPHAVEE), (WPALPHASIGMA) illustrated in Fig. 4. Again, base cases of α use β to compute the wp for the cases of the activation function (a). For instance, for `ReLU` activation function, we compute β using the computation with weight and bias of a layer as follows,

$$relu(f(\overline{x})) = f(relu(\overline{W}.\overline{x} + \overline{b})) = \begin{cases} 0, & (\overline{W}.\overline{x} + \overline{b}) < 0 \\ \overline{W}.\overline{x} + \overline{b}, & (\overline{W}.\overline{x} + \overline{b}) \geq 0 \end{cases}$$

We solve this non-linear equation of $relu(\overline{W}.\overline{x}+\overline{b})$ for postcondition ($\overline{z} \bowtie \overline{n}$) and obtain the precondition as stated in the (BETARELU) rule in Fig. 4. Similary, for other kinds of activation functions, we have derived β rules e.g., (BETALINEAR), (BETASIGMOID), (BETATANH) rules illustrated in Fig. 4. The derivation details of each kind of those rules are in the appendix of open-source repository [7].

$$\beta(relu(W.\overline{x}+b), \overline{z} \bowtie \overline{n}) \equiv \overline{z} \bowtie ((\overline{\gamma}.\overline{n})-\overline{b}) \wedge \overline{z} \bowtie \overline{\gamma}.(-\overline{b}), \overline{\gamma} = (\overline{W}^T.\overline{W})^{-1}$$

Next, we describe the challenges towards layer-wise weakest precondition reasoning using a DNN model.

3.3 Layer-wise weakest precondition reasoning

In order to obtain wp by asserting the model statement using postcondition from layer to layer, there are some challenges. First, the layer function computation using the activation function is not always linear. Different non-linear activation functions operate using weight and bias along with the input in each layer computation. For instance, `sigmoid` activation function computes ($\sigma(x) = \frac{1}{1+e^{-x}}$), `tanh` activation function computes ($tanh(x) = \frac{2}{1+e^{-2x}}$ -1), `ReLU` activation function computes ($relu(x) = x, x \geq 0 | 0, x < 0$), `ELU` activation function computes ($elu(x) = x, x \geq 0 | e^x - 1, x < 0$), `softmax` activation function computes ($softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$) [69], etc.

Second, there is a challenge to tackle the variability of the matrix dimension of weight, bias, input, and output in each layer. For instance, an example model (in Fig. 5) contains 3 `Dense` layers which perform `linear, linear, sigmoid` activation function computation using weight and bias vector with input in each layer. To obtain the layer-wise wp, the dimension of weight and bias matrices should be taken into account. The dimension of weight matrices varies from layer to layer in the network. As the weight vector ($\overline{W}$) is multiplied by the input vector ($\overline{X}$), the dimension must be consistent with the bias vector ($\overline{b}$) and output ($\overline{y}$) in forward propagation. In terms of backward computation, it is challenging to get the appropriate matrix dimension on the precondition of input data in each layer. In Fig. 5, the dimension of weight, bias, input, and output of last layer is (1×8), (1×1), (8×1), (1×1) respectively. In the second layer, the dimension of weight, bias, input, and output is (8×12), (8×1), (12×1), (8×1), respectively. In the first layer, the dimension of weight, bias, input, and output is (12×8), (12×1), (8×1), (12×1), respectively. If we assert using postcondition with a single dimension of δ, as a data precondition in the first layer should be a dimension of 8×1 in this scenario. We encounter here that the weight, bias, input, and output of each layer appear non-linearly in the equations of the activation function, where there are nonlinear constraints among the parameters. To address these challenges, we have adopted the least square solution [39] for nonlinear activation computation. One of our contributions is to derive wp rules for each kind of activation function (shown in Fig. 4) for layer-wise weakest precondition reasoning.

Next, we describe the weakest precondition computation of a DNN model to infer data preconditions of the input layer using the derived rules (in Fig. 4). Our approach is generalized to a DNN with any number of hidden layers with linear or non-linear activation functions. For simplicity, we demonstrate the wp computation process using derived rules with a canonical example DNN model.

3.4 Infer data preconditions of the input layer

For computing the weakest precondition using DNN models, we take the statements from the model structure consisting of layers with input dimensions, number of output, activation function, etc. We consider the prediction interval (n) as the postcondition. The rationale behind choosing prediction interval as a postcondition to DNN classification or regression model is that it [46] provides how good the model prediction is. Also, the prediction interval helps gauge the weight of evidence available when comparing models. Prediction intervals facilitate trade-offs between models favoring less complex or more interpretable models [17].

To infer data preconditions, starting from the last layer statement, we assert using the wp equation to determine the weakest precondition. The equation of *Dense* layer is as follows:

$$output = activation(dot(input, weight) + bias)$$

So, for given postcondition $\delta : y \leq n$, the statement S_3 can be written for output of last layer (y_3) with corresponding weight ($\overline{w_3}$)

$$\text{(WP)}\ \frac{wp(N_0,\delta')=\delta''\quad \delta'=wp(N_1,\delta)}{wp(N_0.N_1,\delta)=\delta''} \qquad \text{(WPALPHA)}\ \frac{\delta'=\alpha(\delta,\beta(a(f(\overline{x}))))}{wp(a(f(\overline{x})),\delta)=\delta'} \qquad \text{(ALPHATRUE)}\ \frac{}{\alpha(true,\beta(a(f(\overline{x}))))=true}$$

$$\text{(ALPHAWEDGE)}\ \frac{\alpha(\delta_0,\beta(a(f(\overline{x}))))=\delta_0'\quad \alpha(\delta_1,\beta(a(f(\overline{x}))))=\delta_1'}{\alpha(\delta_0\wedge\delta_1,\beta(a(f(\overline{x}))))=\delta_0'\wedge\delta_1'}$$

$$\text{(ALPHAVEE)}\ \frac{\alpha(\delta_0,\beta(a(f(\overline{x}))))=\delta_0'\quad \alpha(\delta_1,\beta(a(f(\overline{x}))))=\delta_1'}{\alpha(\delta_0\vee\delta_1,\beta(a(f(\overline{x}))))=\delta_0'\vee\delta_1'} \qquad \text{(ALPHASIGMA)}\ \frac{\delta'=\beta(a(f(\overline{x})),\overline{z}\bowtie\overline{n})}{\alpha(\overline{z}\bowtie\overline{n},\beta(a(f(\overline{x}))))=\delta'}$$

$$\text{(BETARELU)}\ \frac{\delta_1'=\overline{z}\bowtie(\overline{\gamma}.\overline{n})-\overline{b}\quad \delta_2'=\overline{z}\bowtie\overline{\gamma}.(-\overline{b})\quad \overline{\gamma}=(\overline{W}^T.\overline{W})^{-1}.(\overline{W^T})}{\beta(relu(\overline{W}.\overline{x}+\overline{b}),\overline{z}\bowtie\overline{n})=\delta_1'\wedge\delta_2'}$$

$$\text{(BETALINEAR)}\ \frac{\delta'=\overline{z}\bowtie(\overline{\gamma}.\overline{n})-\overline{b}\ \ \overline{\gamma}=(\overline{W}^T.\overline{W})^{-1}.(\overline{W^T})}{\beta(linear(\overline{W}.\overline{x}+\overline{b}),\overline{z}\bowtie\overline{n})=\delta'} \qquad \text{(BETASIGMOID)}\ \frac{\delta'=\overline{z}\bowtie(\overline{\gamma}.ln(\frac{\overline{n}}{1-\overline{n}})-\overline{b})\ \ \overline{\gamma}=(\overline{W}^T.\overline{W})^{-1}.(\overline{W^T})}{\beta(sigmoid(\overline{W}.\overline{x}+\overline{b}),\overline{z}\bowtie\overline{n})=\delta'}$$

$$\text{(BETATANH)}\ \frac{\delta'=\overline{z}\bowtie(\overline{\gamma}.\frac{1}{2}ln(\frac{\overline{n}-1}{\overline{n}+1})-\overline{b})\ \ \overline{\gamma}=(\overline{W}^T.\overline{W})^{-1}.(\overline{W^T})}{\beta(tanh(\overline{W}.\overline{x}+\overline{b}),\overline{z}\bowtie\overline{n})=\delta'}$$

Figure 4: Rules for computing wp over inductive type N, α over inductive type δ, β over inductive type $a(f(\overline{x}))$

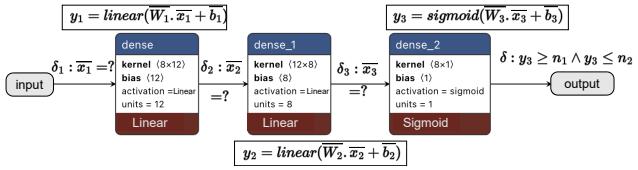

Figure 5: Data precondition (δ_1) computation from an example DNN model (N) with 3 layers and postcondition (δ)

and bias ($\overline{b_3}$) as,

$$y_3 = sigmoid(\overline{w_3^T}.\overline{x_3}+\overline{b_3})$$

We have `Dense` layers with `linear, linear, sigmoid` activation functions for this example. Now, for given neural network (N) and postcondition (δ),

$$N : linear(\overline{W_1}.\overline{x_1}+\overline{b_1}).linear(\overline{W_2}.\overline{x_2}+\overline{b_2}).sigmoid(\overline{W_3}.\overline{x_3}+\overline{b_3});$$
$$\delta : y_3 \geq n_1 \wedge y_3 \leq n_2$$

Our proposed technique is generalized to DNN models with multiple layers. For example, a DNN model presented in Fig. 5 has 3 layers and different activation functions. In that model, the output layer has a single class, i.e., the output value $y \in \mathbf{R}$. The given postcondition is an instance of ($\delta \wedge \delta$) and will be in the range between $[n_1, n_2]$. Now, we utilize wp rules over N and δ using (WP), (WPALPHA) rules to get the precondition for this multiple layer neural network as follows,

$$\delta_1 = wp(N,\delta) = wp(N_0.N_1,\delta) \equiv$$
$$wp(linear(\overline{W_1}.\overline{x_1}+\overline{b_1}).linear(\overline{W_2}.\overline{x_2}+\overline{b_2}).sigmoid(\overline{W_3}.\overline{x_3}+\overline{b_3}),\delta)$$
$$\delta_1 = wp(linear(\overline{W_1}.\overline{x_1}+\overline{b_1},\delta_2);$$
$$\delta_2 = wp(linear(\overline{W_2}.\overline{x_2}+\overline{b_2}).sigmoid(\overline{W_3}.\overline{x_3}+\overline{b_3}),\delta);$$
$$\delta_2 = wp(linear(\overline{W_2}.\overline{x_2}+\overline{b_2}),\delta_3); \delta_3 = wp(sigmoid(\overline{W_3}.\overline{x_3}+\overline{b_3}),\delta)$$

Then, we apply (WPALPHASIGMA), (WPALPHAWEDGE), (BETASIGMOID) rules consecutively to get the precondition as follows,

$$\delta_3 = wp(sigmoid(\overline{W_3}.\overline{x_3}+\overline{b_3}),\delta) \equiv \alpha(\delta_3,\beta(sigmoid(\overline{W_3}.\overline{x_3}+\overline{b_3})))$$
$$\equiv \beta(sigmoid(\overline{W_3}.\overline{x_3}+\overline{b_3}),\delta_3) \equiv \beta(sigmoid(\overline{W_3}.\overline{x_3}+\overline{b_3}),y_3 \geq n_1 \wedge y_3 \leq n_2)$$
$$\equiv \overline{x_3} \geq ((\overline{\gamma_3}.ln\frac{n_1}{1-n_1})-\overline{b_3}) \wedge \overline{x_3} \leq ((\overline{\gamma_3}.ln\frac{n_2}{1-n_2})-\overline{b_3})$$

Here, $\overline{x_3}$ is an array of input that has been obtained from the second layer and fed into the third layer, and the predicate of $\overline{x_3}$ denotes the precondition of the data in layer 3, which is a postcondition of layer 2. Here, $\overline{\gamma_3}$ is an inverse function of the layer's weight matrix ($\overline{W_3}$). Then, we obtain δ_2 similarly using the wp rules (WPALPHA), (WPALPHAWEDGE), (BETALINEAR) consecutively,

$$\delta_2 = wp(linear(\overline{W_2}.\overline{x_2}+\overline{b_2}),\delta_3)$$
$$\equiv \alpha(\delta_3,\beta(linear(\overline{W_2}.\overline{x_2}+\overline{b_2}))) \equiv \beta(linear(\overline{W_2}.\overline{x_2}+\overline{b_2}),\delta_3)$$
$$\equiv \beta(linear(\overline{W_2}.\overline{x_2}+\overline{b_2}),\overline{x_3} \geq ((\overline{\gamma_3}.ln\frac{n_1}{1-n_1})-\overline{b_3})\wedge$$
$$\overline{x_3} \leq ((\overline{\gamma_3}.ln\frac{n_2}{1-n_2})-\overline{b_3})) \equiv \overline{x_2} \geq ((\overline{\gamma_2}.((\overline{\gamma_3}.ln\frac{n_1}{1-n_1})-\overline{b_3}))-\overline{b_2})\wedge$$
$$\overline{x_2} \leq ((\overline{\gamma_2}.((\overline{\gamma_3}.ln\frac{n_2}{1-n_2})-\overline{b_3}))-\overline{b_2}), \overline{\gamma_2} = (\overline{W_2^T}.\overline{W_2})^{-1}.(\overline{W_2^T})$$

In this step, we obtain the precondition which is an array of the input ($\overline{x_2}$) that has been obtained from the first layer and fed into the second layer, and the predicate of $\overline{x_2}$ denotes the precondition of the input in layer 2, which is a postcondition of layer 1. After asserting with this postcondition, we obtain δ_1 similarly using the wp rules (WPALPHA), (WPALPHAWEDGE), (BETALINEAR),

$$\delta_1 = wp(linear(\overline{W_1}.\overline{x_1}+\overline{b_1}),\delta_2) \equiv \alpha(\delta_2,\beta(linear(\overline{W_1}.\overline{x_1}+\overline{b_1})))$$
$$\equiv \beta(linear(\overline{W_1}.\overline{x_1}+\overline{b_1}),\delta_2)$$
$$\equiv \overline{x_1} \geq ((\overline{\gamma_1}.((\overline{\gamma_2}.((\overline{\gamma_3}.ln\frac{n_1}{1-n_1})-\overline{b_3}))-\overline{b_2})-\overline{b_1})\wedge$$
$$\overline{x_1} \leq ((\overline{\gamma_1}.((\overline{\gamma_2}.((\overline{\gamma_3}.ln\frac{n_2}{1-n_2})-\overline{b_3}))-\overline{b_2})-\overline{b_1})$$

Finally, we obtain the precondition, which is an array of the data ($\overline{x_1}$) for each feature that has been assumed by this DNN with multiple layers where $\overline{\gamma_1} = (\overline{W_1^T}.\overline{W_1})^{-1}.(\overline{W_1^T})$. In our proposed technique, the entire process of data precondition inference from a DNN model is automated and generalized for other models which is performed after the training stage. Next, we discuss how we utilize inferred data preconditions for determining the trustworthiness of the model's prediction using unseen data.

3.5 Implying trustworthiness on the model's prediction using inferred data preconditions

Regarding the design choice, we determine the data preconditions for the inputs to the first layer in a DNN model. These data preconditions for the inputs to a DNN model indicate the trained model's assumption about the data. Furthermore, these input data preconditions must hold true for the data before it is fed to the model, which is important for its prediction. Prior work regarding the conformance constraints approach [26] uses the training dataset to establish a "safety envelope" that characterizes the inputs and demonstrates that conformal constraint violation is related to a model's trustworthy predictions. We leverage a similar notion in our approach that the violation of obtained data preconditions for the input to a DNN model indicates the trustworthiness of the model's prediction.

The overall process has two parts shown in Algorithm 1. The procedure computeThreshold computes the violation threshold for input features using the validation set, and checkPrediction uses these computed values to check prediction for unseen data. Given the neural network representation N and the postcondition δ, the first step is to acquire the data preconditions (line 2), set of input features, and data points from the validation dataset D_{test} (lines 3-5). The algorithm proceeds by collecting feature-wise violations using the helper procedure on lines 11–18, which checks precondition violation for each input in the validation set and accumulates the precondition violations by features. Finally, we calculate the mean number of data precondition violations for all features (V), which serve as a threshold (on line 9). For the unseen data, procedure

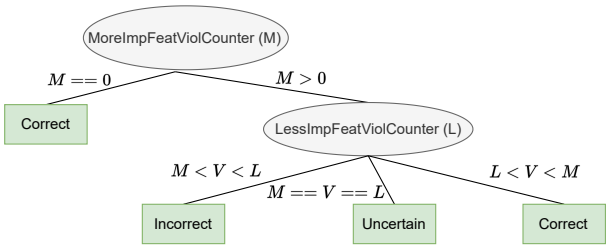

Figure 6: Utilizing computed count vectors of the data precondition violations using decision-tree

checkPrediction computes the violation count for each feature (line 21). Next, for each feature the procedure checks whether the number of violations are above (L) or below (M) the violation threshold. To be more specific regarding the design choice of the decision tree (in Fig. 6) of data preconditions violation, we have utilized more feature violations and fewer feature violations as indicative of the model's correct and incorrect prediction. The decision tree logic is in Fig. 6. First leaf (from the left) of this decision tree is immediate, if there are no more violations compared to the threshold then the model's prediction is correct. If $L == V == M$, then the procedure is unsure about the output of the model and therefore we assign it uncertain (leaf 3). If $L < V < M$, then there are more features for which the precondition violation is below the threshold and fewer features for which the violation is above. That means the overall violation is less, leading to correct prediction (leaf 4). Finally, if $M < V < L$, there are more precondition violations above the threshold, and thus the model output is incorrect (leaf 2). To be more specific regarding the design choice of the decision-tree of data preconditions violation is that we utilized the more feature violations and less feature violations as indicative of the model's correct and incorrect prediction.

Algorithm 1 Data Precondition Violation Procedure

```
1:  procedure computeThreshold(N, D_test, δ)
2:      δ' ← wp(N, δ)                          ▷ Obtain data precondition given N and postcondition δ
3:      f̄ ← dom(D_in)                          ▷ Set of input features, D_in ∈ D_test
4:      d ← range(D_in)                        ▷ Set of data points, v ∈ D_in
5:      wp_vDict ← ∅
6:      v̄ ← collectFeatureWiseViolations(d, f̄, δ')
7:      for each i ∈ |f̄| do
8:          wp_vDict ← wp_vDict ∪ {⟨i, v̄[i]⟩}
9:      V ← (1/n) Σ_{n=1}^{n=|f̄|} (v_wp) | v_wp ∈ wp_vDict   ▷ Mean violation threshold
10:     return V, δ', f̄
11: procedure collectFeatureWiseViolations(d, f̄, δ')
12:     v̄ ← 0                                  ▷ Violation count array indexed by features
13:     for each t̄ ϵ d do
14:         v̄_wp ← δ'(t̄)                       ▷ Check precondition violation for input
15:         for each i ∈ |f̄| do                ▷ Collect precondition violation for each feature
16:             if v̄_wp[i] then
17:                 v̄(i) ← v̄(i) + 1
18:     return v̄
19: procedure checkPrediction (d, V, δ', f̄)
20:     M, L ← 0, wp_warn ← ∅
21:     v̄ ← collectFeatureWiseViolations(d, f̄, δ')
22:     for each i ∈ |f̄| do
23:         if (v̄[i] ≤ V) then                 ▷ Compare violation count with threshold
24:             M ← M + 1
25:         else
26:             L ← L + 1
27:     wp_warn ← decisionTree(M, V, L)        ▷ Correct/incorrect/uncertain?
28:     return wp_warn
```

Time Complexity. The procedure checkPrediction doesn't compute over the DNN. It uses preconditions computed by the procedure computeThreshold that *runs once per DNN* after training. The time complexity of the procedure computeThreshold is dominated by the wp function, whose complexity is akin to the back-propagation algorithm of a FCNN. The time complexity is primarily determined by matrix multiplications, that has the complexity $O(n^{log_2 7})$ for Strassen's method [18]. The time complexity of wp is $O(|N| + n^{log_2 7})$ where, $|N|$ is the length of layers of model and n is the dimension of the weight matrix. The time complexity of checkPrediction is $O(|d|.|\overline{f}|)$ where $|d|$ is the size of unseen data and $|\overline{f}|$ is the number of features. So, our approach for inferring data precondition from a large DNN model with many layers is scalable because of quadratic time complexity.

4 EVALUATION

This section describes the evaluation of *DeepInfer*. First, we discuss the experimental setup in §4.1. Next, we describe research questions and present the results and discussion in §4.2.

4.1 Experiment

4.1.1 Benchmark. We have gathered four canonical real-world datasets from Kaggle competitions [41]. The train and test datasets are converted to numerical values if those are in any other data types during the data preprocessing stage. We have gathered models

intended for classification problems from the Kaggle and used by prior work [9, 14, 64, 76]. In table 1, we present the total number of features in a dataset, number of neurons, and layers of the models.

Table 1: DNN Benchmark for inferring data preconditions

Dataset	# Features	Model	Source	# Layers	# Neurons
Pima Diabetes [61]	8	PD1	Kaggle	3	221
		PD2	Kaggle	3	221
		PD3	Kaggle	3	221
		PD4	Kaggle	4	293
House Price [6]	10	HP1	Kaggle	3	273
		HP2	Kaggle	3	273
		HP3	Kaggle	3	273
		HP4	Kaggle	4	383
BankCustomer [4]	28	BM1	[14]	4	97
		BM2	[14]	4	65
		BM3	[9]	3	117
		BM4	[14]	5	318
		BM5	[14]	4	49
		BM6	[14]	4	35
		BM7	[14]	4	145
		BM8	[76]	7	141
		BM9	Kaggle	3	627
		BM10	Kaggle	3	627
		BM11	Kaggle	3	627
		BM12	Kaggle	4	1439
GermanCredit [5]	22	GC1	[14]	3	64
		GC2	[64]	3	114
		GC3	[14]	3	23
		GC4	[14]	4	24
		GC5	[76]	7	138
		GC6	Kaggle	3	2397
		GC7	Kaggle	3	2397
		GC8	Kaggle	3	2397
		GC9	Kaggle	4	2949

4.1.2 Prediction interval. We have adopted high-quality prediction intervals for deep learning models for classification and regression models from prior work [55]. Therefore, for the experimental evaluation, we selected a prediction interval (≥ 0.95) as the postcondition for determining the data precondition from a deep learning model.

4.1.3 Experimental Setup. To perform our experiments and evaluation, we implemented our techniques using *Python* and *Keras*. We have used mathematical packages (*numpy*, *pandas*) to compute the data precondition from a *Keras* model and to evaluate the implied trustworthiness of model's prediction using inferred data preconditions. We have conducted all the experiments on a machine with a 2 GHz Quad-Core Intel Core i7 and 32 GB 1867 MHz DDR3 RAM running the macOS 11.14.

4.1.4 Evaluation Metrics. : To determine the efficiency *DeepInfer*, we measure the Pearson Correlation Coefficient (*pcc*) following prior work [26]. We define true positive (TP), false positive (FP), false negative (FN), and true negative (TN) following prior work [72]. We also measure precision, recall, TPR, FPR, F-1 score following prior work [72] from TP, FP, and FN to determine the efficiency of our approach in predicting the correct prediction of a DNN model.

4.2 Results

4.2.1 Research Questions. To evaluate the utility, efficiency, and performance, we answer the following research questions:

RQ1(Utility): *Do data precondition violations imply incorrect model prediction, and data precondition satisfaction implies correct model prediction, i.e., to trust the model?*

We first obtain the preconditions on data for each feature using the respective model and dataset to measure the utility of data for implying the model's prediction. Then using Algorithm 1, we imply "Correct" or "Incorrect" or "Uncertain" prediction for unseen data based on data precondition violation and satisfaction for each feature. For RQ1, the model has been trained with the seen i.e., training data, and validated with the second portion of training data. Following the experimental procedure [72], we have used all the test datasets as unseen data. For evaluation purposes, we determine the ground truth from the actual label and the model's predicted label and we consider "Uncertain" prediction as "Incorrect".

RQ2 (Effectiveness): *How effective DeepInfer is to imply trustworthiness in the model's prediction compared to the prior approach?*

To determine the effectiveness of our proposed approach *DeepInfer*, we measure true positive, false positive, false negative, and true negative as discussed in §4.1.4 . We reported the false positive and true positive ground truth where "ActFP" denotes if the actual label and predicted label by a model are not equal and "ActTP" denotes if the actual label and predicted label by a model are equal. This suggests whether the model is properly trained or not and also explains how *DeepInfer* performs compared to the "ActFP" and "ActTP". We compare our approach with *SelfChecker* [72] using same 29 models and 4 datasets. We have compared our approach against *SelfChecker* [72] in terms of how effective each approach is in predicting DNN misclassifications in deployment. We have used the open-source implementation of *SelfChecker* and utilized the same hardware setup. We communicated with the authors to ensure their tool is applicable to these models and datasets.

RQ3 (Efficiency): *What is the performance of DeepInfer with respect to time, and what is the runtime overhead using unseen data during deployment compared to prior work?*

To compute the efficiency of our proposed technique, we compute the training time of all the models. We computed the runtime of *DeepInfer* and *SelfChecker* for all the models and all unseen datasets. We consider the runtime measure important for determining trust on the model's prediction with unseen data in the deployment stage for safety-critical issues. Considering resource constraints such as processing data and generating prediction timely and limited computing power or memory, it is crucial to ensure that models are suitable for deployment in safety-critical scenarios to prevent accidents or mitigate risks. For instance, a self-driving Uber car struck and killed a woman in March 2018 as an investigation [3] revealed that the model couldn't correctly predict her path and it needed to brake just 1.3 seconds before it struck her. Therefore, it is important to measure the runtime of such techniques.

4.2.2 Results and Analysis. In this section, we discuss the results and analysis for each of the research questions utilizing 4 different real-world tabular datasets with 29 different *Keras* real-world models (discussed in §4.1.1) targeting binary classification problems.

RQ1 (Utility): For RQ1, we present the results of all 29 real-world models for four different datasets in Table 2. We report the model's accuracy and the number of test instances. Then, we reported the total number of "Correct" and "Incorrect" labels for all the test datasets as the ground truth of the model's prediction and actual label. Next, we report the total number of data precondition

Table 2: *DeepInfer* implying correct and incorrect model prediction for unseen data

Dataset	Model	Accuracy	# Features	# Unseen data	Ground Truth		DeepInfer					Time (sec)
					# Correct	# Incorrect	# Violation	# Satisfaction	# Correct	# Incorrect	#Uncertain	
Pima Diabetes	PD1	77.98%	8	153	119	34	192	1032	108	43	2	0.67
	PD2	65.10%			99	54	0	1224	153	0	0	0.66
	PD3	65.49%			98	55	129	1095	74	79	0	0.65
	PD4	77.47%			111	42	132	1092	37	116	0	0.65
House Price	HP1	85.22%	10	292	147	145	341	2579	188	98	6	0.86
	HP2	89.77%			147	145	341	2579	188	98	6	0.83
	HP3	45.45%			145	147	0	2920	292	0	0	0.97
	HP4	87.50%			147	145	188	2732	107	184	1	0.87
BankCustomer	BM1	81.10%	28	2116	1072	1044	18814	40434	616	1500	0	3.42
	BM2	82.11%			1066	1050	14855	44393	1492	624	0	3.48
	BM3	80.19%			1054	1062	7370	51878	734	1382	0	3.78
	BM4	79.10%			1067	1049	17486	41762	1061	1055	0	3.78
	BM5	82.10%			1052	1064	7703	51545	807	1306	3	3.39
	BM6	82.00%			1059	1057	17868	41380	1099	1017	0	3.48
	BM7	81.00%			1074	1042	12762	46486	1375	741	0	3.40
	BM8	82.00%			1075	1041	15213	44035	1089	1027	0	3.90
	BM9	81.30%			1058	1058	21395	37853	1392	724	0	3.32
	BM10	81.90%			1092	1024	9931	49317	820	1296	0	3.21
	BM11	83.60%			1092	1024	27241	32007	945	1171	0	3.10
	BM12	80.70%			1075	1041	20051	39197	1164	952	0	3.23
GermanCredit	GC1	99.00%	22	200	198	2	1044	3356	200	0	0	1.94
	GC2	99.00%			198	2	959	3441	188	12	0	2.18
	GC3	99.00%			198	2	1569	2831	73	127	0	2.03
	GC4	99.00%			198	2	2401	1999	200	0	0	2.01
	GC5	99.00%			198	2	1193	3207	195	5	0	1.93
	GC6	99.00%			198	2	1627	2773	67	133	0	1.99
	GC7	99.00%			198	2	1074	3326	195	5	0	1.96
	GC8	99.00%			198	2	1360	3040	143	57	0	2.07
	GC9	99.00%			198	2	1360	3040	143	57	0	1.93

violations and satisfaction. Then, we report "Correct" and "Incorrect" implications in "#Correct" and "#Incorrect", "#Unseen" columns using our proposed technique *DeepInfer*. We also measure the total runtime and report in the "Time" column in Table 2. From the results, we observe that for the model with high accuracy, the total number of "Correct" and "Incorrect" implied using *DeepInfer* is comparable to the ground truth. For example, for the German Credit dataset and GC1 and GC4 model with accuracy 99.00%, *DeepInfer* obtained 200 "#Correct" and 0 "#Incorrect" where Ground Truth contains in total 198 "#Correct" and 2 "#Incorrect" labels. The reason behind incorrectly implying a number of incorrect and correct predictions in models like BM11 is that the model itself was not trained well, as low accuracy suggests. Based on our findings, we conclude that the model with high accuracy implies a better comparable number of "Correct" and "Incorrect" predictions for all the unseen datasets. Despite several models exhibiting high accuracy, we observed a lack of correlation between the number of violations and the accuracy of these models. This finding suggests the presence of underlying issues that warrant further investigation. We investigated further to determine the correlation between the number of violations in data preconditions and the frequency of "Correct" and "Incorrect" predictions based on the ground truth. Using the Pearson Correlation Coefficient (pcc) following prior work [26], we found a positive correlation of 0.88 between data precondition violations and incorrect model predictions, indicating that as the number of violations increases, the likelihood of incorrect predictions by the model also rises. This highlights the importance of data preconditions in determining the trustworthiness of the model's predictions. Additionally, we saw a strong correlation of 0.98 between precondition satisfaction and correct model predictions, indicating that the model tends to make accurate predictions when data preconditions are satisfied. To assess the statistical significance of these correlations, we conducted a t-test to compute p-values following prior work [26], yielding p-values of 0.0001 for the correlation between data precondition violation and incorrect prediction and 0.0003 for the correlation between data precondition satisfaction and correct prediction. Based on the commonly used significance level of 0.05, these p-values indicate that the correlations are statistically significant [57]. A p-value below 0.05 suggests strong evidence against the null hypothesis, supporting the presence of a significant correlation between the variables.

In summary, DeepInfer implies that data precondition violations and Incorrect model prediction are highly correlated (0.88) between prediction ground truth and violation. Also, the precondition satisfaction and correct model prediction are strongly correlated (0.98).

RQ2 (Effectiveness): In Table 3, we highlighted the best values with high model accuracy from each set of the dataset. We also observe how close the values are obtained from *DeepInfer* compared to the ground truth FP and TP. Some of the models, e.g., BM6, BM7, BM9, BM10, BM11 in the Bank Customer dataset throw *numpy.linalg.LinAlgError:Singular matrix* error during KDE generation steps using *SelfChecker* tool. We communicated with the authors of *SelfChecker*, and they explained that the models they used for evaluation contained only `relu,softmax` having more than 8 layers for image datasets. Furthermore, we obtain 0 FP and 0 TP and the same number of FN and TN for many models under experiments e.g., PD2, PD3, HP2, Hp3, BM4, BM5, BM8, GC5, GC6, GC7, GC8, and GC9 etc. We have investigated further and found that *SelfChecker* approach does not handle a model if the last layer contains `sigmoid, relu, tanh` activation functions with single output and the threshold of KDE values performs well for `softmax` activation functions with multiple outputs to determine true misbehavior of the model.

Next, we compute the precision, recall, and accuracy for all the models and present the results in Table 3. We computed average precision, recall, and accuracy for each dataset and obtained that

Table 3: Efficiency of *DeepInfer* for implying model's prediction

Dataset	Model	Test Acc.	Ground Truth		SelfChecker										DeepInfer									
			ActFP	ActTP	FP	TP	FN	TN	Precision	Recall	Accuracy	TPR	FPR	F-1	FP	TP	FN	TN	Precision	Recall	Accuracy	TPR	FPR	F-1
Pima Diabetes	PD1	77.98%	34	119	46	90	5	12	0.66	0.95	0.67	0.95	0.79	0.78	33	118	1	1	0.78	0.99	0.78	0.99	0.97	0.87
	PD2	65.10%	54	99	0	0	59	94	-	0.00	0.61	0.00	0.00	0.00	54	99	0	0	0.65	1.00	0.65	1.00	1.00	0.79
	PD3	65.49%	55	98	0	0	59	94	-	0.00	0.61	0.00	0.00	0.00	55	98	0	0	0.64	1.00	0.64	1.00	1.00	0.78
	PD4	77.47%	42	111	60	77	7	9	0.56	0.92	0.56	0.92	0.87	0.70	42	111	0	0	0.73	1.00	0.73	1.00	1.00	0.84
House Price	HP1	85.22%	145	147	59	114	13	105	0.66	0.90	0.75	0.90	0.36	0.76	127	145	2	18	0.53	0.99	0.56	0.99	0.88	0.69
	HP2	89.77%	145	147	0	0	139	153	-	0.00	0.52	0.00	0.00	0.00	127	145	2	18	0.53	0.99	0.56	0.99	0.88	0.69
	HP3	45.45%	147	145	0	0	139	153	-	0.00	0.52	0.00	0.00	0.00	147	145	0	0	0.50	1.00	0.50	1.00	1.00	0.66
	HP4	87.50%	145	147	51	168	15	57	0.77	0.92	0.77	0.92	0.47	0.84	143	146	1	2	0.51	0.99	0.51	0.99	0.99	0.67
BankCustomer	BM1	81.10%	1044	1072	0	0	1024	1092	-	0.00	0.52	0.00	0.00	0.00	984	987	85	60	0.50	0.92	0.49	0.92	0.94	0.65
	BM2	82.11%	1050	1066	0	0	1024	1092	-	0.00	0.52	0.00	0.00	0.00	869	866	200	181	0.50	0.81	0.49	0.81	0.83	0.62
	BM3	80.19%	1024	1092	387	798	332	599	0.67	0.71	0.66	0.71	0.39	0.69	474	559	533	550	0.54	0.51	0.52	0.51	0.46	0.53
	BM4	79.10%	1049	1067	0	0	1024	1092	-	0.00	0.52	0.00	0.00	0.00	916	906	161	133	0.50	0.85	0.49	0.85	0.87	0.63
	BM5	82.10%	1064	1052	0	0	1024	1092	-	0.00	0.52	0.00	0.00	0.00	1001	970	82	63	0.49	0.92	0.49	0.92	0.94	0.64
	BM6	82.00%	1057	1059	-	-	-	-	-	-	-	-	-	-	1004	977	82	53	0.49	0.92	0.49	0.92	0.95	0.64
	BM7	81.00%	1042	1074	-	-	-	-	-	-	-	-	-	-	987	984	90	55	0.50	0.92	0.49	0.92	0.95	0.65
	BM8	82.00%	1041	1075	0	0	1024	1092	-	-	-	-	-	-	985	986	89	56	0.50	0.92	0.49	0.92	0.95	0.65
	BM9	81.30%	1058	1058	-	-	-	-	-	-	-	-	-	-	923	888	170	135	0.49	0.84	0.48	0.84	0.87	0.62
	BM10	81.90%	1024	1092	-	-	-	-	-	-	-	-	-	-	982	989	103	42	0.50	0.91	0.49	0.91	0.96	0.65
	BM11	83.60%	1062	1054	-	-	-	-	-	-	-	-	-	-	1062	1054	0	0	0.50	1.00	0.50	1.00	1.00	0.66
	BM12	80.70%	1041	1075	0	0	1024	1092	-	0.00	0.52	0.00	0.00	0.00	860	866	209	181	0.50	0.81	0.49	0.81	0.83	0.62
GermanCredit	GC1	99.00%	2	198	37	91	1	71	0.71	0.99	0.81	0.99	0.34	0.83	2	198	0	0	0.99	1.00	0.99	1.00	1.00	0.99
	GC2	99.00%	2	198	0	74	18	108	1.00	0.80	0.91	0.80	0.00	0.89	2	186	12	0	0.99	0.94	0.93	0.94	1.00	0.96
	GC3	99.00%	2	198	42	96	1	61	0.70	0.99	0.79	0.99	0.41	0.82	1	72	126	1	0.99	0.36	0.37	0.36	0.50	0.53
	GC4	99.00%	2	198	0	2	0	198	1.00	1.00	1.00	1.00	0.00	1.00	2	198	0	0	0.99	1.00	0.99	1.00	1.00	0.99
	GC5	99.00%	2	198	0	0	2	198	-	0.00	0.99	0.00	0.00	0.00	2	193	5	0	0.99	0.97	0.97	0.97	1.00	0.98
	GC6	99.00%	2	198	0	0	2	198	-	0.00	0.99	0.00	0.00	0.00	1	66	132	1	0.99	0.33	0.34	0.33	0.50	0.50
	GC7	99.00%	2	198	0	0	2	198	-	0.00	0.99	0.00	0.00	0.00	2	193	5	0	0.99	0.97	0.97	0.97	1.00	0.98
	GC8	99.00%	2	198	0	0	2	198	-	0.00	0.99	0.00	0.00	0.00	1	142	56	1	0.99	0.72	0.72	0.72	0.50	0.83
	GC9	99.00%	2	198	0	0	2	198	-	0.00	0.99	0.00	0.00	0.00	1	142	56	1	0.99	0.72	0.72	0.72	0.50	0.83

* Here, '-' in "FP", "TP", "FN", "TN" column indicates where *SelfChecker* does not provide any output, therefore we can not get any values. For those cases, we get divided by zero error in the "Precision", "Recall", "Accuracy", "TPR", "FPR", "F-1" columns.

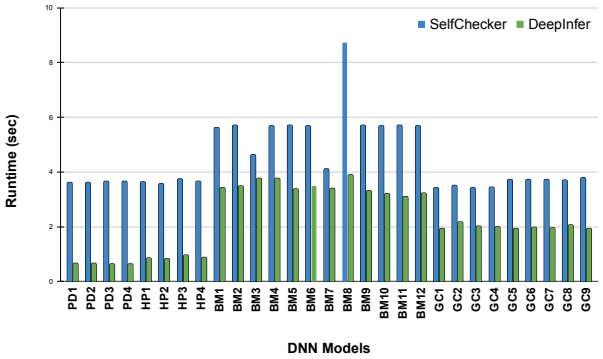

Figure 7: Runtime comparison of *DeepInfer* and *SelfChecker* for all models across different datasets

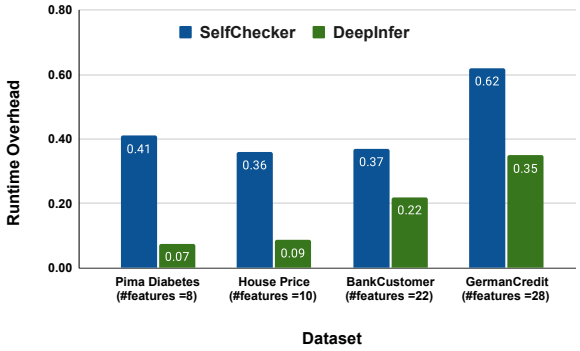

Figure 8: Runtime overhead comparison of *DeepInfer* and *SelfChecker* for all unseen data

for the high-accuracy models, the average precision, recall, and accuracy are 0.76, 0.98, and 0.76, respectively. Higher precision means that *DeepInfer* implies accurate results than inaccurate ones, and high recall means that *DeepInfer* returns most of the accurate results. The average precision and accuracy are low for models with performance-related underlying issues, which calls for further research. Furthermore, we also compared against the *SelfChecker* and found that *SelfChecker* produced identical results in terms of TP, FP, FN, and TN for certain models on a specific dataset. However, the assumption of using density functions and selected layers in the training module might not work properly. Also, measuring density function using training and representative test datasets might not be independent of model architectures, and it might not work well on different model structures which learned the training data differently.

In summary, DeepInfer effectively implies the correct and incorrect prediction of higher accuracy models with recall (0.98) and F-1 score (0.84), compared to SelfChecker with recall (0.59) and F-1 score (0.52).

RQ3 (Efficiency): We computed the runtime overhead of *DeepInfer* and *SelfChecker* with respect to original training time for all models in each kind of dataset which are unseen and plotted in Fig. 7. From the results, we observed that the average runtime of *DeepInfer* is 0.66 sec, 0.88 sec, 3.46 sec, 2.00 sec compared to average training time of 8.88 sec, 10.15 sec, 15.67 sec, and 5.74 sec in Pima Diabetes, House Price, Bank Customer, German Credit dataset respectively. On the other hand, the average runtime of *SelfChecker* is 3.65, 3.66, 5.73, and 3.61 sec using all the models of Pima Diabetes, House Price, Bank Customer, and German Credit dataset, respectively. We observe that the runtime is proportional to the number of features, which is consistent with our theoretical complexity results. Furthermore, we computed the runtime overhead of *DeepInfer* and *SelfChecker* for all unseen datasets over the training time for all models in each kind of dataset and plotted it in Fig. 8. We have observed that, the average runtime overhead of *SelfChecker* and *DeepInfer* is 0.41 and 0.07, 0.36 and 0.09, 0.37 and 0.22, 0.62 and 0.35 respectively, for Pima Diabetes, House Price, Bank Customer, German Credit dataset. During the deployment phase, we found that *DeepInfer* outperforms *SelfChecker* in terms of speed, being approximately 3.27 times faster. Additionally, we calculated the average runtime overhead for all unseen datasets and

models, which is 0.22 seconds. This runtime overhead is relatively minimal when compared to the original training time. An advantage of our proposed approach is that we eliminate the need to repeatedly retrain the model for overhead computation. In contrast, *SelfChecker* requires extensive computations for all training and test datasets, along with different layer combinations, in order to calculate statistical measures like KDE values. Consequently, this process incurs a substantial runtime overhead.

In summary, the average runtime overhead of DeepInfer is fairly minimal (0.22 sec for all the unseen data). The runtime overhead of DeepInfer is 3.27 times faster than SelfChecker during deployment.

4.2.3 Limitation. In this study, we conducted experiments to evaluate our proposed technique for inferring preconditions from real-valued features. We focused on these features because they are easier for humans to understand, and our datasets only included numerical values. While our current algorithms and derived *wp* rules are specific to certain layer computations and activation functions of fully connected layers, we believe that the fundamental idea of inferring data preconditions from deep neural network (DNN) models after training and using them for trustworthy prediction in deployment can be applied to other types of DNNs. For example, in popular models that utilize convolution and attention layers, we can extend the concept of computing data preconditions by extracting features from raw input data, such as images or text, and inferring preconditions from the classifier similarly.

4.2.4 Discussion on the state-of-the-art (SOTA) metrics and approaches. Some classifiers produce a confidence measure, such as confidence score and class prediction, typically by applying a softmax function to the raw numeric prediction values. However, such confidence measures need to be better-calibrated [40]; therefore, they cannot be reliably used as a measure of trust in prediction [26]. Surprise coverage relies on the concept of surprise adequacy [45, 70], which measures the dissimilarity between a test and the training data set. Surprise adequacy has a high computational cost. Surprise adequacy aims to quantitatively measure how surprising each new test input is when compared to the training data. It is used to detect out-of-bound with respect to the distribution of the training data, and the input is also more likely to cause unexpected model behavior. However, given an input, it captures the activation trace, the collection of neuron outputs produced by the model under test, which is expensive even for a simple model. Moreover, it does not indicate whether a particular prediction of the model is correct or incorrect with an unseen data point. DeepGini score [27] mainly provides a way to calculate a test prioritization to improve the quality of DNN. It determines a score by using only on the test input activations of the DNN's softmax output layer, limiting the approach's applicability to only classification problems with softmax activation function in the last layer. Moreover, it does not provide a mechanism to imply whether a particular prediction of the model is correct or incorrect during deployment. Some classifiers provide a level of confidence [55] or certainty when making predictions about which class something belongs to. They usually calculate this confidence using the softmax function. However, these confidence scores are often not very accurate and can't be trusted to tell us how confident the classifier is about its prediction and imply whether it is correct or incorrect prediction. None of these SOTA metrics learns input constraints from the trained model and utilizes that during the deployment to imply trust in the model's prediction using unseen data. For the evaluation with publicly available fully connected DNNs and datasets with numerical values, the SOTA techniques *SELFORACLE* [63], *DISSECTOR* [68], *ConfidNet* [20] are not applicable (details in §5).

5 RELATED WORK

We are inspired by the vast body of seminal work on weakest precondition calculus [16, 21, 23, 24, 33, 34, 56, 75].

Trusted Machine Learning. The closest idea related to trusted machine learning in the database and machine learning community is Conformance Constraint Discovery (*CCSynth*) [26] to quantify the degree of non-conformance in a dataset, allowing for the effective characterization of whether or not inference over a given tuple is reliable. They demonstrated the application for detecting unsafe tuples in trustworthy machine learning. However, their approach is model-independent and will result in the same constraints for different models with the same dataset. Our approach resolves this issue and works as a model-specific approach to identify how to imply trust in different DL models' predictions using a dataset with unseen data during deployment. In the software engineering community, *SELFORACLE* [63] has proposed an approach that monitors the performance of the DNN at runtime to predict unsupported driving scenarios by computing a confidence estimation. In contrast, our approach produces preconditions from the model using offline computation. *SELFORACLE* also focuses on image-based models and temporally ordered inputs, such as video frames, and does not apply to data with numerical attributes. Another technique, *SelfChecker* [72], assesses model consistency during deployment and assumes that the density functions and layers chosen by the training module can be applicable to new test instances. However, this assumption is contingent upon whether the training and validation datasets accurately represent the characteristics of test instances. *SelfChecker* operates through a layer-based approach, which necessitates white-box access and may have limited capabilities in detecting issues in shallow DNNs with a few layers. *SelfChecker++* [71] has been designed to target both unintended abnormal test data and intended adversarial samples. *InputReflector* [73], introduced a runtime approach to identify and fix failure-inducing inputs in DL systems inspired by traditional input-debugging techniques. Wang et al. introduced *DISSECTOR* [68] to identify inputs that deviate from the norm, by training several sub-models on top of a pre-trained deep learning model. However, generating these sub-models is manual and time-consuming [72]. Further, *DISSECTOR* is only applicable to image-based models such as ImageNet [8]. Researchers in the deep learning community have developed learning-based models to measure a model's confidence during deployment [20, 22, 40, 47, 48, 54]. However, these models can be untrustworthy and suffer from overfitting. Corbière et al. [20] proposed *ConfidNet*, a model built on top of pre-trained models that uses true class probability for failure prediction. However, overfitting can occur due to being trained on a small number of incorrect predictions in training dataset. *ConfidNet* technique has ConvNet architecture in its implementation and it would not be applicable for DNNs with only dense layers and datasets with numerical values. In contrast, our approach infers the

model's assumption of the data after training and utilizes that to imply the trustworthiness of model's prediction.

Neural Network Abstraction. There are a number of research ideas that focuses on abstracting neural network as DNN verification is NP-hard due to the number of nodes in DNN slowing the algorithms exponentially [62]. Singh *et al.* [60] proposes an abstract domain based on floating-point polyhedra and intervals along with abstract transformers for neural network functions for certifying deep neural networks. Gehr *et al.* [29] introduces the idea of abstract transformers that capture the behavior of common neural network layers to certify convolutional and large fully connected networks. There are other abstractions of neural networks, e.g., interval universal approximation [69], neural interval abstraction, neural zonotope abstraction, and neural polyhedron abstraction [11] None of these abstractions of the neural network works for *wp* reasoning with neural network functions as code statements and expected output as a postcondition which *DeepInfer* demonstrates.

Neural Network Specification and Verification. The related ideas in the specification of DNNs [32, 58, 66]. [58] discusses formalizing and reasoning about properties of DNN; however, [58] does not propose any precondition inference using model architecture and post condition. [32] proposed a technique to compute input and layer properties from a feed-forward network and utilize formal contracts for the network. The application of inferred properties has been demonstrated to explain predictions, guarantee robustness, simplify proofs, and network distillation. [66] introduced a constraint-based technique for repairing neural network classifiers by inferring correctness specifications. [25] proposes a technique to apply formal methods to ML components e.g., perception systems, and analyze system behavior in an uncertain environment. However, [25, 32, 66] did not consider abstracting neural networks and introduce a technique for computing data preconditions from trained DNN models and utilizing those inferred preconditions for implying trust in the model's prediction during the deployment stage. There is a recent study [59] on reducing DNN properties to enable falsification with adversarial attacks using a correctness problem comprised of a DNN and robustness problems property. In a recent study [19], a rule induction-based technique has been proposed to facilitate the debugging process of trained statistical models only that generates an interpretable characterization of the data on which the predictive machine learning model performs poorly. In another study [30], a bias-guided misprediction explanation technique has been proposed that generates explanation rules with higher misprediction explanation and also improves the machine learning model's robustness utilizing a mispredicted area upweight sampling algorithm. Recently, an empirical study [44] characterizes different kinds of ML contracts, which may help ML API developers to write contracts. Another research study [10] proposed a technique for checking contracts for deep learning libraries by specifying DL APIs with preconditions and postconditions. None of these recent papers along with the work [67, 74] related to neural network specification and verification utilizes a DNN model's model architecture and expected output to infer assumptions on data that our approach emphasizes. We demonstrate the utility of inferred data preconditions to imply the trustworthiness in predicting unseen data during deployment.

6 THREATS TO VALIDITY

In the context of inferring preconditions from a deep learning model, internal threats to validity include an incorrect model structure where the DNN model may not fully capture the underlying system's complexity or dynamics, leading to inaccurate precondition inference. External threats to validity include lack of representativeness in the unseen data where the data used to evaluate the model may not accurately reflect the real-world scenario, leading to the inaccurate implication of the model's prediction by our approach. To mitigate these threats, we have collected a large and diverse dataset that accurately represents the real-world scenario. This can help ensure the model is exposed to various variations and can generalize well to unseen data. Also, we have used more complex models with more `Dense` layers, which have the ability to learn complex patterns and features in the real-world dataset.

7 CONCLUSION AND FUTURE WORK

We propose a novel technique, *DeepInfer*, for inferring data preconditions from a DNN. *DeepInfer* uses an abstract representation of the DNN model and derived *wp* rules for different types of DNN functions, by solving challenges of non-linear computation with different dimensions of matrices, to infer preconditions for the model. A DNN can be deployed with these preconditions, and their violation can imply trust in the model's predictions during deployment. We have evaluated *DeepInfer* on 29 models using 4 real-world datasets and found substantial results compared to prior work regarding effectiveness and efficiency. We find that data precondition violations and incorrect model prediction are highly correlated. *DeepInfer* effectively implies the correct and incorrect prediction of higher accuracy models with recall (0.98) and F-1 score (0.84), which is a significant improvement compared to prior work. *DeepInfer* is 3.29 times faster than the state-of-the-art technique. In future, our approach can be extended to automatically validate the temporal properties of DNN models. We can also explore the use of predicate abstraction and symbolic reasoning for DNN models to further explain the black-box DNN models. Recent studies on decomposing DNN into modules [36, 52, 53], we intend to infer input preconditions of each DNN module for its expected and reliable behavior. We want to extend our data precondition inference technique to mitigate model's unfairness [12, 13, 31] in different stages of the ML pipeline [15]. We can enhance techniques [50, 51] by inferring preconditions from mined models, considering improved accuracy for trustworthy prediction.

8 DATA AVAILABILITY

The replication packages and results are available in this repository [7] that can be leveraged by software engineering for machine learning research in the future.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2018. Uber's fatal self-driving crash reportedly caused by software. https://www.cnet.com/roadshow/news/uber-reportedly-finds-false-positive-self-driving-car-accident/. [Online; accessed Mar-2023].

[2] 2019. AI in Medicine Is Overhyped. https://www.scientificamerican.com/article/ai-in-medicine-is-overhyped/. [Online; accessed Mar-2023].

[3] 2019. Self-driving Uber car that hit and killed woman did not recognize that pedestrians jaywalk. https://www.nbcnews.com/tech/tech-news/self-driving-uber-car-hit-killed-woman-did-not-recognize-n1079281. [Online; accessed Mar-2023].

[4] 2022. Bank Customer dataset. https://www.kaggle.com/datasets/kidoen/bank-customers-data. [Online; accessed Aug-2022].

[5] 2022. German Credit Risk Classification dataset. https://www.kaggle.com/code/twunderbar/german-credit-risk-classification-with-keras/data. [Online; accessed Aug-2022].

[6] 2022. House Price Prediction dataset. here. [Online; accessed Aug-2022].

[7] 2023. Repository of DeepInfer. https://github.com/shibbirtanvin/DeepInfer. [Online; accessed September-2023].

[8] 2023. Repository of DISSECTOR. https://github.com/ParagonLight/dissector. [Online; accessed July-2023].

[9] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black Box Fairness Testing of Machine Learning Models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 625–635. https://doi.org/10.1145/3338906.3338937

[10] Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. 2023. Design by Contract for Deep Learning APIs. In *ESEC/FSE'2023: The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, California). https://doi.org/10.1145/3611643.3616247

[11] Aws Albarghouthi. 2021. Introduction to Neural Network Verification. *Found. Trends Program. Lang.* 7, 1–2 (dec 2021), 1–157. https://doi.org/10.1561/2500000051

[12] Sumon Biswas and Hridesh Rajan. 2020. Do the Machine Learning Models on a Crowd Sourced Platform Exhibit Bias? An Empirical Study on Model Fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 642–653. https://doi.org/10.1145/3368089.3409704

[13] Sumon Biswas and Hridesh Rajan. 2021. Fair Preprocessing: Towards Understanding Compositional Fairness of Data Transformers in Machine Learning Pipeline. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 981–993. https://doi.org/10.1145/3468264.3468536

[14] Sumon Biswas and Hridesh Rajan. 2023. Fairify: Fairness Verification of Neural Networks. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1546–1558. https://doi.org/10.1109/ICSE48619.2023.00134

[15] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The Art and Practice of Data Science Pipelines: A Comprehensive Study of Data Science Pipelines in Theory, in-the-Small, and in-the-Large. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2091–2103. https://doi.org/10.1145/3510003.3510057

[16] Marcello M Bonsangue and Joost N Kok. 1994. The weakest precondition calculus: Recursion and duality. *Formal Aspects of Computing* 6, 1 (1994), 788–800.

[17] Leo Breiman. 2001. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science* 16, 3 (2001), 199–231.

[18] Murat Cenk and M Anwar Hasan. 2017. On the arithmetic complexity of Strassen-like matrix multiplications. *Journal of Symbolic Computation* 80 (2017), 484–501.

[19] Jürgen Cito, Isil Dillig, Seohyun Kim, Vijayaraghavan Murali, and Satish Chandra. 2021. Explaining Mispredictions of Machine Learning Models Using Rule Induction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 716–727. https://doi.org/10.1145/3468264.3468614

[20] Charles Corbière, Nicolas THOME, Avner Bar-Hen, Matthieu Cord, and Patrick Pérez. 2019. Addressing Failure Prediction by Learning Model Confidence. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/757f843a169cc678064d9530d12a1881-Paper.pdf

[21] Frank S de Boer. 1999. A wp-calculus for OO. In *International Conference on Foundations of Software Science and Computation Structure*. Springer, 135–140.

[22] Terrance DeVries and Graham W Taylor. 2018. Learning confidence for out-of-distribution detection in neural networks. *arXiv preprint arXiv:1802.04865* (2018).

[23] Ellie D'hondt and Prakash Panangaden. 2006. Quantum weakest preconditions. *Mathematical Structures in Computer Science* 16, 3 (2006), 429–451.

[24] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (aug 1975), 453–457. https://doi.org/10.1145/360933.360975

[25] Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A Seshia. 2019. Verifai: A toolkit for the formal design and analysis of artificial intelligence-based systems. In *International Conference on Computer Aided Verification*. Springer, 432–442.

[26] Anna Fariha, Ashish Tiwari, Arjun Radhakrishna, Sumit Gulwani, and Alexandra Meliou. 2021. Conformance Constraint Discovery: Measuring Trust in Data-Driven Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 499–512. https://doi.org/10.1145/3448016.3452795

[27] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 177–188.

[28] Sainyam Galhotra, Anna Fariha, Raoni Lourenço, Juliana Freire, Alexandra Meliou, and Divesh Srivastava. 2022. DataPrism: Exposing Disconnect between Data and Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 217–231. https://doi.org/10.1145/3514221.3517864

[29] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 3–18. https://doi.org/10.1109/SP.2018.00058

[30] Jiri Gesi, Xinyun Shen, Yunfan Geng, Qihong Chen, and Iftekhar Ahmed. 2023. Leveraging Feature Bias for Scalable Misprediction Explanation of Machine Learning Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1559–1570. https://doi.org/10.1109/ICSE48619.2023.00135

[31] Usman Gohar, Sumon Biswas, and Hridesh Rajan. 2023. Towards Understanding Fairness and Its Composition in Ensemble Machine Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1533–1545. https://doi.org/10.1109/ICSE48619.2023.00133

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

[33] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[34] Charles Antony Richard Hoare and Jifeng He. 1987. The weakest prespecification. *Inform. Process. Lett.* 24, 2 (1987), 127–132.

[35] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety verification of deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*. Springer, 3–29.

[36] Sayem Mohammad Imtiaz, Fraol Batole, Astha Singh, Rangeet Pan, Breno Dantas Cruz, and Hridesh Rajan. 2023. Decomposing a Recurrent Neural Network into Modules for Enabling Reusability and Replacement. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1020–1032. https://doi.org/10.1109/ICSE48619.2023.00093

[37] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[38] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1135–1146. https://doi.org/10.1145/3377811.3380378

[39] William H Jefferys. 1980. On the method of least-squares. *The Astronomical Journal* 85 (1980), 177.

[40] Heinrich Jiang, Been Kim, Melody Y. Guan, and Maya Gupta. 2018. To Trust or Not to Trust a Classifier. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 5546–5557.

[41] Kaggle. 2010. The world's largest data science community with powerful tools and resources to help you achieve your data science goals. www.kaggle.com.

[42] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*. Springer, 97–117.

[43] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. 2019. The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*. Springer, 443–452.

[44] Samantha Syeda Khairunnesa, Shibbir Ahmed, Sayem Mohammad Imtiaz, Hridesh Rajan, and Gary T. Leavens. 2023. What Kinds of Contracts Do ML APIs Need? *Empirical Software Engineering* 1, 1 (March 2023).

[45] Jinhan Kim, Robert Feldt, and Shin Yoo. 2023. Evaluating Surprise Adequacy for Deep Learning System Testing. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 42 (mar 2023), 29 pages. https://doi.org/10.1145/3546947

[46] Max Kuhn, Kjell Johnson, et al. 2013. *Applied predictive modeling*. Vol. 26. Springer.

[47] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6405–6416.

[48] Yan Luo, Yongkang Wong, Mohan S Kankanhalli, and Qi Zhao. 2021. Learning to Predict Trustworthiness with Steep Slope Loss. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 21533–21544. https://proceedings.neurips.cc/paper_files/paper/2021/file/b432f34c5a997c8e7c806a895ecc5e25-Paper.pdf

[49] Denis Mazzucato and Caterina Urban. 2021. Reduced products of abstract domains for fairness certification of neural networks. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings 28*. Springer, 308–322.

[50] Giang Nguyen, Sumon Biswas, and Hridesh Rajan. 2023. Fix Fairness, Don't Ruin Accuracy: Performance Aware Fairness Repair using AutoML. In *ESEC/FSE'2023: The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, California).

[51] Giang Nguyen, Md Johirul Islam, Rangeet Pan, and Hridesh Rajan. 2022. Manas: Mining Software Repositories to Assist AutoML. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1368–1380. https://doi.org/10.1145/3510003.3510052

[52] Rangeet Pan and Hridesh Rajan. 2020. On Decomposing a Deep Neural Network into Modules. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 889–900. https://doi.org/10.1145/3368089.3409668

[53] Rangeet Pan and Hridesh Rajan. 2022. Decomposing Convolutional Neural Networks into Reusable and Replaceable Modules. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 524–535. https://doi.org/10.1145/3510003.3510051

[54] Nicolas Papernot and Patrick McDaniel. 2018. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765* (2018).

[55] Tim Pearce, Alexandra Brintrup, Mohamed Zaki, and Andy Neely. 2018. High-Quality Prediction Intervals for Deep Learning: A Distribution-Free, Ensembled Approach. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 4075–4084. https://proceedings.mlr.press/v80/pearce18a.html

[56] Christopher M Poskitt and Detlef Plump. 2010. A Hoare calculus for graph programs. In *International Conference on Graph Transformation*. Springer, 139–154.

[57] William R Rice. 1989. Analyzing tables of statistical tests. *Evolution* 43, 1 (1989), 223–225.

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

[59] David Shriver, Sebastian Elbaum, and Matthew B. Dwyer. 2021. Reducing DNN Properties to Enable Falsification with Adversarial Attacks. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 275–287. https://doi.org/10.1109/ICSE43902.2021.00036

[60] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An Abstract Domain for Certifying Neural Networks. *Proc. ACM Program. Lang.* 3, POPL, Article 41 (jan 2019), 30 pages. https://doi.org/10.1145/3290354

[61] Jack W Smith, James E Everhart, WC Dickson, William C Knowler, and Robert Scott Johannes. 1988. Using the ADAP learning algorithm to forecast the onset of diabetes mellitus. In *Proceedings of the annual symposium on computer application in medical care*. American Medical Informatics Association, 261.

[62] Matthew Sotoudeh and Aditya V Thakur. 2020. Abstract neural networks. In *International Static Analysis Symposium*. Springer, 65–88.

[63] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour Prediction for Autonomous Driving Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 359–371. https://doi.org/10.1145/3377811.3380353

[64] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated Directed Fairness Testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 98–108. https://doi.org/10.1145/3238147.3238165

[65] Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Perfectly parallel fairness certification of neural networks. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[66] Muhammad Usman, Divya Gopinath, Youcheng Sun, Yannic Noller, and Corina S. Păsăreanu. 2021. NNrepair: Constraint-based Repair of Neural Network Classifiers. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 3–25. https://doi.org/10.1007/978-3-030-81685-8_1

[67] Chengpeng Wang, Gang Fan, Peisen Yao, Fuxiong Pan, , and Charles Zhang. ICSE 2023. Verifying Data Constraint Equivalence in FinTech Systems. (ICSE 2023).

[68] Huiyan Wang, Jingwei Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. Dissector: Input Validation for Deep Learning Applications by Crossing-Layer Dissection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 727–738. https://doi.org/10.1145/3377811.3380379

[69] Zi Wang, Aws Albarghouthi, Gautam Prakriya, and Somesh Jha. 2022. Interval universal approximation for neural networks. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29.

[70] Michael Weiss and Paolo Tonella. 2022. Simple techniques work surprisingly well for neural network test prioritization and active learning (replicability study). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 139–150.

[71] Yan Xiao, Ivan Beschastnikh, Yun Lin, Rajdeep Singh Hundal, Xiaofei Xie, David S. Rosenblum, and Jin Song Dong. 2022. Self-Checking Deep Neural Networks for Anomalies and Adversaries in Deployment. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–18. https://doi.org/10.1109/TDSC.2022.3200421

[72] Yan Xiao, Ivan Beschastnikh, David S. Rosenblum, Changsheng Sun, Sebastian Elbaum, Yun Lin, and Jin Song Dong. 2021. Self-Checking Deep Neural Networks in Deployment. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 372–384. https://doi.org/10.1109/ICSE43902.2021.00044

[73] Yan Xiao, Yun Lin, Ivan Beschastnikh, Changsheng Sun, David Rosenblum, and Jin Song Dong. 2023. Repairing Failure-Inducing Inputs with Input Reflection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 85, 13 pages. https://doi.org/10.1145/3551349.3556932

[74] Chenyang Yang, Rachel A Brower-Sinning, Grace A Lewis, and Christian Kästner. 2022. Data leakage in notebooks: Static detection and better processes. (2022).

[75] Mingsheng Ying. 2012. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 6 (2012), 1–49.

[76] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. 2020. White-Box Fairness Testing through Adversarial Sampling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 949–960. https://doi.org/10.1145/3377811.3380331