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

Shibbir Ahmed  
Dept. of Computer Science  
Iowa State University  
Ames, IA, USA  
hshibbir@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

KEYWORDS
Deep neural networks, weakest precondition, trustworthiness

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICSE ’24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
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
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.

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.

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

Let us consider the Dense layer computation denoting \( f(\mathcal{X}) \). In the grammar, we denote \( N \) as a neural network with activation function \( a(f(\mathcal{X})) \) 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 \( \mathcal{X} \) with corresponding weight \( W \) and bias \( b \) 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 \( \mathcal{X} \) and the predicate as \( \Rightarrow P \), where \( n \in R \) and \( \Rightarrow \) represent the logical comparison operators. Here, \( \mathcal{F} \) denotes an
We solve this non-linear equation of the input vector for the intermediate layers. We consider \( x \) as the input/output vector for the intermediate layers. We consider \( \tilde{y} \) the output of the last layer, and \( n \) is the prediction interval for a DNN model's prediction. Here, \( x \) 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 \( \delta \) 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 \( \delta \) holds after \( N \) terminates. Deeplyfeeder 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 \( \delta \) using the rules illustrated in Fig. 4. Here, the rules (\( wp)\), (\( wp\alpha\phi\)) represent recursion over inductive type \( N \) by the function \( wp \), eventually satisfying base cases of \( N \). These base cases of \( wp \) use \( \alpha \) to compute the precondition where \( \alpha \) does recursion over the cases of the inductive type \( \phi \) represented using rules (\( wp\alpha\phi\true\)), (\( wp\alpha\phi\wedge\)), (\( wp\alpha\phi\vee\)), (\( wp\alpha\phi\vee\)) illustrated in Fig. 4. Again, base cases of \( \alpha \) use \( \beta \) to compute the \( wp \) for the cases of the activation function \( \phi \). For instance, for \( \text{ReLU} \) activation function, we compute \( \beta \) using the computation with weight and bias of a layer as follows,

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

We solve this non-linear equation of \( \text{relu}(\overline{W}x+b) \) for postcondition \( (\overline{x} \Rightarrow \overline{y}) \) and obtain the precondition as stated in the (\( \text{BetaReLU} \)) rule in Fig. 4. Similary, for other kinds of activation functions, we have derived \( \beta \) rules e.g., (\( \text{BetaLINEAR} \)), (\( \text{BetaSIGMOID} \)), (\( \text{BetaTANH} \)) rules illustrated in Fig. 4. The derivation details of each kind of these rules are in the appendix of the DNN repository [7].

\[
\beta(\text{relu}(\overline{W}x+b), \overline{z} \Rightarrow \overline{y}) \equiv \overline{z} \Rightarrow (\overline{f} \overline{y} - \overline{b}) \land \overline{z} \Rightarrow \overline{f} \equiv (\overline{W}^T \overline{W})^{-1}
\]

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 \( \text{relu}(x) = x, x \geq 0 \), \( 0, x < 0 \), ELU activation function computes \( \text{elu}(x) = x, x \geq 0 \), \( e^x - 1, x < 0 \), softmax activation function computes \( \text{softmax}(x) = \frac{e^x}{\sum_{j=0}^{n-1} e^{x_j}} \) [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. 3) 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 \((8 \times 1), (8 \times 1), (1 \times 1)\) respectively. In the second layer, the dimension of weight, bias, input, and output is \((8 \times 12), (8 \times 1), (12 \times 1), (8 \times 1)\) respectively. In the first layer, the dimension of weight, bias, input, and output is \((12 \times 8), (12 \times 1), (8 \times 1), (12 \times 1)\) respectively. If we assert using postcondition with a single dimension of \( \delta \), as a data precondition in the first layer should be a dimension of \( 8 \times 1 \) 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 non-linear 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 postcondition. The equation of Dense layer is as follows:

\[
\text{output} = \text{activation}(\text{dot}(\text{input}, \text{weight}) + \text{bias})
\]

So, for given postcondition \( \delta : y \leq n \), the statement \( S_1 \) can be written for output of last layer \( (y_s) \) with corresponding weight \( \overline{w} \) as a data precondition.
In our proposed DNN model is automated and generalized for other models which is multiple layers where \( \delta_2 \) denotes the precondition of the data layer 3, which is a postcondition of layer 2. Here, \( \overline{\delta_2} \) is an inverse function of the layer’s weight matrix \( \overline{W} \). Then, we obtain \( \delta_2 \) similarly using the wp rules (\( \text{wpAlpha} \), \( \text{wpAlphaWedge} \), \( \text{BetaLinear} \)) consecutively,
\[
\delta_2 = \text{wp}(\overline{W_0}W_3x_3 + b_3, \overline{\delta_3})
\]
\[
\text{\( \text{\large \alpha}(\delta_3, \beta(\text{linear}(\overline{W_0}W_3x_3 + b_3))) \equiv \beta(\text{linear}(\overline{W_0}W_3x_3 + b_3), \delta_3) \)
\]
\[
\text{\( \beta(\text{linear}(\overline{W_0}W_3x_3 + b_3), \overline{\delta_3}) \equiv (\overline{\delta_2}.(\overline{W_0}W_2x_2 + b_2) - b_2) - b_2 = (\overline{W_0}W_2W_3)^{-1}.(\overline{W_0}W_2)^{-1} \))
\]

In this step, we obtain the precondition which is an array of the input \( \overline{\delta_3} \) that has been obtained from the first layer and fed into the second layer, and the predicate of \( \overline{\delta_3} \) denotes the precondition of the input in layer 2, which is a postcondition of layer 1. After asserting with this postcondition, we obtain \( \delta_3 \) similarly using the wp rules (\( \text{wpAlpha} \), \( \text{wpAlphaWedge} \), \( \text{BetaLinear} \)),
\[
\delta_3 = \text{wp}(\overline{W_0}W_3x_3 + b_3, \overline{\delta_3}) \equiv \alpha(\delta_2, \beta(\text{linear}(\overline{W_0}W_1x_2 + b_1)))
\]
\[
\text{\( \beta(\text{linear}(\overline{W_0}W_3x_3 + b_3), \overline{\delta_3}) \equiv (\overline{\delta_2}.(\overline{W_0}W_1W_3)^{-1}.(\overline{W_0}W_0)^{-1} - b_2) - b_2 = (\overline{W_0}W_1W_3)^{-1}.(\overline{W_0}W_2)^{-1} \))
\]

Finally, we obtain the precondition, which is an array of the input \( \overline{\delta_3} \) for each feature that has been assumed by this DNN with multiple layers where \( \overline{\delta_3} = (\overline{W_0}W_1W_3)^{-1}.(\overline{W_0}W_0)^{-1} \). 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.

Algorithm 1 Data Precondition Violation Procedure

```
1: procedure COMPUTETHRESHOLD(N, Dtest, δ)
2: δ' ← wp(N, δ)  // Obtain data precondition given N and postcondition δ
3: f ← dom(Din)  // Set of input features, Din ∈ Dtest
4: d ← range(Din)  // Set of data points, σ ∈ Din
5: wpDict ← ∅
6: for each (d, f) do
7: wpDict ← wpDict ∪ {σ, wp}  // Collect violation count for each feature
8: V ← 1/|σ| ∑(wp(σ)) wp ∈ wpDict  // Mean violation threshold
9: return V, δ', f
10: procedure CHECKPREDICTION(d, V, δ', f)
11: M, L ← 0, wp warn ← ∅  // Violation count array indexed by features
12: for each i ∈ |f| do
13: wp warn ← wp warn ∪ {σ, wp}  // Check precondition violation for input
14: for each i ∈ |f| do  // Collect precondition violation for each feature
15: if |wp warn| = 0 then  // Compare violation count with threshold
16: M ← M + 1
17: else
18: L ← L + 1
19: wp warn ← decisionTree(M, V, L)  // Correct/incorrect/uncertain?
20: 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 $O(|d||f|)$ where $|d|$ is the size of unseen data and $|f|$ 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

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Features</th>
<th>Model</th>
<th>Source</th>
<th># Layers</th>
<th># Neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pima Diabetes [65]</td>
<td>8</td>
<td>PD1</td>
<td>Kaggle</td>
<td>3</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PD2</td>
<td>Kaggle</td>
<td>3</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PD3</td>
<td>Kaggle</td>
<td>3</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PD4</td>
<td>Kaggle</td>
<td>4</td>
<td>293</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PD5</td>
<td>Kaggle</td>
<td>3</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HP1</td>
<td>Kaggle</td>
<td>3</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HP2</td>
<td>Kaggle</td>
<td>3</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HP3</td>
<td>Kaggle</td>
<td>3</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HP4</td>
<td>Kaggle</td>
<td>4</td>
<td>382</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM2</td>
<td>[4]</td>
<td>4</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM3</td>
<td>[5]</td>
<td>3</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM4</td>
<td>[4]</td>
<td>5</td>
<td>315</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM5</td>
<td>[4]</td>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM6</td>
<td>[4]</td>
<td>4</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM7</td>
<td>[76]</td>
<td>7</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM8</td>
<td>[76]</td>
<td>7</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM9</td>
<td>[76]</td>
<td>7</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM10</td>
<td>[76]</td>
<td>8</td>
<td>622</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM11</td>
<td>[76]</td>
<td>3</td>
<td>622</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BM12</td>
<td>[76]</td>
<td>4</td>
<td>1439</td>
</tr>
<tr>
<td>BankCustomer [4]</td>
<td>28</td>
<td>GC1</td>
<td>[64]</td>
<td>3</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC2</td>
<td>[64]</td>
<td>3</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC3</td>
<td>[16]</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC4</td>
<td>[16]</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC5</td>
<td>[76]</td>
<td>7</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC6</td>
<td>Kaggle</td>
<td>3</td>
<td>2397</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC7</td>
<td>Kaggle</td>
<td>3</td>
<td>2397</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC8</td>
<td>Kaggle</td>
<td>3</td>
<td>2397</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC9</td>
<td>Kaggle</td>
<td>4</td>
<td>2949</td>
</tr>
</tbody>
</table>

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 improved 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 (rcc) 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, FN, and FP 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
Additionally, we saw a strong correlation of 0.98 between precondition violations and satisfaction. Then, we report “Correct” and “Incorrect” implications in “#Correct” and “#Incorrect”, “#Unseen” predictions 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 models 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 violation and incorrect model 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
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 adequacy relies on the concept of surprise adequacy [45, 70], which measures the dissimilarity between a test and the training dataset. 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 possess 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 of where the DNN model may not properly 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.