Self-Checking Deep Neural Networks in Deployment

Yan Xiao*, Ivan Beschastnikh†, David S. Rosenblum‡, Changsheng Sun*, Sebastian Elbaum¶,
Yun Lin* and Jin Song Dong*
*School of Computing, National University of Singapore, Singapore
dcsxan@nus.edu.sg, changsheng_sun@outlook.com, dcsliny@nus.edu.sg, dcsdjs@nus.edu.sg
†Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, bestchai@cs.ubc.ca
‡Department of Computer Science, George Mason University, Fairfax, VA, USA, dsr@gmu.edu
¶Department of Computer Science, University of Virginia, Charlottesville, VA, USA, selbaum@virginia.edu

***Abstract*—The widespread adoption of Deep Neural Networks (DNNs) in important domains raises questions about the trustworthiness of DNN outputs. Even a highly accurate DNN will make mistakes some of the time, and in settings like self-driving vehicles these mistakes must be quickly detected and properly dealt with *in deployment*.**

Just as our community has developed effective techniques and mechanisms to monitor and check programmed components, we believe it is now necessary to do the same for DNNs. In this paper we present *DNN self-checking* as a process by which internal DNN layer features are used to check DNN predictions. We detail *SelfChecker*, a self-checking system that monitors DNN outputs and triggers an alarm if the internal layer features of the model are inconsistent with the final prediction. SelfChecker also provides *advice* in the form of an alternative prediction.

We evaluated SelfChecker on four popular image datasets and three DNN models and found that SelfChecker triggers correct alarms on 60.56% of wrong DNN predictions, and false alarms on 2.04% of correct DNN predictions. This is a substantial improvement over prior work (SELFORACLE, DISSECTOR, and ConfidNet). In experiments with self-driving car scenarios, SelfChecker triggers more correct alarms than SELFORACLE for two DNN models (DAVE-2 and Chauffeur) with comparable false alarms. Our implementation is available as open source.

***Index Terms*—deep learning, trustworthiness, deployment**

I. INTRODUCTION

Deep Neural Networks (DNNs) are now used in a variety of domains, including speech processing [1], NLP [2], medical diagnostics [3], image processing [4], robotics [5] and even reconstruction of brain circuits [6]. The power and accuracy of DNNs have led to deployments of Deep Learning (DL) systems in safety- and security-critical domains, including self-driving cars [7], malware detection [8] and aircraft collision avoidance systems [9]. Such domains have a low tolerance for mistakes. The software systems in a self-driving car, for example, must have high assurance in deployment.

Unfortunately, the stochastic nature of DL virtually ensures that DL models will not achieve 100% accuracy, even on the training dataset. Since in mission-critical applications a wrong DNN decision could be costly, we believe that such applications must include logic to (1) *check* the trustworthiness of a DNN's output, and (2) raise an *alarm* when there is low confidence in the output. Our community has developed such methods for programmed components [10]–[12] and now is the time to do so for learned ones like DNNs.

Trustworthiness of a simple DNN can be measured with softmax probabilities [13], or information theoretic metrics, such as entropy [14] and mutual information [15]. However, in complex DNNs with many layers and neurons, softmax probabilities and entropy are unreliable confidence estimators of the prediction [16], [17]. Even for abnormal samples, DNNs may still produce overconfident posterior probabilities. For example, when we built classifiers for VGG-16 [18] on CIFAR-10 [19], we found that 75% of predictions that were incorrect had maximum softmax probabilities over 70%; and 63% incorrect predictions had maximum softmax probabilities over 80%. We had similar results on other datasets and models. This illustrates the unreliability of the softmax probabilities as confidence estimators of the final prediction.

Our goal is to build a general-purpose system that monitors a deployed DNN's predictions during inference, raises an *alarm* if there is low confidence in the predictions, and provides an alternative prediction that we call an *advice*. A key challenge in building such a system is finding a source of additional information to check DNN outputs. The inspiration for our work comes from Kaya et al., who study internal DNN behavior [20]. They found that a DNN can reach a correct prediction *before* the final layer. In fact, the final layer of a DNN may change a correct internal prediction into an incorrect prediction. This work illustrates that features extracted from internal layers of a DNN contain information that can be used to cross-check a model's output.

Inspired by Kaya et al.'s work, we define *self-checking* as a process by which internal DNN layer features are used to check DNN predictions. In this paper we describe a novel self-checking system, called **SelfChecker**, that triggers an alarm if the internal layer features of the model are inconsistent with the final prediction. SelfChecker also provides *advice* in the form of an alternative prediction. SelfChecker assumes that the training and validation datasets come from a distribution similar to that of the inputs that the DNN model will face in deployment.

SelfChecker uses kernel density estimation (KDE) to extrapolate the probability density distributions of each layer's

1558-1225/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00044

output by evaluating the DNN on the training data. Based on these distributions, the density probability of each layer's outputs can be inferred when the DNN is given a test instance. SelfChecker measures how the layer features of the test instance are similar to the samples in the training set. If a majority of the layers indicate inferred classes that are different from the model prediction, then SelfChecker triggers an alarm. In addition, not all layers can contribute positively to the final prediction [20]. SelfChecker therefore uses a search-based optimization to select a set of optimal layers to generate a high quality alarm and advice.

We evaluated SelfChecker's alarm and advice mechanisms with experiments on four popular and publicly-available datasets (MNIST, FMNIST, CIFAR-10, and CIFAR-100) and three DNNs (ConvNet, VGG-16, and ResNet-20) against three competing approaches (SELFORACLE [21], DISSECTOR [22], and ConfidNet [17]). Our results show that SelfChecker achieves the highest F1-score (68.07%), which is 8.77% higher than the next best approach (ConfidNet). Our evaluation of SelfChecker's DNN prediction checking runtime shows an acceptable time overhead of 34.98ms. We also compared SelfChecker to the state-of-the-art approach for self-driving car scenarios (SELFORACLE [21]), and found that SelfChecker triggers more correct alarms and a comparable number of false alarms.

Our paper makes the following three contributions:

- ⋆ We present the design of SelfChecker, which uses density distributions of layer features and a search-based layer selection strategy to trigger an alarm if a DNN model output has low confidence. We show that SelfChecker achieves better alarm accuracy than previous work.
- ⋆ Unlike existing work, SelfChecker provides *advice* in the form of an alternative prediction. We find that models on a 10-class dataset can use this advice to achieve higher prediction accuracy.
- ⋆ We demonstrate the effectiveness of SelfChecker's alarms and advice on publicly available DNNs, ranging from small models (ConvNet) to large and complex models (VGG-16 and ResNet-20), and self-driving car scenarios. Our implementation is open-source[1].

II. BACKGROUND AND MOTIVATION

In a deep neural network (DNN), an input is fed into the input layer, then passed through a series of hidden layers that extract features from the input using activation functions attached to neurons, and the process concludes with the output layer, which uses the extracted features to output a prediction using either *classification* (from a categorical set of classes) or *regression* (in the form of real-valued ordinals). The behavior of a layer during inference thus can be characterized by its vector of neuron activation outputs. In what follows, we refer to these layer-wise vectors of activation outputs as the *layer features* analyzed by our approach.

[1]https://github.com/self-checker/SelfChecker

A. The Promise of Using Layer Features

DNNs make decisions based on features extracted from training data. But how can we judge if a model is making a wrong decision for a given test instance? One way is to check whether the model has previously observed a similar instance during training. This raises the question of how to define the similarity between a test instance $\mathbf{x}$ and a training instance $\mathbf{x}'$. Most existing studies use a distance-based measure [23], such as L_p or cosine similarity. We think this is problematic since the inputs are complex enough and need DNNs to extract features, so we doubt that a distance measure defined directly on the inputs can properly capture similarity.

Instead, we use the features of the inputs extracted by internal layers in DNNs to capture similarity. Specifically, we define the similarity as the likelihood of the DNN having seen a similar layer features during training. We use probability density distributions extrapolated from the training process to measure the similarity between layer features of a given input and those observed for training data.

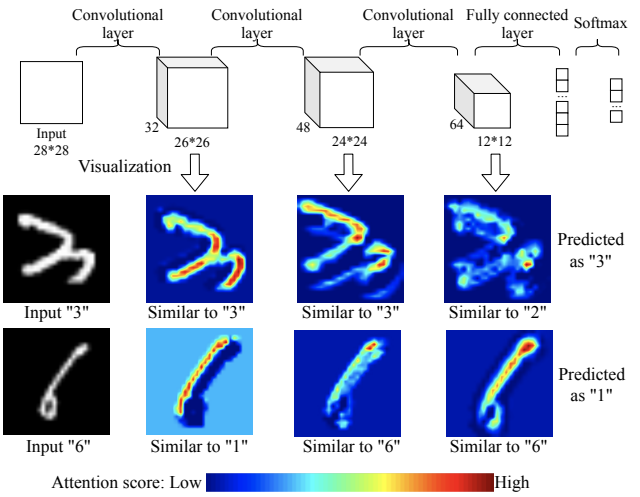


Fig. 1. The top of the figure depicts the architecture of a Convolutional Neural Network with three convolutional layers to classify digit images. The two rows of images at the bottom depict attention heatmaps for the associated layers when given test inputs for digits 3 and 6, respectively.

Fig. 1 presents a motivating example where a Convolutional Neural Network (CNN) with three convolutional layers trained on MNIST is used to classify images of digits 3 and 6, while outputting labels "3" and "1" as the respective predictions. To visualize *where* the features of each layer focus, we apply Grad-CAM [24] to highlight the attention heatmap on the original images as shown in the bottom two rows of images in Fig. 1. The heatmap images show that different layers have different points of focus. For example, the first and second images of digit 3 are similar to 3 itself, but the third image is closer to digit 2. Similarly, the first image of digit 6 is similar to digit 1, but the second and third images are similar to 6.

Although the CNN misclassifies the second image, in both cases the images appear to be recognized correctly by one or more hidden layers. This example thus illustrates the promise

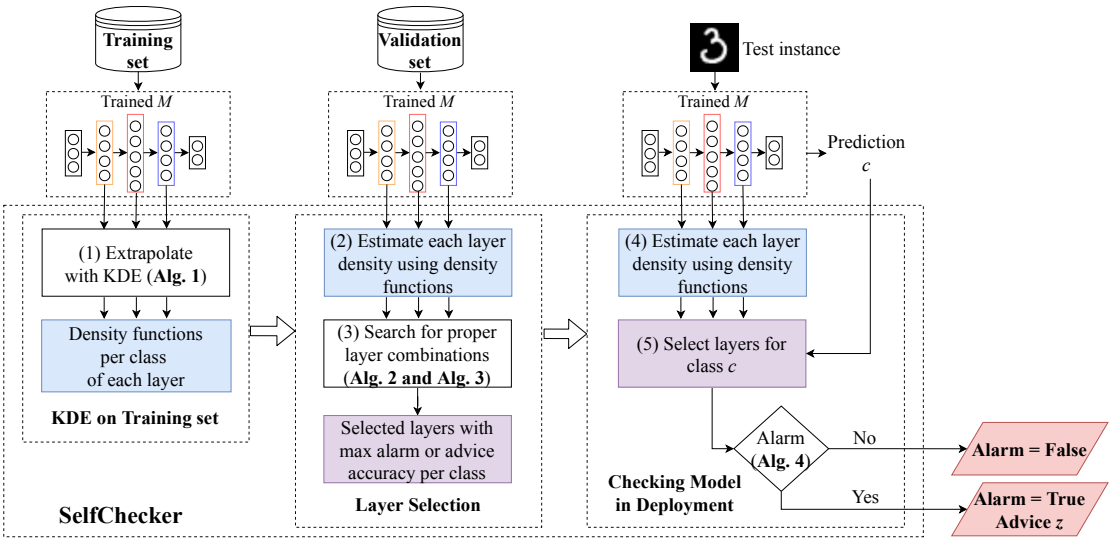


Fig. 2. The design of SelfChecker and its integration with a trained model and model predictions.

of using layer features to check the model's classification of a test instance.

DNNs exist in many variants and can be combined to form more complex models. For example, models used in urban flow prediction [25], [26] combine convolutional, graph and recurrent neural nets. However, all these DNNs extract features using internal layers, and that is the focus of our research.

The design we present targets DNN classifiers with convolutional layers and fully-connected layers. Our system also works for regression networks by transforming the network into a binary classification problem. Since our design uses layer features, it should work on other types of DNNs, such as recurrent neural networks. We leave the evaluation of our system on other DNN types to future work.

B. The Challenges of Using Layer Features

The preceding example also raises two challenges that a technique using layer features must resolve:

- Which layers should be selected for checking the classification of a test instance? For example, does selecting more layers lead to a better checker?
- How should the features from the different layers be aggregated — either to determine if an alarm should be raised, or to produce alternative advice?

Resolving these questions is the goal of this paper.

Problem statement. Given a trained DNN classifier and a test instance, we aim to develop a systematic method called SelfChecker for determining whether the DNN will misclassify the test instance, based on extensive checking the DNN's internal features. First, SelfChecker should trigger an *alarm* if it detects a potential misclassification of the test instance. Second, and going beyond the previous studies [17], [21], [22], SelfChecker should provide *advice* once an alarm is triggered, in the form of an alternative classification. Our goal is for SelfChecker to achieve high accuracy in both triggering alarms and offering advice.

III. Design of SelfChecker

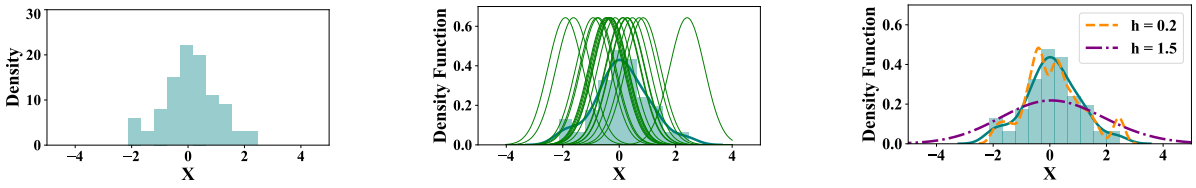
The goals of SelfChecker are (1) to check a DNN's prediction, (2) to raise an *alarm* if the DNN's prediction is determined to be incorrect, and (3) to provide an *advice*, or an alternative prediction.

SelfChecker's *training module* is used after the model has been trained to configure SelfChecker's behavior in deployment. The training module uses the training and validation datasets, as well as the trained model to generate a deployment configuration.

SelfChecker's *deployment module* runs along with the inference process: it analyses the internal features of a DNN when the model is given a test instance and provides an alarm as well as an advice if it detects an inconsistency in the model's output. To detect these inconsistencies, the deployment module uses the configuration supplied to it by the training module.

Note that although SelfChecker analyses the features extracted from the internal layers of a DNN, the training module is independent from the architecture of the model and requires no model modifications or retraining. The deployment module, however, is specific to a DNN.

Fig. 2 overviews our approach. Given a DNN model M trained on training dataset D_{train} and validated on validation set D_{valid}, for each layer in M, SelfChecker's training module first (1) computes layer-wise density distributions of each class using kernel density estimation (KDE) [27] on D_{train} (Section III-A). Based on the distributions, (2) SelfChecker can estimate the density values of each validation or test instance on each class. The higher the values of the class, the more similar the features of the instance in this layer are to the specific class. After SelfChecker obtains all of estimated density values on D_{valid} across all layers, SelfChecker (3) finds the



(a) Histogram (b) KDE (c) Bandwidth

Fig. 3. An example to illustrate KDE computation with *(a)* showing the set of input 1D points, *(b)* showing how to obtain the distribution using a KDE, and *(c)* showing the distributions obtained by using different bandwidths.

optimal layer combinations to reach the best alarm and advice accuracy. Since different classes produce distinctive feature behaviors in different layers, SelfChecker uses global search to find the optimal layer combinations per class (Section III-B).

Finally, when the model is presented with a test instance in deployment, SelfChecker's deployment module decides whether to provide an alarm as well as an advice by using (4) the density values and (5) specific layer combinations (Section III-C). We now detail each step in our approach.

A. KDE of the Training Set

Given a trained classifier M with L layers (except for the input layer) and C classes, let $\mathcal{X}^t = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$ and $\mathcal{Y}^t = \{y_1, \ldots, y_n\}$ in D_{train} be the set of training inputs and corresponding ground truth labels. Similarly, let $\mathcal{X}^v$, $\mathcal{Y}^v$, and $\hat{\mathcal{Y}}^v$ in D_{valid} be the validation inputs, corresponding ground truth labels, and model predictions.

We denote the outputs of all layers in the training set as feature vectors $\mathcal{V}^t = \{\mathbf{v}_1^t, \ldots, \mathbf{v}_L^t\}$, where the feature vectors of the layer l with n_l neurons are $\mathbf{v}_l^t \in \mathbf{R}^{n_l}$. We note that the feature vectors are trivially available after each execution of the trained model M over a given input. In general, M focuses on different features in different layers for different classes. SelfChecker's aim is to compute the density probability of feature vectors in each layer for each class based on the training set D_{train}. Using these density probabilities SelfChecker will then estimate how close the features in a specific layer (for a certain input) are to those of the training set.

KDE is a non-parametric method for estimating a probability density function by using a finite number of samples from a population [27], [28]. The resulting density function allows the estimation of relative likelihood of a given random variable. In this paper we use the Gaussian kernel, which works well for the multivariate data common to most datasets and produces smooth functions. Given a data sample $\{x_1, x_2, \ldots, x_m\}$, SelfChecker estimates the kernel density function f as follows:

$$\hat{f}(x) = \frac{1}{mh} \sum_{i=1}^{m} K(\frac{x - x_i}{h}) \quad (1)$$

where K is the Gaussian kernel function and h is *bandwidth*.

To see how a KDE with Gaussian kernels works, consider Fig. 3. First, each observation in the sample is replaced with a Gaussian curve centered at that value (green curves); these work as a kernel. The green curves are then summed to compute the value of the density at each point. Fig. 3(b) also shows the normalized curve (in blue) whose area under the curve is 1. The bandwidth parameter h of the KDE controls how tightly the estimate is fit to the sample data. It corresponds to the width of the kernels (green lines in Fig. 3(b)). Fig. 3(c) shows that if h is large, the curve is smooth but flat. And, if h is small, the curve is peaked and oscillating. The choice of h is based on the number of sample points and their dimensions.

For each combination of class and layer, SelfChecker uses Gaussian KDE to estimate the density function that the training data for the class induces on the layer's feature vector. Then given a test instance, SelfChecker estimates the probability density for each class within each layer from the computed density functions. Finally, SelfChecker uses these probability densities to infer classes for each layer, defined as follows:

Definition 1 (Inferred class for a layer): Given a test instance, the *inferred class for layer* l is the class for which the test instance induces the maximum estimated probability density among l's per-class density functions.

Algorithm 1 details SelfChecker's procedure for KDE estimation and inference. Lines 1-10 show the Gaussian KDE used to extrapolate the density distribution functions of feature

Algorithm 1: KDE Estimation and Inference

Input: Input instances in D_{train}, D_{valid}: $\mathcal{X}^t$, $\mathcal{X}^v$, true labels in D_{train}: $\mathcal{Y}^t$;
Trained model M with L layers and C classes;
Variance threshold: t_{var}
Output: KDE functions for each combination of class and layer: $kdes$;
Inferred classes for all layers on D_{valid}: $kdeInferL^v$

1 # Estimation
2 **for** *c in C* **do**
3 Obtain instances $\mathcal{X}_c^t$ whose true label is c;
4 **for** *l in L* **do**
5 $\mathbf{v}_{lc}^t = M.output_l(\mathcal{X}_c^t)$;
6 Remove elements in $\mathbf{v}_{lc}^t$ whose variance is less than t_{var};
7 $\hat{f}(x) = \frac{1}{|\mathbf{v}_{lc}^t| h} \sum_{i=1}^{|\mathbf{v}_{lc}^t|} K(\frac{x - \mathbf{v}_{lc}^t[i]}{h})$;
8 $kdes[l][c] = \hat{f}(x)$;
9 **end**
10 **end**
11 # Inference
12 **for** $\mathbf{x}$ *in* $\mathcal{X}^v$ **do**
13 **for** *l in L* **do**
14 $\mathbf{v}_l = M.output_l(\mathbf{x})$;
15 Remove values of the neurons filtered in the training set from $\mathbf{v}_l$;
16 **for** *c in C* **do**
17 $kde\_values[c] = kdes[l][c](\mathbf{v}_l)$;
18 **end**
19 $kdeInferL^v[\mathbf{x}.index][l] = max(kde\_values).index$;
20 **end**
21 **end**

vectors per class in each layer. As illustrated with Fig. 1, we want to extrapolate the patterns of the attention overlaid on the raw input. Since the input instances with different classes perform differently in different layers, the attentions in the first layer of digit 3 are different from the first one of 6 that is also different from the second one of 6 itself. SelfChecker therefore splits the original training input instances according to their true classes (Line 3). Based on these it obtains the outputs of each layer given the trained model M (Line 5). SelfChecker also uses mean-pooling to reduce dimensions for convolutional layers and then filters out neurons whose values show variance lower than a pre-defined threshold, t_{var}, to reduce the dimension of feature vectors as these neurons do not contribute much information to the KDE (Lines 6). SelfChecker then uses the filtered feature vectors to extrapolate the density functions for each layer and class, and stores them (Lines 7-8) so that they can be used for inference on new examples, such as D_{valid} (Lines 11-21).

During inference on a given input instance, SelfChecker first obtains the outputs in each layer (Line 14), from which it removes the values of the neurons filtered in Line 6 (Line 15). It then generates the estimated density values of each class, given the corresponding KDE functions (Lines 16-18). Finally, the layer inference for the input instance is the class that has the maximum density value (Line 19), which indicates that the feature vectors of the input instance in this layer are close to those in training set that belong to this specific class. For instance, in Fig. 1, the class inferences given by Algorithm 1 in the three layers are 3, 3, 2 for digit 3, and 1, 6, 6 for digit 6, respectively.

B. Layer Selection

In Section II we noted that different layers have different attentions, but some of these focus on a particular part of the image and may be misleading. For example, in Fig. 1 the second and third layers for 6 are different from the final prediction. If SelfChecker would consider the outputs of these layers, it can detect that the model is not confident about the final output. And, if SelfChecker considers just these layers and uses maximum voting, then it can also provide an alternative prediction that correctly classifies this image. Therefore, the design of *robust layer selection* in SelfChecker is important to accurately raise an alarm and to provide a high quality advice.

We first explain what we mean by a model output's *confidence*. Our definition is based on an observation: given a test instance, if the features of DNN layers are different from the final prediction, then the decision made by the model on the test instance will tend to be incorrect. For example, in Fig. 1 the attentions in the second and third images of a 6 are more similar to those of a 6 instead of the final prediction of 1. In this case the model misclassifies the 6 as 1. We evaluated this observation by using Spearman rank-order correlation coefficient and p-values [29]. Spearman rank-order measures the relationship between the prediction correctness and the consistency of inferred layer classes and final predictions. Our results show that they are correlated with p-value much less than 0.05 (at most 3.09e-26) on all evaluated four image datasets and three DNN models listed in Table I.

We formally define the confidence (δ) of a model output ($\hat{y}$) given a test instance $\mathbf{x}$ as follows:

$$\delta = \frac{N_{kdeInferL_x == \hat{y}}}{N_{selectedLayerC_{alarm}[\hat{y}]}} \tag{2}$$

where $N_{kdeInferL_x == \hat{y}}$ is the number of selected layers whose inferred class is the same as the final prediction $\hat{y}$ and $N_{selectedLayerC_{alarm}[\hat{y}]}$ is the number of selected layers for the class $\hat{y}$. Based on the maximum voting, if δ is lower than 0.5, we say that a DNN has *low confidence* in prediction $\hat{y}$ for a test instance $\mathbf{x}$.

We now discuss how SelfChecker selects the proper layer combinations for each class to reach a high alarm accuracy (Algorithm 2). We use the training set to estimate the density function, from which the inferred class for each layer can be obtained for a given input instance. As mentioned in Section II, different layers have different attentions but some of these may be misleading, we thus use the validation dataset to select layers. Given the validation dataset D_{valid},

Algorithm 2: Layer Selection for Alarm

Input: Input instances in D_{valid}: $\mathcal{X}^v$, true labels and predictions: $\mathcal{Y}^v$, $\hat{\mathcal{Y}}^v$;
Total classes: C;
Inferred classes for all layers on D_{valid}: $kdeInferL^v$
Output: Selected layers for all classes: $selectedLayerC_{alarm}$

1 **for** *c in C* **do**
2 Obtain the indexes idx_c of instances $\mathcal{X}^v_c$ whose prediction $\hat{\mathcal{Y}}^v$ is c;
3 Generate all kinds of layer combinations $combL$;
4 **for** *layers* l_s *in* $combL$ **do**
5 **for** *l in* l_s **do**
6 y_s.add($kdeInferL^v[idx_c][l]$);
7 **end**
8 $KdePredPos$.add(index of $sum(y_s$!= $\hat{\mathcal{Y}}^v[idx_c]) >= sum(y_s == \hat{\mathcal{Y}}^v[idx_c]))$;
9 $TrueMisBehavior$.add(index of $\hat{\mathcal{Y}}^v[idx_c] != c)$;
10 TP = $TrueMisBehavior$ & $KdePredPos$;
11 FP = $\neg TrueMisBehavior$ & $KdePredPos$;
12 FN = $TrueMisBehavior$ & $\neg KdePredPos$;
13 $F1 = 2 * TP/(2 * TP + FN + FP)$;
14 **if** *F1 is max* **then**
15 $selectedLayerC_{alarm}[c] = l_s$;
16 **end**
17 **end**
18 **end**

SelfChecker splits the input instances into C subsets based on their predictions (Line 2). SelfChecker then generates all possible layer combinations with lengths in range 1 through L, from which it searches for the best combination for each class to reach the highest accuracy (Lines 4-17). To calculate the alarm accuracy, SelfChecker first obtains the inferred class of each layer in the given layer combination (Lines 5-7) based on the generated KDE inferences across all layers on D_{valid} ($kdeInferL^v$) by Algorithm 1. To conclude whether or not the model has made a wrong prediction for an input, SelfChecker considers the layers in the layer combination. If a majority of the layers indicate inferred classes that are different from the model prediction (the confidence δ is less than 0.5), then SelfChecker concludes that the model is wrong (Line 8). In

this case, if the model prediction is indeed different from the true label of this input, the alarm is correct (True Positive), otherwise, it is incorrect (False Positive). SelfChecker uses the F1-score to measure the alarm accuracy (Lines 10-13), and it selects the layer combination with the highest accuracy for the corresponding class (Lines 14-16).

Algorithm 3: Layer Selection for Advice

Input: Input instances in D_{valid}: $\mathcal{X}^v$, true labels and predictions: $\mathcal{Y}^v$, $\hat{\mathcal{Y}}^v$;
Total classes: C;
Inferred classes for all layers on D_{valid}: $kdeInferL^v$;
Selected layers for all classes: $selectedLayerC_{alarm}$
Output: Selected layers and weights per class: $selectedLayerPosC_{advice}$, $\mathbf{W}_{pos}$, $selectedLayerNegC_{advice}$, $\mathbf{W}_{neg}$;

1 **for** c_p *in* C **do**
2 Obtain the indexes idx_{c_p} of instances $\mathcal{X}^v_{c_p}$ whose prediction $\hat{\mathcal{Y}}^v$ is c_p;
3 Generate y_s given $selectedLayerC_{alarm}[c_p]$;
4 Generate all kinds of layer combinations $combL$;
5 $KdePredPos$.add(index of $sum(y_s$!= $\hat{\mathcal{Y}}^v[idx_{c_p}]) >= sum(y_s == \hat{\mathcal{Y}}^v[idx_{c_p}])$);
6 $TrueMisBehavior$.add(index of $\mathcal{Y}^v[idx_{c_p}]! = c_p$);
7 $FP = \neg TrueMisBehavior$ & $KdePredPos$;
8 **for** c_t *in* C **do**
9 idx_{c_t}.add(index of $KdePredPos$ where $\mathcal{Y}^v[KdePredPos] = c_t$);
10 Select layers $selectedLayerPosC_{advice}$ with highest accuracy acc_{max} from $combL$;
11 **if** $c_t = c_p$ **then**
12 $\mathbf{W}_{pos}[c_p][c_t] = len(idx_{c_t}) * acc_{max}/len(KdePredPos)$
13 **else**
14 $\mathbf{W}_{pos}[c_p][c_t] = len(idx_{c_t}) * acc_{max}/(len(KdePredPos) - FP)$
15 **end**
16 **end**
17 $KdePredNeg$.add(index of $sum(y_s$!= $\hat{\mathcal{Y}}^v[idx_{c_p}]) < sum(y_s == \hat{\mathcal{Y}}^v[idx_{c_p}])$);
18 $TN = \neg TrueMisBehavior$ & $KdePredNeg$;
19 Iterate Lines 8-16 to obtain $selectedLayerNegC_{advice}$ and $\mathbf{W}_{neg}$
20 **end**

After selecting the layer combinations for the alarm, SelfChecker must determine the layer combinations that give a good advice whenever SelfChecker raises an alarm about a prediction. Algorithm 3 details SelfChecker's procedures for layer selection to achieve the best advice accuracy. First, SelfChecker splits the validation set D_{valid} into C subsets (Line 2), and for each subset it searches for the best layer combination. Given the layers selected for alarms by Algorithm 2, SelfChecker generates the KDE inferred classes in these layers as in Lines 5-7 in Algorithm 2. Given a test instance, if the confidence of the model prediction (δ) is less than 0.5, SelfChecker concludes that the model misbehaved (Line 5). SelfChecker then searches for the best layer combination where the model predicts the input with label c_t as c_p (Lines 9-10). Since not all classes have correlation, SelfChecker obtains weights for different combinations (Lines 11-15). For example, 1 is prone to be misclassified as 7 but has little chance to be misclassified as 2. Subsequently, in Lines 17-19, SelfChecker finds the layer combination that achieves the highest accuracy for the case where the selected layers by Algorithm 2 indicate a negative decision (the model behaves normally).

Boosting strategy: SelfChecker searches for both positive and negative decisions made by the selected layers in Algorithm 2 in order to boost the quality of the alarm. In particular, if the layers selected by Algorithm 2 indicate an alarm but the advice given by $selectedLayerPosC_{advice}$ (Line 10) is the same as the model prediction, then SelfChecker does not raise an alarm. Similarly, if the layers selected by Algorithm 2 indicate that the model prediction is correct but the advice given by $selectedLayerNegC_{advice}$ (Line 19) is different from the model prediction, SelfChecker will raise an alarm.

C. Checking the Model in Deployment

SelfChecker checks a trained DNN in deployment. It raises an alarm if it disagrees with the model's prediction of a given test instance and also generates an advice (alternative prediction). Algorithm 4 presents this process.

Algorithm 4: Checking Model in Deployment

Input: Input instance and its prediction by M with L layers: $\mathbf{x}$, $\hat{y}$;
KDE functions for all layers and classes: $kdes$;
Selected layers for all classes: $selectedLayerC_{alarm}$, $selectedLayerPosC_{advice}$, $selectedLayerNegC_{advice}$;
Weights for advice: $\mathbf{W}_{pos}$, $\mathbf{W}_{neg}$
Output: $alarm$ and $advice$ z

1 Generate inferred class for each layer $kdeInferL$ using KDE functions $kdes$;
2 L_{alarm} = $selectedLayerC_{alarm}[\hat{y}]$;
3 Generate y_s given L_{alarm} and $kdeInferL$;
4 **if** $sum(y_s$!= $\hat{y}) >= sum(y_s == \hat{y}))$ **then**
5 initialize $prob$ with C dimensions;
6 **for** c *in* C **do**
7 L_{advice} = $selectedLayerPosC_{advice}[\hat{y}][c]$;
8 **for** l *in* L_{advice} **do**
9 $prob[c] = sum(kdeInferL[l] == c)$;
10 **end**
11 $prob[c] = prob[c] * \mathbf{W}_{pos}[\hat{y}][c]/len(L_{advice})$
12 **end**
13 $advice = max(prob[c]).index$;
14 **if** $advice$!= $\hat{y}$ **then**
15 $alarm = True$, $z = advice$
16 **else**
17 $alarm = False$
18 **end**
19 **else**
20 Iterate 5-18 if the alarm is not triggered initially;
21 **end**

First, SelfChecker generates inferred classes of all layers $kdeInferL$ using layer outputs and KDE functions $kdes$ obtained from Algorithm 1. Then, as in Lines 5-7 in Algorithm 2, SelfChecker generates y_s consisting of inferred classes given the selected layers for $\hat{y}$. If the output class $\hat{y}$ is *not* inferred in the majority of cases in y_s, then SelfChecker has an initial alarm that still needs to go through the boosting strategy (mentioned in the last section).

Lines 5-18 show that SelfChecker first generates the probabilities of each class given $selectedLayerPosC_{advice}[\hat{y}]$, which are weighted by $\mathbf{W}_{pos}$. If the class with the largest probability is still different from the model prediction $\hat{y}$, SelfChecker triggers the alarm and it selects the class with the largest probability as the advice. Otherwise, SelfChecker does not trigger the alarm. A similar strategy is used if the alarm is not triggered initially where the output class $\hat{y}$ is inferred in the majority of cases in y_s.

IV. Evaluation

In this section we present experimental evidence for the effectiveness of SelfChecker. The goal of our evaluation is to answer the following research questions.

TABLE I
DL MODELS AND DATASETS USED IN THE EXPERIMENTS.

Dataset	# Class	# Train	# Valid	# Test	DL models					
					ConvNet		VGG-16		ResNet-20	
					# Layers	*Accuracy%*	*# Layers*	*Accuracy%*	*# Layers*	*Accuracy%*
MNIST	10	50,000	10,000	10,000	8	99.36	16	98.87	-	-
FMNIST	10	50,000	10,000	10,000	8	92.13	16	93.75	20	92.74
CIFAR-10	10	40,000	10,000	10,000	8	80.45	16	92.17	20	92.08
CIFAR-100	100	40,000	10,000	10,000	-	-	16	66.79	20	69.52

ResNet-20 and ConvNet are seldom used for MNIST and CIFAR-100. We omit their results due to space limitation but we will release them with our code. DAVE-2 and Chauffeur for self-driving cars are regression models so we exclude them in this table.

A. Research Questions

RQ1. Alarm Accuracy: *How effective is SelfChecker in predicting DNN misclassifications in deployment?*

To evaluate the effectiveness of SelfChecker for raising alarms in deployment, we compare its alarm accuracy on the test dataset with related techniques, namely, SELFORACLE [21], DISSECTOR [22], and ConfidNet [17]. For the comparison, we chose the variant from SELFORACLE—the VAE (variational autoencoder)—that achieved the best performance against other SELFORACLE variants, with confidence threshold of 0.05. Since DISSECTOR did not provide the threshold for distinguishing beyond-inputs from within-inputs, we used the validation dataset to choose a threshold in the $0-1$ range with the highest F1-score and the best weight growth type from *linear*, *logarithmic*, and *exponential* defined in [22] with the highest Area Under Curve (AUC) for each dataset and DNN classifier. We also used the validation dataset to find the best threshold of failure prediction for ConfidNet to reach the highest F1-score.

RQ2. Advice Accuracy: *Does the advice given by SelfChecker improve the accuracy of a DNN?*

In cases where SelfChecker raises an alarm about a model prediction, we also determine whether it can provide an advice and the accuracy of this advice. To answer this question, we compare the advice accuracy of SelfChecker against the accuracy of the original DL model M. For self-driving cars, we use the dataset released by SELFORACLE. This dataset only includes anomalous/normal labels, which is not enough to provide realistic advice, such as turning right/left.

RQ3. Deployment Time: *What is the time overhead of SelfChecker in deployment for a given test instance?*

We consider what different algorithms do in deployment and evaluate the computation time of their deployment-time components. SelfChecker performs DNN computation, KDE inferences, and alarm and advice analysis. SELFORACLE uses the reconstructor to compute a loss and anomaly detector. DISSECTOR generates probability vectors and performs validity analysis[2]. ConfidNet computes an output using two DNNs.

[2]By contrast, Wang et al. [22] only include validity analysis. We believe that the probability vector generation must also be performed during deployment, since this is the input to validity analysis.

RQ4. Layer Selection: *Does the choice of layers for selection by SelfChecker have an impact on its alarm accuracy?*

Kaya et al. [20] characterized "over-thinking" as a prevalent weakness of DL models, which occurs when a DL model can reach correct predictions before its final layer. Over-thinking can be destructive when a correct prediction within hidden layers changes to a misclassification at the output layer (see Section II). Therefore, it is important to select proper layers for different classes. To evaluate the impact of layer selections on the alarm accuracy, we experimented with three layer selection strategies as discussed in Section IV-C: RQ4.

RQ5. Boosting Strategy: *Does the boosting strategy improve SelfChecker's alarm accuracy, particularly in terms of decreasing the number of false alarms?*

As discussed in Sections III-B and III-C, we use a boosting strategy to check whether or not to raise an alarm.

B. Experimental Setup

We evaluate SelfChecker on four popular datasets (MNIST [30], FMNIST [31], CIFAR-10 [19], and CIFAR-100 [19]) using three DL models (ConvNet [32], VGG-16 [18], and ResNet-20 [33]). We also compare the alarm accuracy of SelfChecker against SELFORACLE [21] for self-driving car scenarios evaluated on two publicly-available DL models, NVIDIA's DAVE-2 [7] and Chauffeur [34]. To reduce the possibility of fluctuation due to randomness, we ran all experiments involving MNIST, FMNIST, CIFAR-10, and CIFAR-100 three times and computed the average of all metrics. For the experiments involving the driving datasets, we ran each experiment just once, since we used pre-trained models released by the authors of SELFORACLE [21]. We conducted all experiments on an Ubuntu 18.04 server with Intel i9-10900X (10-core) CPU @ 3.70GHz, one RTX 2070 SUPER GPU, and 64GB RAM.

Datasets and DL models. Table I lists the number of classes and the number of training, validation, and test instances in each dataset, as well as the number of layers and the testing accuracy of all trained DL models. These datasets are widely used and each is a collection of images. ConvNet, VGG-16, and ResNet-20 are commonly-used DL models whose sizes range from small to large, with the number of layers ranging from 8 to 20. Table I presents the accuracy of each model we obtained for each dataset; these accuracies are similar to the

TABLE II
ALARM ACCURACY.

Dataset	DL	↑ TPR %				↓ FPR %				↑ F1 %			
		SO	*DT*	*CN*	*SC*	*SO*	*DT*	*CN*	*SC*	*SO*	*DT*	*CN*	*SC*
MNIST	*ConvNet*	18.75	60.94	60.94	**62.50**	4.39	0.24	0.58	**0.23**	4.69	61.42	48.45	**62.99**
	VGG-16	20.35	68.14	61.95	**74.34**	4.29	0.32	0.46	**0.31**	8.21	69.37	61.40	**73.68**
FMNIST	*ConvNet*	9.53	**47.65**	38.12	41.55	5.60	4.03	0.73	**0.51**	10.89	48.92	51.99	**56.33**
	VGG-16	8.00	**48.48**	43.36	46.88	5.75	4.16	0.98	**0.86**	8.24	45.98	54.86	**58.66**
	ResNet-20	9.64	**54.96**	47.66	51.79	5.69	3.76	1.14	**0.98**	10.57	54.14	58.74	**63.03**
CIFAR-10	*ConvNet*	5.01	61.43	58.57	**61.89**	3.97	9.83	2.29	**2.04**	8.26	60.86	69.73	**72.69**
	VGG-16	6.39	**53.77**	43.17	49.30	3.94	4.47	3.03	**1.16**	8.36	52.10	48.29	**60.50**
	ResNet-20	7.07	47.98	49.87	**52.15**	3.96	4.93	1.03	**0.64**	9.23	46.74	61.62	**65.35**
CIFAR-100	*VGG-16*	10.48	82.78	78.20	**84.22**	7.88	23.78	16.17	**6.57**	16.59	71.79	74.22	**85.31**
	ResNet-20	11.25	75.16	61.15	**80.97**	7.64	21.63	13.56	**7.09**	17.49	66.96	63.67	**82.14**
Driving	*DAVE-2*	76.85	-	-	**99.01**	**7.29**	-	-	9.37	46.43	-	-	**49.88**
	Chauffeur	81.15	-	-	**93.44**	4.77	-	-	**4.56**	32.25	-	-	**37.25**

SO, DT, CN, and SC stand for SELFORACLE, DISSECTOR, ConfidNet, and SelfChecker, respectively.

state-of-the-art. As mentioned in Section III, SelfChecker has a training module and a deployment module. The training and validation dataset were used in the training module, and the test dataset were used on the deployment module to evaluate the performance of SelfChecker.

For our experiments with NVIDIA's DAVE-2 [7] and Chauffeur [34] for self-driving cars, we used the dataset and pre-trained models released by the authors of SELFORACLE. There are 37,947 training images, 9,486 validation images and 134,820 testing images for DAVE-2 and 250,830 for Chauffeur. The testing images are collected by the self-driving car respectively equipped with the two trained DL models. The collection process stops when the car has collisions or out-of-bound episodes. Therefore, the testing images are different for the two DL models. DAVE-2 contains five convolutional layers followed by three fully-connected layers, while Chauffeur consists of six convolutional layers followed by one fully-connected layer.

Configurations. As discussed in Section III, we filter out neurons whose activation values show variance lower than a pre-defined threshold (t_{var} in Algorithm 1), as these neurons do not contribute much information to the KDE. For all research questions, the default variance threshold is set to 10^{-5}, and the bandwidth for KDE is set using Scott's Rule [35] based on the number of data points and dimensions.

Metrics. Given the KDE inferences of the selected layers, if more layers disagree than agree with the model output, SelfChecker triggers an alarm. We compute the confusion metrics (TP, FP, TN, and FN) as our measurement. Consequently, a True Positive (TP) is defined when SelfChecker triggers an alarm to predict a misclassification where the model output is indeed wrong. Conversely, a False Negative (FN) occurs when SelfChecker does not trigger an alarm on a real misclassification by the model. A False Positive (FP) represents a false alarm by SelfChecker, whereas True Negative (TN) cases occur when SelfChecker is silent on correct classifications. Our goal is to achieve (1) a high true positive rate (TPR = TP / (TP+FN)), (2) a low false positive rate (FPR = FP / (TN+FP)), and (3) a high F1-score (F1 = (2 * TP) / ((2 * TP) + FN + FP)).

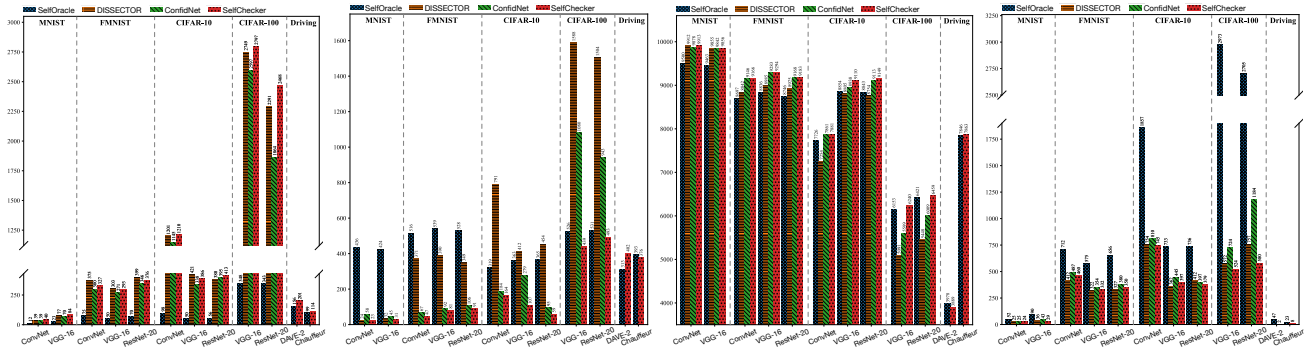
C. Results and Analyses

We now present results that answer our research questions.

RQ1. Alarm Accuracy

Table II presents the alarm accuracies of three DL models (ConvNet, VGG-16, and ResNet-20) in deployment on four datasets (MNIST, FMNIST, CIFAR-10, and CIFAR-100) checked by SELFORACLE, DISSECTOR, ConfidNet and SelfChecker, and the alarm accuracies of two self-driving car DL models checked by SelfChecker and SELFORACLE [21], in terms of TPR, FPR and F1-score. Fig. 4 shows the average confusion metrics of all datasets and DL models. SelfChecker can always trigger more correct alarms (TP) and miss fewer true alarms (FN) than SELFORACLE and ConfidNet.

On traditional DNN classifiers, SelfChecker correctly triggers an alarm on over half of the misclassifications (average TPR 60.56%), which is much higher than that of SELFORACLE (average TPR 10.65%) and ConfidNet (average TPR 54.30%), and comparable to DISSECTOR (average TPR 60.13%). In particular, the highest TPR of SelfChecker is 84.22%; this means that over 80% of misclassifications can be detected by SelfChecker. However, there are four cases on which DISSECTOR achieves higher TPR. Similar to SelfChecker, DISSECTOR also benefits from the internal layer features. It builds several sub-models that are retrained on top of internal layers. Therefore, additional information may be learned by the training process that SelfChecker lacks. But, SelfChecker outperforms DISSECTOR on TPR in the majority of cases, which indicates that the additional information is limited. Significantly, SelfChecker outperforms SELFORACLE, which has no internal information and ConfidNet, which only considers high-level representations on all datasets and DNN classifiers on TPR. We thus conclude that the internal layer features obtained by SelfChecker are important to detecting misclassifications. On the other hand, SelfChecker achieves lower FPR than all the competitors. The low FPR indicates that SelfChecker triggers few false alarms. This is expected since



(a) # True Positives (TP) (b) # False Positives (FP) (c) # True Negatives (TN) (d) # False Negatives (FN)

Fig. 4. Confusion metrics comparing the performance of all approaches.

the boosting strategy (Section III-B) makes SelfChecker very prudent in triggering alarms. Finally, SelfChecker has a higher F1-score than all the competing approaches with an average values of 68.07% against 10.25%, 57.83%, and 59.30% for SELFORACLE, DISSECTOR, and ConfidNet, respectively. The reason SELFORACLE has worse accuracy on traditional DNN classifiers is that it is tailored for time series analysis on video frame sequences that change little over short periods of time. ConfidNet is trained on top of the original DL model whose weights of feature extraction are frozen using the training dataset and it uses the loss function based on true class probability. Since there are few wrong predictions in the training dataset after the original model is trained, overfitting leads to limited performance of ConfidNet. Note that the results of ConfidNet shown in Table II are different from those in [17] since our study regards wrong predictions as positive cases (discussed in Metrics in Section IV-B) while [17] regards correct predictions as positive cases.

In the self-driving car scenarios, we transformed the regression network that predicts steering angles into a binary classification network that classifies steering angles as either normal or anomalous. Since the true class probability is the base of ConfidNet, and the first and second highest class probabilities are necessary for DISSECTOR, both of these cannot be used in the self-driving car scenarios. Given the validation dataset, a Gamma distribution is fitted to the errors between the predictions and the real-valued angles (MSE), and density values of each layer generated by Algorithm 1, respectively. Given an ϵ value of 0.05 (the same as used in SELFORACLE) from the Gamma fitting distribution, if the error of an instance in the validation dataset is larger than the value corresponding to ϵ, it is labeled as an anomaly. Similarly, if the density value is less than the values corresponding to ϵ, it is predicted as an anomaly. We then use SelfChecker to solve the regression problem as a binary classification problem. Table II shows that SelfChecker achieves a higher TPR than SELFORACLE on both DAVE-2 and Chauffeur, indicating that SelfChecker can trigger more correct alarms. Even though SelfChecker triggers more false alarms for DAVE-2, it also triggers more true alarms (201 against 156 by SELFORACLE) and misses only 2 true alarms. In addition, the F1-score for SelfChecker is higher than for SELFORACLE on both models.

For RQ1, we conclude that SelfChecker effectively triggers alarms that predict misbehaviors of DL models in deployment with high TPR and low FPR.

RQ2. Advice Accuracy

Table III compares the accuracies of the original model M to those of M having advice provided by SelfChecker. Even though SelfChecker achieves high alarm accuracies, it is challenging for it to provide correct advice as we regard the advice as correct only if the inferred classes of most selected internal layers are the same as the true label. This condition is more strict than triggering an alarm that requires the inferred classes of most selected internal layers to be different from the model's prediction. Our results show that

TABLE III
ADVICE ACCURACY.

Accuracy	Strategies	ConvNet	VGG-16	ResNet-20
MNIST	M	99.36	98.87	-
	M+SC	**99.37**	**99.21**	-
FMNIST	M	92.13	93.75	92.74
	M+SC	**92.34**	**93.78**	**92.80**
CIFAR-10	M	80.45	92.17	92.08
	M+SC	**80.63**	**92.41**	**92.11**
CIFAR-100	M	-	**66.79**	**69.52**
	M+SC	-	66.16	68.85

SC stands for SelfChecker.

even though the trained DL models have achieved state-of-the-art accuracies, the advice can still improve model's prediction accuracy by about 0.138% for datasets with 10 classes but decrease the prediction accuracy by about 0.65% for datasets with 100 classes. There are two reasons for this. First, finding a correct prediction from 100 classes is a harder problem. Second, the validation set per class is more limited: CIFAR-10 has 1000 samples per class but CIFAR-100 only has 100 samples per class. We empirically find that SelfChecker's advice can improve model's prediction accuracy when the number of samples per class is over 200. The results also show that the advice provided by SelfChecker can improve

the prediction accuracy at most 0.34% without retraining with additional inputs or changing the architecture. Even though this difference is small, for a safety-critical domain such as self-driving cars, which make tens of decisions per second, a difference of 0.2% in 10,000 decisions translates to 20 fewer misclassifications.

For RQ2, we showed that SelfChecker's advice can improve the accuracy of the original models beyond their state-of-the-art performance with a sufficiently large validation dataset.

RQ3. Deployment Time

We measured the average time that it takes a method to check a model's inference on a single input. Table IV lists the average times for all the datasets in Table II for each DNN classifier. The results for DAVE-2 and Chauffeur are for their corresponding self-driving datasets. SELFORACLE and ConfidNet take the least time since they use an additional DL model and their deployment checking time is the time it takes for two DL models to compute their outputs. However, these methods have alarm accuracies that are lower than DISSECTOR and SelfChecker. DISSECTOR takes longer than SelfChecker (average of 50.47ms vs 34.98ms) on traditional DNN classifiers.

TABLE IV
DEPLOYMENT TIME.

Time (ms)	SO	DT	CN	SC
ConvNet	0.96	29.74	0.98	26.47
VGG-16	1.35	58.34	1.02	35.83
ResNet-20	1.79	63.33	1.36	42.63
DAVE-2	45.80	-	-	67.78
Chauffeur	42.66	-	-	63.12

SO, DT, CN, and SC stand for SELFORACLE, DISSECTOR, ConfidNet, and SelfChecker, respectively

We believe that these checking times are acceptable across a variety of application domains. As is, SelfChecker can be used for applications ranging from medical image-based diagnosis to airport security screening. For real-time applications (e.g., autonomous driving), the latency of SelfChecker and SELFORACLE needs to improve. The checking time in the self-driving car scenarios is high because 32 frames must be analyzed before raising an alarm. Efficiency is not this paper's focus, but we acknowledge its importance for cyber-physical systems. We plan to parallelize SelfChecker by using a process per class density function to decrease latency by 1/(number of classes).

RQ4. Layer Selection

As discussed in Section III-B, we use search-based optimization to select suitable layers for improving alarm accuracy. We present the results of checking VGG-16 on FMNIST and Chauffeur on the self-driving car dataset in Table V; we omit results for the other models and dataset since they have similar properties. We evaluate three layer selection strategies for triggering alarms and compare them in terms of alarm accuracy. The first strategy involves random selection of layers for each class, with the number of layers selected for each class being the same as the number selected using our approach, in order to make a fair comparison. The second strategy uses the full set of layers. The third strategy is our own approach described in Section III-B, which selects suitable layers based on the validation dataset. To ensure a fair comparison, none of the strategies use the boosting strategy.

TABLE V
IMPACT OF LAYER SELECTION ON ALARM ACCURACY.

FMNIST	TP	FP	TN	FN	↑ TPR	↓ FPR	↑ F1
Random	280	482	8893	345	44.80	5.14	40.37
Full	209	230	9145	416	33.44	**2.45**	39.29
SC-layer[a]	317	329	9046	308	**50.72**	3.51	**49.88**
Chauffeur	**TP**	**FP**	**TN**	**FN**	**↑ TPR**	**↓ FPR**	**↑ F1**
Random	112	3059	5180	10	91.80	37.13	6.80
Full	99	2596	5643	23	81.15	**31.51**	7.03
SC-layer[a]	116	2978	5261	6	**95.08**	36.15	**7.21**

[a] SC-layer stands for SelfChecker's layer selection.

The results in Table V indicate that SelfChecker's layer selection strategy always achieves the highest TPR and F1-score compared to random selection and full selection. Even though using all layers to decide whether triggering an alarm achieves lower FPR than our approach, it sacrifices the number of correct alarms by 108 and 17 for FMNIST and driving dataset, respectively. Therefore, selecting more layers does not lead to a better checker.

For RQ4, we conclude that a careful selection of layers allows SelfChecker to identify more misclassifications and raise more correct alarms.

RQ5. Boosting Strategy

Table VI presents the alarm accuracies of SelfChecker both with (*SC*) and without (*SC-b*) the boosting strategy described in Section III-B, for ResNet-20 on FMNIST and CIFAR-100; we omit results for the other models and dataset since they have similar properties. As indicated in Table VI, adopting the boosting strategy achieves much lower FPR (the lower the better) than *SC-b*, with larger F1-score (the higher the better).

TABLE VI
IMPACT OF BOOSTING ON ALARM ACCURACY CHECKING RESNET-20.

FMNIST	TP	FP	TN	FN	↑ TPR	↓ FPR	↑ F1
SC-*b*	402	323	8951	324	55.37	3.48	55.41
SC	376	91	9183	350	51.79	**0.98**	**63.03**
CIFAR[a]	**TP**	**FP**	**TN**	**FN**	**↑ TPR**	**↓ FPR**	**↑ F1**
SC-*b*	2571	930	6022	477	84.35	13.38	78.52
SC	2468	493	6459	580	80.97	**7.09**	**82.14**

[a] CIFAR stands for CIFAR-100

For RQ5, we showed that the boosting strategy significantly improves alarm accuracy by reducing false alarms.

V. RELATED WORK

Most studies that check DL model trustworthiness focus on the process of model engineering: generate adversarial test instances [36]–[41], increase test coverage [42]–[44], and improve robust accuracy [32], [45]. Unlike our work, which

checks the model in production, these approaches rely heavily on manually supplied ground truth labels. Our focus is on non-adversarial inputs, which require different considerations [46]. We plan to consider adversarial inputs in our future work.

SelfChecker's performance will depend on the difference in distribution. We conducted preliminary experiments by *slightly* changing the testing dataset with random noise to push the dataset embeddings of the first fully-connected layer after all convolutional layers away from the training dataset. In this setup, SelfChecker performs similarly to the normal in-distribution dataset. Besides, there are existing studies detecting out-of-distribution data [47]–[49]. For example, recent work [49] uses temperature scaling and an input preprocessing strategy to make the max class probability a more effective score for detecting out-of-distribution data. Such studies are complementary to SelfChecker: they could first check for the input being out-of-distribution, and then SelfChecker can check the prediction. In addition, our problem cannot be subsumed by confidence calibration. As stated in ConfidNet [17], confidence calibration helps to create confidence criteria but ConfidNet's focus is failure prediction. Comparing SelfChecker against a technique with temperature scaling is inappropriate because using temperature scaling to mitigate confidence values doesn't affect the ranking of the confidence score on different classes and therefore cannot separate errors from correct predictions.

In the SE community, several studies consider checking a DL model's trustworthiness in *deployment*. SELFORACLE, proposed by Stocco et al. [21], estimates the confidence of self-driving car models. In their work, an alarm is triggered if the confidence of the model output is lower than a predefined threshold, in which case a human is then involved. It is designed for the scenario in which inputs are temporally ordered, such as video frames. Its performance is limited on other DNN types (see Section IV). Wang et al. [22] propose DISSECTOR to detect inputs that deviate from normal inputs. It trains several sub-models on top of the pre-trained DL model for validating samples fed into this DL model. But the generation of sub-models is manual and time-consuming, and DISSECTOR does not provide an explicit design of the threshold for distinguishing inputs, which depends on the model and dataset. In the DL community, researchers have developed new learning-based models to measure confidence [17], [50]–[53]. These models may also be untrustworthy and may suffer from, e.g., overfitting. In [52], [53], nearest-neighbor classifiers are built to measure the model confidence. A clear drawback of both approaches is the lack of scalability, since computing nearest neighbors in large datasets and complex models is expensive. Corbière et al. [17] propose a new confidence model, namely ConfidNet, on top of the pre-trained model to learn the confidence criterion based on True Class Probability for failure prediction, which outperforms [53] in both effectiveness and efficiency. But its performance is limited due to overfitting since it is trained on the training dataset where there are few wrong predictions. Except for [52], which cannot scale to large datasets and models, none of the above papers provide alternative advice. In contrast, SelfChecker achieves both high alarm and advice accuracy (with sufficient validation data per class) using internal features extracted from the DNN.

VI. LIMITATIONS AND CONCLUSION

Limitations. SelfChecker builds on an assumption that the density functions and selected layers determined by the training module can be used to check model consistency in deployment. This assumption depends on whether the training and validation datasets are representative of test instances. SelfChecker is a layer-based approach that requires white-box access and will have more limited power on shallow DNNs with few layers.

Conclusion. To be used in mission-critical contexts, DNN outputs must be closely monitored since they will inevitably make mistakes on certain inputs.

In this paper we hypothesized that features in internal layers of a DNN can be used to construct a *self-checking* system to check DNN outputs. We presented the design of such a general-purpose system, called SelfChecker, and evaluated it on four popular publicly-available datasets (MNIST, FMNIST, CIFAR-10, CIFAR-100) and three DNNs (ConvNet, VGG-16, ResNet-20). SelfChecker produces accurate alarms (accuracy of 60.56%), and SelfChecker-generated advice improves model accuracy on the 10-class dataset by 0.138% on average, within an acceptable deployment time (about 34.98ms). As compared to alternative approaches, SelfChecker achieves the highest F1-score with 68.07%, which is 8.77% higher than the next best approach (ConfidNet). In the self-driving car scenarios, SelfChecker triggers more correct alarms than SELFORACLE for both DAVE-2 and Chauffeur models with a comparable number of false alarms. SelfChecker is open source: *https://github.com/self-checker/SelfChecker*.

ACKNOWLEDGMENT

This work was supported in part by the National Research Foundation, Singapore and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office under the Trustworthy Software Systems – Core Technologies Grant (TSSCTG) award no. NSOE-TSS2019-05.

REFERENCES

[1] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[2] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[3] D. Ciresan, A. Giusti, L. M. Gambardella, and J. Schmidhuber, "Deep neural networks segment neuronal membranes in electron microscopy images," in *Advances in neural information processing systems*, 2012, pp. 2843–2851.

[4] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 2012, pp. 3642–3649.

[5] F. Zhang, J. Leitner, M. Milford, B. Upcroft, and P. Corke, "Towards vision-based deep reinforcement learning for robotic motion control," in *Proceedings of the Australasian Conference on Robotics and Automation 2015:*. Australian Robotics and Automation Association, 2015, pp. 1–8.

[6] M. Helmstaedter, K. L. Briggman, S. C. Turaga, V. Jain, H. S. Seung, and W. Denk, "Connectomic reconstruction of the inner plexiform layer in the mouse retina," *Nature*, vol. 500, no. 7461, pp. 168–174, 2013.

[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv:1604.07316*, 2016.

[8] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-Sec: Deep learning in android malware detection," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 371–372.

[9] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *Digital Avionics Systems Conference (DASC)*. IEEE, 2016, pp. 1–10.

[10] J. M. Chimento, W. Ahrendt, G. J. Pace, and G. Schneider, "StaRVOOrS: a tool for combined static and runtime verification of java," in *Runtime Verification*. Springer, 2015, pp. 297–305.

[11] S. Mitsch and A. Platzer, "ModelPlex: Verified runtime validation of verified cyber-physical system models," *Formal Methods in System Design*, vol. 49, no. 1-2, pp. 33–74, 2016.

[12] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 393–403.

[13] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," *arXiv:1610.02136*, 2016.

[14] J. Steinhardt and P. S. Liang, "Unsupervised risk estimation using only conditional independence structure," in *Advances in Neural Information Processing Systems*, 2016, pp. 3657–3665.

[15] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[16] V. T. Vasudevan, A. Sethy, and A. R. Ghias, "Towards better confidence estimation for neural models," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 7335–7339.

[17] C. Corbière, N. Thome, A. Bar-Hen, M. Cord, and P. Pérez, "Addressing failure prediction by learning model confidence," in *Advances in Neural Information Processing Systems*, 2019, pp. 2902–2913.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.

[19] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[20] Y. Kaya, S. Hong, and T. Dumitras, "Shallow-deep networks: Understanding and mitigating network overthinking," in *International Conference on Machine Learning*, 2019, pp. 3301–3310.

[21] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 359–371.

[22] H. Wang, J. Xu, C. Xu, X. Ma, and J. Lu, "Dissector: Input validation for deep learning applications by crossing-layer dissection," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 727–738.

[23] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Testing deep neural networks," *arXiv:1803.04792*, 2018.

[24] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[25] Z. Pan, Y. Liang, W. Wang, Y. Yu, Y. Zheng, and J. Zhang, "Urban traffic prediction from spatio-temporal data using deep meta learning," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 1720–1730.

[26] Y. Liang, K. Ouyang, L. Jing, S. Ruan, Y. Liu, J. Zhang, D. S. Rosenblum, and Y. Zheng, "UrbanFM: Inferring fine-grained urban flows," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 3132–3142.

[27] G. R. Terrell and D. W. Scott, "Variable kernel density estimation," *The Annals of Statistics*, pp. 1236–1265, 1992.

[28] R. A. Davis, K.-S. Lii, and D. N. Politis, "Remarks on some nonparametric estimates of a density function," in *Selected Works of Murray Rosenblatt*. Springer, 2011, pp. 95–100.

[29] D. Zwillinger and S. Kokoska, *CRC standard probability and statistics tables and formulae*. Crc Press, 1999.

[30] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[31] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[32] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1039–1049.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[34] T. Chauffeur. (2019) Steering angle model: Chauffeur. [Online]. Available: https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/chauffeur

[35] D. W. Scott, *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015.

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

[37] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the International Conference on Automated Software Engineering*, 2018, pp. 132–142.

[38] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2574–2582.

[39] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 427–436.

[40] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 408–426.

[41] Q. Li, Y. Qi, Q. Hu, S. Qi, Y. Lin, and J. S. Dong, "Adversarial adaptive neighborhood with feature importance-aware convex interpolation," *IEEE Transactions on Information Forensics and Security*, 2020.

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

[43] L. Ma, F. Zhang, M. Xue, B. Li, Y. Liu, J. Zhao, and Y. Wang, "Combinatorial testing for deep learning systems," *arXiv:1806.07723*, 2018.

[44] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[45] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "MODE: Automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 175–186.

[46] X. Zhang, X. Xie, L. Ma, X. Du, Q. Hu, Y. Liu, J. Zhao, and M. Sun, "Towards characterizing adversarial defects of deep learning software from the lens of uncertainty," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 739–751.

[47] S. Liang, Y. Li, and R. Srikant, "Enhancing the reliability of out-of-distribution image detection in neural networks," in *International Conference on Learning Representations*, 2018.

[48] K. Lee, H. Lee, K. Lee, and J. Shin, "Training confidence-calibrated classifiers for detecting out-of-distribution samples," in *International Conference on Learning Representations*, 2018.

[49] Y.-C. Hsu, Y. Shen, H. Jin, and Z. Kira, "Generalized odin: Detecting out-of-distribution image without learning from out-of-distribution data," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 10 951–10 960.

[50] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," in *Advances in neural information processing systems*, 2017, pp. 6402–6413.

[51] T. DeVries and G. W. Taylor, "Learning confidence for out-of-distribution detection in neural networks," *arXiv:1802.04865*, 2018.
[52] N. Papernot and P. McDaniel, "Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning," *arXiv:1803.04765*, 2018.
[53] H. Jiang, B. Kim, M. Guan, and M. Gupta, "To trust or not to trust a classifier," in *Advances in neural information processing systems*, 2018, pp. 5541–5552.