CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries

Hung Viet Pham, Thibaud Lutellier
University of Waterloo, Canada
{hvpham, tlutelli}@uwaterloo.ca

Weizhen Qi
USTC, China
zkdqwz@mail.ustc.edu.cn

Lin Tan
Purdue University, USA
lintan@purdue.edu

***Abstract*—Deep learning (DL) systems are widely used in domains including aircraft collision avoidance systems, Alzheimer's disease diagnosis, and autonomous driving cars. Despite the requirement for high reliability, DL systems are difficult to test.**

Existing DL testing work focuses on testing the DL models, not the implementations (e.g., DL software libraries) of the models. One key challenge of testing DL libraries is the difficulty of knowing the expected output of DL libraries given an input instance. Fortunately, there are multiple implementations of the same DL algorithms in different DL libraries.

Thus, we propose CRADLE, a new approach that focuses on finding and localizing bugs in DL software libraries. CRADLE (1) performs cross-implementation inconsistency checking to detect bugs in DL libraries, and (2) leverages anomaly propagation tracking and analysis to localize faulty functions in DL libraries that cause the bugs. We evaluate CRADLE on three libraries (TensorFlow, CNTK, and Theano), 11 datasets (including ImageNet, MNIST, and KGS Go game), and 30 pre-trained models. CRADLE detects 12 bugs and 104 unique inconsistencies, and highlights functions relevant to the causes of inconsistencies for all 104 unique inconsistencies.

***Index Terms*—deep learning software testing; cross-implementation testing; bugs detection; software testing;**

I. Introduction

Deep learning (DL) is widely used in many domains, including aircraft collision avoidance systems [1], Alzheimer's disease diagnosis [2], autonomous driving cars [3], and romance storytelling [4], [5]. Bugs in such systems can cause disastrous consequences, e.g., a software bug in Uber's self-driving car DL system has resulted in the death of a pedestrian [6].

Users of DL systems have a diverse range of background, including people with little technical backgrounds, e.g., singers/songwriters have used DL to compose music [7]. The pervasive use of DL systems requires them to be highly reliable.

Unfortunately, DL algorithms are complex to understand and use. Average users do not know all the details of DL algorithms. High-level DL Application Programming Interfaces (APIs) have been developed to enable users build DL systems without knowledge of the inner working of neural networks. These high-level APIs rely on lower-level libraries that implement DL algorithms.

Figure 1 presents the structure of typical DL libraries. Developers write code using high-level library APIs (e.g., Keras [8] API). These APIs invoke low-level libraries that implement specific DL algorithms. Low-level libraries such as TensorFlow (Google) [9], Theano [10], and CNTK (Microsoft) [11], implement the same algorithms, e.g., convolutional neural network (CNN) and recurrent neural network (RNN). Low-level libraries use different input formats and provide different APIs, while a high-level library allows users to seamlessly switch among different low-level libraries. The components that invoke low-level libraries are referred to as the *interfaces* between the high-level libraries and the low-level libraries. Each interface and low-level library, referred to as a *backend*, provides an implementation of DL algorithms. The backend trains and tests DL models. A DL *model* contains a DL network and parameters (also known as *weights*).

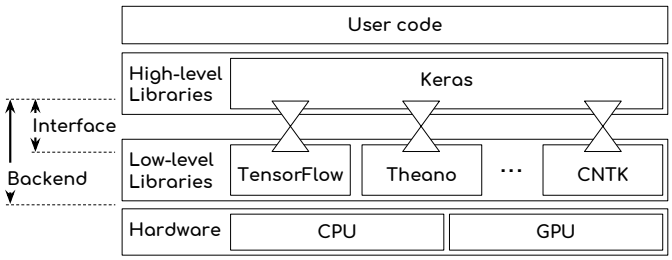
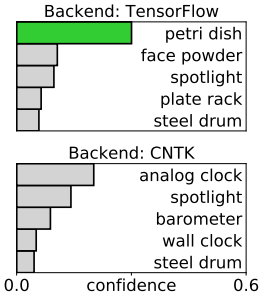


Fig. 1: Overview of DL libraries.

Keras [8] is the most popular high-level library for deep learning [12]. Keras has been used to implement neural networks in critical domains, including aircraft collision avoidance systems [1], inflammatory bowel disease diagnosis [13], chemical reaction predictions [14], medical imaging [15], [16], air quality control [17] and computer network security [18].

The backends and the high-level libraries contain bugs, which are particularly challenging to find and fix [19], [20]. One key challenge is that it is difficult for developers to know the *expected output* given an input instance. DL backends implement DL models that use complex networks and mathematical formula. Thus, it is hard for humans to produce the expected output of a DL backend given an arbitrary input instance, if possible at all. For example, given an input image of digit '1' (*ground truth* '1'), and a digit classification model, the expected output of that model on that image is not necessarily '1', as it is common for a model to misclassify due to its limitations (100% classification accuracy is rarely achieved). Existing DL testing work [19], [21]–[25] focuses on generating input instances that make the ground truth and the model output disagree so that DL users and builders can improve the model.

(a) Input image "Petri dish"



(b) Top-5 InceptiontionResNetV2

```
-   return (x-mean)/(C.sqrt(var)+epsilon)*gamma+beta
+   return (x-mean)/ C.sqrt(var +epsilon)*gamma+beta
```

(c) Bug fix in `batch_normalization` in the CNTK backend.

Fig. 2: A bug found by CRADLE in the CNTK backend, which has been fixed after we reported it.

Models must be implemented by backend libraries. If the backend libraries fail to faithfully implement a model (e.g., due to a bug in the backend), the output from the backend can be wrong even if the model is correct, and vice versa. An incorrectly-implemented DL backend may cause the aforementioned digit classification model to output '9' for the same image of '1', even if the expected output of the DL model is '7'. Alternatively, the DL backend may output '1' accidentally matching the ground truth. The wrong outputs could mislead DL users and builders in their debugging and fixing process. The output masks the implementation bug, which makes it challenging to be detected.

There has been little attention to testing the correctness of the models' *implementation*. Instead many techniques [19], [21]–[25] test the correctness of the *models*, which assume that the backend implementation is correct. Both the model and the backend implementation need to be correct for DL algorithms to produce a correct output. The critically important task of testing DL backend implementation is challenging since the expected output of the backend is hard to obtain as explained.

The multiple implementations (i.e., the DL backends) of the same functionality (i.e., the same DL algorithm) provide us a unique opportunity to detect inconsistencies among these implementations to find bugs in DL backend libraries. For example, if the **same** CNN model—which is the same CNN network with identical weights—behaves differently when running on the two CNN implementations (e.g., TensorFlow and CNTK), one of the CNN implementations is likely to be incorrect, without knowing the expected output.

Figure 2 shows a bug that causes two backends to be inconsistent. The input image (Figure 2a) is manually labeled as a petri dish (the ground truth) in ImageNet (a popular dataset of manually labeled images) [26]. Figure 2b shows the classification results of this image by the pre-trained model, InceptionResNetV2 [27], on Keras 2.2.0 with TensorFlow and CNTK backends respectively. While the model with TensorFlow backend classifies the image as a petri dish correctly as its first choice, the same model with CNTK classifies the image as an analog clock, with petri dish not in the top-5.

Once an inconsistency is detected, a big challenge is to identify the faulty functions among many functions in the DL backend libraries. For example, one run that exposes the inconsistency bug in Figure 2 contains 781 invocations of backend functions. Following the complex invocation path of the InceptionResNetV2 model, it is difficult for developers to tease out that the `batch_normalization` function is faulty.

To automatically detect and localize such inconsistencies across DL backends, we propose and implement a novel approach—*CRADLE*. Given a DL model and its input data, CRADLE (1) uses two distance metrics to compare the output of a model on different backends to detect inconsistent output, and (2) identifies the location of the inconsistency by tracking the anomaly propagation through the execution graph. By identifying the spike in the magnitude of the difference between two backends, CRADLE points out the inconsistent functions in the backend that introduces the inconsistency, which should be very useful for developers to debug and understand the bug.

Including the example in Figure 2, CRADLE identifies 580 images (out of a 5,000 random sample from ImageNet) that trigger inconsistent classifications for InceptionResNetV2 model. CRADLE then successfully localizes the faulty function (`bath_normalization`). After we reported this bug in the interface, developers have fixed the bug since Keras 2.2.1. Figure 2c shows the fix. The batch normalization formula was implemented incorrectly in CNTK backend's function `batch_normalization`: it should take the square root of `(var + epsilon)` instead of the square root of `var`.

To evaluate the effectiveness of CRADLE, we answer the following research questions:

RQ1: Can CRADLE detect bugs and inconsistencies in deep learning backends?
RQ2: Can CRADLE localize the source of inconsistencies?
RQ3: What is CRADLE's detection and localization time?

In this paper, we make the following contributions:

- A new approach to testing DL software by cross-checking multiple *implementations* of the same model to detect inconsistencies and bugs;
- The first approach to localizing the faulty function of a cross-model inconsistency, using anomaly propagation tracking and analysis; and
- An evaluation of the testing and localization technique on 30 DL models, 11 datasets (including ImageNet, MNIST, Udacity challenge 2, and KGS Go game), and 15 Keras versions (including the latest version).

Our results show that CRADLE detects **12 bugs** (9 have been fixed by developers) in DL software that cause inconsistencies for 28 out of 30 models, 3 of which are previously unknown bugs, 2 of which have already been confirmed by developers (RQ1). CRADLE highlights functions relevant to the causes of inconsistencies for all 104 unique inconsistencies (RQ2). CRADLE's median end-to-end running time is less than 5 minutes, suggesting that CRADLE is practical (RQ3).

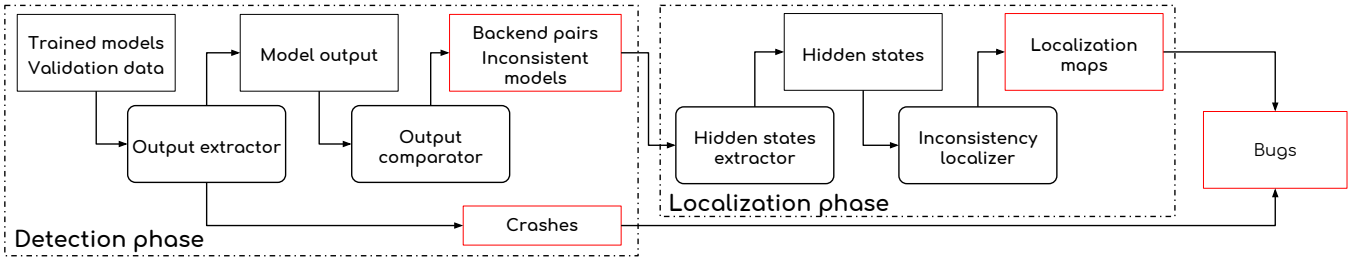


Fig. 3: Overview of CRADLE. Red boxes indicate CRADLE outputs.

II. Background

A DL *network* is a structure (i.e., a graph) that contains nodes or *layers* that are stacked to perform a specific task (e.g., regression or classification). Each layer represents a specific low-level transformation (e.g., convolution, pooling, etc.) of the input data with specific parameters (e.g., *weights*).

Each layer maps to a function invocation that converts weight and the input data to output. While multiple layers in a network can have the same type, the operation performed is generally different because the parameters of these layers are different. This is analogous to, in a traditional program, the same methods/functions, defined in one specific place in the source code, are called many times with different input parameters. Similarly, in a DL network, the same *layer type* can be called several times (i.e., in multiple layers) with different input parameters (i.e., weights). Fed with one input instance, a model maps to an *execution graph* of those low-level functions (i.e., layers).

As a DL network generally consists of more than two layers, there are many intermediate layers. Each intermediate layer produces an internal state that is fed to the next layers. We call such states *hidden states* because they are internal, to which normal users have no access.

To obtain the correct weights for each layer, the network needs to be trained on a *training set*. We call this phase the training phase. Once the training phase is over, the weights (or parameters) of each layer are fixed and do not change, and the model can be used in the inference phase. A *validation set* is a set of input, different from the training set, that is used to tune a model. In this work, we use it as input to the models because we know the ground-truth labels of such input.

A *pre-trained* model is a network that had been trained (and saved) in prior work. In the context of this paper, trained model also refers to pre-trained model. Its network structure and weights are fixed and do not change. While the training phase is often non-deterministic (e.g., the weights of the network can be initialized randomly), a pre-trained model is expected to behave deterministically in the inference phase because the weights of each layer do not change.

III. Approach

In this section, we describe how CRADLE detects and localizes inconsistencies among multiple backends. Recall that a backend consists of low-level libraries and the interface to high-level libraries (e.g., Keras). For example, the TensorFlow backend contains the TensorFlow library, the interface between Keras and TensorFlow, and the GPU computation library Nvidia CUDA invoked by TensorFlow.

A. Overview and Challenges

Figure 3 shows the two phases of CRADLE: the detection phase and the localization phase. The detection phase takes pre-trained DL models and their corresponding validation data as input. We focus only on the inference stage because of the non-deterministic nature of DL training.

CRADLE runs a pre-trained model using multiple DL backends. Specifically, the *Output extractor* feeds the validation set to the trained model as input and extracts the sets of output from the model on multiple backends. In general, we represent the output as a matrix of numbers. If a DL backend crashes during this extraction stage, the failure is recorded and later reviewed and reported. Otherwise, the *Output comparator* performs pairwise comparisons of the output for each model evaluated on different backends to detect inconsistencies.

Once an inconsistency is detected, CRADLE performs the localization phase. Specifically, the *Hidden states extractor* records hidden states of each inconsistent model on different backends. These hidden states are fed to the *Inconsistency localizer*, which produces localization maps where significant spikes in deviations propagating between hidden states on different backends are highlighted, indicating faulty locations.

To detect and localize cross-backend inconsistencies and bugs effectively, we need to address two main challenges:

1. How to determine if a model's outputs with two backends are inconsistent? Since different backends optimize the computational process differently, the results of the same calculation are almost always slightly different [28]. A naive approach that expects the output to be identical will detect inconsistencies for practically all models on all backends, which will not be useful for identifying bugs in DL systems. As shown by our experiment, Theano and CNTK backends always output slightly different values (the differences vary from 10^{-5} to less than 10^{-10}).

It is difficult to know how big of a difference indicates a bug-revealing inconsistency, due to the diversity of models, DL tasks, and datasets. It is not possible to have a single threshold to distinguish between bug-revealing inconsistencies and uninteresting inconsistencies for all models and datasets. For example, LeNet1, a model performing a simple image classification task has an average top-1 confidence level of

95%. This means that for this model, a small variation (e.g., a change in confidence level from 95% to 80%) is unlikely to make the label change. On the other hand, Betago is a model performing a complex task (i.e., playing Go). For this model, the average top-1 confidence level is only 60%. In this case, the same output variation (from 60% to 45%) might change the predicted label. Therefore, different models need different thresholds. Determining the correct threshold is a challenging problem as it depends on many parameters (e.g., dataset, model structure, training, etc.).

To address this challenge of identifying bug-revealing inconsistencies without the need for complex hard-coded heuristics, we use two distance metrics (refer to later sections for details) that emphasize the deviation between the output of both backends and the ground truth. These metrics effectively differentiate bug-revealing inconsistent runs from consistent runs and uninteresting inconsistent runs.

For these metrics, we compare the differences of outputs against the ground-truth instead of comparing individual outputs directly to the expected output. Recall that it is difficult to obtain the expected output as explained in the Introduction. We cannot directly compare the output of one backend to the ground truth to detect bugs because when one backend produces a wrong label it does not necessarily indicate a bug in the backend, as it is common for DL models to produce incorrect labels for some inputs (e.g., due to the limitation of the algorithm/model, not a bug in the implementation).

2. How to precisely localize the source of an inconsistency? After an inconsistency is detected, the internal source of the inconsistency is often challenging to localize, due to the complexity of DL backends. For example, one run that exposes the inconsistency bug in Figure 2 contains 781 invocations of backend functions that have complex mathematical connections. We propose a novel localization and visualization method that localizes faulty functions in the backend library which introduces inconsistencies by analyzing internal input and output of these backend functions and localizing the error spikes that propagate through the execution graph.

B. Detection Phase

In the detection phase, CRADLE identifies pairs of backends that are inconsistent for a specific model.

Output extractor takes as input a pre-trained model and its corresponding validation instances. It loads the provided weights (no training required) and performs classification or regression tasks using the loaded models. It produces the model output using all backends under test for each input instance. For example, comparing 5,000 validation instances and one associated model on 3 different backends will generate 15,000 output vectors. During this phase, CRADLE detects crashes on specific backends and we report them to developers.

Output comparator loads previously stored output matrices and performs pair-wise comparisons for each given validation instance to detect inconsistencies. These pair-wise comparisons are between a specific pair of backends using a particular model, its associated validation data, and a particular Keras version. The *Output comparator* then groups inconsistencies into unique inconsistencies. We use two metrics to compare a pair of backends—the Class-based distance for classification and the MAD-based distance for regression.

A straightforward metric to use is top-k accuracy on the entire validation set. Top-k accuracy calculates the portion of correct instances—an instance's ground-truth label is within the top-k predicted labels—among the total number of instances classified. Top-k accuracy could fail to identify certain inconsistencies. For example, the Dog species classification model, affected by the presented *Batch Normalization* bug, induces inconsistency between Tensorflow and CNTK. However, when ran on those backends, the model has identical top-1 (29.9%) and top-5 (64.4%) accuracies.

To overcome this problem, we calculate the portion of inconsistent input instances over the validation set. Because of the way inconsistent input instances are detected, we will not aggregate inconsistencies in the same way as top-k accuracy metric. In the following sections, we introduce Class-based and MAD-based distances as the ways to measure the severity of inconsistent instances. Once we have the severities of all validation instances between a pair of backends, we can apply two thresholds to see if that pair of backends is inconsistent.

Class-based distance is specific to classification models. It calculates the distance between two classifications based on the relative distances of the ground-truth label ranks in the output matrices. Here, we leverage the mapping between the syntax of the model output (the output vector) and its semantic meaning (the classification). Without this mapping, it would be difficult to come up with a universal metric and threshold that could work across different model configurations (e.g., the output vector size of a classifier can vary from 1000 for ImageNet models to 1 for binary classifiers).

A classification model with N classes outputs a vector of size N containing confidence level p_i corresponding to class C_i, where $0 < i \leq N$. Confidence level p_i shows how confident the model is in predicting class C_i as the correct label for that input instance. Given an output vector of a classification model as Y and the ground-truth label C of the input, we calculate the score of classification $\sigma_{C,Y}$ as:

$$\sigma_{C,Y} = \begin{cases} 2^{k-rank_{C,Y}} & \text{if } rank_{C,Y} \leq k \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

$rank_{C,Y}$ is the rank of the ground-truth label C in the classification Y. For example $rank_{C,Y} = 1$ if C is predicted as top-1 in classification Y. The score $\sigma_{C,Y}$ emphasizes on classifications that predict ground-truth label with higher rank. We consider $rank_{C,Y}$ out of top-k not interesting.

Given the confidence level output of the same model on a different backend as Y', the Class-based distance $\text{D\_CLASS}_{C,Y,Y'}$ is calculated as the absolute difference between two scores $\sigma_{C,Y}$ and $\sigma_{C,Y'}$:

$$\text{D\_CLASS}_{C,Y,Y'} = |\sigma_{C,Y} - \sigma_{C,Y'}| \quad (2)$$

TABLE I: Example of inconsistencies found using the Class-based metric. TF is TensorFlow and CN is CNTK.

				Inconsistency pattern					
Id	Keras	Backends	Model	16	15-8	7-4	3-2	1	0
1	2.2.2	TF-CN	Xception	10	202	147	100	85	4456
2	2.2.2	TF-CN	NASNetLarge	5	132	86	77	65	4635
3	2.2.1	TF-CN	Xception	10	202	147	100	85	4456
4	2.2.1	TF-CN	NASNetLarge	5	132	86	77	65	4635

We define our Class-based metric based on the top-k rankings with $k = 5$. For example, in Figure 2, $\sigma_{petridish,Y_{TF}}$ = 2^{5-1} = 16 as the rank of petri dish label by the TensorFlow backend $rank_{petridish,Y_{TF}}$ is 1. Similarly, $\sigma_{petridish,Y_{CN}} = 0$ because petri dish is not in CNTK's top-5 for that image. Then the Class-based distance D_CLASS$_{petridish,Y_{TF},Y_{CN}}$ is 16. If another backend generates the ground-truth label in rank 3, then its $\sigma_{petridish,Y}$ is 4, and D_CLASS$_{petridish,Y_{TF},Y}$ is 12. The maximum value of D_CLASS$_{C,Y,Y'}$ is 16, and the minimum is 0 with $k = 5$.

Mean absolute deviation (MAD)-based distance is a metric that could be used for both classification and regression models. However, the main purpose of the MAD-based distance is detecting inconsistencies in regression models where our Class-based distance would not work.

Given two predicted vectors Y and Y' of size N for a pair of backends using a model and an input instance, we first calculate the Mean Absolute Distance (MAD), $\delta_{Y,O}$ and $\delta_{Y',O}$, between the two output vectors and the ground-truth vector O. $\delta_{Y,O}$ is calculated as followed:

$$\delta_{Y,O} = \frac{1}{N}\sum_{i=1}^{N} |Y_i - O_i| \tag{3}$$

The MAD-based distance D_MAD$_{O,Y,Y'}$ is calculated as:

$$\text{D\_MAD}_{O,Y,Y'} = \frac{|\delta_{Y,O} - \delta_{Y',O}|}{\delta_{Y,O} + \delta_{Y',O}} \tag{4}$$

MAD is used here (instead of the more common Euclidean distance) because it does not inflate due to outliers.

For example, Dave-2 [29] is a model that outputs the steering angle (measured in radian) of a car given a dashboard camera image as input. For a given input image I, the recorded (ground-truth) steering angle is $O = 0.0$. Using the same image as input, Dave-2 outputs $Y = 0.4$ and $Y' = -0.1$ using two different backends. We have $\delta_{Y,O} = |0.4 - 0.0| = 0.4$ and $\delta_{Y,O} = |-0.1 - 0.0| = 0.1$. We can then calculate D_MAD$_{O,Y,Y'}$ as $|0.4 - 0.1|/(0.4 + 0.1) = 0.6$. MAD-based metric produces values between 0 and 1.

Before we can use this metric with classification models, we first need to convert the ground-truth labels to one-hot vectors. In multi-class classification, a one-hot vector is a vector of all zero except the value at the ground-truth label index is 1. This vector indicates a perfect classification with 100% confidence in the ground-truth label.

Identifying Inconsistencies: Given a model (and its validation set), two backends, and one version of Keras, we consider this pair of backends *inconsistent* if at least $p\%$ of validation input instances cause the distance between those two sets of output to be larger than a given threshold T (T_C denotes the threshold for the Class-based metric and T_M for MAD-based metric). We call such input instances *inconsistency-triggering*.

For Class-based metric with $k = 5$, using threshold $T_C = 16$ is the most strict. This means that an input instance is considered inconsistency-triggering if one backend ranks the ground-truth label top-1, while the other ranks it outside of the top-5. Using threshold $T_C = 1$ means that an input instance is inconsistency-triggering if there is any difference in the top-5 labels of the two backends and the ground-truth label is in the top-5 of at least one backend (e.g., if one backend ranks the ground truth label in the top-5, while the other backend ranks it outside of the top-5). In Figure 2, the petri dish image is an inconsistency-triggering input instance.

Similarly, for MAD-based metric, using $T_M = 1$ is the most strict. For example, with the Dave-2 model, an input image is *inconsistency-triggering* with $T_M = 1$ if it causes one backend to predict an angle matching the recorded angle exactly, while causing the other to predict a different angle. On the other hand, using $T_M = 0$ means that we consider any input image inconsistency-triggering.

The stricter the thresholds are the fewer inconsistencies are detected, however, the detected inconsistencies will be more severe (higher T_C or T_M means each inconsistency-triggering instance is more severe, while higher p means more output instances are inconsistent). If covering all inconsistencies is the priority, lower and more relaxed thresholds should be used (e.g., the recommended thresholds in Section IV). However, if finding severe bugs that significantly affect models' accuracies is the priority, then stricter settings would ensure that those severe bugs will be found and fixed quicker with less inspection effort.

Identifying Unique Inconsistencies: Table I shows four examples of inconsistencies. These inconsistencies are identified using the Class-based metric. Column '7–4' is the number of validation input instances that cause the two backends to have Class-based distances of 7, 6, 5, or 4. Inconsistency in row one (inconsistency 1) indicates that the model Xception is inconsistent between TensorFlow and CNTK (Keras 2.2.2) on its associated ImageNet validation set where 10 input instances trigger a Class-based distance of 16, 202 instances trigger distances in the range of 15–8, etc.

The same inconsistencies may exist in different Keras versions (different interface versions in the backend). To avoid finding duplicate inconsistencies, the output comparator also automatically groups certain inconsistencies together into unique inconsistencies based on inconsistency patterns.

An *inconsistency pattern* is the distribution of the distances over the entire validation data. It expresses the characteristics of the inconsistencies. Table I shows two unique inconsistency patterns: pattern 1 (for inconsistencies 1 and 3) and pattern 2 (for inconsistencies 2 and 4).

Since the range of MAD-based metric is between 0 and 1, we choose 5 equal sized bins between 0 and 1 to calculate the inconsistency patterns. Similar to Class-based metric, the number in bin 0.6-0.8 is the number of input instances that

trigger the MAD-based distance $0.6 \leq$ D_MAD < 0.8 for each pairwise comparison.

C. Localization Phase:

Given each unique inconsistency, the *Hidden states extractor* and the *Inconsistency localizer* produce a localization map. A localization map is an execution deviation graph of two implementations (backends), which highlights inconsistent executions (hidden states) of a function (layer type), pointing to potential faulty functions in one of the backends.

Recall that an execution of a model produces one execution graph (Section II). Each execution graph contains connected layers, where the output of one layer is the input of subsequent layers. Given a model and an input instance, there is one execution graph for each implementation of libraries. An *execution deviation graph* is a graph that represents the differences between two execution graphs of the same model. Since both execution graphs are from executions of the same model, they have the same structure i.e., the network structure. Thus, the execution deviation graph also has that same structure but contains the deviation between each pair of layer type executions. We describe the deviation calculation below.

For each unique inconsistency, we only perform localization on the most inconsistent input instance. *The most inconsistent input instance* triggers the largest Class-based distance (classification tasks) or MAD-based distance (regression tasks) between the output of two backends.

Hidden states extractor produces execution graphs in a similar way to the *Output extractor* described previously. Both execute the model on validation input instances to extract output. However, the latter also retrieves the intermediate function output (hidden state) of each hidden layer (internal execution) in the model. Hidden states are presented as vectors of floating point numbers.

Inconsistency localizer produces a localization map for each unique inconsistency by first extracting the execution deviation graphs. It does this by calculating the mean absolute deviation (MAD) between each pair of corresponding hidden state from two executions of the same layer type on two different backends. It is important not to confuse the usages of MAD here to the MAD-based metrics mentioned previously. Here, MAD is used to calculate the distances between corresponding intermediate outputs of hidden layers to represent the internal deviations of two execution graphs. Given the intermediate states S_L and S'_L of layer L executed on two backends, the deviation is calculated using Equation 3 as δ_{S_L,S'_L}.

Due to the sequential nature of a model, a noticeable MAD deviation at a particular layer does not indicate inconsistency at that layer as deviation can propagate through the execution graph and get amplified along the way. Ideally, we want to localize the source of the inconsistency. To do this, the *Inconsistency localizer* calculates the rate of change in deviation between consecutive function executions. Finally, it generates the localization maps by highlighting functions in the execution deviation graph that have inconsistent executions.

To calculate the rate of change, we first need to calculate the MAD deviation for all executions (layers output) in the set $pre(L)$ as δ_{S_l,S'_l} with $l \in pre(L)$ ($pre(L)$ is the set of inbound layers which hidden states are the input to layer L). We calculate the representative deviation of inbound executions, δ_{pre}, simply as the maximum deviation:

$$\delta_{pre} = \max_{l \in pre(L)} (\delta_{S_l,S'_l}) \tag{5}$$

The rate of change in deviations at layer L is:

$$R_L = \frac{\delta_{S_L,S'_L} - \delta_{pre}}{\delta_{pre} + \epsilon} \tag{6}$$

We use a smoothing constant $\epsilon = 10^{-7}$ to prevent $R_L = \infty$ in the case where $\delta_{pre} = 0$ (e.g., L is the first layer).

We call R_L the inconsistency introduction rate of a layer L, i.e., how much diversion layer L (executions of a pair of function implementations) introduces due to inconsistent implementations. R_L values of all layers provide an overall picture of how the inconsistency is introduced through the model so that we can localize the function that is the source of the inconsistency. To generate the localization map, we overlay the MAD and R_L values for each layer on the model structure graph (e.g., maps in Figure 4). A node, representing a layer L, shows the layer type (i.e., low-level transformation function), the MAD value δ, and the inconsistency introduction rate R_L. We select the third quantile of R_L distribution of all nodes in each map as the highlighting threshold. We highlight a node red if its R_L is higher than this threshold.

IV. Datasets and experimental settings

Trained Models and Datasets: To evaluate CRADLE, we collect 11 public datasets and 30 DL models that are pre-trained from these datasets. Table II lists the datasets.

We collected the models by looking for pre-trained models compatible with Keras from prior work and GitHub. To avoid low-quality models (e.g., class projects and simple demos), we only examine repositories with at least two stars. Overall, we collected 13 ImageNet [26] models (Xception, VGG16-19, ResNet50, InceptionV3, InceptionResNetV2, MobileNetV1-V2, DenseNet121-169-201, NASNetLarge-Mobile [8]), 3 self-driving models used in previous work (DaveOrig-Norminit-Dropout [19], [29]), 3 MNIST models (LeNet1-4-5 [30]), and various models trained for other tasks (Thai number detector – ThaiMnist [31], Go game player – Betago [32], anime faces recognition – AnimeFaces [33], cat and dog classifiers – CatDog(Basic, Augmented) [34], [35], dog species classifier – Dog [36], gender detection – Gender [37], Pokemon classifier – Pokedex [38], and GTSRB traffic sign recognition – TrafficSigns(1, 2, 3) [39]–[41]). We use provided validation dataset for each model to run our experiment. For ImageNet, we use a random sample of 5,000 images from over 80,000 provided cropped validation images.

Experimental settings: We run CRADLE on 15 versions of Keras (2.0.5–2.2.2). For the low-level libraries, we use the latest versions of CNTK (2.5.1), Theano (1.0.1), and

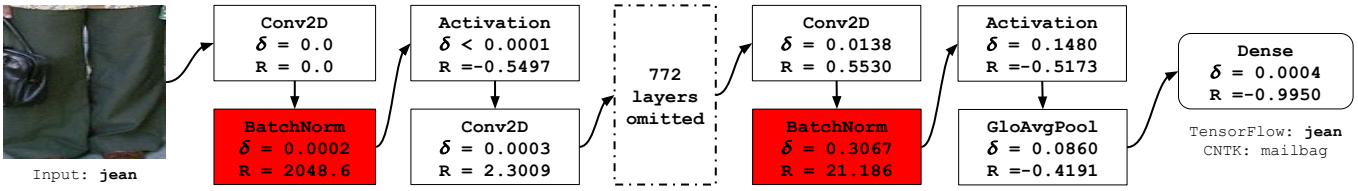


Fig. 4: Batch normalization bug's localization map for InceptionResNetV2 between TensorFlow and CNTK with Keras 2.2.0.

TABLE II: Number of inconsistencies found by CRADLE. The numbers outside and (inside) brackets are the unique and (total) number of inconsistencies respectively. TF is TensorFlow, TH is Theano, and CN is CNTK.

Dataset	Instances	# of Inconsistencies		
		TH-TF	TF-CN	CN-TH
ImageNet	5,000	10(34)	21(54)	18(46)
Driving	5,614		3(9)	3(12)
MNIST	10,000		3(9)	3(12)
Thai MNIST	1,665		1(3)	1(4)
KGS Go game	12,288	2(14)	3(12)	3(15)
Anime Faces	14,490	1(5)		1(6)
Dogs VS Cats	832		2(6)	2(8)
Dog species	835		3(8)	3(9)
Faces	466	2(14)	3(8)	6(15)
Pokedex	1,300	1(14)	1(3)	2(15)
GTSRB sign	12,630	2(14)	2(5)	2(7)
Total		18(95)	42(117)	44(149)
		104(361)		

TensorFlow (1.7.0). For regression models, i.e., Dave variants, we only use the MAD-based metric because the Class-based metric does not apply. For the classification models, we use both Class and MAD-based metrics. Some models are not supported with older versions of Keras and result in crashes. Since the crash is the expected behavior, we do not consider them as bugs and exclude those runs from our experiment.

We vary the thresholds (T_C, T_M, and p) and found the optimal setting (covering the most inconsistency without false positives and false negatives) for Class-based metric are $T_C = 8$ and $p = 0\%$ and for MAD-based metric are $T_M = 0.2$ and $p = 0\%$. We use cross-validation with 80-20% of models to confirm that the thresholds consistently perform across all 5 folds. These are the thresholds we use in RQ1 and RQ2.

Hardware and Infrastructure: We utilize multiple Anaconda environments to switch between multiple versions of Keras and different backends. We run all experiments on an Intel Xeon E5-2695 machine with 128 GB of RAM and two Nvidia Titan XP GPUs. For the performance analysis, we run the output extraction step utilizing a single GPU.

V. Results

A. RQ1: Can CRADLE detect bugs and inconsistencies in deep learning backends?

CRADLE detects **12 bugs** in DL software for 28 out of 30 models that cause **104 unique inconsistencies**. The 12 bugs (9 have been fixed) consist of 7 inconsistency bugs (3 previously unknown, 2 out of 3 have already been confirmed by developers, e.g., the bug in Figure 2 has been fixed by developers after we reported it), and 5 crash bugs that crash either Keras or one of the backend libraries. None of the 12 bugs

Fig. 5: Inconsistency-triggering inputs for the pooling bug (left column), the padding bug (middle column), and the batch normalization bug (right column). Faulty backends are bold.

is detected by the test cases that come with Keras (including the interface), which does simple unit and integration testing. The results demonstrate that cross-backend inconsistencies are frequent and CRADLE is effective in detecting them.

Our approach does not report false inconsistencies as it is a dynamic approach: for each inconsistency, we have inputs that trigger two backends to disagree. Theoretically speaking, some true inconsistencies may indicate a false bug, as our approach may identify uninteresting inconsistencies (e.g., natural computation difference explained in Section III-A). In our experiment, all 12 bugs are real (i.e., no false bugs detected).

Inconsistencies and inconsistency-triggering input: Using the Class-based metric on classification tasks and the MAD-based metric on regression tasks, CRADLE detected a total of 361 inconsistencies. Based on the inconsistency patterns, CRADLE automatically groups the inconsistencies into 104 unique inconsistencies (Section III-B).

Table II shows the number of inconsistencies found by CRADLE for each dataset and pair of backends. For example, CRADLE detects '21(54)' inconsistencies between the two backends TensorFlow and CNTK triggered by 13 ImageNet models. Here '21(54)' indicates that CRADLE detects 54 inconsistencies which map to 21 unique inconsistencies corresponding to 21 unique inconsistency patterns. Table I shows two of such patterns (the first and second rows).

On average, these inconsistencies are triggered by 21.9% of input instances in a dataset (22.2% for classification tasks and 13.9% for regression tasks). Figures 5 provide examples of inconsistency-triggering inputs. The image of a groom was identified correctly by TensorFlow but incorrectly as an Indian elephant by the faulty Theano. In some extreme cases, the faulty TensorFlow backend *accidentally* labels an image of bananas "correctly" while CNTK identifies it as tennis balls.

Inconsistency bugs: We use CRADLE to localize the source function of all 104 detected unique inconsistencies (detailed

TABLE III: Bugs found by CRADLE. '# Inc. bugs' indicates the number of inconsistency bugs per root inconsistency.

Root inconsistency	Localized layers (functions)	Affected backends	# Affected models	# Inc. bugs
Batch normalization	BatchNomalization	CNTK	11	2
Padding scheme	Conv2D, DepthwiseConv2D, SeparatableConv2D	TensorFlow, Theano	15	2
Pooling scheme	AveragePooling2D	Theano	3	1
Parameter organization	Trainable convolution	CNTK, Theano	18	2

localization results are in Section V-B). We find that they are caused by 7 bugs in the backend libraries (Table III). Some bugs have the same root inconsistency because they are either different bugs in the same function or affect several backends which required multiple fixes to multiple backends. For example, in addition to the batch normalization bug we presented earlier, we found another bug in the batch normalization function affecting an older version of Keras.

We manually check the fault localization maps for each cluster of inconsistencies and confirm whether it indicates a bug. If we find a corresponding bug fixing commit in a more recent version, we consider the bug has been fixed by developers. If not, we consider it previously unknown. Once two authors agree that it is a bug, we report it to developers.

If the same invocation of functions is identified for multiple bugs that are triggered by the same model in the same pair of backends across successive Keras versions (which affect the interface code between Keras and low-level libraries), we consider them one unique bug. However, if the bugs are in nonconsecutive versions, and the inconsistency pattern changes for some versions of Keras, this indicates that the issue was partially fixed (or a new bug introduced) in some Keras versions, then we consider them different bugs (e.g., the new inconsistency is likely to be a regression bug).

In addition to the batch normalization bug in Figure 2, we detail two additional confirmed bugs that CRADLE found.

Padding scheme bugs: Padding artificially increases the size of an input image so that a kernel function can be applied to all the pixels of the original image and produces an output of the same shape as the input. The `SAME` padding scheme behaves inconsistently across backends when applied on different combination of odd or even sizes of input and kernel. This creates a shift in the input that propagates through the model and caused the model to sometimes completely miss some of the shapes it was trained to recognize. Eventually, it results in inconsistencies between Theano or TensorFlow (depends on the different combination of input and kernel sizes) and the other two backends. The middle column of Figure 5 shows an example of input images revealing this bug. Although it has not been fixed yet in the interface source code, this bug has been confirmed to be a significant problem because various models (i.e., ResNet50, MobileNet, NASNetsLarge-Mobile, and MobileNetV2) have been updated by their developers to include workarounds that makes their models consistent across backends.

Pooling scheme bug: This bug in Theano backend causes Gender, InceptionResNetV2, and InceptionV3 models to misbehave. In Keras 2.1.4 and earlier, the 2D pooling layer in Theano interface determined the average pooling scheme based on the padding scheme. If the padding is `SAME`, it used the pooling `average_inc_pad` scheme which includes padding in the average calculation. However, if there is no padding, then they use the `average_exc_pad` scheme. This creates inconsistencies for models that use the AveragePooling layer with `SAME` padding. Figure 6 presents the fix where `average_exc_pad` is used regardless of the padding scheme.

```
-   if padding == 'same':
-     th_avg_pool_mode = 'average_inc_pad'
-   elif padding == 'valid':
-     th_avg_pool_mode = 'average_exc_pad'
  ...
-                   mode=th_avg_pool_mode)
+                   mode='average_exc_pad')
```

Fig. 6: Pooling scheme bug fix in `pool2d` in Theano backend.

Crashes bugs: Excluding crashes caused by unsupported models, we encounter 86 crashes out of 1173 possible runs. We identified 3 Keras bugs (happened with all backends) and 2 specific backend bugs. In total, 4 of the crash bugs have already been fixed and a workaround has been added to the crashing model to address the last issue. They are often caused by incorrect object's shape (e.g., incorrect weight or convolution kernel shapes).

Comparison between Class-based metric and top-k accuracy: One alternative to our Class-based metric is top-k accuracy. To measure its effectiveness in detecting inconsistencies, we integrate it into CRADLE by calculating the top-k accuracy differences between pairs of backends. A pair is considered inconsistent if the accuracy difference is larger than a threshold T_{AC}. We vary k (1 to 5) and accuracy threshold T_{AC} (between 0% and 50%).

Using $T_{AC} = 0\%$ and $k = 1$, the accuracy metric detects the most number of inconsistencies (305) but still misses 35 inconsistencies found by our Class-based metric. These are 35 valuable test cases that developers could use to test, localize, and fix detected bugs. In addition, our Class-based metric enables the generation of inconsistency patterns which help remove duplicates to reduce 340 detected inconsistencies to 98 unique inconsistencies. This reduction is not possible with top-k accuracy. The results show that our Class-based metric is more effective than top-k accuracy.

MAD-based metric usage for classification models: To demonstrate the usefulness of our Class-based metric, we compare the ability of both metrics in detecting unique inconsistencies for classification models.

Using the MAD-based metric for classification tasks, CRADLE can only find 10 unique inconsistencies, 4 of which are inconsistent in confident level but do not trigger inconsistent classifications. On the other hand, with the Class-based metrics, CRADLE correctly identifies 98 unique inconsistencies in classification models, including all inconsistencies correctly

found using the MAD-based metric. These results show that Class-based metric help CRADLE find more inconsistencies with no false positive.

B. RQ2: Can CRADLE localize the source of inconsistencies?

For each of the 104 unique inconsistencies, CRADLE generates a localization map for the most inconsistent input instance (Section III-C). By focusing on the first localized inconsistent execution and executions with high inconsistency introduction rates in each map, we manually cluster the 104 unique inconsistencies into 7 bugs. CRADLE's localization maps enable us to do this clustering. This manual process takes 1–2 hours per bug. A technique to automatically cluster unique inconsistencies based on the first localized function executions or similarity between localization maps remains as future work.

Overall, CRADLE highlights executions that are relevant to the causes of inconsistencies for all 104 unique inconsistencies. For 4 of the bugs, the first localized inconsistent executions are exactly the executions of faulty functions that were fixed by developers. This suggests that the localization technique is effective in pinpointing the faulty functions, which should help developers to understand and fix the bugs. For example, the reduction is 13 to 1 in one case, meaning that the developers only need to examine one function instead of 13 functions with complicated formula and interactions to understand and fix the bug. When we consider all (instead of only the first) localized inconsistent executions, the faulty methods are invoked in one of the localized inconsistent executions for 5 of the bugs. For the fifth bug, this represents a reduction of 22–44% for the number of functions to examine. For the remaining 2 bugs, the localized inconsistent executions are related to the bug fixes. In fact, the localized executions helped us tremendously in understanding the bugs so that we were able to write good bug reports.

Figure 4 shows a part of a localization map for the batch normalization bug (for the unique inconsistency involving InceptionResNetV2, TensorFlow and CNTK backends, and Keras 2.2.0). The input image shown is the most inconsistent input instance for this unique inconsistency. The Dense box shows the output: "jean" from TensorFlow, and "mailbag" from CNTK, while the ground truth is "jean". The map includes 781 invocations of backend functions, for presentation purposes, 772 of which are omitted. Each box represents an invocation of a neural network function, the arrows indicate the flow of data. Function names are indicated in each box, while δ is the MAD distance between the hidden states (defined in Equation 3), and R is the inconsistency introduction rate (defined in Equation 6). In this example, executions of function `batch_normalization` are localized as faulty (shown in red). The white boxes indicate executions with low or negative R (i.e., they are unlikely the source of inconsistency). This map correctly highlights the earliest invocation of the function `batch_normalization` as the source of inconsistency. We examine localization maps for the other affected models (e.g., InceptionV3, DenseNets (121, 169, 201)) and notice that they

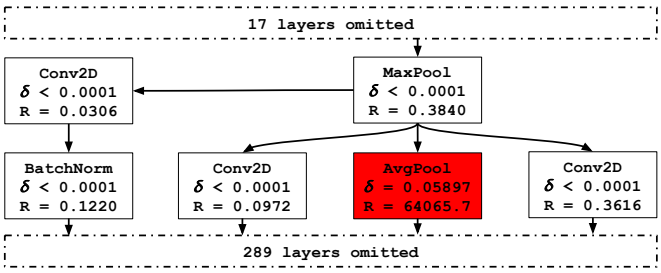


Fig. 7: Pooling scheme bug's localization map for model InceptionV3 between TensorFlow and Theano with Keras 2.1.4 on the "groom" input image in Figure 5.

all point to the `batch_normalization` function. We reported this bug to developers and it has been fixed in Keras 2.2.1.

Figure 7 shows a section of the localization map highlighting the faulty executions for pooling scheme bug with model InceptionV3 between TensorFlow and Theano on Keras 2.1.4. The first highlighted execution indicate correctly the source of this unique inconsistency as the function `average_pooling`. We look at the source code of `average_pooling` which points to the faulty `pool2d` function in the Theano backend. Figure 6 shows the fix (for Keras 2.1.5) in the Theano backend source code where the average pooling scheme is set to `average_exc_pad` regardless of the padding scheme.

C. RQ3: What is CRADLE's detection and localization time?

We measure the execution time of CRADLE on the latest version of Keras (2.2.2) using all 30 models. Overall, CRADLE's detection and localization time is quite reasonable with a typical end-to-end execution time lower than 5 minutes.

The running times of *Output extractor* and *Hidden states extractor* are dominantly the model execution times, which depends on the model complexity, validation dataset size, and performance of the backend. The extractor is slow in rare cases, e.g., nearly 10 hours with the large NASNetLarge model containing over 1,000 layers. However, the typical running time is within minutes with the median of less than 2 minutes.

The *Output comparator* and *Inconsistency localizer* are much faster with the median running time of less than 20 seconds and the maximum of less than 5 minutes. The running time is independent of the backend implementation; it depends on the dataset size and the model complexity respectively.

VI. Limitations and Threats to Validity

Since we focus on detecting bug-revealing inconsistencies, CRADLE may miss inconsistencies that cause internal errors but not failures (i.e., incorrect external behaviors). This is our design choice to avoid detecting too many false alarms.

We assume that the same algorithms are implemented with similar specifications in all backends due to the interchangeability of DL backends. In theory, it is possible for our technique to find false positive inconsistencies because of this assumption. However, our results show that the inconsistencies found by our approach indicate real bugs because 11 of them have already been confirmed or fixed by developers.

Our approach might not be generalizable to other models or DL libraries. To mitigate this threat, we use 30 models extracted from different GitHub projects and evaluate our approach on Keras, the most popular high-level DL library [12], and three popular backends. Our approach of detecting and localizing inconsistencies should be applicable to other models and libraries with little work.

It is possible that some complex DL systems contain non-deterministic layers so that given the same input the output might be slightly different. To mitigate this issue, we make sure none of the layers contains intentional sources of randomness and we apply two metrics that are designed to be robust even in the existence of small inconsistencies.

Our approach uses pre-trained models, which is our design choice in believing that those pre-trained models that are used by real users are likely to cause bugs that developers care. Alternatively, we can use dummy models or mutated models to test backends in order to find more bugs, which remains as future work.

VII. Related work

To the best our knowledge, we are the first to detect and localize inconsistencies between DL backends.

Testing machine learning (ML) libraries: Recently, automatic testing of ML libraries becomes active [42]–[44]. Srisakaokul et al. [43] detect inconsistencies between multiple implementations of common ML algorithms (i.e., kNN or Naive Bayes (NB)). This approach uses majority votes to estimate the expected output. However, it requires many implementations of the same algorithm (19 kNNs and 7 NBs used) with the assumption that most of them are correctly implemented. In contrast, CRADLE performs pairwise comparisons, which as shown by our experiments, detects inconsistencies without knowing the expected output and works with a minimum of two implementations. Another major difference is that Srisakaokul et al. define deviation based on the inconsistency of top-1 classifications without comparing them to the ground truth. CRADLE, on the other hand, define inconsistency as deviations in predicted ranks of the ground-truth label because we want to focus on inconsistent implementations that affect the performance of DL models on real world validation dataset. Dwarakanath et al. [42] test ML libraries by applying transformations on the training and testing data to detect inconsistencies. However, they were only able to identify artificially injected bugs. Dutta et al. [44] used fuzzing to test probabilistic programming systems. None of these techniques performs localization.

Benchmarking DL Libraries: Liu et al. [20] observe that the same DL algorithm with identical configurations, such as training set and learning rate, produces different execution time and accuracy when trained with different low-level DL libraries. However, this work aims to benchmark DL libraries, not to detect or localize inconsistency bugs, as it does not compare the exact same model on different backends. Since each model is re-trained on each backend and the training process contains non-determinism (e.g., the seed for the optimization function), small accuracy differences are expected. DL libraries have been compared in the literature [45]–[49]. However, the prior work focuses on performance comparison only and does not detect or localize non-performance bugs in DL libraries.

Adversarial Testing of DL Models: Much recent work focuses on testing DL models [19], [21]–[25], [50]–[55]. Many techniques generate adversarial examples [21]–[25]. Some work [50]–[52] verifies DL software. DeepXplore [19] introduces neuron coverage to measure testing coverage in CNN models. These approaches are orthogonal to our work as they test the correctness of DL models, while we test the correctness of the implementations of models in the DL software libraries.

Differential Testing and Inconsistency Detection: Differential testing [56] consists of testing whether different compilers produce the same results. Much work uses differential testing to find bugs in compilers by comparing the output of multiple compilers [57]–[59] or different compiler optimization levels [57], [60]. Inconsistency detection has been used in other domains such as cross-platform [61], [62], web browsers [63]–[66] or document readers [67]. Our work is a new application of differential testing and inconsistency detection for DL software, which has its unique challenges such as identifying bug-triggering inconsistencies (Section III-A). In addition, we localize the inconsistencies to the faulty functions.

Debugging and Fault Localization: We are not aware of prior work that localizes inconsistency bugs in DL libraries, despite the large volume of debugging and fault localization work for general software bugs [68]–[75]. While these approaches could be used to debug DL networks, applying such techniques to localize faulty functions in DL networks may have unique challenges such as scalability, which remains as future work.

VIII. Conclusion

We propose CRADLE, a new approach to find and localize bugs in the implementations of DL models by cross-checking multiple backends. We evaluate CRADLE on three backends and 30 pre-trained models and find 12 bugs and 104 unique inconsistencies in the backends for 28 models. This paper calls for attention for testing DL implementations not just DL models. In the future, we plan to design approaches to identify bugs even if they do not cause observable differences in backends. It is also conceivable to expand the set of trained models with mutants for CRADLE to find more bugs.

Acknowledgment

The authors thank Carmen Kwan for her contribution in collecting evaluation models from GitHub, Yitong Li for the reproduction and validation of experimental results, and the anonymous reviewers for their invaluable feedback. This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 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), 2016 IEEE/AIAA 35th*. IEEE, 2016, pp. 1–10.

[2] S. Liu, S. Liu, W. Cai, S. Pujol, R. Kikinis, and D. Feng, "Early Diagnosis of Alzheimer's Disease with Deep Learning," in *Biomedical Imaging (ISBI), 2014 IEEE 11th International Symposium on*. IEEE, 2014, pp. 1015–1018.

[3] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning Affordance for Direct Perception in Autonomous Driving," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2722–2730.

[4] R. Kiros, Y. Zhu, R. Salakhutdinov, R. S. Zemel, A. Torralba, R. Urtasun, and S. Fidler, "Skip-Thought Vectors," *arXiv preprint arXiv:1506.06726*, 2015.

[5] J. R. Kiros, "Recurrent Neural Network that Generates Little Stories About Images," https://github.com/ryankiros/neural-storyteller, 2018.

[6] A. Efrati, "Uber Finds Deadly Accident Likely Caused by Software Set to Ignore Objects on Road," *The information*, 2018.

[7] "Taryn Southern's new album is produced entirely by AI," https://www.digitaltrends.com/music/artificial-intelligence-taryn-southern-album-interview/, 2018.

[8] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a System for Large-Scale Machine Learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[10] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron *et al.*, "Theano: Deep learning on GPUs with Python," in *NIPS 2011, BigLearning Workshop, Granada, Spain*, vol. 3. Citeseer, 2011, pp. 1–48.

[11] F. Seide and A. Agarwal, "CNTK: Microsoft's Open-Source Deep-Learning Toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 2135–2135.

[12] "The Data Incubator," https://github.com/thedataincubator/data-science-blogs/blob/master/output/DL_libraries_final_Rankings.csv, 2018.

[13] D. Fioravanti, Y. Giarratano, V. Maggio, C. Agostinelli, M. Chierici, G. Jurman, and C. Furlanello, "Phylogenetic Convolutional Neural Networks in Metagenomics," *BMC bioinformatics*, vol. 19, no. 2, p. 49, 2018.

[14] S. Kwon and S. Yoon, "DeepCCI: End-to-End Deep Learning for Chemical-Chemical Interaction Prediction," in *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*. ACM, 2017, pp. 203–212.

[15] K. Chang, N. Balachandar, C. Lam, D. Yi, J. Brown, A. Beers, B. Rosen, D. L. Rubin, and J. Kalpathy-Cramer, "Distributed Deep Learning Networks Among Institutions for Medical Imaging," *Journal of the American Medical Informatics Association*, 2018.

[16] K.-H. Thung, P.-T. Yap, and D. Shen, "Multi-Stage Diagnosis of Alzheimer's Disease with Incomplete Multimodal Data via Multi-Task Deep Learning," in *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*. Springer, 2017, pp. 160–168.

[17] B. S. Freeman, G. Taylor, B. Gharabaghi, and J. Thé, "Forecasting Air Quality Time Series Using Deep Learning," *Journal of the Air & Waste Management Association*, pp. 1–21, 2018.

[18] N. N. Diep, "Intrusion Detection Using Deep Neural Network," *Southeast Asian Journal of Sciences*, vol. 5, no. 2, pp. 111–125, 2017.

[19] 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*. ACM, 2017, pp. 1–18.

[20] L. Liu, Y. Wu, W. Wei, W. Cao, S. Sahin, and Q. Zhang, "Benchmarking Deep Learning Frameworks: Design Considerations, Metrics and Beyond," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018.

[21] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing Properties of Neural Networks," *arXiv preprint arXiv:1312.6199*, 2013.

[22] 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.

[23] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial Examples in the Physical World," *arXiv preprint arXiv:1607.02533*, 2016.

[24] N. Papernot, P. McDaniel, A. Swami, and R. Harang, "Crafting Adversarial Input Sequences for Recurrent Neural Networks," in *Military Communications Conference, MILCOM 2016-2016 IEEE*. IEEE, 2016, pp. 49–54.

[25] N. Narodytska and S. P. Kasiviswanathan, "Simple Black-Box Adversarial Attacks on Deep Neural Networks," in *CVPR Workshops*, 2017, pp. 1310–1318.

[26] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[27] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," in *AAAI Conference on Artificial Intelligence*, 02 2016.

[28] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991. [Online]. Available: http://doi.acm.org/10.1145/103162.103163

[29] Z. Chen and X. Huang, "End-to-End Learning for Lane Keeping of Self-Driving Cars," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, June 2017, pp. 1856–1860.

[30] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[31] "Thai Handwriting Number," https://kittinan.github.io/thai-handwriting-number/, 2018.

[32] "BetaGo: AlphaGo for the Masses," https://github.com/maxpumperla/deep_learning_and_the_game_of_go, 2018.

[33] "Anime Face Dataset," http://www.nurs.or.jp/~nagadomi/animeface-character-dataset, 2018.

[34] "Cat vs. Dog Models," https://github.com/rajshah4/image_keras, 2017.

[35] "Dog Model," https://github.com/humayun/dl-dataday-workshop, 2018.

[36] "Model Weights File of Dog Project," https://github.com/humayun/dl-dataday-workshop/blob/master/code/dog_project/saved_models/weights.best.from_scratch.hdf5, 2018.

[37] "Gender Model," https://github.com/oarriaga/face_classification/, 2018.

[38] "Pokedex," https://github.com/Robert-Alonso/Keras-React-Native-Pokedex, 2018.

[39] "TrafficSigns1 Model," https://github.com/jaeoh2/CoreML-Traffic-Sign-Classifier, 2018.

[40] "TrafficSigns2 Model," https://github.com/inspire-group/advml-traffic-sign, 2018.

[41] "TrafficSigns3 Model," https://github.com/MidnightPolaris/gtsdb_cnn, 2018.

[42] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. P. J. C. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 118–128.

[43] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software," in *Proc. AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS)*, 2018.

[44] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, 2018*, 2018, pp. 574–586.

[45] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "DAWNBench: An End-to-End Deep Learning Benchmark and Competition," *Training*, vol. 100, no. 101, p. 102, 2017.

[46] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking State-of-the-Art Deep Learning Software Tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, Nov 2016, pp. 99–104.

[47] S. Dutta, B. Manideep, S. Rai, and V. Vijayarajan, "A Comparative Study of Deep Learning Models for Medical Image Classification," in *IOP Conference Series: Materials Science and Engineering*, vol. 263, no. 4. IOP Publishing, 2017, p. 042097.

[48] S. Shams, R. Platania, K. Lee, and S.-J. Park, "Evaluation of Deep Learning Frameworks Over Different HPC Architectures," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1389–1396.

[49] A. Shatnawi, G. Al-Bdour, R. Al-Qurran, and M. Al-Ayyoub, "A Comparative Study of Open Source Deep Learning Frameworks," in *Information and Communication Systems (ICICS), 2018 9th International Conference on*. IEEE, 2018, pp. 72–77.

[50] 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.

[51] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 97–117.

[52] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 3–29.

[53] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 303–314. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180220

[54] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 120–131. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238202

[55] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "AI 2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation," in *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[56] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[57] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[58] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random Program Generator for Java JIT Compiler Test System," in *Quality Software, 2003. Proceedings. Third International Conference on*. IEEE, 2003, pp. 20–23.

[59] F. Sheridan, "Practical Testing of a C99 Compiler Using Output Comparison," *Software: Practice and Experience*, vol. 37, no. 14, pp. 1475–1488, 2007.

[60] V. Le, M. Afshari, and Z. Su, "Compiler Validation via Equivalence Modulo Inputs," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 216–226.

[61] M. Fazzini and A. Orso, "Automated Cross-platform Inconsistency Detection for Mobile Apps," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 308–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155604

[62] M. E. Joorabchi, M. Ali, and A. Mesbah, "Detecting Inconsistencies in Multi-platform Mobile Apps," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, pp. 450–460.

[63] S. Roy Choudhary, H. Versee, and A. Orso, "WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609723

[64] S. R. Choudhary, "Detecting Cross-browser Issues in Web Applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1146–1148. [Online]. Available: http://doi.acm.org/10.1145/1985793.1986024

[65] S. Roy Choudhary, M. R. Prasad, and A. Orso, "CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications," in *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, 04 2012, pp. 171–180.

[66] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-PERT: a Web Application Testing Tool for Cross-Browser Inconsistency Detection," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 417–420.

[67] T. Kuchta, T. Lutellier, E. Wong, L. Tan, and C. Cadar, "On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files," *Empirical Software Engineering*, pp. 1–34, 2018.

[68] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and Improving Fault Localization," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 609–620.

[69] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 273–282.

[70] L. Naish, H. J. Lee, and K. Ramamohanarao, "A Model for Spectra-based Software Diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, p. 11, 2011.

[71] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the Mutants: Mutating Faulty Programs for Fault Localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2014, pp. 153–162.

[72] M. Papadakis and Y. Le Traon, "Metallaxis-FL: Mutation-based Fault Localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[73] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[74] G. Misherghi and Z. Su, "HDD: Hierarchical Delta Debugging," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 142–151.

[75] R. Hodován and Á. Kiss, "Modernizing Hierarchical Delta Debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. ACM, 2016, pp. 31–37.