Sanity Checker: Evaluations for Deep Neural Network Interpretability Tools

Brandon Liu
Department of Computer Science
University of Virginia
Charlottesville, VA 22901
bl6aw@virginia.edu

Yanjun Qi
Assistant Professor, Department of Computer Science
University of Virginia
Charlottesville, VA 22901
yq2h@virginia.edu

Abstract

Significant work has been done on interpretability in machine learning, but there is little agreement on how to measure the effectiveness of interpretability tools and evaluate them for benchmarking. We present Sanity Checker, a Python library as a novel solution to allow researchers to easily conduct evaluations on state-of-the-art interpretability tools. We explore the methods and implementation behind the design of the library and demonstrate its usage in a classic image classification task as well as in a novel DNA classification task.

1 Introduction

Effective interpretability and explanation tools are increasingly vital as deep neural network architectures become more ubiquitous in machine learning. Neural nets are used everywhere from self-driving cars and smart cameras to healthcare policy and DNA sequencing. Although they play a important role in peoples' livelihoods, they are often difficult to trust. Just last year, it was revealed that Amazon's candidate screening model was unexpectedly rating candidates in a gender-biased way. In an effort to address this problem and make these black-box models more transparent, researchers have placed a large focus on interpretability tools to be able to better understand how models work and why they make certain predictions.

There are many new interpretability tools out there, however not all can be easily evaluated by the researcher for the quality, scope and usefulness of the explanations they provide [1]. For this reason, it is important for researchers to have access to an easy method of verifying that the explanation method they have chosen actually yields "meaningful" explanations. There is currently little consensus on how to evaluate interpretability for benchmarking (towards a rigorous), and there are numerous examples of researchers relying on the wrong explanations for their desired goals.

Motivated by this problem, we introduce our Sanity Checker Python library as a novel solution to enable researchers to more easily perform evaluations on state-of-the-art deep learning interpretability tools. We explore the usage of this tool in two classification case studies. The first is on evaluating image-based interpretation tools using the MNIST hand-written digit dataset. The second takes a closer look at a biomedical setting with a DNA classification task. Sanity Checker mainly focuses on gradient based saliency explanations for the different settings however can be expanded to encompass many more methods. To the best of the authors' knowledge, this is the first library that makes a unified approach at evaluating interpretability methods.

2 Background and Related Work

Doshi-Velez [2] states that, in the context of machine learning, interpretability is the degree to which humans can understand the cause of a decision made by a model. Although there exists much

literature exploring the nature of how to make ML models interpretable for humans from a theoretical [3] as well as computational perspective[4], it is nonetheless difficult for researchers to quantify exactly how "interpretable" a model is and what exactly "interpretability" is in the first place. For example, Kim et al [3] presents a conceptual taxonomy for interpretability that is based on a hierarchy of the levels of human interactions involved in evaluations, while Guidotti et al [4] argues for a taxonomy based completely on different types of explanatory approaches. There are many other papers that argue for constraints or rules that make models interpretable [2, 5, 6, 3].

Despite the lack of consensus, there is nevertheless a growing number of tools researchers can rely on for explaining models, and thus a growing need to be able to evaluate those tools. Kim et al. [3] showed that saliency methods are unreliable when the explanation is significantly manipulated with a small perturbation in the input. Jain et al. [7] showed that attention weights do not necessarily explain model outputs using an adversarial approach. Kim et al. found that some widely used explanation methods are independent of the data the model was trained on as well as the model parameters and thus unfit for certain tasks [1].

Gradient explanation methods, the focus of this paper, began as a visualization tool for allowing better interpretability in machine learning models like convolutional neural networks. At first, feature activation methods like Deconvolutional Neural Networks (DeconvNets) were used to visualize the layers in a CNN by reconstructing the layers from the output of the network in order to obtain semantic image segmentation in classification tasks [8]. Saliency (gradient) map visualization began as a popular generalization of this DeconvNet procedure. Determining a saliency score involves computing the gradient of the class score with respect to the input. Saliency has traditionally been used in the image space [8], where visualizations are able to make meaningful interpretations for the researcher to highlight which parts of the image were "important" for the classification, but has since grown to include applications like text classification as well as DNA classification [9].

3 Methods

The central idea in evaluating how "meaningful" a given explanation is revolves around observing how the explanation changes with respect to a change in the model or a change in the data the model was trained on. Our library follows a simple structure: we combine an explanation, a sanity check method, and a similarity metric to make an informed evaluation.

3.1 Explanations

Explanation methods are what Sanity Checker uses to actually compute the importance of a certain feature in the output. We choose to focus mainly on the gradient-based explanations that are discussed above, however we show that any general structure of scores for an input will work in our model. The saliency gradients output a vector that represent a series of scores that determine how "important" each individual feature of the input is in predicting its final output class.

In our experiments, a model describes a classification function $S : \mathbb{R}^n \to \mathbb{R}^C$, where n represents the dimensionality of the input and C represents the number of classes. The explanations that we have implemented are listed below.

Saliency A technique first presented by Simonyan et al for the purpose of visualizing the output of deep Convolutional Networks. It involves just a single back propagation pass through a ConvNet to compute the gradient. Also known as the gradient explanation, it is $\frac{\partial S}{\partial x}$ for an input vector x. [8]

GradInput Gradient $\odot$ input, an element-wise mask of the input and the gradient, calculated $x \odot \frac{\partial S}{\partial x}$. [10]

Integrated Gradients A variation of the gradient explanation that also summs over scaled version of the input. [11]

ELRP Also known as epsilon-LRP, computes a layer-wise relevance propagation for non-linear classifier decisions. [12]

Guided Back-propagation Only propagates positive gradients by setting negative gradients to zero. The idea behind this is that we are only concerned with the features the neuron detects, not what it does not.

Guided GradCAM A generalization of the Class Activation Mapping, this detects important regions in a coarse localization map of the input. [13]

3.2 Sanity Checks

The core feature of the Sanity Checker is to randomly permute the trained model in different ways so as to be able to measure the impact on the above explanation methods and compare the results to the baseline model with no randomization.

Complete Model Randomization The model randomization sanity checker acts on a trained model and randomly permutes the weights of the model for each layer. This is equivalent to reinitializing the model with new, untrained weights but while retaining the same structure. When picking the new random weight values, we employ a uniform distribution [-1, 1]. For an explanation method to yield *meaningful* results, we expect that the model should depend highly on the learned parameters of the model and thus should change significantly as the weights change. If we make many changes to the weights of the model and the resulting explanation does not change as much, we can be sure that the explanation is not fit for tasks depending on model weights such as debugging the output class.

Cascading Model Randomization A variant of the above method, instead of reinitializing all of the weights of the model, we randomize the layers individually starting from the bottom up and compare the resulting explanations with the original. The purpose of this check is to see how the explanations handle increasing randomness.

Data Randomization We randomly permute the training labels of the input data before retraining the model. Doing this test ensures that we can check if the explanation truly depends on the labels of the training data. Much like above, if we are presented with a task requiring the explanations related to how the model was trained, we can use this check to determine if the method is truly making meaningful explanations.

3.3 Similarity Metrics

Similarity metrics are used to compare between a set of new explanation scores and original explanation scores after the model randomization has been performed. For the cascading model randomization, the similarity score here is calculated after every successive layer is randomized.

L2 Norm Measures the mean squared error between two differing sets of scores. With increased randomization, we expect this value to also be randomized among explanation scores.

Pearson Rank Correlation Measures the degree of similarity in how well the explanations can be described using a monotonic function. With increased randomization, we expect correlation near 0 among original and randomized explanations.

4 Experiments

We show example results of running Sanity Checker in two different settings. First we show its output in a classical machine learning setting using the MNIST dataset for image classification on hand-written digits. We then show Sanity Checker's effectiveness on a novel and much less common discrete sequence classification problem using a CHIPSEQ dataset for transcription factor binding in DNA sequences.

4.1 Case Study: MNIST Image Classification

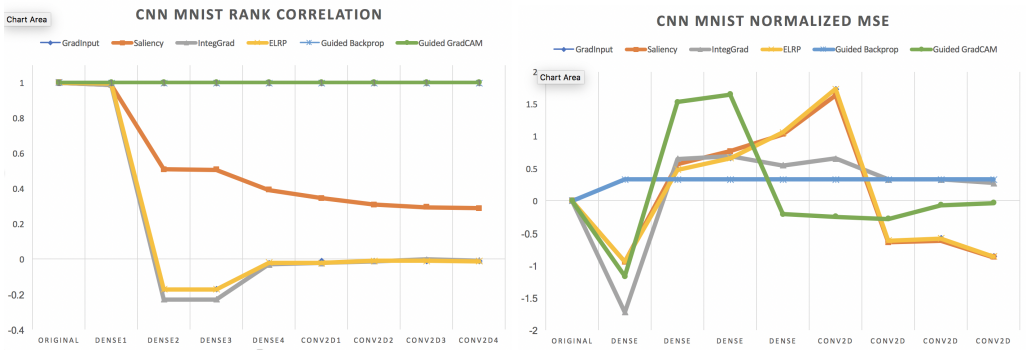
We train a CNN-based image classifier on 60000 hand-written digit training samples of size 28x28 to 98% accuracy. The goal of the model is to classify and recognize the digits from zero to nine. This

Table 1: Aggregate Sanity Check Results

Model	Method	Saliency	GradIn	IntGrad	ELRP	BackProp	GramCAM)
MNIST	Labels	0.0047	0.2713	0.0045	0.0031	0.9982	1.0000
MNIST	Model	-0.0124	0.2876	-0.0095	-0.0125.	0.9962	1.0000
DeepBind	Labels	0.0023	0.4357	-0.0036	0.0122	0.9883	1.0000
DeepBind	Model	0.0024	0.3519	0.0405	-0.0152	0.9952	1.0000

is a classic machine-learning problem and a good example of sanity checks in the image domain. The model itself contains four Dense layers and four convolutional layers with (3,3) kernels. We use a simple Keras model in conjunction with our Sanity Checker library to run 1) Complete Model Randomization, 2) Cascading Model Randomization, and 3) Data Label Randomization. The results of 2) and 3) are shown in Figure 1.

After performing a cascading randomization of the layer weights, we find that GradInput, IntegGrad, ELRP, and Saliency all drop from their initial correlation values as expected. However, we see that Guided GradCAM and Guided Back-propagation (stacked under GradCAM) actually do not change throughout multiple randomization layers. Our results here are also reflected on Figure 1b where Guided Back-propagation has the same normalized MSE throughout. We can conclude here that these two methods may not be desirable if we are trying to debug the output of the model.



(a) Aggregate Rank Correlation

(b) Aggregate Normalized MSE

Figure 1: Results for Sanity Checking MNIST Image Classification

4.2 Case Study: DeepBind DNA Classification

For the DNA classification setting, we use an online genomics model zoo called Kipoi. It contains pre-trained models for different tasks in genomics including transcription-factor binding site classification, DNA sequence classification, and protein prediction. Models stored on Kipoi can have a variety of backends (TensorFlow, Keras, PyTorch, etc.), making Kipoi essentially a wrapper around those trained functions. The repository itself has plugins for visualization and interpretability tools such as saliency, however it does not have the capability to perform sanity checks and evaluations on those tools.

The model we chose, DeepBind, performs a sequence-based DNA classification task for performing transcription factor site binding prediction. For an input DNA sequence of length 100, we predict a 0 if the sequence does not represent a binding site and a 1 if the site does. Since Kipoi is only a repository for pre-trained models, we gather 4000 input data DNA sequences from a ChipSEQ dataset for use in our data randomization test.

We show that our implementation is easily compatible with the models in Kipoi, and should thus be extensible to other pretrained online models as well. The results of the sanity check are displayed in Figure 2 and we are able to draw a similar conclusion as the sanity checks for our previous model.

(a) Aggregate Rank Correlation

(b) Aggregate Normalized MSE

Figure 2: Results for Sanity Checking DeepBind DNA Classification

5 Implementation

Sanity Checker is implemented in Python in order to be most up to date with state-of-the-art machine learning libraries and neural network tools. The goal of the library is to function both as a standalone package providing API functionality and as a command line utility. Furthermore, the library is built to be as modular as possible to be able to promote future extensibility. Currently, Sanity Checker supports models from Keras and TensorFlow, as well as hosted models from Kipoi. It can also be easily extended to support base PyTorch models as well as other standards like ONNX.

A significant challenge for this project revolved around the fact that many of these open-source machine learning frameworks behave differently under the hood when interacting with actual model objects. For example, for the task of randomizing weights of a pre-trained model's layers, the API functionality and way of randomizing layers was completely different between Keras and PyTorch. For one, Keras had more simple ways of accessing the inner layer weights data structure of every model, however PyTorch required a full reinitialization of the model in order to work. Additionally, functionality for saving and loading models was framework specific and sometimes required specific directory structures in order to work. For example, a PyTorch .h5 weights file did not actually contain class-specific model structure and required the actual model class Python file in order to function. By writing convenient API interfaces that provide functionality for randomization, saving, and loading models from different frameworks, we are able to protect the researcher from much complexity and frustration.

A large part of the implementation challenge for this project was to determine how to best make the functionality of the library as modular as possible. We adhered to software development best practices by making use of abstract base classes to define the functionality that would be implemented by framework-specific Sanity Checkers. In addition, the modules to compute explanations and perform comparisons can be used as standalone utility APIs for anybody seeking to make computations with NumPy arrays.

We use the Python ArgParse library to create an intuitive command line interface for interacting with the library. Running the command line interface gives information on how the library should be run and options to run different sanity checker algorithms given a path to a valid pre-trained model. The case study examples we show above can also be run directly from the command line.

6 Conclusion and Future Work

The goal of Sanity Checker was to provide researchers with an easy way to evaluate interpretability tools and explanation methods which may provide more insight into the models that they build. Sanity Checker successfully implements an easy-to-use command line interface that also pairs with a standalone library that researchers can use in conjunction with existing models.

The need for more work in this area is clear. Model builders need to understand how models reached a certain decision in order to make a more informed decision about how to improve them. Engineers

need to be able to better debug models and ensure that models do not carry unfair biases from their inputs to outputs. Model users need to be able to establish trust in black-box models.

We provide implementations of several explanations, sanity checks, and similarity metrics in our initial version of Sanity Checker that work well with Keras, Tensorflow, and PyTorch models. However, there is room for improvement in model compatibility. Future work should address support for ONNX-compatible models (a standard model format), as well as custom formats for model-building.

Sanity Checker can be easily extended to encompass more explanations, randomization methods, and similarity metrics. Examples include Structural Similarity Index (SSIM) for images and In-Silico Mutagenesis for DNA.

References

[1] Adebayo, Julius, Gilmer, Justin, Michael, Goodfellow, Ian, Hardt, Moritz, Kim, and et al., "Sanity checks for saliency maps," Oct 2018.

[2] Ross, Slavin, Doshi-Velez, and Hughes, "Right for the right reasons: Training differentiable models by constraining their explanations," May 2017.

[3] B. Kim, "Towards a rigorous science of interpretable machine learning," Jan 1970.

[4] Guidotti, Riccardo, Anna, Ruggieri, Salvatore, Franco, Dino, Giannotti, and Fosca, "A survey of methods for explaining black box models," Jun 2018.

[5] Samek, Wojciech, Wiegand, Thomas, and Klaus-Robert, "Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models," Aug 2017.

[6] A. Vellido, J. D. Martín-Guerrero, and P. J. G. Lisboa, "Making machine learning models interpretable - semantic scholar," Jan 1970.

[7] Jain, Sarthak, Wallace, and B. C., "Attention is not explanation," Apr 2019.

[8] Simonyan, Karen, Vedaldi, Andrea, Zisserman, and Andrew, "Deep inside convolutional networks: Visualising image classification models and saliency maps," Apr 2014.

[9] J. Lanchantin, R. Singh, Z. Lin, and Y. Qi, "Deep motif: Visualizing genomic sequence classifications," *CoRR*, vol. abs/1605.01133, 2016.

[10] Peyton and Anshul, "Learning important features through propagating activation differences," Apr 2017.

[11] Taly, Ankur, Yan, and Qiqi, "Axiomatic attribution for deep networks," Jun 2017.

[12] S. Bach, A. Binder, G. Montavon, K.-R. M. Frederick Klauschen, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," Oct 2018.

[13] Ramprasaath, Michael, Abhishek, Ramakrishna, Parikh, Devi, Batra, Dhruv, and Cogswell, "Grad-cam: Visual explanations from deep networks via gradient-based localization," Mar 2017.

[14] Adebayo, Julius, Gilmer, Justin, Goodfellow, Ian, and Kim, "Local explanation methods for deep neural networks lack sensitivity to parameter values," Oct 2018.

[15] Lundberg, Scott, and Lee, "A unified approach to interpreting model predictions," Nov 2017.

[16] C. Molnar, "Interpretable machine learning," Apr 2019.