Generating Adversarial Examples with SAT Solvers

COMP 597

Owen Smith
owen.smith@mail.mcgill.ca (260788385)

November 12th 2021

Abstract

In this paper we explore automated white-box testing for deep neural networks using a SAT solver to search for adversarial examples. Adversarial examples are curated inputs with the sole purpose of confusing a neural network, resulting in a misprediction. A major advantage of this approach is that the program is quite simple to attach to your own models. It requires the weights and biases of a pretrained model and an associated input and label. Without the need of gradients and a loss function, encoding the feedforward pass of a network as SAT is relatively straight forward. We include two demos as a proof of concept: one on a toy dataset and another on the MNIST dataset.

Background

Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models used for regression, machine learning, decision-making, and pattern recognition problems. At their core, ANNs aim to adjust its parameters to minimize a given loss function. Inspired by human brains with millions of biological neurons, an ANN consists of multiple *layers*, each containing numerous *artificial neurons* (Figure 1).

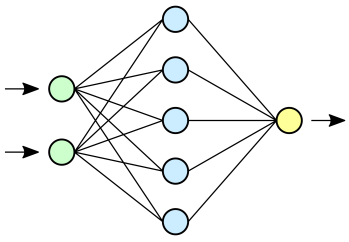

Figure 1: An example of a two layer[1] ANN. Where the circles represents the neurons and the edges connecting them are called the weights. Here, each layer is defined by the colour of its neurons. We start at the input layer (green), then move to the next layer (blue) which is commonly referred to as the hidden layer, and the last (yellow) is the output layer.

While the biological analogies are admissible, it is perhaps more convenient to think of a ANN as purely a mathematical function that maps inputs to outputs. Under the hood, a ANN includes five main components.

- An *input layer* $x \in \mathbb{R}^n$
- An *output layer* $\hat{y} \in \mathbb{R}^m$
- Any number of *hidden layers*
- A set of *weights* W and *biases* b between each layer
- A choice of an *activation function* σ for each layer[2]

The output $\hat{y}$ can then be simplified to one equation, for the two layer network in Figure 1, we have:

$$\hat{y} = \sigma(W_2\,\sigma(W_1\,x + b_1) + b_2)$$

Where W_1, b_1 are the weights and biases associated with entering the hidden layer (blue), and W_2, b_2 similarly for the output layer (yellow).

You may notice from the above equation that the only parameters affecting our prediction $\hat{y}$ is our weights and biases (if we fix the input). Thus, fine tuning the values of these components will be the key to our ANN making better predictions.

[1] We commonly exclude the input layer when counting the number of layers in a neural network.
[2] For this text we will use the Rectified Linear Unit (ReLU) activation function: $\sigma(u) = \max(u, 0), u \in \mathbb{R}$

We describe the process of producing $\hat{y}$ from our input x as the *feedforward* of our neural network. Whereas the process of fine tuning our weights and biases (ie. "learning") is called *backpropagation*. At a high level, we use a loss function to quantify the "goodness" or "strength" of our ANN. Higher loss implies less accurate predictions. If we treat the loss function as a multidimensional plane, we can perform gradient descent to travel downhill towards a global minimum. Thus, minimizing the loss and improving our ANN performance.

In this text we will be mostly focusing on the feedforward of a neural network because we will consider our models to be *pretrained*, that is backpropagation has already been conducted and so all the weights and biases are considered fixed.

Classification

In this text we will be focusing on ANNs built for *classification tasks*. Given a *labeled* input, our goal is to classify it to belong to a particular group of similar inputs. A label can be thought of as a ground truth value (Figure 2), generated by some expert (usually humans).

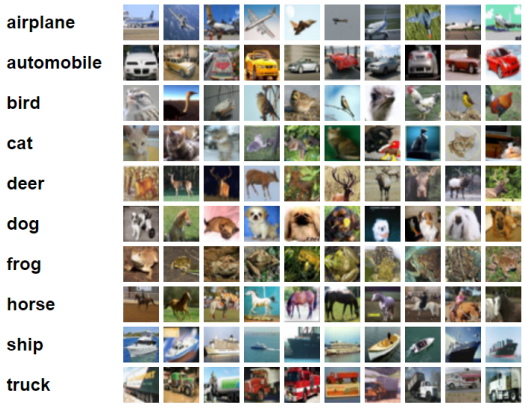

Figure 2: A set of example input images and their associated class label from the CIFAR-10 Dataset

In machine learning, we say that training a classifier is a *supervised learning* task. Here you can think of the "supervisor" as the label associated with an input. Classification is one of the most common applications of ANNs. We could use a ANN classifier to extract animals from images, identify ancient fossils, view traffic signs/signals, and much more. There are other methods (naive bayes, decision trees) to build a classifier, so we will refer to classifiers with an ANN as Deep Learning Classifiers (DLC).

Testing Deep Learning Classifier

Deep Learning Classifiers (DLC) are being increasingly deployed in safety and security critical domains including self-driving cars and malware detection, where the correctness and predictability of a system's behavior for corner case inputs are of great importance.

Existing DLC testing depends heavily on manually labeled data. So there exists a problem of finding such corner cases and as well as generating a correct associated label. While there are numerous techniques (such as colour-jitters, random cropping, blurring[3]) to manipulate your training dataset to generate more samples, networks often fail to identify and cover the behaviors for rare inputs.

Adversarial Examples

Adversarial Examples serve to counter the aforementioned problem. They are the outcome of modifications (referred to as a *mask*) applied to an input image with the sole purpose of causing a misclassification of the input (Figure 3).

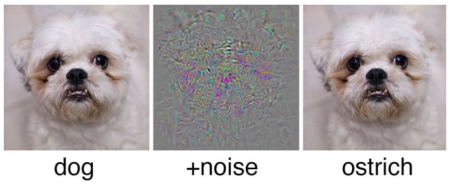

Figure 3: Our input image (left) is correctly predicted to be a dog, but after applying some noise onto the image (middle), we produce an adversarial example (right) where the model now predicts that it is an ostrich.

This process not only exposes the model's behaviours on rare inputs, but can also lead to major improvements in the performance of our model. By feeding these curated inputs back into our training set, we now have a set of new high-quality inputs for our network to learn from.

[3] pytorch.org/vision/stable/transforms.html

Related Work

Most of the motivation for this project stems from the work done for DeepXplore in 2017. Here the authors develop the first white-box framework for systematically testing real-world systems that include DLCs.

DeepXplore is capable of efficiently finding thousands of incorrect corner case behaviors in state-of-the-art deep learning models. For each of the five popular dataset used in the paper, DeepXplore was able to generate a test input demonstrating incorrect behaviour within one second.

Test result show that samples generated by DeepXplore was able to improve classification accuracy by up to 3%[1].

DeepXplore works to maximize *neuron coverage*. Neuron coverage of a set of test inputs is defined as the ratio of the number of unique activated neurons for all test inputs and the total number of neurons in the DLC.

More formally, if we assume all neurons of a deep neural network are represented by the set $N = \{n_1, n_2, ...\}$, all test inputs are represented by the set $T = \{x_1, x_2, ...\}$, and $out(n, x)$ is a function that returns the output value of a neuron n for a given test input x. The authors define neuron coverage $NCov$ as follows,

$$NCov(T, x) = \frac{|\{n|\forall x \in T, out(n, x) > t\}|}{|N|}$$

DeepXplore considers a neuron to be activated if its output is higher than a threshold value t.

Problem

A major drawback of DeepXplore is the accessibility to incorporate your own models. In fact, the current release of DeepXplore on GitHub[4] does not support training your own models. Our goal is to leverage the algorithms of a SAT solver to easily encode a model and effectively search for adversarial examples.

[4] github.com/peikexin9/deepxplore
[5] ericpony.github.io/z3py-tutorial/guide-examples.htm

Methodology

Overview

At a high level, we will be given a pretrained model and a labelled input image. The objective is to find some mask which when applied to the input, causes the model to predict the class. So the program will encode the input and the feed-forward of a neural network into a single equation and passed into a SAT solver. Additionally we add placeholder variables for the mask which will be the only unknown values for the solver to find. From there, the solver will search for any mask that produces a misclassification of the input (Figure 4).

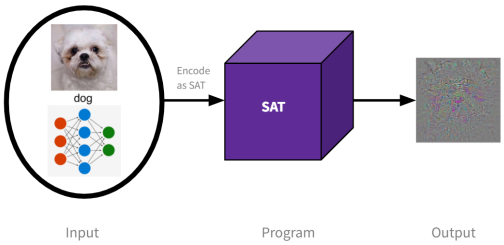

Figure 4

To formalize our approach: given a model $M: \mathbb{R}^n \rightarrow \mathbb{R}^m$, an image $x \in \mathbb{R}^n$, and label $\hat{y} \in \mathbb{R}^m$, such that $M(x) = \hat{y}$. We are searching for a mask $t \in \mathbb{R}^n$ such that,

$$M(x + t) \neq \hat{y}, \qquad \sum_{i}^{n} t_i \leq \epsilon$$

where $\epsilon \in \mathbb{R}$, is a bound for how drastic or "heavy" our transformation can be. The larger the value of ϵ, the more impact our mask has on x.

Implementation

The solution was implemented in Python 3 using Microsoft's Z3 SMT Solver[5] as the backend for our search. There are some pros and cons to using Z3, which will be discussed in later sections.

Evaluation

Evaluating the performance of the program is broken into two demonstrations. The first acts as a proof of concept where we generate a toy dataset from scratch. The second demo is applying the program to the infamous MNIST dataset.

Two Feature Binary Classifier

Consider a set of two featured inputs $X = \{x \in \mathbb{R}^2\}$ and a known decision boundary[6] f. Since every element of X has only two features, we can easily visualize our data in the Cartesian plane. The label y associated with $(x_1, x_2) \in X$ is whether the point sits above (class 1) or below (class 0) the decision boundary.

$$y = \begin{cases} 0 & \text{if } x_2 < f(x_1) \\ 1 & \text{otherwise} \end{cases}$$

We uniformly sample 4000 points to make up our trainset (Figure 5) and another 1000 points for testing.

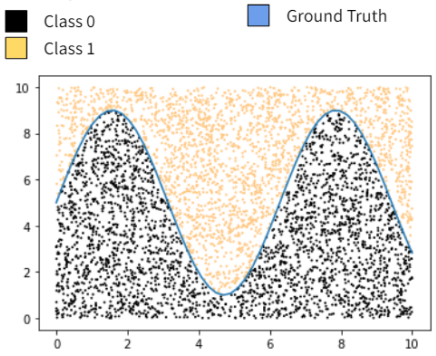

Figure 5

We then use the train-test sets to train and evaluate (Figure 6) the performance of a 4 layer neural network with 24 total neurons (input=2, hidden_1=5, hidden_2=10, hidden_3=5, output=2).

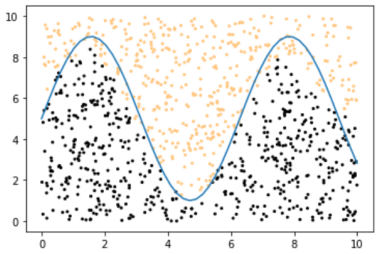

Figure 6: Near the decision boundary we see some mispredictions. This is expected as the model is attempting to learn the decision boundary.

[6] In this toy example we use $f(x) = 4\sin(x) + 5$

Now all thats left is to encode the model into a single equation and let SAT solve for the mask. We pick a point and an $\epsilon \in \mathbb{R}^2$ such that applying the mask does not jump the decision boundary (because then a misclassification would be expected). Below (Figure 7) is a visualization of the program in action. After encoding the model we provide a point $(8, 7)$ and our program is able to confidently find a adversarial example.

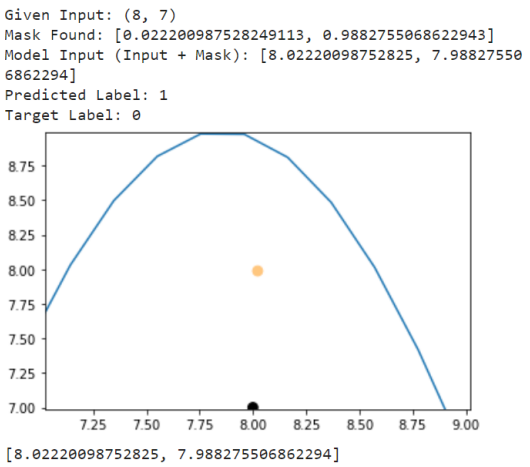

Figure 7

In fact, it is able to do so quicker than expected, the runtime of this program was around 1-3 seconds. Which isn't groundbreaking, but surprising for how simple the setup was.

MNIST

MNIST is a large handwritten digit dataset contain 28x28 pixel images with class labels from 0 to 9. The dataset includes roughly 60,000 training samples and 10,000 testing samples. Below (Figure 8) is a peak at 15 example images from the dataset.

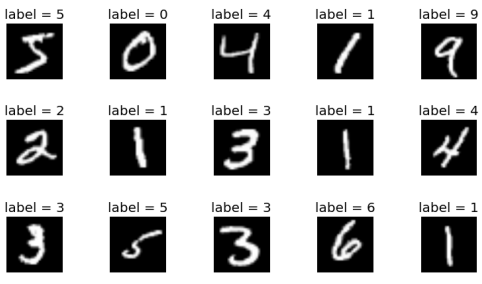

Figure 8

The jump from two features in the previous example, to now 784 features is quite drastic. To combat this increase in scale, we shrink the input images to 8x8 (64 total features). We then train a significantly larger 4 layer model with 174 total neurons (input=64, hidden_1=50, hidden_2=30, hidden_3=20, output=10). Similarly to before, we encode the model and input into our solver and search for a desired mask. After about 30 seconds it is able to find the following result (Figure 9).

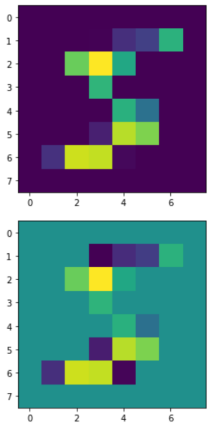

Figure 9: The original image (top) is labelled as a 5 but after applying a mask (bottom), the model predicts a 3.

Discussion

You may notice that with this approach, we do not require access to the gradient or the loss function. Modern white-box methods (like DeepXplore) for generating adversarial examples, rely on this knowledge to perform gradient ascent in search of masks which increase the loss to a particular threshold.

This interestingly can be argued as an advantage of using a SAT solver to search for adversarial examples. The process of attaching a model to this program is relatively simple, all that is needed is just the pretrained model, and an input-label pair. Once you have encoded the forward pass of the network, the rest is automated. This makes the above approach relatively accessible and simple to get started with your own models.

A major issue that was addressed in the proposal was the scalability of this (and DeepXplore's) approach. This is a common flaw in white-box testing. If our SAT solver needs a variable for every weight and bias, we could be keeping track of millions of values as we increase the the size of layers or add new ones. An attractive alternative is to take on a black-box approach, where we simply rely on the model outputs and forget about its internal structure[2]. The obvious trade-off is the time and space complexity. White-box testing will be able to generate results faster but struggles for large networks, whereas black-box testing will generate results slower but is able to scale for larger networks.

There is also a debate of whether or not to purely encode the network in SAT, as mentioned in the Implementation subsection, we leverage Microsoft's Z3 SMT Solver to translate the model purely into a SAT problem. Perhaps exclusively using a tool like PySAT[7] could offer considerable performance gains.

Future Work

The current iteration of this program encodes models with only linear layers. Thus, additions can be made to incorporate convolutional, pooling layers, and much more.

There is an interesting direction to explore quantized neural network encoded in SAT. Quantized neural networks are simply neural networks but with lower precision weights and biases. An extreme case of a quantized neural network is a binary neural network, where each weight and bias in a network can be held in a single bit. The major advantage of using a quantized neural network is the memory reduction. Interestingly, despite the lower precision, quantized neural networks can achieve similar performance to full precision networks[3]. Working with these quantized neural networks would improve the scalability of the program for larger networks. This is because the SAT Solver would have to work with significantly less variables for storing each weight and bias.

[7] pysathq.github.io/

References

[1] Pei, K., Cao, Y., & Jana, S. (2017, September 24). DeepXplore: Automated Whitebox Testing of Deep Learning Systems. arXiv.org. Retrieved October 29, 2021, from https://arxiv.org/pdf/1705.06640.pdf.

[2] Lin, J., Xu, L., Liu, Y., & Zhang, X. (2020, July 30). Black-box adversarial sample generation based on differential evolution. arXiv.org. Retrieved December 1, 2021, from https://arxiv.org/abs/2007.15310.

[3] Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (2016, August 2). XNOR-net: ImageNet classification using binary convolutional Neural Networks. arXiv.org. Retrieved October 28, 2021, from https://arxiv.org/abs/1603.05279.