TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing

Augustus Odena, Ian Goodfellow
Arxiv 2018

Table of Contents

Table of Contents

Introduction

In this work, they introduce automated software testing techniques for neural networks that are well-suited to discovering errors which occur only for rare inputs.

They develop **coverage-guided fuzzing (CGF)** methods for neural networks.

In CGF, random mutations of inputs to a neural network are guided by a coverage metric toward the goal of satisfying user-specified constraints.

Introduction

Machine learning models are notoriously difficult to debug or interpret for a variety of reasons.

Neural networks can be particularly difficult to debug, for example, ReluPlex can formally verify some properties of neural networks but is too computationally expensive to scale to model sizes used in practice.

This is not to criticize ReluPlex but to illustrate the need for additional testing methodologies that interact directly with software as it actually exists in order to correctly test even software that deviates from theoretical models.

Introduction

This work makes following contributions:

- They introduce the notion of CGF for neural networks and describe how fast approximate nearest neighbors algorithms can be used to check for coverage in a general way.
- They open source a software library for CGF called TensorFuzz.
- They use TensorFuzz to find numerical issues in trained neural networks, disagreements between neural networks and their quantized versions, and undesirable behaviors in character level language models.

Table of Contents

Background

Coverage-guided fuzzing

In coverage-guided fuzzing, a fuzzing process maintains an input corpus containing inputs to the program under consideration.

Random changes are made to those inputs according to some mutation procedure, and mutated inputs are kept in the corpus when they exercise new "coverage".

CGF has been highly successful at identifying defects in traditional software, so it is natural to ask whether it could be applied to neural networks.

Background

Coverage-guided fuzzing

In their most basic forms, neural networks are implemented as a sequence of matrix multiplications followed by elementwise operations.

The underlying software implementation of these operations may contain many branching statements but many of these are based on the size of the matrix and thus the architecture of the neural network, *so the branching behavior is mostly independent of specific values of the neural network's input*.

A neural network run on several different inputs will thus often execute the same lines of code and take the same branches, yet produce interesting variations in behavior due to changes in input and output values.

Background

Coverage-guided fuzzing

In this work, they elect to use **fast approximate nearest neighbor algorithms** to determine if two sets of neural network 'activations' are meaningfully different from each other.

This provides a coverage metric producing useful results for neural networks, even when the underlying software implementation of the neural network does not make use of many data-dependent branches.

Background

Testing of Neural Networks

Methods for testing and computing test coverage of traditional computer programs cannot be straightforwardly applied to neural networks.

Thus, we have to think about how to write down useful coverage metrics for neural networks.

A variety of proposals have been made for ways to test neural networks and to measure their test coverage.

For example, neuron coverage (DeepXplore), neuron boundary coverage, MC/DC coverage, concolic testing of DNN, DeepTest and feature guided black-box safety testing.

Background

Opportunities for improvement

It is heartening that so much progress has been made recently on the problem of testing neural networks.

They would implement CGF for neural networks using the coverage metrics from above. However, all of these metrics, though perhaps appropriate in the context originally proposed, lack certain desirable qualities.

Neuron coverage: Sun et al. claim that the metric is too easy to satisfy. This metric is also specialized to work on rectified linear units (ReLUs) which limits its generality.

Neuron boundary coverage: nice in that it doesn't rely on using ReLUs, but it is still easy to exercise all of the coverage with few examples.

Background

Opportunities for improvement

MC/DC coverage: They still treat ReLUs as a special case, they require special modification to work with convolutional neural networks, and they do not offer an obvious generalization that supports *attention* or *residual networks*. They also rely on neural networks being arranged in hierarchical layers, which is often not true for modern deep learning architectures.

What they would really like is a coverage metric that is simple, cheap to compute, and is easily applied to all sorts of neural network architectures.

Thus, they propose storing the activations (or some subset of them) associated with each input, and checking whether coverage has increased on a given input by using an approximate nearest neighbors algorithm to see whether there are any other sets of activations within a pre-specified distance.

Table of Contents

TensorFuzz

Drawing inspiration from the fuzzers like AFL etc, they have implemented a tool that we call TensorFuzz.

Instead of an arbitrary computer program written in C or C++, it feeds inputs to an arbitrary TensorFlow graph.

Instead of measuring coverage by looking at basic blocks or changes in control flow, it measures coverage by looking at the "activations" of the computation graph.

TensorFuzz

The basic fuzzing procecure

The fuzzer starts with a seed corpus containing at least one set of inputs for the computation graph.

Given this seed corpus, fuzzing proceeds as follows: Until instructed to stop, the fuzzer chooses elements from the input corpus according to some component we will call the *Input Chooser*.

Given an input, the *Mutator* component will perform some sort of modification to that input.

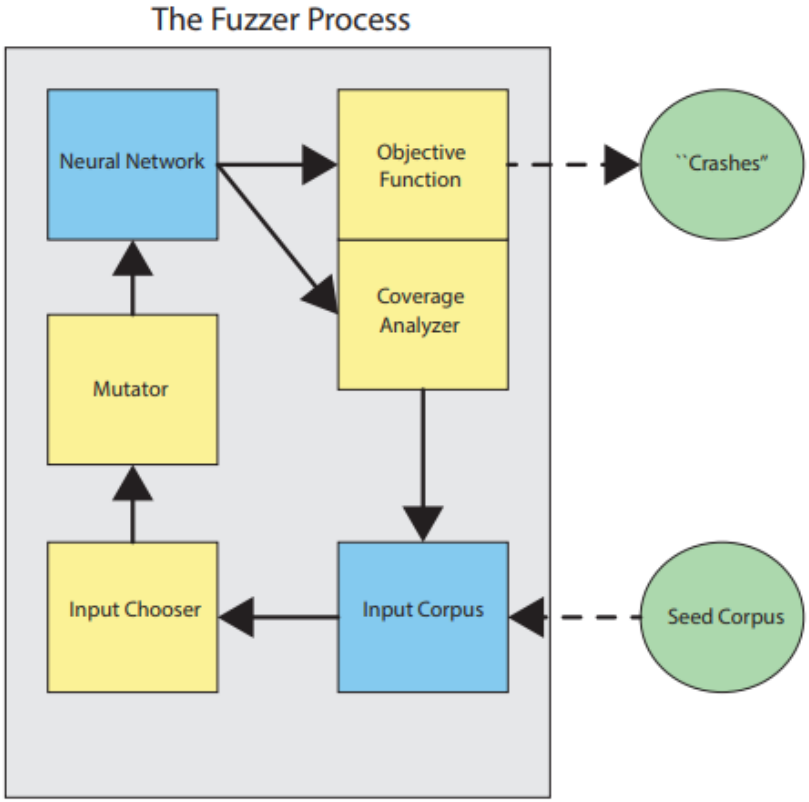
Finally, the mutated inputs can be fed to the neural network. In TensorFuzz, two things are extracted from the neural network:

- a set of coverage arrays, from which the actual coverage will be computed,
- a set of metadata arrays, from which the result of the objective function will be computed.

Once the coverage is computed, the mutated input will be added to the corpus if it exercises new coverage, and it will be added to the list of test cases if it causes the objective function the be satisfied.

TensorFuzz

The basic fuzzing procecure



Data: A seed-corpus of inputs to the computation graph
Result: Test cases satisfying the objective
while *number of iterations* $< N$ **do**
  $parent \leftarrow$ `SampleFromCorpus`;
  $data \leftarrow$ `Mutate`$(parent)$;
  $cov, meta \leftarrow$ `Fetch`$(data)$;
  if `IsNewCoverage`(cov) **then**
    add data to corpus;
  end
  if `Objective`$(meta)$ **then**
    add data to list of test cases;
  end
end

Figure: Coarse descriptions of the main fuzzing loop. Left: A diagram of the fuzzing procedure, indicating the flow of data. Right: A description of the main loop of the fuzzing procedure in algorithmic form.

Input Chooser: For the applications we tested, making a uniform random selection worked acceptably well, but we ultimately settled on the following heuristic, which we found to be faster:

$$p(c_k, t) = \frac{e^{t_k - t}}{\sum e^{t_k - t}}$$

where $p(c_k, t)$ gives the probability of choosing corpus element c_k at time t and t_k is the time when element c_k was added to the corpus.

The intuition behind this is that recently sampled inputs are more likely to yield useful new coverage when mutated, but that this advantage decays as time progresses.

Mutator: For image inputs, they implemented two different types of mutation:

- 1. Just add white noise of a user-configurable variance to the input.
- 2. Add white noise, but to constrain the difference between the mutated element and the original element from which it is descenended to have a user-configurable L_{inf} norm.

For text inputs, they uniformly at random perform one of these operations: delete a character at a random location, add a random character at a random location, or substitute a random character at a random location.

Objective Function: Generally we will be running the fuzzer to make neural network to reach some particular state - maybe a state that we regard as erroneous.

When the mutated inputs are fed into the computation graph, both coverage arrays and metadata arrays are returned as output. The objective function is applied to the metadata arrays, and flags inputs that caused the objective to be satisfied.

Coverage Analyzer: The characteristics of a desirable coverage checker are:

- They want it to check if the neural network is in a *state* that it hasn't been in before, so that we can find misbehaviors that might not be caught by the test set.
- They want this check to be fast, so that find misbehaviors quickly.
- They want it to work for many different types of computation graph without special engineering.
- They want exercising all of the coverage to be hard, otherwise they won't actually cover very much of the possible behaviors.
- They want getting new coverage to help them make incremental progress, so that continued fuzzing yields continued gains.

TensorFuzz

Details of the fuzzing procedure

None the coverage metrics discussed before quite meet all these desiderata. They define another coverage criteria.

A naive, brute force solution is to read out the whole activation vector and treat new activation vectors as new coverage.

However, such a coverage metric would not provide useful guidance, because most inputs would trivially yield new coverage.

It is better to detect whether an activation vector is close to one that was observed previously (approximate nearest neighbors).

When we get a new activation vector, we can look up its nearest neighbor, then check how far away the nearest neighbor is in Euclidean distance and add the input to the corpus if the distance is greater than some amount L.

Table of Contents

Experimental Results

CGF can efficiently find numerical errors

Since neural networks use floating point math, they are susceptible to numerical issues.

These issues are hard to debug, partly because they may only be triggered by rarely encountered inputs.

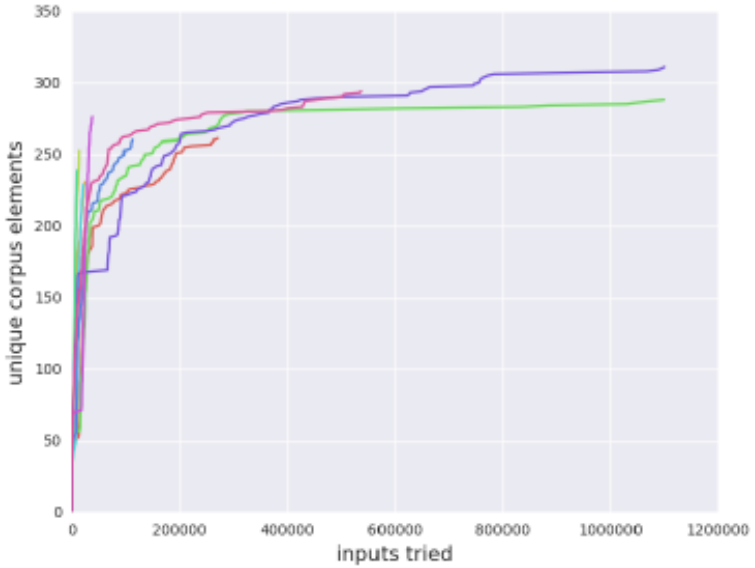
They focus on finding inputs that result in not-a-number (NaN) values.

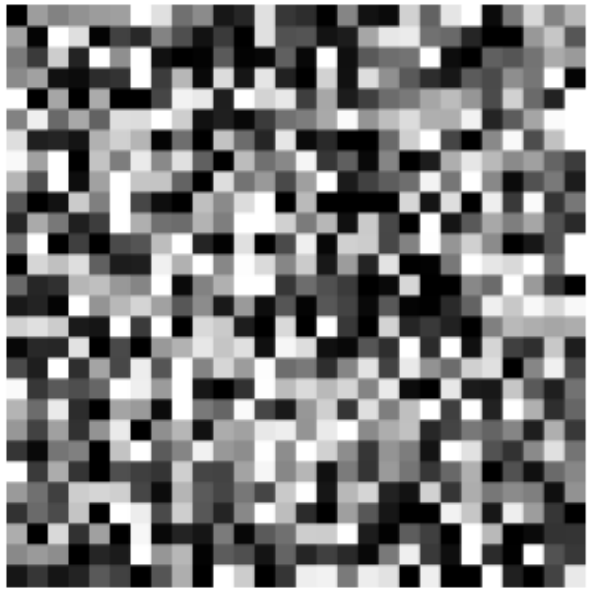
CGF can be used to identify a large number of errors before deployment, and reduce the risk of errors occurring in a harmful setting.

Experimental Results

CGF can efficiently find numerical errors

They trained a fully connected neural network (MNIST). They used a poorly implemented cross entropy loss on purpose so that there would be some chance of numerical errors. The trained model had a validation accuracy of 98%. They then checked that there were no elements in the MNIST dataset that cause a numerical error. Nevertheless, TensorFuzz found NaNs quickly across multiple random initializations.





Experimental Results

CGF surfaces disagreements between models and their quantized versions

Quantization is a process by which neural network weights are stored and neural network computations performed using numerical representations that consist of fewer bits of computer memory.

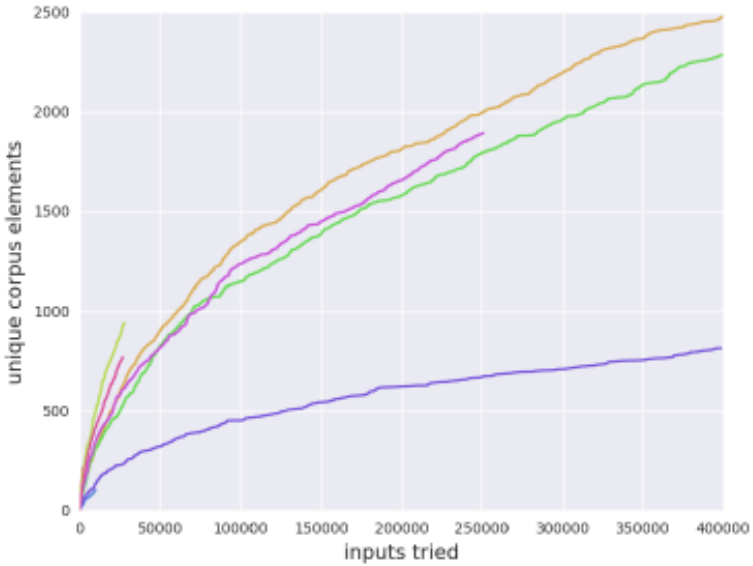
Quantization is a popular approach to reducing the computational cost or size of neural networks, and is widely used for e.g. running inference on cell phones (as in *Android Neural Networks API*).

As a baseline experiment, they trained an MNIST classifier using 32-bit floating point numbers. They then truncated all weights and activations to 16-bits, then compared the predictions of the 32-bit and the 16-bit model on the MNIST test set and found 0 disagreements.

Experimental Results

CGF surfaces disagreements between models and their quantized versions

They then ran the fuzzer with mutations restricted to lie in a radius 0.4 infinity norm ball surrounding the seed images, using the activations of only the 32-bit model as coverage.
They restrict to inputs near seed images because it is less interesting if two versions of the model disagree on out-of-domain garbage data with no true class. With these settings, the fuzzer was able to generate disagreements for 70% of the examples they tried.



Experimental Results

CGF surfaces undesirable behavior in character level language models

They train a character level language model on the Tiny Shakespeare dataset.

They consider the application of sampling from this trained language model given a priming string. One can imagine such a thing being done in an auto-complete application, for example. For illustrative purposes, we identify two desiderata that we can approximately enforce via the fuzzer: First, the model should not repeat the same word too many times in a row. Second, the model should not output words from the blacklist.

We ran an instance of TensorFuzz and an instance of random search for 24 hours each. TensorFuzz was able to generate repeat words, but so was random search. However, TensorFuzz was able to generate six out of ten words from our blacklist while random search only generated one.