TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing

Augustus Odena [1] **Catherine Olsson** [2] **David G. Andersen** [1] **Ian Goodfellow** [3]

Abstract

Neural networks are difficult to interpret and debug. We introduce testing techniques for neural networks that can discover errors occurring only for rare inputs. Specifically, we develop **coverage-guided fuzzing** (CGF) methods for neural networks. In CGF, random mutations of inputs are guided by a coverage metric toward the goal of satisfying user-specified constraints. We describe how **approximate nearest neighbor** (ANN) algorithms can provide this coverage metric for neural networks. We then combine these methods with techniques for **property-based testing** (PBT). In PBT, one asserts properties that a function should satisfy and the system automatically generates tests exercising those properties. We then apply this system to practical goals including (but not limited to) surfacing broken loss functions in popular GitHub repositories and making performance improvements to TensorFlow. Finally, we release an open source library called TensorFuzz that implements the described techniques.

1. Introduction

Machine learning is gradually becoming used in more contexts that affect human lives, including for medical diagnosis (Gulshan et al., 2016), in autonomous vehicles (Huval et al., 2015; Angelova et al.; Bojarski et al., 2016), as input into corporate and judicial decision making processes (Scarborough & Somers, 2006; Berk et al., 2017), in air traffic control (Katz et al., 2017), and in power grid control (Siano et al., 2012).

Machine learning models are notoriously difficult to debug or interpret (Lipton, 2016) for a variety of reasons, ranging from the conceptual difficulty of specifying what the user wishes to know about the model in formal terms to statistical and computational difficulties in obtaining answers to formally specified questions. This property has arguably contributed to the recent "reproducibility crisis" in machine learning (Ke et al., 2017; Henderson et al., 2018; Fedus et al., 2018; Lucic et al., 2017; Melis et al., 2018; Oliver et al., 2018) – it's tricky to make robust experimental conclusions about techniques that are hard to debug.

Neural networks can be particularly difficult to debug because even relatively straightforward formal questions about them can be computationally expensive to answer and because software implementations of neural networks can deviate significantly from theoretical models. For example, Reluplex (Katz et al., 2017) can formally verify some properties of neural networks but is too computationally expensive to scale to model sizes used in practice.

Moreover, Reluplex works by analyzing the description of a ReLU network as a piecewise linear function, using a theoretical model in which all of the matrix multiplication operations are truly linear. In practice, matrix multiplication on a digital computer is not linear due to floating point arithmetic, and machine learning algorithms can learn to exploit this property to perform significantly nonlinear computation (Foerster, 2017). This is not to criticize Reluplex, but to illustrate the need for additional testing methods that interact directly with software as it actually exists in order to correctly test even software that deviates from theoretical models.

In this work, we adapt an existing technique from traditional software testing —**coverage-guided fuzzing** (CGF) (Zalewski, 2007; Serebryany, 2016) — to be applicable to testing neural networks. We then combine this technique with **property-based testing** (PBT) (Claessen & Hughes, 2011) and use the resulting system on a wide variety of neural network testing problems. In particular, this work makes the following contributions:

- First, we introduce the notion of CGF for neural networks and describe how **approximate nearest neighbor** (ANN) algorithms can be used to check for coverage in a general way.
- Second, we describe how PBT techniques can be combined with CGF techniques into a general system that

[1]Google Brain [2]Open Philanthropy Project (work done while at Google Brain) [3]Work done while at Google Brain. Correspondence to: Augustus Odena <augustusodena@google.com>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

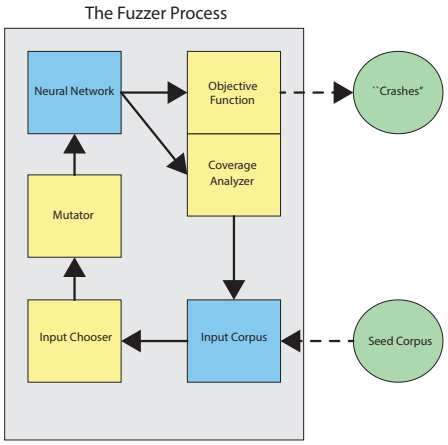

Figure 1. A diagram of the fuzzing procedure, indicating the flow of data.

Algorithm 1 Fuzzer Main Loop

```
Input: seed corpus of inputs to graph
Output: test cases satisfying the objective
for i = 1 to N do
    parent = SampleFromCorpus()
    data = Mutate(parent)
    cov, meta = Fetch(parent)
    if IsNewCoverage(cov) then
        Add element to corpus
    end if
    if ObjectiveFunction(meta) then
        Add element to list of test cases
    end if
end for
```

performs reasonably well out-of-the-box for a variety of applications where we would like to test neural networks.

- Third, we open source a software library called TensorFuzz that implements these ideas.

- Finally, we demonstrate the use of TensorFuzz for finding numerical errors in trained neural networks, exposing disagreements between neural networks and their quantized versions, surfacing broken loss functions in popular GitHub repositories, and making performance improvements to TensorFlow.

2. Background

This section covers necessary background. It may help to keep in mind the following motivating example while reading: imagine that we have a trained neural network and we suspect that some inputs may cause it to yield numerical errors. If those inputs are hard to find, what sort of testing can we perform?

Coverage-guided fuzzing: Coverage-guided fuzzing is used to find many serious bugs in real software (Aizatsky et al., 2016). Two of the most popular coverage-guided fuzzers for normal computer programs are AFL (Zalewski, 2007) and libFuzzer (Serebryany, 2016). These have been expanded in various ways in order to make them faster or to increase the extent to which certain parts of the code can be targeted (Böhme et al., 2017b;a).

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". What is coverage? It depends on the type of fuzzer and on the goals at hand. One common measure is the set of parts of the code that have been executed. By this measure, if a new input causes the code to branch a different way at an if-statement than it has previously, coverage has increased.

CGF has been successful at identifying defects in traditional software, so it is natural to ask whether it could be applied to neural networks. Traditional coverage metrics track which lines of code have been executed and which branches have been taken. 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. Executing an existing CGF tool such as AFL therefore may not find interesting behaviors of the neural network. In this work, we elect to use nearest neighbor algorithms to determine if two sets of neural network "activations" are meaningfully different from each other. This provides a coverage metric that yields useful results for neural networks, even when the underlying software implementation of the neural network does not use many data-dependent branches.

Property-based testing: Property-based testing is a testing technique that automatically generates test cases for software functions. The user specifies a property that their function should satisfy, and the PBT system automatically generates test cases that attempt to violate this property. There are a variety of concrete instantiations of this abstract

idea, but the most well-known is QuickCheck (Claessen & Hughes, 2011). QuickCheck is a Haskell library in which a user specifies that inputs be of a specific type and the library generates random examples satisfying that type constraint. See Listing 1 for an illustrative example.

```
1 prop_a xs = reverse (reverse xs) == xs
2   where types = xs::[Int]
```

Listing 1. An example definition of a QuickCheck property. Line 1 asserts that calling reverse twice on a list is the same as the identity. Line 2 explains to QuickCheck that it should generate random test cases that are lists of type Int.

The concept has proved sufficiently useful that there are now many other PBT libraries servicing languages such as Rust, Java, and Python (Gallant, 2018; Holser, 2018; Hypothesis, 2018).

As with CGF, it is natural to ask whether PBT can be applied fruitfully to neural networks. It is also worth asking whether CGF and PBT can be combined to make a useful general-purpose testing tool. We claim that the answer is yes in both cases. Researchers and practitioners often write functions in TensorFlow (or other libraries) about which they would like to assert invariants. Moreover, if CGF can help us search over the space of possible inputs more efficiently, then it makes sense to combine CGF with PBT.

We are not the first to discuss the relationship between CGF and PBT — though this work is arguably the first to combine them this way. Both Elhage (2017) and MacIver (2017) argue that PBT and fuzzing can be seen as similar in nature, where the property one is checking with a fuzzer is "this program doesn't crash", and the AFL documentation (Zalewski, 2007) discusses the possibility of adding assertions to code so that AFL will find the assertions. In Luu (2015), the author speculates about a combination of CGF and PBT wherein the coverage guidance is meant to improve the efficiency of the search for property-violating counter-examples.

Approximate Nearest Neighbor algorithms: TensorFuzz uses ANN algorithms to decide if a new input should be added to the corpus. The approximate nearest neighbor problem is formulated as follows: Consider a set $P = \{p_1, \cdots, p_n\}$ of points in a metric space (X, D). The problem is to build a data structure taking a query point $q \in (X, D)$ and returning any point p sufficiently close to q.

There are many different ANN techniques, including Locality Sensitive Hashing (Andoni et al., 2015), tree-based methods (Bernhardsson, 2012), and Proximity Graph methods (Malkov et al., 2014; aalgo, 2012). For a benchmark comparing these algorithms, see Bernhardsson (2017). With n points lying in R^d with the Euclidean Norm, modern techniques for ANN (Indyk & Motwani, 1998) allow for preprocessing costs polynomial in n and d and query costs polynomial in d and $\log n$. Empirical performance for these algorithms is in many cases dramatically better than the theoretical worst case performance guarantees (Indyk & Motwani, 1998; Bernhardsson, 2017) and can be made even faster if one uses properties of the specific dataset (Andoni & Razenshteyn, 2015). These techniques have been shown to be reasonably performant on datasets with millions of elements and thousands of dimensions (Bernhardsson, 2017). For more thorough reviews, see Andoni et al. (2018); Shakhnarovich et al. (2006); Andoni & Indyk (2006).

3. Related Work

Testing of neural networks: Methods for testing and computing test coverage of traditional computer programs cannot be straightforwardly applied to neural networks. We can't just naively compute branch coverage, for example, for the reasons discussed above. Thus, we must think about how to write down useful coverage metrics for neural networks. Though this work is the first (as far as we know) to explore the idea of CGF for neural networks, it's not the first to address the issues of testing and test coverage for neural networks. A variety of proposals (many of which focus on adversarial examples (Szegedy et al., 2013)) have been made for ways to test neural networks and to measure their test coverage. We survey these proposals here:

Pei et al. (2017) introduce the metric of neuron coverage for a neural network with rectified linear units (ReLUs) as the activation functions. A test suite is said to achieve full coverage under this metric if for every hidden unit in the neural network, there is some input for which that hidden unit has positive value. They then cross reference multiple neural networks using gradient based optimization to find misbehavior.

Ma et al. (2018) generalize neuron coverage in two ways. In k-multisection coverage, they take – for each neuron – the range of values seen during training, divide it into k chunks, and measure whether each of the k chunks has been "touched". In neuron boundary coverage, they measure whether each activation has been made to go above and below a certain bound. They then evaluate how well these metrics are satisfied by the test set.

Sun et al. (2018a) introduce a family of metrics inspired by Modified Condition / Decision Coverage (Hayhurst et al., 2001). We describe their ss-coverage proposal by example: Given a neural network arranged into layers, a pair of neurons (n_1, n_2) in adjacent layers is said to be ss-covered by a pair (x, y) of inputs if the following 3 things are true: n_1 has a different sign for each of x, y; n_2 has a different sign

for each of x, y; all other elements of the layer containing n_1 have the same sign for x, y.

Tian et al. (2017) applies the neuron coverage metric to deep neural networks that are part of self-driving car software. They perform natural image transformations such as blurring and shearing and use the idea of metamorphic testing (Chen & Yiu) to find errors.

Wicker et al. (2017) perform black box testing of image classifiers using image-specific operations. Concurrent with our work, Sun et al. (2018b) leverage a complementary approach called concolic execution. Whereas our approach is analogous to AFL or libFuzzer, their approach is analogous to CUTE (Sen et al., 2005).

Opportunities for improvement: It is heartening that so much progress has been made recently on the problem of testing neural networks. However, the success of AFL and libFuzzer in spite of the existence of more sophisticated techniques suggests that there is a role for an analogous tool that works on neural networks. Ideally we would implement CGF for neural networks using the coverage metrics above. However, all of these metrics, though perhaps appropriate in the context originally proposed, lack certain desirable qualities. We describe below why this is true for the most relevant metrics.

Sun et al. (2018a) claim that the neuron coverage metric is too easy to satisfy. In particular, they show that 25 randomly selected images from the MNIST test set yield close to 100% neuron coverage for an MNIST classifier. This metric is also specialized to work on rectified linear units (ReLUs) which limits its generality.

Neuron boundary coverage (Ma et al., 2018) is nice in that it doesn't rely on using ReLUs, but it also still treats neurons independently. This causes it to suffer from the same problem as neuron coverage: it's easy to exercise all of the coverage with few examples.

The metrics from Sun et al. (2018a) improve upon neuron coverage and may be useful in the context of more formal methods, but for our desired application, they have several disadvantages. 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 (Bahdanau et al., 2014) or residual networks (He et al., 2016). They also rely on neural networks being arranged in hierarchical layers, which is often not true for modern deep learning architectures.

What we 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, we 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 ANN algorithm to see whether there are any other sets of activations within a pre-specified distance. We discuss this idea in more detail in Section 5.

4. An Overview of the TensorFuzz Library

Drawing inspiration from the fuzzers described in the previous section and from the notion of property-based testing, we have implemented a tool that we call TensorFuzz. It works in a way that is analogous to those tools, but that differs in ways that make it more suitable for neural network testing. 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 (roughly speaking) looking at the "activations" of the computation graph.

Overview of the fuzzing procecure: The overall structure of the fuzzing procedure is very similar to the structure of coverage-guided fuzzers for normal computer programs. The main difference is that instead of interacting with an arbitary computer program that we have instrumented, TensorFuzz interacts with a TensorFlow graph that it can feed inputs to and get outputs from.

The fuzzer starts with a seed corpus containing at least one set of inputs for the computation graph. In traditional CGF, inputs don't generally have to be perfectly valid — they may just be random bytes. In TensorFuzz, we restrict the inputs to those that are valid neural network inputs. If the inputs are images, we restrict our inputs to have the correct size and shape, and to lie in the same interval as the input pixels of the dataset under consideration. If the inputs are sequences of characters, we only allow characters that are in the vocabulary extracted from the training set.

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. For the purpose of this section, it's ok to imagine the Input Chooser as choosing uniformly at random, though we describe more complicated strategies in 5.

Given an input, the Mutator component will perform some sort of modification to that input. The modification can be as simple as just flipping the sign of an input pixel in an image, and it can also be restricted to obey some kind of constraint on the total modification made to a corpus element over time — see Section 5 for more on this.

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, and 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. The mutated input will be added to the list of test cases if it causes the objective function to be satisfied. The objective function is a user defined function that asserts a property that the graph being fuzzed should obey, as in (Claessen & Hughes, 2011). This property can be something like "this graph should not report zero loss when the gradient is very large" or "this graph should not contain non-finite elements".

See Figure 1 and Algorithm 1 for complementary depictions of this procedure.

5. Details of the TensorFuzz Library

Input Chooser: At any given time, the fuzzer must choose which inputs from the existing corpus to mutate. The optimal choice will of course be problem dependent, and traditional CGFs rely on a variety of heuristics to make this determination. For the applications we tested, making a uniform random selection worked acceptably well, but was sometimes slow.

There were some contexts in which we attained a considerable speed-up by changing the Input Chooser to prioritize certain inputs over others. For instance, when conducting the experiment in Section 6, we found that fuzzing was sped up considerably by preferentially sampling more recently discovered corpus elements. This is somewhat analogous to doing depth first search in the input-space, since it makes us quickly explore some points far from the seed.

Mutator: Once the Input Chooser has chosen an element of the corpus to mutate, the mutations must be applied. In TensorFuzz, we have implemented mutation functions that act on arbitary dense vectors of floats and mutations that are specialized to image inputs, since a common use-case for deep neural networks is in image classification.

For image inputs, we implemented two different types of mutation. The first is to just add white noise of a user-configurable variance to the input. The second is to add white noise, but to constrain the difference between the mutated element and the original element from which it is descended to have a user-configurable L_∞ norm. This type of constrained mutation can be useful if we want to find inputs that satisfy some objective function, but are still plausibly of the same "class" as the original input that was used as a seed. In both types of image mutation we clip the image after mutation so that it lies in the same range as the inputs used to train the neural network being fuzzed.

Objective Function: Generally we will be running the fuzzer with some goal in mind. That is, we will want the neural network to reach some particular state — maybe a state that we regard as erroneous. The objective function is used to assess whether that state has been reached. In CGF, the goal is to find an input causing the program to crash. Since we are interested in doing PBT, we allow for arbitrary user-defined goals. 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. In this way, the objective function asserts the property to be tested and the fuzzer attempts to find a violation of this property.

Coverage Analyzer: The coverage analyzer is in charge of reading arrays from the TensorFlow runtime, turning them into Python objects representing coverage, and checking whether that coverage is new. The algorithm by which new coverage is checked is central to the proper functioning of the fuzzer.

The characteristics of a desirable coverage checker are as follows: First, we 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. We want this check to be fast (so we probably want it to be simple), so that we can find many of those misbehaviors quickly. We also want it to work for many different types of computation graphs without special engineering, so that practitioners can use our tooling without having to make special adaptations. Moreover, we want it to be appropriately difficult to exercise all of the coverage. If every new input generated new coverage, then we would just be doing random search over the whole input space. If inputs never generated any new coverage, then we would just be performing random mutations on the initial corpus elements. Finally, we want getting new coverage to help us make incremental progress, so that continued fuzzing yields continued gains.

As alluded to in Section 3, none the coverage metrics discussed in Section 3 quite meet all these desiderata, but we can design from first principles a coverage metric that comes closer to meeting them.

A brute-force solution to this problem is to read out the whole activation vector (that is, the whole vector of intermediate states from the neural network) 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. One way to achieve this is to use an ANN algorithm, (as used for Neural Turing Machine (Graves et al., 2014) memory by Rae et al. (2016)). 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. This is the approach we have chosen in the TensorFuzz prototype. Currently, we use an open source libarary called FLANN (Muja & Lowe, 2014) to compute the approximate nearest neighbors. The specific search algorithm used is user configurable. This set-up raises a few questions, which we address now:

Which set of activations should you choose to observe? In general, you may not need to see all the activations. There are a few reasons to believe that this is true: First, the TensorFlow graph can be represented as a DAG. Thus, certain subsets of the activations deterministically imply the values for the rest of the activations. Second, it's well known that trained neural networks are heavily compressible (see Cheng et al. (2017) for a survey), so it can in principle suffice to choose only a small subset of the activations. We leave it to future work to determine the optimal way to exploit this compressibility, but one speculative idea we have in this vein is to use something like the procedure in Raghu et al. (2017) to project activations onto their most important directions, and then consider only those directions for coverage.

In practice, the set of activations to use is presently a matter for empirical tuning. When fuzzing whole classifiers, we find it is often possible to obtain good results by tracking only the logits, or the layer before the logits. The compressibility considerations mentioned above may hint at why this works OK: the "intrinsic dimension" of the activations is just not that large. When fuzzing smaller graphs, it may make sense to just use all of the activations.

How do you choose the distance L? If the distance is too small, coverage is too easy to find. If it's too large, coverage is too hard to find. Both of these situations cause issues, as described above. In practice, we do the simplest thing possible: we tune the distance for a given fuzzing problem by tracking whether the number of new coverage elements per mutation is neither too high nor too low. This represents another potential optimization: we could instead automatically modify the distance to maintain a constant rate of new coverage per mutation.

What are the performance characteristics of ANN in this context? First we note that, when doing CGF, the size of the corpus tends to grow quite slowly in comparison with the number of mutations performed, and we should never expect the corpus to grow that large. Thus, we ought to care much more about the cost of checking for new coverage than the memory use of the corpus. That being said, the memory usage of the corpus can be quite small for many choices of the nearest-neighbor algorithm.

Second, our use case seems to require that new elements be added to the corpus as they are produced, but only some ANN algorithms support incremental additions to the index, and incremental additions may require periodic rebalancing. For now, we simply reconstruct the index periodicially while keeping a buffer of elements over which the exact nearest neighbor is computed temporarily. See the open source release for more details on this.

Third, given the cost of moving data back and forth between the CPU and the GPU and the large cost of evaluating many machine learning models, we expect that much of the work involved in the nearest neighbor computation can be moved to a "background thread", so that the GPU is always saturated, as it would be with random search. Anecdotally, this seems to be true for our current experiments, but we have not conducted a thorough study of this point.

Finally, **note that our particular use case does not require that we actually retrieve an approximate nearest neighbor. We only need to know whether such a neighbor exists.** This means that we don't need to store all of the corpus elements. There are moderately-exotic data structures (such as distance-sensitive Bloom filters (Kirsch & Mitzenmacher, 2006)) that support precisely this type of query, but we haven't found a usable open-source implementation of one, so in future work we plan to implement something similar ourselves. We haven't done this yet out of both a general wariness of premature optimization and a feeling that it's possible that storage of the elements won't become a bottleneck for realistic use-cases.

6. Experimental Results

The goal of these results is to establish that TensorFuzz is (or can be) a useful general purpose tool that practitioners could reach for first when trying to perform some kind of testing or debugging task for neural networks. The model we aspire to is AFL, which is easy to use on new code-bases and tasks and performs acceptably well in most of those cases. We don't aspire to build a tool that is the very best at any specific task — this often requires specialization to that task and sacrifices generality or ease of use. Instead, we aspire to build a good general purpose tool for neural network testing and debugging.

This section presents experimental results from four different settings. For some of these results we compare with a random search baseline, and for some we don't compare to any sort of baseline. Why don't we show comparisons to some of the methods from Section 3? First, as discussed in that section, many of those coverage definitions are trivial to completely exercise, and so quickly reduce to random search. For the rest, it's not obvious how to apply them to arbitrary TensorFlow graphs. Moreover, none of those methods seem to support arbitrary objective functions — that is, they don't do generic PBT.

TensorFuzz can efficiently find numerical errors in trained neural networks: Because neural networks use floating point math (Goldberg, 1991), they are susceptible to numerical issues, both during training and at evaluation time. These issues are notoriously hard to debug, partly because they may only be triggered by a small set of rarely encountered inputs. This is one case where CGF can help. We focus on finding inputs that result in not-a-number (NaN) values.

Numerical errors, especially those resulting in NaNs, could cause dangerous behavior of important systems if these errors are first encountered "in the wild". CGF can be used to identify errors before deployment, and reduce the risk of errors occurring in a harmful setting.

With CGF, we should be able to simply add operations to the metadata that check for non-finite elements and then run our fuzzer. To test this hypothesis, we trained a fully connected neural network to classify MNIST (LeCun et al., 1998) digits. We performed fault injection by using a poorly implemented cross entropy loss so that there would be a chance of numerical errors. We trained the model for 35000 steps with a mini-batch size of 100, at which point it had a validation accuracy of 98%. We then checked that there were no elements in the MNIST dataset that caused a numerical error. As shown in Figure 2, TensorFuzz found NaNs quickly across 10 different random initializations.

One potential objection to using CGF techniques is that gradient-based search techniques might be more efficient. However, it is not obvious how to specify this objective for a gradient based search. There is not a straightforward way to measure how similar a real-valued output of the model is to a NaN value[1].

To establish that random search is insufficient and that coverage guidance is necessary for efficiency, we compared to random search. [2] We implemented a baseline random search algorithm and ran it for 10 million mutations with 10 different random initializations. The baseline was not able to find a non-finite element in any of these trials.

TensorFuzz surfaces disagreements between models and their quantized versions: Quantization (Hubara et al., 2016) 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 or TFLite and in the context of custom machine learning hardware – e.g., Google's Tensor Processing Unit (Jouppi et al., 2017) or NVidia's TensorRT. Of course, quantization is not very useful if it reduces the accuracy of the model too dramatically. Given a quantized model, it would thus be nice to check how much quantization reduced the accuracy.

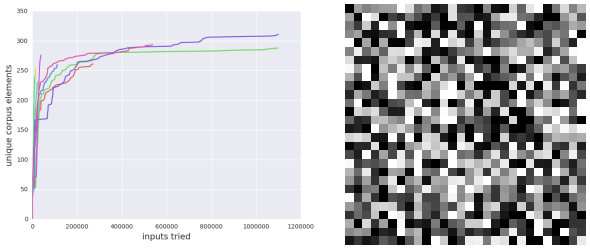

Figure 2. We trained an MNIST classifier with some unsafe numerical operations. Klees et al. (2018) recommend evaluating fuzzers with multiple random restarts, so we ran the fuzzer 10 times on random seeds from the MNIST dataset. The fuzzer found a non-finite element every run. Random search never found a non-finite element. Left: the accumulated corpus size of the fuzzer while running, for 10 runs. Right: an example satisfying image found by the fuzzer. Inspecting the classifier when exposed to the image on the right, we see that the NaN surfaces because there was one logit with a very positive value and one logit with a very negative value, which broke the loss calculation.

As a baseline experiment, we trained an MNIST classifier (this time without intentionally introducing numerical issues) using 32-bit floating point numbers. We then truncated all weights and activations to 16-bits. We then compared the predictions of the 32-bit and the 16-bit model on the MNIST test set and found 0 disagreements.

We 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. We restrict to inputs near seed images because these inputs nearly all have unambiguous class semantics. 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 we tried. Thus, CGF allowed us to find real errors that could have occured at test time. See Figure 3 for more details.

As in Section 6 we tried a baseline random search method in order to demonstrate that the coverage guidance specifically was useful in this context. The random search baseline was not able to find any disagreements when given the same

[1] In principle, if you know exactly where in the neural network you expect to find the NaN, and you know that e.g., it will come from a division by 0, then you could use gradient-based optimization to find an input that triggered it, but this technique won't help you if all you know is that your code has (or may have) a numerical issue somewhere.

[2] By random search we mean a search where we randomly decide whether a given activation qualifies as new coverage, rather than using ANN to decide.

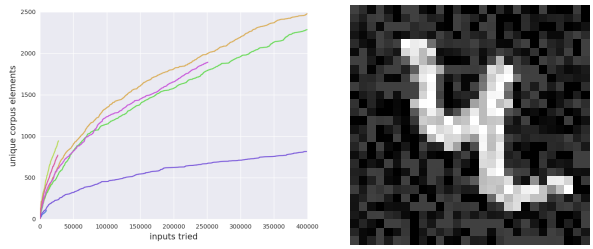

Figure 3. We trained an MNIST classifier with 32-bit floats and then truncated the associated TensorFlow graph to 16-bit floats. Both the original and the truncated graph made the same predictions on all 10000 elements of the MNIST test set, but the fuzzer was able to find disagreements within an infinity-norm ball of radius 0.4 around 70% of the test images that we tried to fuzz. Left: the accumulated corpus size of the fuzzer while running, for 10 runs. Lines that go all the way to the right correspond to failed fuzzing runs. Right: an image found by the fuzzer that is classified differently by the 32-bit and 16-bit neural networks.

number of mutations as the fuzzer.[3]

TensorFuzz can find real bugs in popular model implementations: Another application of TensorFuzz is to help with testing of TensorFlow code. If we have a library of Python functions that instantiate TensorFlow graphs, we can write down properties that we would like each function to satisfy, express the violation of this property as the TensorFuzz objective function, and then run TensorFuzz for as long as we think is appropriate. If TensorFuzz satisfies the objective, we consider the test failed. This application seems promising to us because it allows for fuzzing to be easily integrated with the workflow of developing machine learning models.

To corroborate this intuition, we wrote several tests for a popular TensorFlow implementation of a machine learning model. In particular, we add tests to the code from https://github.com/carpedm20/DCGAN-tensorflow. This is a very popular (it has roughly 4700 GitHub stars) implementation of the DCGAN model (Radford et al., 2015). We already knew that the loss function had a certain issue such that if you made the learning rate too high, the loss would get stuck at a very high value.

Thus, we hooked up TensorFuzz to the sub-graph defined by the loss function and wrote an objective that would be satisfied if the loss was high and the gradient was close to zero.

TensorFuzz was able to quickly find a satsifying input. The input was one in which the output of the discriminator had too high a magnitude, which caused the incorrectly implemented loss function to have saturating gradients. In this way, TensorFuzz was able to reproduce a known issue in a real, heavily used piece of TensorFlow code.

TensorFuzz can help make semantics-preserving code transformations: When implementing a machine learning model, it's common to have the following problem: You would like to make a change to your code that preserves the semantics of the model. However, this can be difficult, because you may not be able to reason formally about how different operations compose, or you might be worried about numerical differences between seemingly equivalent implementations. You might want to make one of these transformations for a variety of reasons. Maybe the change is preparation for a future change that will modify semantics. Maybe the the change is just a refactoring. Maybe you are trying to optimize your code. We will consider an example of this last case. In particular, we optimized TensorFlow's implementation of batch-wise random flipping of images.

First, we "attached" TensorFuzz to two copies of the old implementation and wrote an objective function asserting that both implementations are the same. Having done this, we could then make incremental changes to one of the implementations while asserting that each change did not modify the semantics for a large set of possible inputs. Since the operation involves pseudorandom number generators (PRNGs), this involved carefully setting all of the graph level and operation level seeds, but once this was done it provided a high degree of assurance. Of course, this procedure can yield false negatives, but in the absence of the abililty to do equational reasoning it is a strong tool.

7. Conclusion

This paper has introduced coverage-guided fuzzing for neural networks and described how to build a useful coverage checker in this context. We have demonstrated the practical utility of TensorFuzz by finding numerical errors, exposing disagreements between neural networks and their quantized versions, surfacing broken loss functions in popular repositories, and making performance improvements to TensorFlow. Finally, we have released an open source libary so that other researchers can build on this work and so that practitioners can use it to test their machine learning code.

[3] It might not be obvious why we have chosen to compare based on number of mutations rather than wall-clock time. There are two reasons. First, for the experiments described here, the computation is theoretically bottlenecked on the neural network forward pass, not the ANN lookup time. Second, to keep the comparison straightforward, we implemented random search inside TensorFuzz by just accepting all new inputs into the corpus. The lookup technically still happens, which means that the wall-clock time would not be reduced.

Acknowledgments

We thank Kostya Serebryany for helpful explanations of libFuzzer. We thank Rishabh Singh, Alexey Kurakin, and Martín Abadi for general input.

References

aalgo. Kgraph, 2012. URL https://github.com/aaalgo/kgraph.

Aizatsky, M., Serebryany, K., Chang, O., Arya, A., and Whittaker, M. Announcing oss-fuzz: Continuous fuzzing for open source software. *Google Testing Blog*, 2016.

Andoni, A. and Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pp. 459–468. IEEE, 2006.

Andoni, A. and Razenshteyn, I. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pp. 793–801. ACM, 2015.

Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L. Practical and optimal lsh for angular distance. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 28*, pp. 1225–1233. Curran Associates, Inc., 2015.

Andoni, A., Indyk, P., and Razenshteyn, I. Approximate nearest neighbor search in high dimensions. *arXiv preprint arXiv:1806.09823*, 2018.

Angelova, A., Krizhevsky, A., Vanhoucke, V., Ogale, A. S., and Ferguson, D. Real-time pedestrian detection with deep network cascades.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Berk, R., Heidari, H., Jabbari, S., Kearns, M., and Roth, A. Fairness in criminal justice risk assessments: the state of the art. *arXiv preprint arXiv:1703.09207*, 2017.

Bernhardsson, E. Approximate nearest neighbors oh yeah, 2012. URL https://github.com/spotify/annoy.

Bernhardsson, E. ann-benchmarks, 2017. URL https://github.com/erikbern/ann-benchmarks.

Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344. ACM, 2017a.

Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017b.

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

Chen, T. Y. and Yiu, S. M. Metamorphic testing: a new approach for generating next test cases. Technical report.

Cheng, Y., Wang, D., Zhou, P., and Zhang, T. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017. URL http://arxiv.org/abs/1710.09282.

Claessen, K. and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

Elhage, N. Property-based testing is fuzzing, 2017. URL https://blog.nelhage.com/post/property-testing-is-fuzzing/.

Fedus, W., Rosca, M., Lakshminarayanan, B., Dai, A. M., Mohamed, S., and Goodfellow, I. Many paths to equilibrium: GANs do not need to decrease a divergence at every step. *International Conference on Learning Representations*, 2018.

Foerster, J. Nonlinear computation in deep linear networks, 2017. URL https://blog.openai.com/nonlinear-computation-in-linear-networks/.

Gallant, A. Quickcheck for rust, 2018. URL https://github.com/BurntSushi/quickcheck.

Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Gulshan, V., Peng, L., Coram, M., Stumpe, M. C., Wu, D., Narayanaswamy, A., Venugopalan, S., Widner, K., Madams, T., Cuadros, J., et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *Jama*, 316(22): 2402–2410, 2016.

Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., and Rierson, L. K. A practical tutorial on modified condition/decision coverage. 2001.

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

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Holser, P. junit-quickcheck, 2018. URL https://github.com/pholser/junit-quickcheck/.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

Huval, B., Wang, T., Tandon, S., Kiske, J., Song, W., Pazhayampallil, J., Andriluka, M., Rajpurkar, P., Migimatsu, T., Cheng-Yue, R., et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

Hypothesis. Hypothesis, 2018. URL https://github.com/HypothesisWorks/hypothesis.

Indyk, P. and Motwani, R. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613. ACM, 1998.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12. ACM, 2017.

Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pp. 97–117. Springer, 2017.

Ke, R. N., Goyal, A., Lamb, A., Pineau, J., Bengio, S., and Bengio, Y. (eds.). *Reproducibility in Machine Learning Research*, 2017.

Kirsch, A. and Mitzenmacher, M. Distance-sensitive bloom filters. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 41–50. SIAM, 2006.

Klees, G. T., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

Lipton, Z. C. The mythos of model interpretability. *arXiv preprint arXiv:1606.03490*, 2016.

Lucic, M., Kurach, K., Michalski, M., Gelly, S., and Bousquet, O. Are GANs created equal? A large-scale study. *arXiv preprint arXiv:1711.10337*, 2017.

Luu, D. Afl + quickcheck = ?, 2015. URL https://danluu.com/testing/.

Ma, L., Juefei-Xu, F., Sun, J., Chen, C., Su, T., Zhang, F., Xue, M., Li, B., Li, L., Liu, Y., Zhao, J., and Wang, Y. Deepgauge: Comprehensive and multi-granularity testing criteria for gauging the robustness of deep learning systems. *CoRR*, abs/1803.07519, 2018. URL http://arxiv.org/abs/1803.07519.

MacIver, D. R. What is property based testing, 2017. URL https://hypothesis.works/articles/what-is-property-based-testing/.

Malkov, Y., Ponomarenko, A., Logvinov, A., and Krylov, V. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45: 61–68, 2014.

Melis, G., Dyer, C., and Blunsom, P. On the state of the art of evaluation in neural language models. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

Muja, M. and Lowe, D. G. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence*, 36(11): 2227–2240, 2014.

Oliver, A., Odena, A., Raffel, C., Cubuk, E. D., and Goodfellow, I. J. Realistic evaluation of deep semi-supervised learning algorithms. *CoRR*, abs/1804.09170, 2018. URL http://arxiv.org/abs/1804.09170.

Pei, K., Cao, Y., Yang, J., and Jana, S. Deepxplore: Automated whitebox testing of deep learning systems. *CoRR*, abs/1705.06640, 2017. URL http://arxiv.org/abs/1705.06640.

Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015. URL http://arxiv.org/abs/1511.06434.

Rae, J. W., Hunt, J. J., Harley, T., Danihelka, I., Senior, A., Wayne, G., Graves, A., and Lillicrap, T. P. Scaling Memory-Augmented Neural Networks with Sparse Reads and Writes. *ArXiv e-prints*, October 2016.

Raghu, M., Gilmer, J., Yosinski, J., and Sohl-Dickstein, J. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In *Advances in Neural Information Processing Systems*, pp. 6076–6085, 2017.

Scarborough, D. and Somers, M. J. *Neural networks in organizational research: Applying pattern recognition to the analysis of organizational behavior.* American Psychological Association, 2006.

Sen, K., Marinov, D., and Agha, G. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pp. 263–272. ACM, 2005.

Serebryany, K. Libfuzzer: A library for coverage-guided fuzz testing (within llvm), 2016.

Shakhnarovich, G., Darrell, T., and Indyk, P. *Nearest-neighbor methods in learning and vision: theory and practice (neural information processing)*. 2006.

Siano, P., Cecati, C., Yu, H., and Kolbusz, J. Real time operation of smart grids via FCN networks and optimal power flow. *IEEE Transactions on Industrial Informatics*, 8(4):944–952, 2012.

Sun, Y., Huang, X., and Kroening, D. Testing Deep Neural Networks. *ArXiv e-prints*, March 2018a.

Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., and Kroening, D. Concolic Testing for Deep Neural Networks. *ArXiv e-prints*, April 2018b.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Tian, Y., Pei, K., Jana, S., and Ray, B. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. *CoRR*, abs/1708.08559, 2017. URL http://arxiv.org/abs/1708.08559.

Wicker, M., Huang, X., and Kwiatkowska, M. Feature-guided black-box safety testing of deep neural networks. *CoRR*, abs/1710.07859, 2017. URL http://arxiv.org/abs/1710.07859.

Zalewski, M. American fuzzy lop, 2007.