TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing

A. Odena, I. Goodfellow

Google Brain

arXiv: 1807.10875

Reviewed by : Bill Zhang
University of Virginia
https://qdata.github.io/deep2Read/

Outline

Introduction

Basic Premise and Motivation

- Tricky to make robust conclusions about techniques that are hard to debug; the 'reproducibility crisis' in machine learning
- Even straightforward questions about a network can be computationally expensive to answer; current methods like ReluPlex do not scale well
- Draw from a traditional software engineering technique, Coverage-Guided Fuzzing (CGF), and create similar method for neural networks

Background

Coverage-Guided Fuzzing

- Common technique used to find serious bugs in software; common fuzzers include AFL and libFuzzer
- Fuzzing maintains an input corpus, randomly changes the inputs using some mutation procedure, and keeps the changed input if it adds additional coverage
- "Coverage" is usually set to be the lines of code which have been executed given an input
- This would not really work for neural networks because even different inputs usually take the same branches when being executed; branches are independent of the specific values of the network's input
- Instead, let coverage be determined by the network's activations

Background

Testing of Neural Networks

- Pei et al.: Introduced metric of neuron coverage for network with ReLU units
- Ma et al.: Took, for each neuron, the range of values seen during training, divided it into k chunks, and measured whether each of the chunks had been "touched"; also, measured whether activation was ever above or below some given bounds
- Sun et al.: Introduced metric inspired by Modified Condition/Decision Coverage
- Tian et al.: Applies neuron coverage metric to DNNs in self-driving car software; performed natural image transformations and used principle of metamorphic testing
- Wicker et al.: Performed blackbox testing of image classifiers using image-specific operations

Background

Opportunities for Improvement

- Success of CGF methods suggests that there should be an analogous method for neural networks
- Other metrics like neuron coverage was shown to be too easy to satisfy (25 randomly selected images from MNIST was enough to activate all neurons)
- Some metrics also only apply to ReLUs, or are difficult to generalize beyond ReLUs
- Want a metric that is simple, cheap to compute, and easily applied to all architectures

The TensorFuzz Library

Basic Fuzzing Procedure

- Instead of interacting with computer program, interact with TensorFlow graph which we can feed inputs and get outputs from
- Unlike traditional CGF, restrict inputs to valid inputs
  - Images are correct shape and size, values restricted to range of values in actual dataset under consideration
  - For text, restrict to characters present in dataset

The TensorFuzz Library

Basic Fuzzing Procedure

- Given the seed corpus, until instructed to stop, the fuzzer chooses an input, mutates it, and feeds it to the neural network
- Then, a set of coverage arrays and metadata arrays (from which the objective function will be computed) are extracted from the network
- If the mutated input adds coverage, add to corpus; add to test cases if objective function is satisfied

The TensorFuzz Library

Detailed Fuzzing Procedure

- Input chooser
  - Uniform random selection worked well
  - Faster to use a heuristic $p(c_k, t) = \frac{e^{t_k - t}}{\sum e^{t_k - t}}$ where $p(c_k, t)$ is the probability of choosing element c_k at time t and t_k is when the element was added to the corpus
- Mutator
  - For images, either add white noise with user-specified variance or add white noise but constrain the difference between input and mutation to some l_∞ norm; afterwards, clip image to original input range
  - For text, randomly delete a character, add a character, or substitute a character

The TensorFuzz Library

Detailed Fuzzing Procedure

- Objective function
  - Generally, we run the fuzzer with the goal of having the network in some state - maybe a state regarded as erroneous
  - Objective function is applied to metadata arrays and inputs which cause errors are flagged
- Coverage analyzer
  - Want to check if network is in a state that it has not been in before
  - Want the check to be fast (and simple)
  - Want it to work for many types of computation graphs
  - Want it to be hard to exercise full coverage to cover as many behaviors as possible
  - Want new coverage to provide incremental progress

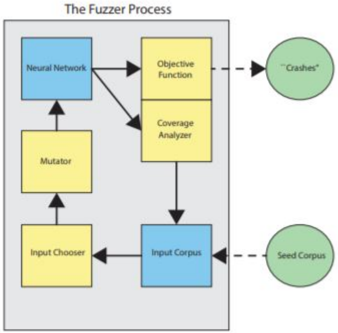
The TensorFuzz Library

Detailed Fuzzing Procedure

- Coverage analyzer (cont.)
  - Naive method would treat each activaton vector as new coverage - all inputs trivially increase coverage
  - Instead, look up nearest neighbor and add new activation vector it is sufficient far away (greater than some L)
  - Use open-source FLAN to compute nearest neighbors
  - Found that performance is good even if only tracking logits or layer before logits
  - Potential optimization: do not actually need to know nearest neighbor; just need existence within some L so could use distance-sensitive bloom filter at the expense of missing some "new" coverage vectors

The TensorFuzz Library

Detailed Fuzzing Procedure



```
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 1: 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.

The TensorFuzz Library

Batching and Nondeterminism

- TensorFlow graphs are made to take advantage of hardware-parallelism, so process batch of inputs and analyze coverage for batch of arrays each iteration
- Computation graphs often give nondeterministic results due to intrinsic random functions and large accumulations on GPUs; deal with this in naive way: if same input produces different coverage, then simply have the input appear twice in the corpus

Experimental Results

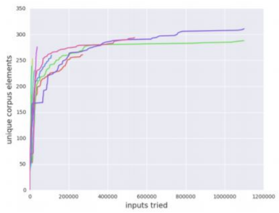
Numerical Errors in Trained Models

- Since neural networks use floating point operations, they are susceptible to numerical issues during training and evaluation; focus on finding inputs which result in NaN values
- These are usually difficult to find because they are triggered by very specific inputs
- Numerical errors are important to find because they can be very dangerous if first encountered in productions; CGF can find a large number of these errors
- CGFs can quickly find these errors by adding check numeric ops to metadata

Experimental Results

Numerical Errors in Trained Models

- Gradient descent based methods might be faster, but it is unclear what kind of objective function you would use to find NaN errors
- Random search is prohibitively inefficient for finding these errors; unable to find a non-finite element



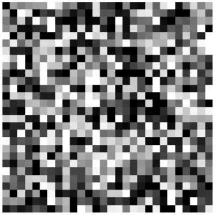


Figure 2: We trained an MNIST classifier with some unsafe numerical operations. We then ran the fuzzer 10 times on random seeds from the MNIST dataset. The fuzzer found a non-finite element every run. Random search never found a non-finite element. Left: the accumulated corpus size of the fuzzer while running, for 10 runs. Right: an example satisfying image found by the fuzzer.

Experimental Results

Disagreements from Quantized Versions

- Quantization is where neural network weights are stored and neural network computations are performed using numerical representations which have fewer bits
- This technique reduces computational cost and size of networks
- It's important to find errors in quantized models because quantization is not useful if accuracy is lowered too much
- Just checking existing data does not work: MNIST trained on 32-bits and truncated to 16-bits had no disagreements on existing data

Experimental Results

Disagreements from Quantized Versions

- CGF can quickly find errors in small regions around data
  - Running fuzzer with mutations restricted to 0.4 l_∞ orb around seed images resulted in disagreements in 70% of examples tried
  - These images have unambiguous class semantics due to the small perturbation
- Random search once again failed to find new disagreements

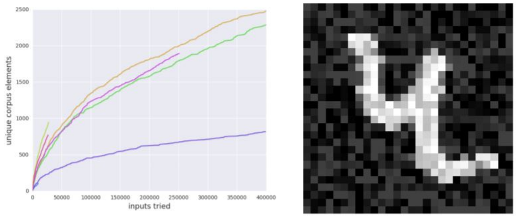


Figure 3: We trained an MNIST classifier with 32-bit floats and then truncated the associated TensorFlow graph to 16-bit floats. Both the original and the truncated graph made the same predictions on all 10000 elements of the MNIST test set, but the fuzzer was able to find disagreements within an infinity-norm 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.

Experimental Results

Undesirable Behavior in Character-Level Language Models

- Train a character-level language model using 2 layer LSTM on the Tiny Shakespeare dataset
- Sample from model given priming string
- Enforce two "errors": should not repeat same word too many times in a row, should not output words from a "blacklist"
- Fuzz the model using hidden state of LSTM; use fixed random seed which we reset at every sampling and mutation function described earlier
- After 24 hours, both TensorFuzz and random search generated repeat words
- TensorFuzz generated 6/10 words from blacklist compared to 1/10 from random search

Conclusion

- Introduced concept of CGF for neural networks and described how to build a useful fuzzer in this context
- Demonstrated practical applicability of TensorFuzz by finding numerical errors, finding disagreements between networks and their quantized versions, and surfacing undesirable behavior in RNNs
- Released an open source version of TensorFuzz

References

- https://arxiv.org/pdf/1807.10875.pdf