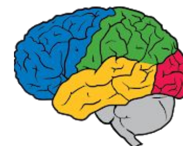
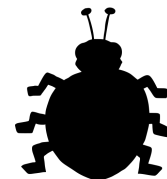
TensorFuzz: Debugging Neural Networks with Coverage Guided Fuzzing

Augustus Odena, Catherine Olsson,
David G. Andersen, Ian Goodfellow





How Should We Test Neural Networks?

What are we testing for?

- No crashes
- No NaNs
- Same answer for mutated image if difference is small
- Same answer from quantized & non-quantized network
- Same answer from base and refactored network

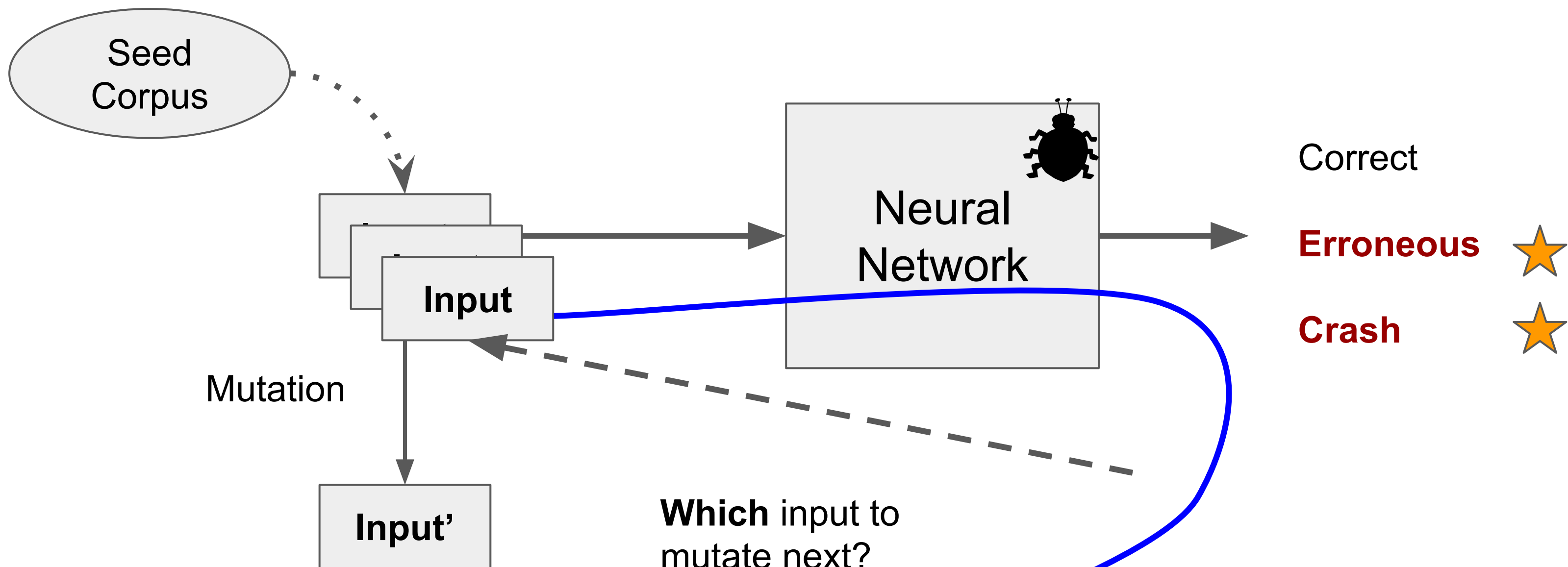
Some related work on this: Pei et al, Sun et al, Ma et al, Tian et al, Wicker et al - see paper for more details

Our idea: neural networks are programs, so let’s use an idea from testing programs: coverage guided fuzzing.

Fuzzing Normal Programs

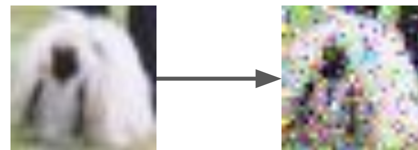
- 2 big fuzzers: AFL and libFuzzer
- Maintain a corpus of inputs, mutate elements in that corpus, and put new mutations in the corpus if they ‘exercise’ new coverage
- What is coverage? Basic blocks hit, say.
- These are used *a lot* in the security community and have found *a lot* of serious security flaws.

Fuzzing to Find Bugs in Deep Neural Networks



Coverage-guided!
Run example through network: Did it explore new states?
Pick next examples to maximize exploration.

Fuzzing Challenges for DNNs:



- **Mutation**: How to change input to better find bugs?
  - For crashes, anything goes.
  - For errors, if we scramble an image, is it still a dog?
  - We keep it simple: Gaussian noise, sometimes with constraints to ensure image remains in domain, etc.

- **Guidance**: How do we define “coverage” for a DNN?

- **Errors**: What defines an error?

Defining Coverage: (Approx.) Nearest Neighbors

Code coverage doesn't apply well to DNNs:

- All examples trigger identical code paths, only the data differs

Prior work used, e.g., "Neuron Coverage" (though not for fuzzing):

- Ensure each ReLU is explored in both 0 and > 0 states.
- Unfortunately, too easy to satisfy: Not enough exploration

Our contribution: Nearest neighbor algorithm on activations

- In practice, use activations of last layer before softmax, often a ~1024 dimension vector.

Nearest Neighbors on Activations is Practical

Inputs already tested represent N points in R^D
Given a query point, find the distance *dist* to the closest of those N points
- Cosine or Euclidean distance

TensorFuzz considers an input “interesting” if $dist > thresh$

This technique is practical:

- Approximate nearest neighbor has preprocessing cost polynomial in N and D and query costs polynomial in D and log(N).
- Real-world performance is even better when data has structure
- And we only need distance, not actual point, so can relax further.

Defining Errors: Property Based Testing

In PBT, assert a property of the function that should always hold.

PBT tool generates random examples trying to find violations.

- Best known example: Haskell's QuickCheck

In TensorFuzz, the user specifies the properties to hold. Examples:

- No crashes
- No NaNs
- Same answer for mutated image if difference is small
- Same answer from quantized & non-quantized network
- Same answer from base and refactored network

Important Detail: Input Chooser

Choice isn't just based on novelty - in practice, many inputs are novel.

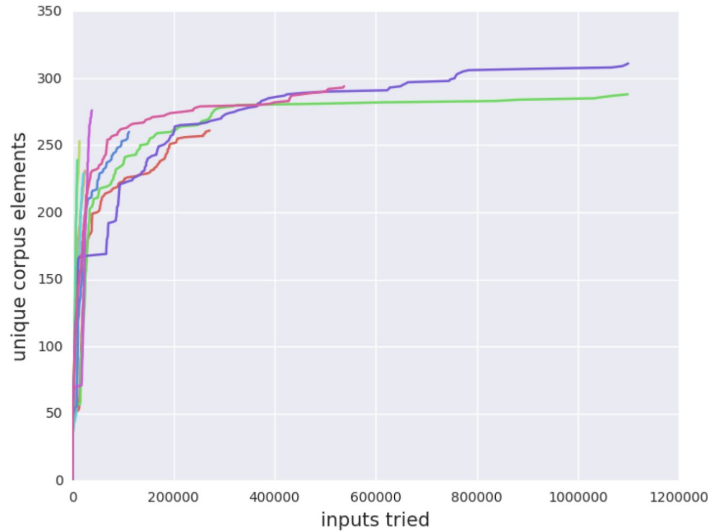
Bias to recently added elements → depth-first search

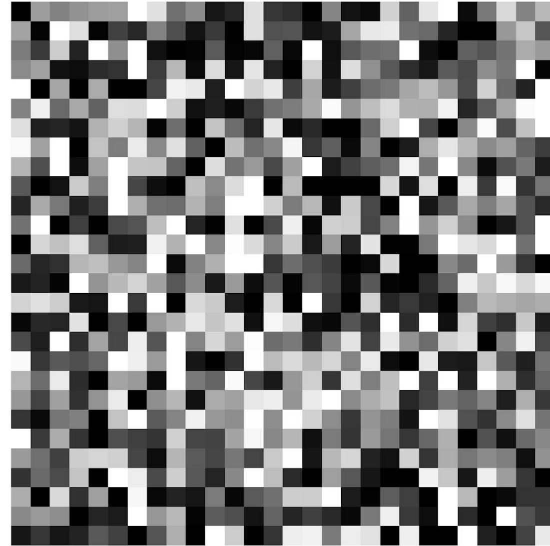
- Will use mutations from a single seed until they stop being interesting

Otherwise, "breadth-first": Explore all interesting examples equally

TensorFuzz has a hyperparameter that biases slightly to recently added elements.

TensorFuzz finds NaNs

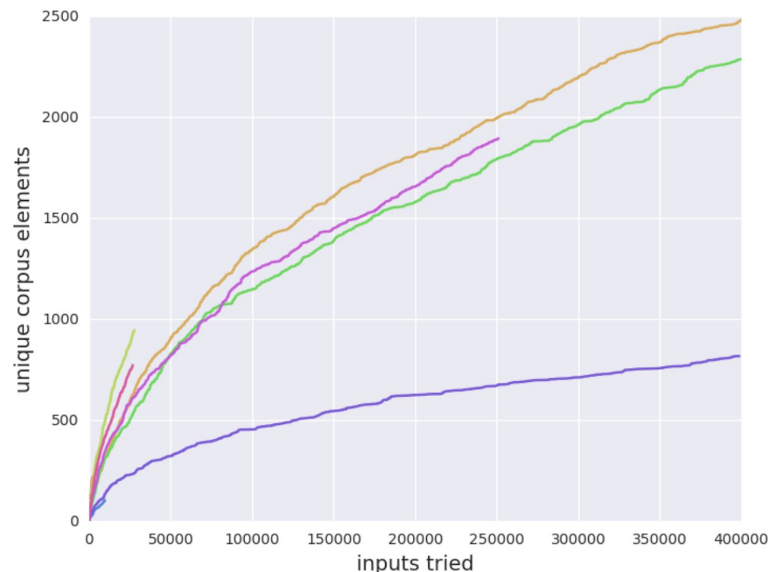


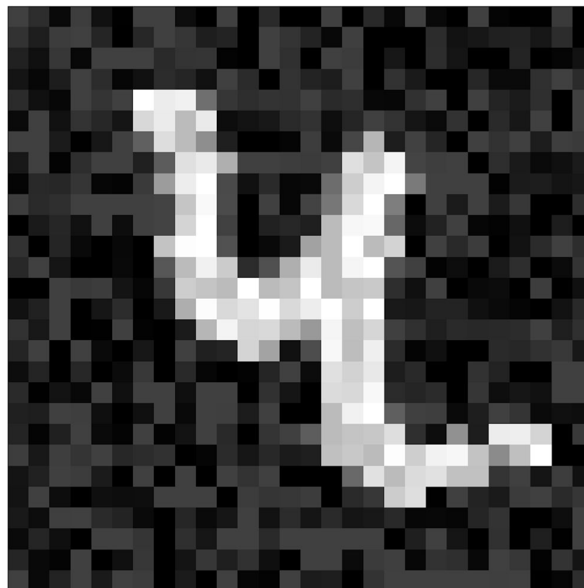


NaNs cause trouble for everyone, but it's hard to track them down.

We 'fault-injected' a bad loss function into a neural network and showed that TensorFuzz could find NaNs substantially faster than a baseline random search.

TensorFuzz surfaces Quantization ‘Errors’





We often want to ‘quantize’ neural networks.

How do we test we’ve done this correctly? We could look at differences on test set, but often few show up. Instead, fuzz for inputs that yield differences

TensorFuzz Facilitates Refactoring

```
functional_ops.map_fn(lambda x:
      array_ops.reverse(x, [flip_index]),
      image, dtype=image.dtype)
```

```
flipped_input =
   array_ops.reverse(image,
     [flip_index + 1])
```

45x speedup at 32x32

Inefficient flipping in tensorflow random flip.

A 6 line of code change sped up flipping by 2.6x to 45x.

Key technique during refactoring:

Fuzz the difference between new code and old code.

Tensorfuzz finds Bugs in open source models

Took popular open source GAN implementation and added tests.

We knew that if learning rate too high, loss would NaN.

```
def objective_function(corpus_element):
    loss = corpus_element.metadata["loss"]
    grad = corpus_element.metadata["grad"]
    if loss > 0.1 and abs(grad) < 0.0001:
        return True
    return False
```

✔️ TensorFuzz found a NaN-producing input to this loss function in seconds.

Try TensorFuzz:

https://github.com/brain-research/tensorfuzz

Questions?