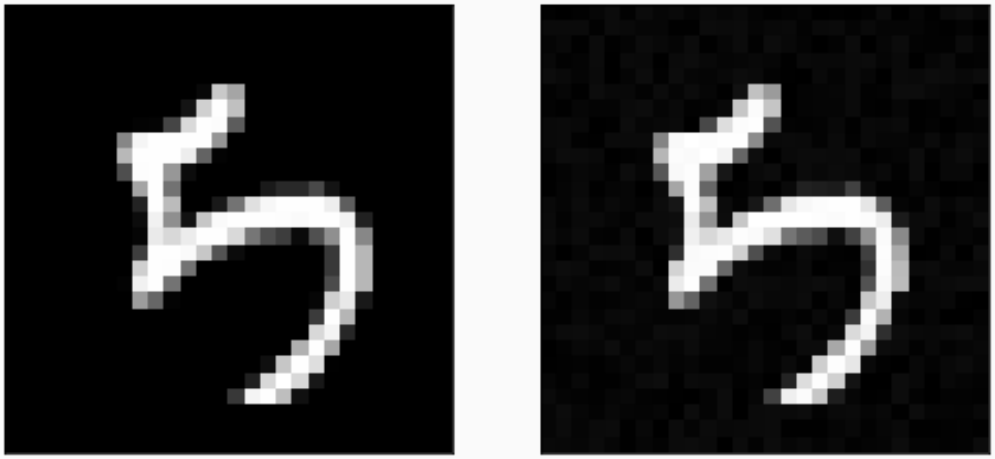
A Comparison of Adversarial Attack Methods

ECE653 Project

Vineel Nagisetty, Laura Graves, Joseph Scott

Overview

Adversarial examples are specifically chosen examples that cause a neural network to make an incorrect classification. They are usually regular examples that have been slightly perturbed via some means.



Left: original image. Right: perturbed image.

Most perturbations are limited to a small bound (usually an ℓ_2 or ℓ_∞ bound) so they appear imperceptible to human observers.

Adversarial Attack Methods

Most adversarial attacks are found using **gradient descent**.

FGSM algorithm:

$$x_{adv} = x + \epsilon sign \Delta_x J(\theta, x, y)$$

Problem: The gradient does not always help. Defense mechanisms such as **adversarial training** or **obfuscated gradients** make it difficult to gain useful information from the gradient, and consequently these attacks are frequently ineffective.

Solution: Find better attack methods that don't use the gradient. In this paper we examine non-gradient methods such as **fuzzing**, **genetic algorithms**, and **symbolic execution** and compare them against state-of-the-art gradient-based methods.

Defense Methods

In this paper we use two common defense methods:

Adversarial training works by training a model as usual, and then generating a set of adversarial examples that that model fails to correctly classify. That set of adversarial examples is correctly labeled and added to the training set, and the model is trained further. This process is repeated iteratively, reinforcing the robustness of the model against adversarial attacks with each iteration.

Gradient Obfuscation works by making the gradient give less useful information, and as a result making gradient-based attacks ineffective. The gradient obfuscation technique we use here is called **thermometer encoding**, and it works by downscaling the input to a significantly lower resolution. This not only makes the gradient difficult to follow, but reduces the effect of small perturbations, forcing attackers to make larger changes to have an effect.

Our Attacks: Fuzzing

We've developed 4 fuzzing-based adversarial attacks:

- **StepFuzz**: $x_{adv} = x + \epsilon z$, where z is a vector where each value is randomly assigned to a value in [-1, 0, 1].
- **LaplaceFuzz**: $x_{adv} = x + z$, where $z \in Laplace(\mu, b)$
- **xAIFuzz**: $x_{adv} = x + \epsilon(z * i)$, where z is sampled from a Gaussian distribution and i is an importance vector calculated using an explainable AI system
- **VinFuzz**: A fuzzing method that sets lower and upper bounds for each feature and iteratively randomly assigns a value within those bounds and updates the bounds based on the gradient (algorithm in appendix)

Our Attacks: Genetic Algorithm

A **Genetic Algorithm** (GA) is a subset of AI that utilizes mechanisms inspired by biological evolution to solve various problems. At a high level, a population of candidate solutions to the problem at hand are maintained. This population is randomly modified and combined (akin to evolutionary processes in biology) based on their fitness - a scalar score that tracks their efficacy.

Our implementation utilizes standard GA mechanisms. The notable difference is the fitness function which combines the target models confidence in the true label with the similarity distance.

$$\text{fitness}(x_{adv}) = y_{adv} + \textit{similarity}(x_{adv}, x)$$

Our Attacks: Symbolic Execution

We consider three approaches to symbolic execution to generate adversarial examples on neural networks:

1. ESBMC – C prover
2. SMT Solvers
   - 2.1 Z3
   - 2.2 CVC4
   - 2.3 Bitwuzla
3. Marabou

Our Attacks: Symbolic Execution

Off-the-shelf Attacks

As a comparison, we evaluate a number of state-of-the-art attacks. These are:

- **DLFuzz**: A gradient-based fuzzer that guides fuzzing in the direction of greater neuron activation
- **Fast Gradient-Sign Method**: $x_{adv} = x + \epsilon sign \Delta_x J(\theta, x, y)$
- **Basic Iterative Method**: $x_{adv} = x_0 + \sum_{i=0}^{n} \epsilon sign \Delta_{x_i} J(\theta, x_i, y)$, where $x_k = x_{k-1} + \epsilon sign \Delta_{x_{k-1}} J(\theta, x_{k-1}, y)$
- **Carlini-Wagner Attack:** A gradient-based attack that uses multiple-start gradient descent and a custom objective function to create examples that are adversarial with minimal perturbation
- **Madry Attack**: A modified version of projected gradient descent that has been shown to perform well against defended models

Experiment Setup

1. We implemented using **python 3**, **keras** framework for **tensorflow**. DLFuzz and Genetic Algorithm are implemented in keras, and the library **cleverhans** implements FGSM, BIM, C&W, Madry, and multiple other attack algorithms in tensorflow.
2. We evaluated using a python framework that tests each algorithm over a collection of benchmarks. When an adversarial attack is found, the algorithm moves on to the next. If an algorithm is unable to find an adversarial example within a timeout limit, it is stopped and moves on to the next example with a time score of twice the timeout length (consistent with how the Par2 scores are calculated in the SAT community).
3. We ran experiments using a local hardware running on Ubuntu 16.04 with four Intel i5-4300U (CPU 1.9 GHz) cores and 16 GB RAM.

Benchmarks

We ran our experiments on benchmarks against a suite of target models chosen to highlight the effectiveness of different attack methods.

Target models:

- **FCNN and Robust FCNN**: A fully connected network with a single hidden layer of width 128. Robust FCNN is made robust through adversarial training.
- **CNN and Robust CNN**: A convolutional network with two convolutional layers and a fully connected layer. Robust CNN is made robust through adversarial training.
- **Thermometer CNN**: A CNN using thermometer encoding, a gradient obfuscation technique that downscales the input resolution to reduce the impact of small perturbations

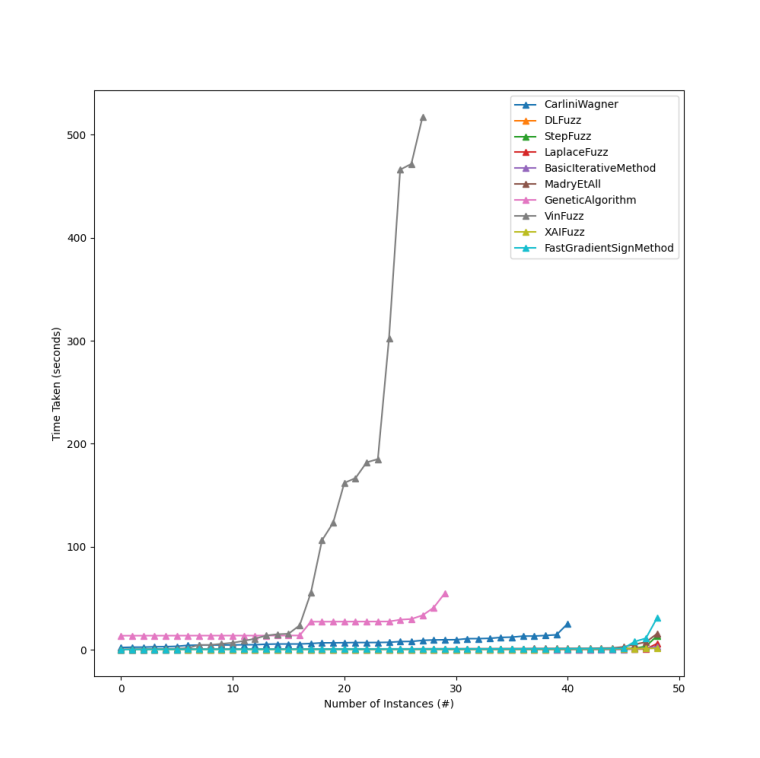
Our benchmarks are:

- **Main-Similar**: Attacking FCNN and Robust FCNN with an allowable ℓ_2 perturbation bound of 5 and a 600 second timeout
- **Main**: Attacking FCNN and Robust FCNN with an allowable ℓ_2 perturbation bound of 10 and a 600 second timeout
- **CNN**: Attacking CNN and Robust CNN with an allowable ℓ_2 perturbation bound of 10 and a 600 second timeout
- **Thermometer**: Attacking thermometer CNN with an allowable ℓ_2 perturbation bound of 10 and a 600 second timeout

Results - Demo

An example of the results of various algorithms is given in the Figure above. The original image (on the left) is that of a 5, and the result of each algorithm (along with the new label) is displayed.

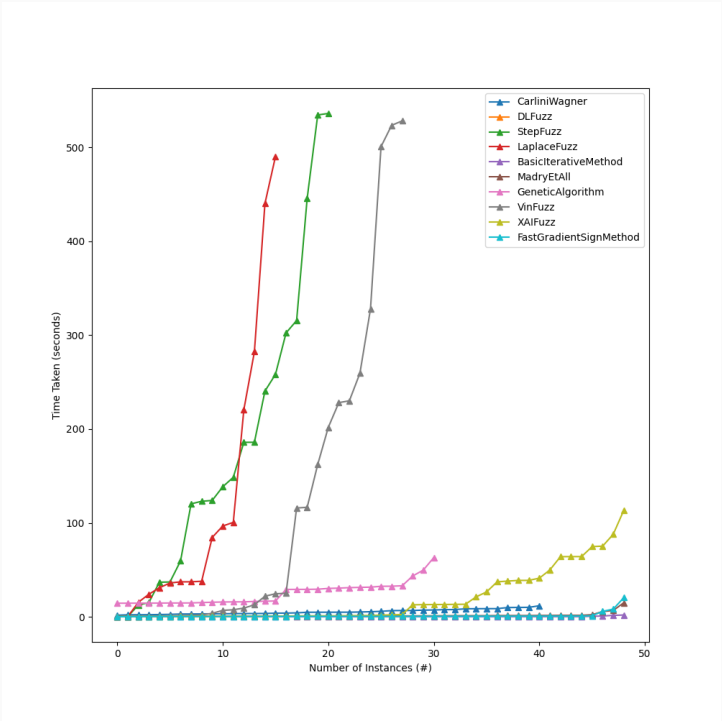
Results - Main Benchmark



	Main	
Generator	**Time**	**Sim**
BasicIterativeMethod	10.9916	38.3133
LaplaceFuzz	16.8971	370.0474
XAIFuzz	18.8156	236.3747
StepFuzz	34.4126	350.8247
FastGradientSignMethod	58.535	25.2557
MadryEtAll	68.2202	25.3125
CarliniWagner	9,915.0678	197.7778
GeneticAlgorithm	23,439.0246	420.2257
VinFuzz	28,050.7742	457.1261
DLFuzz	58,800	980

Results of experiments run on the Main Benchmark are given in the cactus plot and table above.

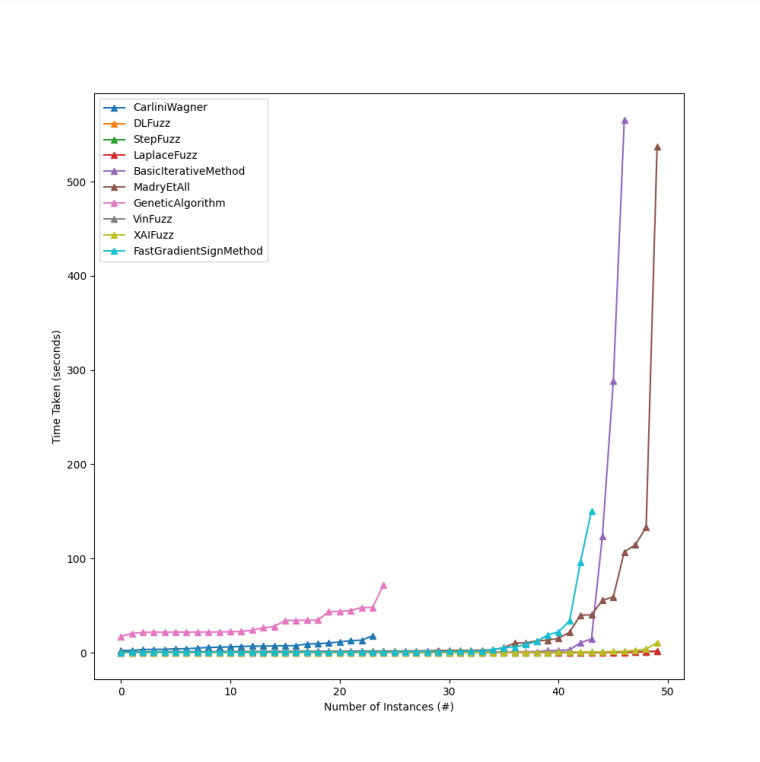
Results - Main-Similar Benchmark



	Main-Similar	
Generator	**Time**	**Sim**
BasicIterativeMethod	14.2705	38.3133
MadryEtAll	71.9455	25.315
FastGradientSignMethod	80.2188	25.2557
XAIFuzz	938.7496	199.0752
CarliniWagner	9,821.9288	117.7778
GeneticAlgorithm	22,371.8497	221.0096
VinFuzz	28,514.1743	248.1306
StepFuzz	37,421.9183	374.4402
LaplaceFuzz	41,535.0076	395.6257
DLFuzz	58,800	490

Results of experiments run on the Main-Similar Benchmark are given in the cactus plot and table above.

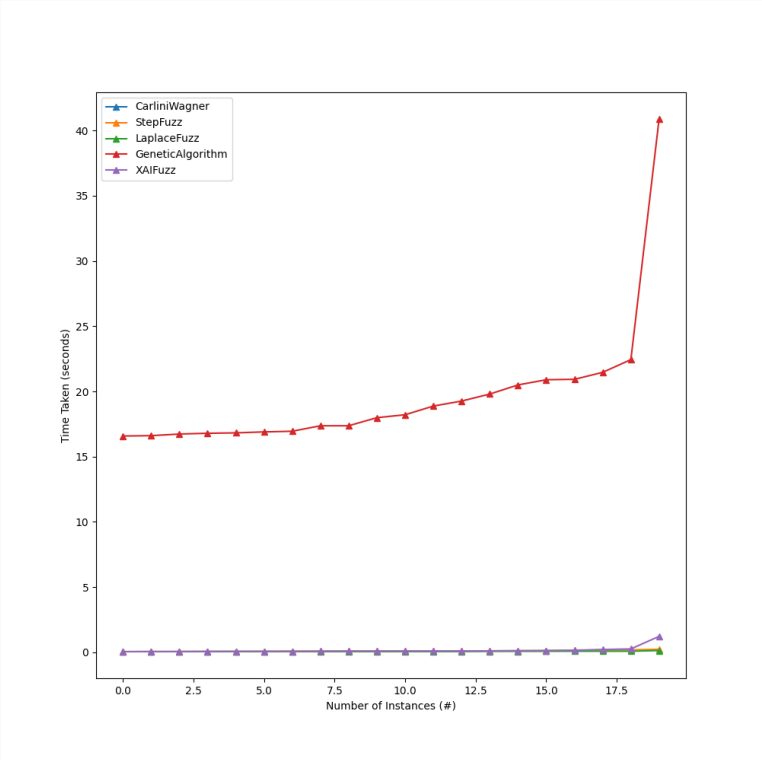
Results - Main-CNN Benchmark



	Main-CNN	
Generator	**Time**	**Sim**
LaplaceFuzz	10.6816	319.0187
StepFuzz	15.7379	314.1623
XAIFuzz	31.5575	207.5959
MadryEtAll	1,230.6365	28.747
BasicIterativeMethod	4,623.3858	90.8382
FastGradientSignMethod	7,589.8691	138.8464
GeneticAlgorithm	30,773.1413	532.6955
CarliniWagner	31,373.3318	534.5619
VinFuzz	58,800.4578	980.4263
DLFuzz	60,000	1,000

Results of experiments run on the Main-CNN Benchmark are given in the cactus plot and table above. Note that gradient based algorithms do not work on models with thermometer encoding.

Results - Thermometer Benchmark



	Main-Similar	
Generator	**Time**	**Sim**
LaplaceFuzz	1.3311	64.082
StepFuzz	2.0283	63.1283
XAIFuzz	3.2711	54.3104
GeneticAlgorithm	393.3436	26.2774

Results of experiments run on the Thermometer Benchmark are given in the cactus plot and table above.

Discussions

1. The lower the similarity distance limit, the harder it seems to be to find adversarial examples.
2. The performance of each type of algorithm (except for genetic algorithm) depends on model architecture and defense method.
3. Gradient based methods work remarkably well on undefended fully connected networks. On the other hand, fuzzing algorithms outperform others on defended cnn models. Finally, genetic algorithm shows consistent performance on all benchmarks.

Conclusion

1. The more complex a model is and the more difficult it is to gain meaningful information from the gradient, the less useful standard gradient-based attacks become.
2. This highlights the need for more sophisticated attack methods that don't rely on the gradient, as well as shows the shortcomings of the state of the art attacks that rely heavily on gradients.
3. Future research would do well to focus on logic-based methods for analysis, utilizing logic solvers and symbolic execution tools to be able to efficiently find counterexamples (in this case, adversarial examples) that show where models are vulnerable.

Appendix: VinFuzz Algorithm

`VinFuzz`$(\theta, x, y, n, \epsilon)$

θ = model parameters, x = original example, y = true label, n = number of iterations, ϵ = perturbation range

nx = new Array[x.size]

lb = new Array[x.size]

ub = new Array[x.size]

set the initial lower and upper bound and generate a random example in those ranges

for $i = 0;\ i < x.size;\ i++$ **do**

    lb[i] = x[i] - ϵ

    ub[i] = x[i] + ϵ

    nx[i] = random(min=lb[i], max=ub[i])

end

run n iterations of the loop **for** $j = 0;\ j < n;\ j++$ **do**

    use the sign of the gradient to update the lower and upper bounds and generate a new example

    grad = $sign\Delta_{nx}J(\theta, nx, y)$

    for $i = 0;\ i < x.size;\ i++$ **do**

        if *grad[i] == -1* **then**

            ub[i] = nx[i]

        if *grad[i] == 1* **then**

            lb[i] = nx[i]

        nx[i] = random(min=lb[i], max=ub[i])

    end

end

return **nx**
