

A Comparison of Adversarial Attack Methods

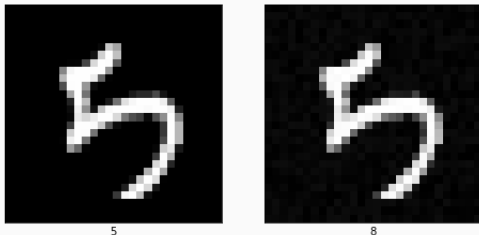
ECE653 Project

Code: <https://github.com/vin-nag/checkYourPerturbations>

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:

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

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 $z \in \text{Laplace}(\mu, b)$
- **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.

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

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:** $x_{adv} = x + \epsilon \text{sign} \Delta_x J(\theta, x, y)$
- **Basic Iterative Method:** $x_{adv} = x_0 + \sum_{i=0}^n \epsilon \text{sign} \Delta_{x_i} J(\theta, x_i, y)$,
where $x_k = x_{k-1} + \epsilon \text{sign} \Delta_{x_{k-1}} J(\theta, x_{k-1}, y)$
- **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

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 Attacks – ESBMC

- ESBMC is a C prover that takes as input a C program with annotated preconditions and postconditions and determines validity.
- ESBMC is state of the art in validating floating-point programs.
- How we generate adversarial inputs:
 1. Translate the neural network into a C program in SSA
 2. **Precondition:** each input feature needs to be within a perturbation budget
 3. **Postcondition:** The maximum output neuron is not the label of the target image
- Bugs!!! ESBMC cannot soundly generate adversarial examples. Fails multiple unit tests.

Our Attacks: Symbolic Attacks – SMT Solvers

- We translate each neural network to an SMT-LIB benchmark.
- Unit tests that ESBMC failed are passed by SMT Solvers.
- Neural networks trained on MNIST with a single hidden layer results in an SMT equation with 100,000 multiplications
- All solvers fail to solve with an hour long timeout.

Our Attacks: Symbolic Attacks – Marabou

1. Marabou is an NN verification engine that implements the Reluplex algorithm by Katz et. al.
2. Marabou works by converting the problem of neural network verification to a linear program with special handling of the the relu activation function
3. Marabou allows for preconditions of setting inputs and outputs within a fixed interval over literal values
4. However, this is insufficient in generating adversarial examples, as we need to be able to compare the outputs of neuron values and is excluded from our results.

Experiment Setup

1. We implemented using **Python 3**, **keras** framework for **tensorflow** and **cleverhans**. The code can be found at:
<https://github.com/vin-nag/checkYourPerturbations>
2. We evaluated using a framework that tests each algorithm over a collection of **benchmarks**. The framework runs each of the algorithm on model, input pairs until the algorithm produces a result (or results in a timeout). For each, we record the time and similarity using Par2 scoring.
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:

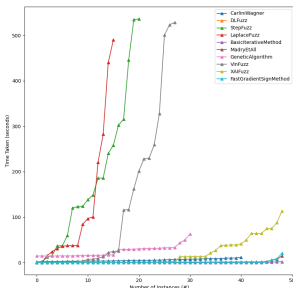
- **FCNN and Robust FCNN:** A fully connected network with a single hidden layer of width 128. Robust FCNN is made robust through adversarial training.
- **CNN and Robust CNN:** A convolutional network with two convolutional layers and a fully connected layer. Robust CNN is made robust through adversarial 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:

Benchmark	Models	Similarity (l2 Distance)	Time Limit (sec)
Main-Similar	Regular and Robust FCNN	5	600
CNN	Regular and Robust CNN	10	600
Main	Regular and Robust FCNN	10	600
Thermometer	Thermometer CNN	10	600

Table 1: Benchmarks created and used in this work.

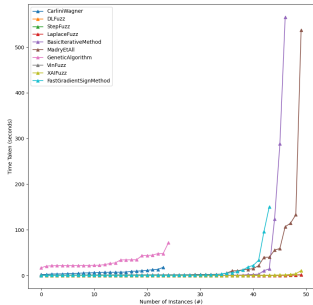
Results on the Main-Similar Benchmark



Generator	Main-Similar	
	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 on Main-CNN Benchmark



Generator	Main-CNN	
	Time	Sim
LaplaceFuzz	10.6816	319.0187
StepFuzz	15.7379	314.1623
XAI Fuzz	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.

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. Fuzzing algorithms surprisingly outperform gradient based ones on defended CNN models. However, they perform poorly when the perturbation budget is lowered.
3. Symbolic based methods do not scale well, and do not provide the right functionality to check for adversarial attacks.
4. Genetic algorithm shows consistent yet average performance on all benchmarks.

Conclusion and Future Work



1. This work 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.
2. Symbolic methods can potentially 'certify' adversarial robustness, but do not currently scale to regular sized DNNs.
3. Perhaps there may be a way to combine gradient-based and symbolic methods in order to create a method that can leverage the benefits of each.

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

$nx = \text{new Array}[x.\text{size}]$

$lb = \text{new Array}[x.\text{size}]$

$ub = \text{new Array}[x.\text{size}]$

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

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

$lb[i] = x[i] - \epsilon$

$ub[i] = x[i] + \epsilon$

$nx[i] = \text{random}(\text{min}=lb[i], \text{max}=ub[i])$

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

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

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

if $\text{grad}[i] == -1$ **then**

$ub[i] = nx[i]$

if $\text{grad}[i] == 1$ **then**

$lb[i] = nx[i]$

$nx[i] = \text{random}(\text{min}=lb[i], \text{max}=ub[i])$

end

end

return nx

Appendix: Model Architectures

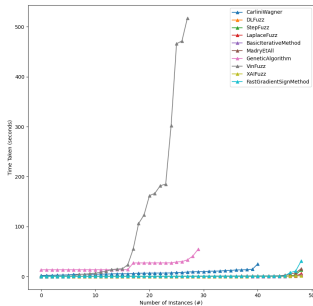
Fully Connected Neural Network (FCNN)		
Layer Name	Shape	Parameters
Input	(None, 784)	0
Dense	(784, 128)	100,480
Output	(128, 10)	1290
Total Params:		101,770

tableArchitecture of FCNN.

Convolutional Neural Network (CNN)		
Layer Name	Shape	Parameters
Conv2D	(None, 14, 14, 20)	520
MaxPooling2D	(None, 7, 7, 20)	0
Conv2D	(None, 3, 3, 20)	10,020
MaxPooling2D	(None, 1, 1, 20)	0
Dense	(None, 64)	1344
Output	(64, 10)	650
Total Params:		12,534

tableArchitecture of CNN.

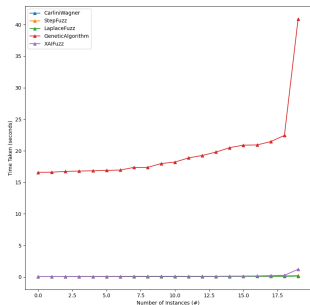
Appendix: Results on the Main Benchmark



Generator	Main	
	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
MadryEtAl	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.

Appendix: Results on Thermometer Benchmark



Generator	Main-Similar	
	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. Note that gradient based algorithms do not work on models with thermometer encoding.