Compare Common Adversarial Attack Generators for Deep Learning Systems

Jiarui Xu, ID: 20923663, Email: j572xu@uwaterloo.ca
Wanyue Quan, ID: 20880990, Email: w8quan@uwaterloo.ca
Zhifan Wu, ID: 20918890, Email: z392wu@uwaterloo.ca

Abstract

Deep learning has evolved into a widely used and efficient framework of artificial intelligence, especially in the realm of computer vision and classification for complex data. However, deep learning models can be extremely sensitive to small changes in input data and thus can be easily fooled by perturbations in the input, which are known as adversarial attacks. Multiple test methods are invented to identify this problem. This report introduces a fuzzing method that only changes a small subset of inputs, and compares multiple types of adversarial attacks against six carefully chosen and trained neural networks, evaluates the methods' performance, scalability with respect to model complexity & input complexity, transfer-ability, and examines the impact of adversarial defenses. Finally, future research opportunities are proposed.

The code are available at `https://github.com/1003366190/ECE653_project.git`

1 Introduction

Recently Deep Learning (DL) has been widely used in artificial intelligence areas, such as computer vision, image classification, object recognition. For example, one of the fascinating technologies, self-driving cars are designed based on image classification and object recognition realized by deep neural networks.

As neural networks (NNs) are more frequently applied to systems, attackers start to focus on fooling the neural network with crafted or modified inputs instead of attacking the system's security vulnerability such as invading the database, this method is known as the adversarial attack. The modified input data are usually the edge cases of the target NN and are targeted to reduce the confidence of the NN's output, causing the NN to misclassify inputs (untargeted adversarial attack), and even control the NN's output (targeted adversarial attack).

These attacks can potentially cause serious problems. For example, during automatic driving, if the NN misidentifies the stop sign that has some physical perturbations, such as paints, dirt, and fails to stop in time, it could lead to serious consequences including car damage and human injury. Therefore, there is a great need to test and improve the robustness of DL systems. In the past, multiple test methods have been proposed to address this problem. In this project, we compare the chosen test methods based on their performance against multiple Convolutional Neural Networks (CNNs) for image classification with different architecture, and measure the method's scalability and transferability. In addition to the existing methods, we introduced new methods to generate adversarial attacks for CNN based on the idea of random fuzzing.

This report presents the following contributions:

1. In this report, we produce a thorough literature survey of approaches for generating adversarial examples.

2. We perform an evaluation on the selected candidates to compare their efficiency and scalability to identify weaknesses, generate adversarial attacks against different benchmark neural networks.

3. We design new methods to generate adversarial examples based on gradients and random fuzzing that changes the top N most influential pixels each iteration.

4. We examined the effectiveness of the adversarial defense method "adversarial training" against multiple methods.

The rest of the report is organized as follows. Section 2 will introduce the background of adversarial attacks. In section 3, we provide a thorough description of the TAV Tools we chose. Section 4 gives the evaluation of the selected methods. Finally, the conclusion is made in Section 5.

2 Background

Szegedy et al.[1] originally presented the notion of adversarial examples in their work, in which they utilize adversarial examples to fool DNNs. They discovered that introducing a small perturbation to the input data, where the changes are not visible to the human eyes, causes the neural network to misclassify it with high confidence. An example is shown in Fig. 1, the original image x is considered as "macaw" by the model with 97.3% confidence. After a slight perturbation is added to the original image, the model classifies it as "bookcase" with 88.9% confidence. However, human eyes cannot notice the differences.

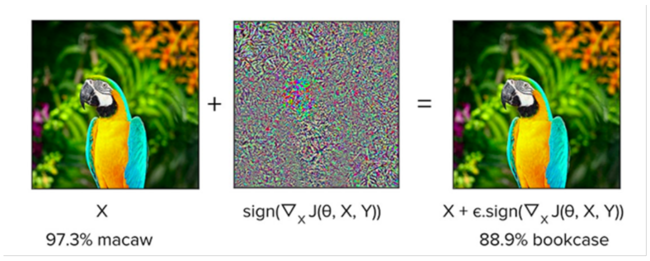


Figure 1: Generating adversarial example using FGSM[2]

Adversarial attacks are either classified as Whitebox attacks or Blackbox attacks[3]. In Whitebox attacks, information about the target model, such as its architecture, parameters, training process, or training data, is given, but in BlackBox attacks, only the output of the target model is known. Although Blackbox attacks are more practical in the actual world since attackers seldom know the internals of the target, Whitebox attacks are needed to assess the target model in worst-case situations[3]. In addition, adversarial examples generated by white-box methods on a transparent model can be used to attack black-box models for the same tasks. Therefore, in this report, we mainly focus on the white-box methods.

One important characteristic of adversarial examples is Transferability[4], which means that adversarial samples created to fool one model can also be used to fool other models which are trained to do the same task and are trained over similar datasets. In this report, we take transferability as one of our measure metrics.

Many attacking methods[2],[5],[6],[7],[8],[9] have been proposed to generate adversarial examples. Currently, researchers have proposed several methods against adversarial attacks with the purpose of better performance in the robustness of models. For example, adversarial training[3] is an intuitive defense method to make models more robust by injecting adversarial samples into the training set.

Some prior research has been proposed to compare different adversarial attack methods. In their work, Zhang et al.[4] review the latest study of adversarial examples, discuss the advantages and disadvantages of the state-of-the-art adversarial examples generation methods, and review the existing defenses and discuss their limitations. Our work differs from this in the following ways:

1. We design a new adversarial attacking method based on mutational random fuzzing and gradients. This method only changes a relatively small number of pixels.

2. We measure the scalability in the metrics of distance and runtime of different methods with respect to the increase in model complexity and input complexity.

3. We examined the impact of adversarial training through FGSM against different types of attacks.

3 Method

We choose 7 common adversarial attack methods and compare them with new methods we introduced.

DeepFool

DeepFool is developed by Moosavi-Dezfooli et al[7] based on the L_2-norm. They assume that the neural network is completely linear, with a hyperplane separating one class from another. The algorithm aims to find the closest distance from the original input x to the decision boundary of a multi-class classifier. For nonlinear neural networks with high dimensions, they use a linear approximation during the iterative attack. In each iteration, they perturb the input by linearizing the model's per-class decision boundaries and obtain an optimal update direction on the linearized model[4]. Then a small perturbation is added to the input in this direction. This process is repeated until misclassifying occurs. According to the authors, DeepFool can generate adversarial examples with perturbations that are smaller than the ones computed by FGSM.

FGSM

Goodfellow et al.[2] demonstrated a fast adversarial example generation method called FGSM (short for Fast Gradient Sign Method), which reliably leads to misclassification of inputs by various models. In this method, a one-step gradient along the direction of the sign of that gradient is performed, thus FGSM is considered as a one-step gradient-based approach.

BIM

Kurakin et al.[5] extended the fast method by introducing a straightforward approach run for multiple iterations with a small step size. In the process of each iteration, pixel values were clipped to ensure small differences from the original ones. The authors also introduced a method called the iterative least-likely method, which attempts to allow an adversarial image to be categorized to a specific desired target class, to achieve a specific class attack.

Madry Attack

Madry et al.[10] used a strategy called PGD (Projected Gradient Descent) as a reliable first-order adversary, which refers to the strongest attack utilizing the local first-order information of the network, to find the adversarial examples. For input x, this strategy finds an adversarial example x' which satisfies the given norm constraint $\|x' - x_p\| \leq \epsilon$. Consider B indicates the l_p-ball of radius , in which x is the center. The attack begins with a random point x_0, and repeatedly sets

$$x_{i+1} = Proj_B(x_i + \alpha \cdot g)$$
$$g = argmax_{\|v\|_p \leq 1} v^\top \bigtriangledown_{x_i} L(x_i, y)$$

Where $Proj_B$ projects an input onto the norm-ball B, α denotes a step-size, g indicates the steepest ascent direction for a given l_p-norm, and $L(x, y)$ refers to a suitable loss-function.

Essentially, the PDG attack is the same as the BIM attack mentioned previously, where the only difference is that PDG initializes to a random point of the ball while BIM starts at the original one.

MI-FGSM (Momentum Iterative Method, MIM)

Dong et al.[11] proposed a large class of iterative algorithms based on momentum to enhance adversarial attacks and integrated this term (i.e., momentum) into FSGM to make adversarial example generation more iteratively, intending to improve update directions stability and escape from poor local maxima in the process of iterations. The aforementioned momentum technique is applied to accelerate the gradient descent algorithms, which accumulate the velocity vector in the loss function gradient direction across iterations. Through experiments, this momentum-based method increases the effectiveness of adversarial attacks.

SPSA

Spall et al.[12] proposed a more efficient numerical gradient estimator called SPSA (short for simultaneous perturbation stochastic approximation). To estimate the gradient, SPSA uses 2 queries. SPSA perturbs all dimensions simultaneously by using a vector, which is sampled from the distribution with elements either +1 or -1 (i.e., Rademacher distribution):

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\Delta v) - f(x-\Delta v)}{2\Delta} \cdot v$$

Uesato et al.[13] illustrated SPSA can be applied to a black box attack. In the SPSA attack algorithm, an adversarial loss function gradient for input x is processed N numerically estimates. The algorithm gives the input perturbations along the negative average gradient direction and projects new input onto a ball originated from the original input.

Algorithm 1 Random Fuzz

```
function RANDOM_FUZZ(model, stepsize, seed_image)
    updates = random(-1,1, seed_image.shape)
    new_image = seed_image + updates * stepsize
    clip(new_image,0,1)
    return new_image
end function
function GENERATE_ADVERSARIAL(model, stepsize, seed_image)
    correct_label = model.predict(seed_image)
    adv_label = correct_label
    while correct_label == adv_label do
        seed_image = Random_Fuzz(model, stepsize, seed_image)
        adv_label = model.predict(seed_image)
    end while
    return seed_image
end function
```

C&W

Carlini and Wagner[14] introduced three gradient-based attacks under previously used distance metrics L_0, L_2, L_∞, which are much more effective than previous algorithms for generating adversarial examples. While finding adversarial examples on defensively distilled neural networks, these proposed new attacks achieve a 100% success rate. When considering choosing the target class in a targeted attack, the authors proposed three different approaches: average case, best case, and worst case. For these three distance metrics: (1) L_0 as a measure of the number of coordinates i, satisfying $x_i \neq x'_i$ (2) L_2 measures the standard Euclidean distance between original valid inputs and targeted adversarial examples (3) L_∞ refers to a measurement of the maximum change to any of the coordinates. The L_2 attack adopts an objective function based on logits, as well as change variables to avoid box constraint, while L_0 and L_∞ attacks are L_2 -attack-based but tailored to different distance metrics. An alternative formulation is used for adversarial examples construction:

$$minimize \|\delta\|_p + c \cdot f(x+\delta)$$

such that

$$x + \delta \in [0,1]^n$$

C&W's Attack can defeat effectively most of the existing adversarial detecting defenses, e.g., defensive distillation unable to make a significant increase in the robustness of neural networks in face of

C&W's Attack.

Random Fuzz

We propose this simple fuzzer that randomly mutates the input image by some magnitude until the model gives a different classification. Theoretically, if such an adversarial exists, and runtime is unlimited, the fuzzer will eventually find the adversarial, however it is impractical in real life. Hence in this method, there is no guarantee for finding an adversarial close enough to the original image. This fuzzer mimics the grammar-based mutational fuzzer used to test performance of traditional software and implements it to test neural networks. The Pseudocode is showed in Algrithm 1.

Pixel Fuzz

This algorithm picks several most influential pixels (the pixels with the greatest impact on the confidence of the classifier, i.e. the pixels with the largest gradient with respect to the loss function of the model) and mutates them each iteration, which ensures that small perturbation to a small number of pixels can lead to a large difference in the result. On average (sample size 50, on the VGG16 benchmark), each 32x32 image requires only 57(0.56%) pixels to be changed. The Pseudocode is showed in Algrithm 2.

Algorithm 2 Pixel Fuzz

```
function PIXEL_FUZZ(model, stepsize, seed_image, num_pixels)
    correct_label = model.predict(seed_image)
    loss_fn = model.layer(-1)
    gradients = gradient(seed_image, correct_label, model, model, loss_fn)
    pixel_gradient_sum = L2Norm(gradients, axis = 3)
    selected_index = select_max_indexes(pixel_gradient_sum, num_pixels)
    selected_pixels = seed_image[selected_index]
    gradient_signs = sign(gradients[selected_index])
    updates = random(-0.5,1, selected_pixels.shape)*gradient_signs*stepsize
    new_image = seed_image.update(selected_index, selected_pixels+updates)
    new_image = clip(new_image,0,1)
    return new_image
end function
function GENERATE_ADVERSARIAL(model, stepsize, seed_image)
    correct_label = model.predict(seed_image)
    adv_label = correct_label
    while correct_label == adv_label do
        seed_image = Pixel_Fuzz(model, stepsize, seed_image)
        adv_label = model.predict(seed_image)
    end while
    return seed_image
end function
```

Method	Black/White-box	Type
DeepFool	White-box	Decision boundary-based
FGSM	White-box	Gradient-based
BIM	White-box	Gradient-based
Madry	White-box	Gradient-based
MIM	White-box	Gradient-based
SPSA	White-box	Iterative optimization
C&W	White-box	Iterative optimization
Random Fuzz	Black-box	mutational-fuzzing
Pixel Fuzz	White-box	Gradient-based fuzzing

Table 1: Adversarial Attack Methods Selected for this report.

4 Evaluation

4.1 Experiment Settings

Definitions

1. **Adversarial images:** images with perturbed pixels to fool the models. Suppose that X is the original image, and there is a ML model M which can correctly classify X. X' is the adversarial image of X. X' is close enough to X such that humans cannot recognize the difference but can be misclassified by M(i.e., distance(X, X')<a threshold).

2. **Runtime:** time consumed by processor to generate one adversarial image for a specific model and image.

3. **Distance:** the distance between two images X, Y (i.e., $distance(X, Y)$) is measured as the euclidean distance (L_2 norm) between the RGB values of the images. The RGB values are normalized to floats between $[0, 1]$ by dividing the original RGB value by 255.

4. **Transfer rate:** Transfer rate measures the probability of one adversarial image generated on model M for image X, ($adv(X, M)$) can also fool another model M' for image X.

Environment and Framework

Programming Language	Framework & Libraries	CPU	GPU	RAM
Python 3.9.1	Tensorflow framework Keras library version 2.6.0 Numpy library version 1.19.5	Intel 10700k (4.85 ghz)	Nvidia RTX 3070	32GB 3600mhz

Table 2: Adversarial Attack Methods Selected for this report.

Our evaluation is implemented using python3, the Tensorflow framework, Keras library, and Numpy library. Tensorflow is the most popular open-source machine learning framework. Keras is a library associated with TensorFlow that provides high-level APIs for training models. Numpy library NumPy offers comprehensive mathematical functions. The experiment environment is showed in table 2.

Methods & Implementations

In this report, we use 9 methods to generate adversarial attacks, including 7 existing methods (DeepFool, FGSM, BIM, Madry, MIM, SPSA, and C&W) and 2 self-written methods (Random Fuzz, and Pixel Fuzz). DeepFool is implemented using code published on GitHub[15]. FGSM, BIM, Madry, MIM, SPSA, and C&W are imported from the "cleverhans"[16] library that provides multiple methods to generate adversarial examples. Random Fuzz and Pixel Fuzz are implemented by ourselves using the aforementioned algorithms.

4.2 Benchmarks

Dataset

We choose CIFAR10[17] as our dataset. This dataset contains 60000 $32 \times 32 \times 3$ images belonging to 10 different classes encoded as (airplane:0, automobile:1, bird:2, cat:3, deer:4, dog:5, frog:6, horse:7, ship:8, truck:9).

Neural network benchmarks

All the neural networks are trained until fully convergence, using augmented images generated by keras imageDataGenerator based on the CIFAR10 training set containing 50000 images and evaluated on the testing set containing the remaining 10000 images. ADAM optimizer and categorical cross entropy loss are used for all models.

VGG16[18]: it is a famous image-recognition model, which was originally designed for imagenet consisting of 224×224 images. In order to fit the CIFAR10 dataset, the architecture of this model is slightly modified according to Liu. et al 's work: (1)the convolution layers are not unchanged; (2)the fully connected layers are reduced to 512, 512, 256 neurons correspondings to the reduction in the input; (3)massive dropout and batch normalization are added to prevent overfitting. Experiment shows this model obtained an accuracy of 87.6% with 15384906 trainable parameters.

RobustVGG16: it has the same architecture as the VGG16 benchmark, but is defended using the "adversarial training" method. Based on VGG16 model, this model is further trained with 50000 images and their adversarial examples generated by FGSM ((250 images + 250 adversarial per batch)$\times$200batches$\times$2epochs). Experiment shows the accuracy decreased from 87.6% to 86% with the impact of adversarial defense.

RegularCNN: it is a much simpler convolutional neural network compared to VGG16. Experiment shows this model obtained an accuracy of 80.9% with only 1868210 trainable parameters. The figure 3 shows the architecture of this model.

SimpleCNN: it is a RegularCNN with a reduced number of neurons in each layer. Experiment shows this model obtained an accuracy of 82.1% with 925310 trainable parameters.

SimpleCNN2: it is based on SimpleCNN and trained using images of 64×64 pixels generated by duplicating each pixel in the 32×32 image 2×2 times. Experiment shows this model obtained an accuracy of 79.0% with 3325310 trainable parameters.

SimpleCNN3: it is based on SimpleCNN and trained using images of 96×96 pixels generated by duplicating each pixel in the 32×32 image 3×3 times. Experiment shows this model obtained an accuracy of 71.8% with 7325310 trainable parameters.

The SimpleCNN, SimpleCNN2, SimpleCNN3 are benchmarks for evaluating the methods' performance to generalize to models with more parameters and to adapt different input sizes.

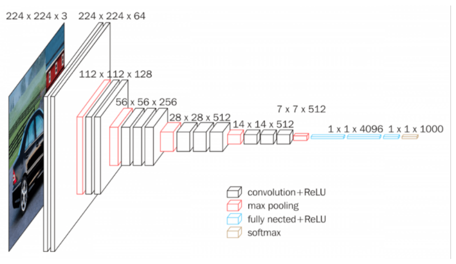
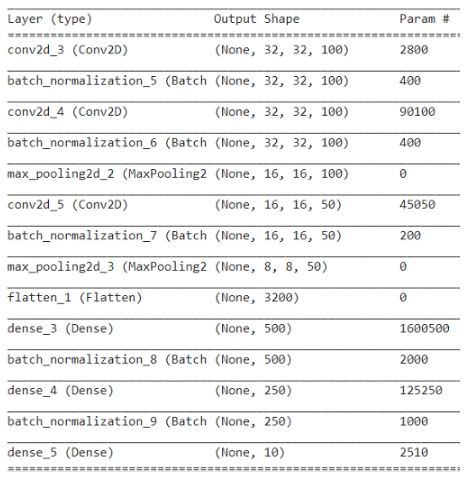


Figure 2: The architecture diagram of the VGG16 network



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 32, 32, 100)       2800
_________________________________________________________________
batch_normalization_5 (Batch (None, 32, 32, 100)       400
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 32, 32, 100)       90100
_________________________________________________________________
batch_normalization_6 (Batch (None, 32, 32, 100)       400
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 100)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 50)        45050
_________________________________________________________________
batch_normalization_7 (Batch (None, 16, 16, 50)        200
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 8, 8, 50)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3200)              0
_________________________________________________________________
dense_3 (Dense)              (None, 500)               1600500
_________________________________________________________________
batch_normalization_8 (Batch (None, 500)               2000
_________________________________________________________________
dense_4 (Dense)              (None, 250)               125250
_________________________________________________________________
batch_normalization_9 (Batch (None, 250)               1000
_________________________________________________________________
dense_5 (Dense)              (None, 10)                2510
=================================================================
```

Figure 3: The architecture of RegularCNN

4.3 Testing

For each benchmark, we use 50 random images from CIFAR10 that the benchmark model successfully classifies. Each method is enclosed in a loop to keep perturbing the selected image iteratively until an adversarial image is successfully generated. This loop also performs the function GENERATE_ADVERSARIAL for fuzzing methods in the pseudo code. We evaluate these methods' ability to attack complex models, generate complex inputs, and transferable adversarial attacks in terms of three metrics: average distance between the original image and adversarial image, average runtime, and transfer rate.

Transfer rate estimation: We donate X to the set of original images that are successfully classified by both models A, B; X' be the set of adversarial examples of X generated by method M on A. The transfer rate of M from A to B is estimated by:

$$transfer\_rate = \frac{count(x' \in X', s.t. B\ misclassifies\ x')}{count(x \in X)}$$

4.4 Results and Discussion

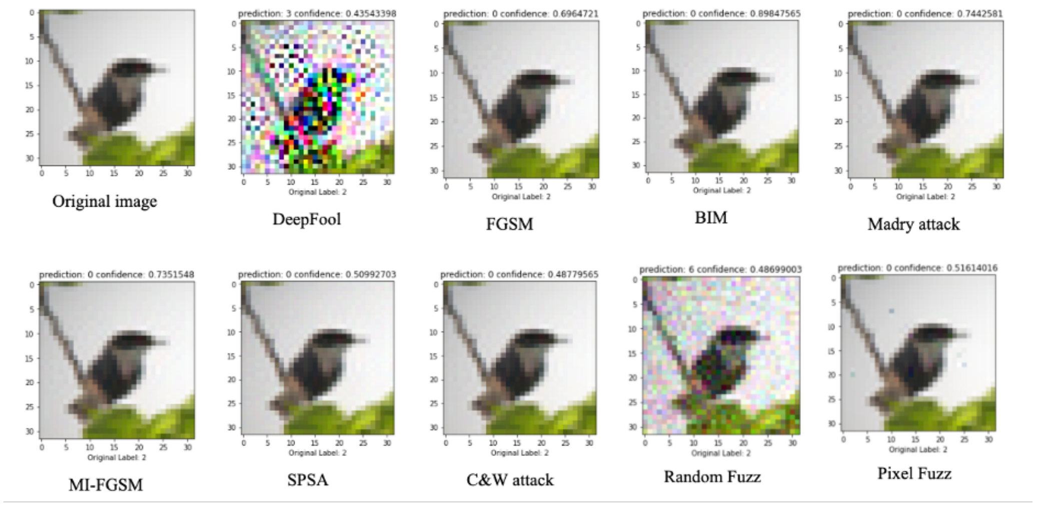

Original image
prediction: 3 confidence: 0.43543398
DeepFool
prediction: 0 confidence: 0.6964721
FGSM
prediction: 0 confidence: 0.89847565
BIM
prediction: 0 confidence: 0.7442581
Madry attack
prediction: 0 confidence: 0.7351548
MI-FGSM
prediction: 0 confidence: 0.50992703
SPSA
prediction: 0 confidence: 0.48779565
C&W attack
prediction: 6 confidence: 0.48699003
Random Fuzz
prediction: 0 confidence: 0.51614016
Pixel Fuzz
Original Label: 2

Figure 4: example of adversarial images generated by each method

	Average Distance to the original image(L2 norm)			Average Runtime(s)			Transfer Rate		
	VGG16	Robust VGG16	Regular CNN	VGG16	Robust VGG16	Regular CNN	VGG16	Robust VGG16	Regular CNN
DeepFool	6.0599	6.1110	10.4057	0.3250	0.3543	0.1138	0.5897	0.5000	0.4783
FGSM	0.2779	0.4837	0.2473	0.03969	0.0631	0.0153	0.1250	0.2973	0.0714
BIM	0.2296	0.4399	0.2149	0.4138	0.7784	0.1744	0.1500	0.2703	0.0714
Madry	0.2614	0.4645	0.2482	0.4663	0.9728	0.1997	0.0750	0.1351	0.0714
MIM	0.3334	0.5062	0.3210	0.0966	0.1469	0.0394	0.2000	0.3784	0.1905
SPSA	0.3207	0.5652	0.2805	1.8856	4.3928	1.1109	0.0750	0.2162	0.0714
C&W	0.1565	0.3762	0.1322	41.6784	28.7869	23.3838	0.0500	0.1622	0.0476
Random Fuzz	4.3418	6.2772	4.1050	3.2425	5.0222	1.6238	0.7000	0.8108	0.3333
Pixel Fuzz	0.3558	0.7505	0.3124	1.1391	2.7213	0.6428	0.1000	0.2162	0.0714

Table 3: The average runtime, average distance, and transfer rate of VGG16, robustVGG16, regularCNN

Figure 4 shows a sample of adversarial images generated by the selected methods on the VGG16 benchmark. Table 2 shows the recorded distance, runtime, and transfer rate of VGG16, robustVGG16, regularCNN. Data for the other three simpleCNN benchmarks are mainly used for scalability analysis and not shown in this table. They are available in the ipython notebook files for testing.

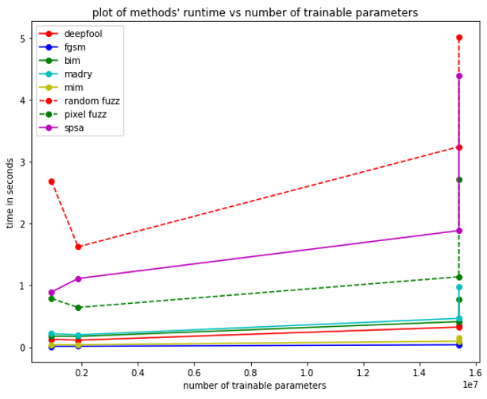
Overall, C&W's method obtained the best distance in all 3 benchmarks, at a cost of extremely high runtime. Random Fuzz does not work well for generating adversarial examples in both metrics of distance and runtime as expected. Surprisingly, DeepFool also works poorly in the metric of distance. In our definition of distance, distance greater than 1.5 implies that the perturbation is human recognizable. Hence, we conclude that DeepFool and Random Fuzz with distance greater than 4 fail to generate the adversarial images and will be excluded in further discussion. Figure 3 shows that significant noise is observed in the adversarial image generated by Deepfool and Random Fuzz.

It's also noticeable that the gradient-based methods has similar performances. Comparing the gradient-based methods, BIM obtained the best distance and FGSM obtained the best runtime. BIM is an iterative version of FGSM, which implies BIM creates a more accurate adversarial image at a higher cost. Performances of MIM and madry lie in between BIM and FGSM.

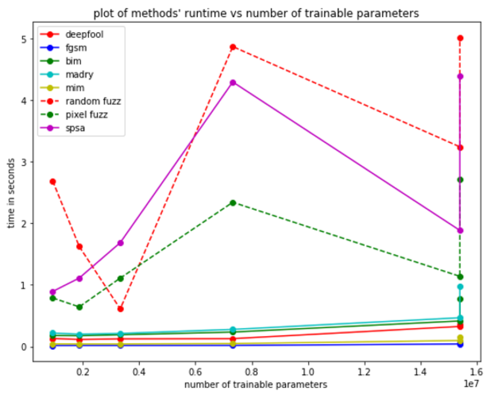
The runtime and distance performance of all attack methods are significantly worse on the robustVGG16 benchmark compared to the VGG16 benchmark. This shows that adversarial defense methods such as adversarial training improved the robustness of the model significantly. Although the robustVGG16 is trained with only the adversarial images generated by FGSM, the model gains robustness against all types of adversarial attacks including SPSA, C&W, Random Fuzz which are non-gradient based methods. It's concluded that the VGG16 model after adversarial training is more robust against small perturbations in general.

We also observe that transfer rate has a positive correlation with distance. For example, C&W has the lowest distance and also has the lowest transfer rate. It's noticeable that MIM has the highest transfer rate, but medium distance. This fact indicates that MIM has a strong ability to generate transferable adversarial examples. The results also show that adversarial images generated on defended networks (robustVGG16) have better transferability compared to those generated on undefended networks (VGG16 & regularCNN).

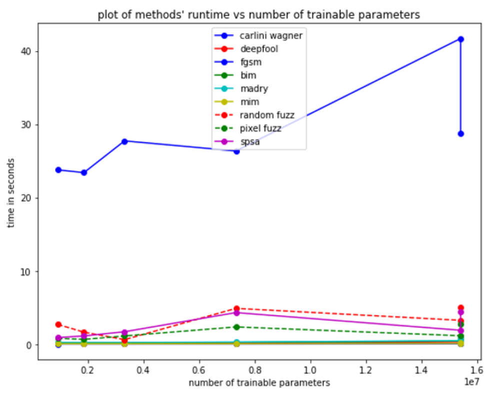
Pixel Fuzz that we proposed, which uses gradients to rank and select pixels to mutate, has achieved a medium performance on distance and runtime but only changes a few pixels.



(a)



(b)



(c)

Figure 5: Results of the trainable parameters vs runtime

Fig. 5 (a) shows results of the trainable parameters vs runtime for 4 different benchmarks: on the x label, from left to right points, there are simpleCNN, regularCNN, VGG16, robustVGG16 respectively. These benchmarks have the same input size and the increasing number of trainable parameters in the model. The figures for simpleCNN2, simpleCNN3 are included in Fig 5(b), since simpleCNN2 and simpleCNN3 have different input sizes of $64 \times 64 \times 3$ and $96 \times 96 \times 3$ respectively. On the x label, from left to right points, there are simpleCNN, regularCNN, simpleCNN2, simpleCNN3, VGG16, robustVGG16 respectively. Since the runtime of C&W method has relatively large magnitude, it is only shown in Fig 5(c) for a better observation of the other methods' runtimes.

Fig 5 indicates that in general, with the increasing number of trainable parameters, the runtime for the methods to generate adversarial examples increases. We proposed 2 possible explanations. One is that the methods need more time to analyze the parameters. The other explanation is that models with more parameters usually have higher accuracy (fewer edge cases & less weakness), therefore harder to confuse. However, the latter explanation is rejected due to the fact that simpleCNN3 is weaker than CNN, has more trainable parameters, and the runtime is longer for all methods.

The gap in runtime (the vertical line) between VGG16 and robust VGG16 demonstrates the impact of adversarial training. It's also observed that runtime of Pixel Fuzz, Random Fuzz, and SPSA is also sensitive to the number of inputs in addition to the number of parameters. This is also shown in Fig 6.

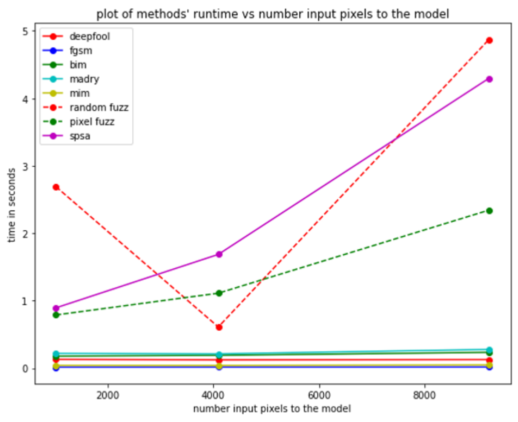
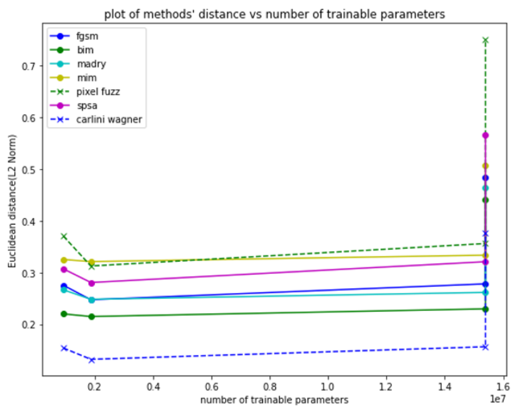


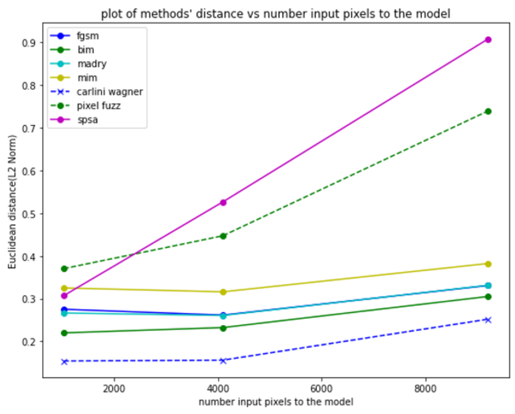
Figure 6: Results of the input pixels vs runtime

Fig 6 demonstrates the relation between runtime and the number of input pixels for benchmarks: simpleCNN, simpleCNN2, simpleCNN3. They are situated at the x label from left to right points, with an increasing number of pixels. It shows that Pixel Fuzz, Random Fuzz, and SPSA are extremely sensitive to the dimension of the input image, which affects their scalability to attack networks that classify complex inputs.

For Pixel Fuzz, this scalability problem can be avoided by increasing the number of pixels changed during each iteration. As more pixels are changed each iteration, fewer iterations are required and thus the total runtime is reduced.



(a)



(b)

Figure 7: Results for Distance metric

Fig 7(a) shows results of the trainable parameters vs distance for 4 different benchmarks: on the x label, from left to right points, there are simpleCNN, regularCNN, VGG16, robustVGG16 respectively. These benchmarks have the same input size and an increasing number of trainable parameters in the model. Fig 7(b) indicates the relation between distance and the number of input pixels for benchmarks simpleCNN, simpleCNN2, simpleCNN3. They are situated at the x label from left to right points, with an increasing number of pixels. Note that Deep Fool and Random Fuzz are excluded from the distance plot due to their scale.

From Fig 7, we observe that there is no strong relationship between distance and the number of trainable parameters, however, the adversarial training significantly weakened the method's ability to generate adversarial images. C&W's method obtained the lowest distance on all benchmarks compared to other methods, while SPSA and Pixel Fuzz are sensitive to the size of the input image.

In summary, the gradient-based methods have better scaling ability compared to the other methods. Traditional fuzzing techniques have worse performance than methods designed for testing neural networks.

5 Conclusion

The results of this project show that more runtime is required to generate adversarial attacks for more complex systems, since the number of computed parameters and classification power of the models increase. The methods require a longer time to find adversarial examples to attack defended systems and the adversarial examples generated are farther away from the original sample. We also show that adversarial training can increase the robustness of the model against all multiple types of adversarial attacks. The transfer rate has a positive relationship with distance and is generally higher on defended models. Through comparison, C&W has the lowest distance and transfer rate, while DeepFool works poorly in the metric of distance, and MIM has the highest transfer rate. Traditional Random Fuzzer fails to generate effective attacks even to undefended neural networks. Pixel Fuzz that we proposed, which uses gradients to rank and select pixels to mutate, has achieved a medium performance but only changes a few pixels. Future works can be done focusing on better fuzzing techniques to generate adversarial images and how to generate adversaries that are highly transferable.

References

[1] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013. [Online]. Available: https://arxiv.org/abs/1312.6199

[2] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," 2014. [Online]. Available: https://arxiv.org/abs/1412.6572

[3] R. R. Wiyatno, A. Xu, O. Dia, and A. de Berker, “Adversarial Examples in Modern Machine Learning: A Review,” 2019. [Online]. Available: https://arxiv.org/abs/1911.05268

[4] J. Zhang and C. Li, “Adversarial Examples: Opportunities and Challenges,” *IEEE transaction on neural networks and learning systems*, vol. 31, no. 7, pp. 2578–2593, 2020, publisher: IEEE.

[5] A. Kurakin, I. Goodfellow, and S. Bengio, “Adversarial examples in the physical world,” 2016. [Online]. Available: https://arxiv.org/abs/1607.02533

[6] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, “Practical Black-Box Attacks against Machine Learning,” 2016. [Online]. Available: https://arxiv.org/abs/1602.02697

[7] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, “DLFuzz: differential fuzzing testing of deep learning systems,” ser. ESEC/FSE 2018. ACM, 2018, pp. 739–743.

[8] P. Zhang, Q. Dai, and P. Pelliccione, “CAGFuzz: Coverage-Guided Adversarial Generative Fuzzing Testing of Deep Learning Systems,” 2019. [Online]. Available: https://arxiv.org/abs/1911.07931

[9] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, “DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks.” IEEE, 2016, pp. 2574–2582, iSSN: 1063-6919.

[10] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, “Towards Deep Learning Models Resistant to Adversarial Attacks,” 2017. [Online]. Available: https://arxiv.org/abs/1706.06083

[11] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, “Boosting Adversarial Attacks with Momentum,” 2017. [Online]. Available: https://arxiv.org/abs/1710.06081

[12] J. C. Spall, “Multivariate stochastic approximation using a simultaneous perturbation gradient approximation,” *IEEE transactions on automatic control*, vol. 37, no. 3, pp. 332–341, 1992, place: NEW YORK Publisher: IEEE.

[13] J. Uesato, B. O’Donoghue, A. v. d. Oord, and P. Kohli, “Adversarial Risk and the Dangers of Evaluating Against Weak Attacks,” 2018. [Online]. Available: https://arxiv.org/abs/1802.05666

[14] N. Carlini and D. Wagner, “Towards Evaluating the Robustness of Neural Networks.” IEEE, 2017, pp. 39–57, iSSN: 2375-1207.

[15] MyRespect, “AdversarialAttack,” May 2021, original-date: 2019-07-02T13:25:18Z. [Online]. Available: https://github.com/MyRespect/AdversarialAttack

[16] “CleverHans (latest release: v4.0.0),” Aug. 2021, original-date: 2016-09-15T00:28:04Z. [Online]. Available: https://github.com/cleverhans-lab/cleverhans

[17] “CIFAR-10 and CIFAR-100 datasets.” [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[18] “VGG16 - Convolutional Network for Classification and Detection,” Nov. 2018. [Online]. Available: https://neurohive.io/en/popular-networks/vgg16/