Iterative, Gradient-Based Adversarial Attacks on Neural Network Image Classifiers

Toomas Liiv and Axel Strömberg

***Abstract*—Deep neural networks (DNN) are used in a wide range of tasks such as voice recognition, image classification and spam-mail detection. However, it has been shown that the networks can misclassify an input when a small, carefully chosen perturbation is added to it. Many adversarial machine learning attacks have been proposed to create such samples, often with the goal of finding the smallest possible perturbation. In this study, three DNN image classification algorithms are constructed. Two adversarial methods (IFGM and DeepFool) are then analyzed and compared in terms of finding the smallest perturbation to cause misclassification of the networks, with the least computational effort. The two iterative, gradient-based methods are implemented in four distance metrics: L_0, L_1, L_2, L_∞. The surprising result is that even though DeepFool uses a more sophisticated optimization strategy it does not perform significantly better than IFGM. Furthermore, IFGM actually finds a smaller perturbation with the same amount of time given. Also, in the targeted regime, IFGM performs better than DeepFool. Lastly, a fast L_0-attack is suggested that strives to perturb as few pixels as possible to cause misclassification.**

***Index Terms*—Adversarial machine learning, neural networks, image recognition, DeepFool, FGSM, IFGM**

TRITA number: TRITA-EECS-EX-2019:135

I. Introduction

Deep neural networks can be used to teach a computer to perform tasks previously only solvable by humans, such as speech recognition [1], image recognition [2], and spam detection [3]. Even though neural networks can be trained to outperform human achievements, it has been shown that they are vulnerable to adversarial samples [4]–[6]. Furthermore, it has been shown that these adversarial examples are transferable between different neural networks trained on the same task and even transferable to real-world examples [7]. Due to the wide usage of neural networks, it is of uttermost importance to thoroughly examine them in adversarial environments, especially for security-critical cases. For example, a stop sign can be altered with colored tape to be identified as a speed limit sign for a wide range of image classifiers that may be used in autonomous driving cars [8]. Furthermore, a few words in a spam mail could be altered so that it is classified as a genuine email.

By studying the properties of the algorithms that produce adversarial samples, possible defenses can be deduced, for example, by training on the adversarial samples themselves. This may even lead to even better classifiers in the non-adversarial setting, and be seen as a way to augment and expand the training data.

For many adversarial methods, the goal is to find the smallest possible perturbation to cause misclassification. They can roughly be divided into optimization-based approaches [4], [9], or gradient-based approaches, which is the focus of this study. Two such examples are *IFGM* [10], a method utilizing the gradient of the cost function, and *DeepFool* [6], a method utilizing the gradient of the classifier. In the DeepFool-article, it is compared to a single step version of IFGM. Thus, the goal of this project is to do a fairer comparison between iterative versions of both, and explore related aspects of the attacks. Our contributions are in short

- We compare DeepFool and IFGM in a juster way.
- We investigate the effect of clipping for DeepFool.
- We propose a step length and normalization for the otherwise hyperparameterless DeepFool.
- We propose a way to make DeepFool and IFGM converge in L_1- and L_0-norm.
- We compare a targeted version of DeepFool with a targeted version of IFGM.

Some examples of the adversarial attacks using the IFGM and DeepFool methods are shown in Fig. 1.

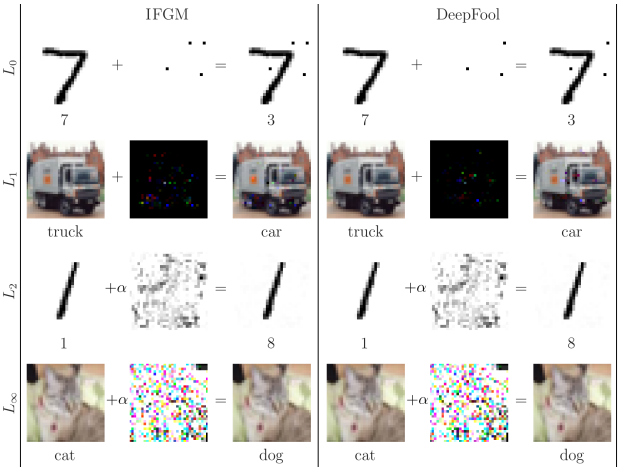
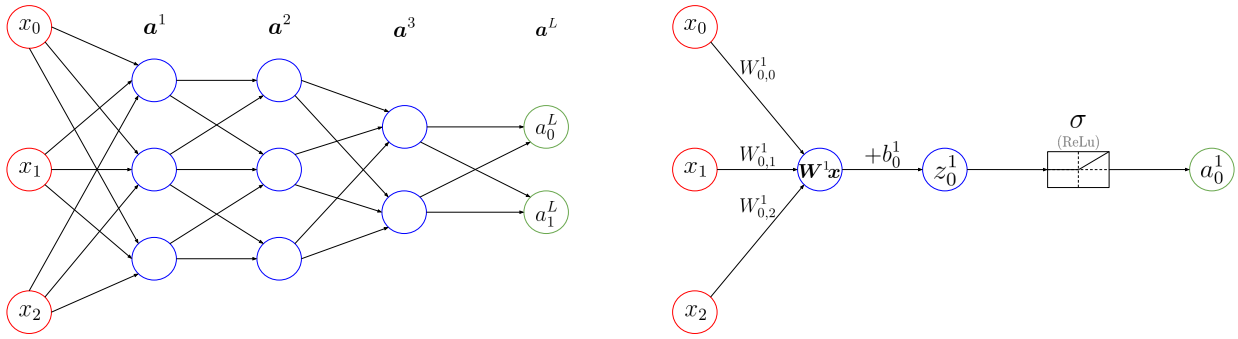


Fig. 1. IFGM and DeepFool attacks on the DNN-MNIST classifier and CNN-CIFAR-10 classifier. The images are chosen so that the required perturbations are similar to the average perturbation required on the entire dataset. The L_0-methods strive to change the least amount of pixels, the L_1-methods find the smallest perturbed sum, the L_2-methods find the smallest Euclidean distance and the L_∞ strives to have the smallest perturbation of the highest altered pixel. Interestingly, even though DeepFool uses a conventionally better optimization method, both methods perform similarly with the same iterative step length. Furthermore, the IFGM method is significantly faster which allows it to compute smaller perturbations in the same amount of time.

II. Neural networks

Artificial neural networks are a biologically-inspired method of training a computer to perform tasks given observational



(a) Schematic over a neural network with a three-dimensional input layer, three hidden layers and a two-dimensional output layer.

(b) Detailed structure of a neural network with a three dimensional input, no hidden layers and a one-dimensional output.

Fig. 2. The schematic of a simple neural network in (a) shows how the neurons are connected in a deep neural network. Even though the complexity may seem daunting when the input dimension goes up to $28 \times 28 = 784$ for the simple MNIST dataset the fundamental operations are the same as for the simple neural network in (b). Firstly, a matrix operation $\boldsymbol{W}$ is applied to the previous layer and a bias vector $\boldsymbol{b}$ is added to the result which gives $\boldsymbol{z}$. Then, the activation function σ is applied to $\boldsymbol{z}$ to give the next layer $\boldsymbol{a}$.

data. A traditional neural network consists of an input layer, several hidden layers, an output layer and is trained on several thousand training samples. In this study, several image classifiers are trained. Here, the input layer takes the raw pixel data, forwards it through several hidden layers and then lastly to the output layer. The n^{th} output neuron is a value from zero to one that corresponds to the network's confidence that the input data should be classified with this label.

A. Deep neural networks

The human vision is a complex, incredible accomplishment of nature. Even though a human easily and instantly recognizes common objects such as numbers or animals, telling a computer to the same is next to impossible. This motivates modern software to mimic the way the human vision and categorization works with artificial neural networks [11].

Fig. 2 shows a simple schematic over a neural network. When this network has a high number of hidden layers it is called a *deep neural network* (DNN). A deep neural network image classifier takes the *input layer* as the raw image pixel values flattened into a vector $\boldsymbol{x}$. With the MNIST dataset [12] classifier, this layer corresponds to a $28 \times 28 = 784$ dimensional vector with inputs between zero and one. This information is transferred to the first hidden layer, $\boldsymbol{a}^1$, with the weight matrix $\boldsymbol{W}^1$, bias vector $\boldsymbol{b}^1$ and activation function σ according to:

$$\boldsymbol{a}^1 = \sigma(\boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1)\,. \tag{1}$$

The σ-function could be one of many conventional activation functions such as:

$$\begin{array}{r|l} \text{ReLu} & [\sigma(\boldsymbol{x})]_i = \max(0, x_i) \\ \text{Sigmoid} & [\sigma(\boldsymbol{x})]_i = \dfrac{1}{1+e^{-x_i}} \\ \text{(or) Softmax} & [\sigma(\boldsymbol{x})]_i = \dfrac{e^{x_i}}{\sum_j e^{x_j}}\,. \end{array} \tag{2}$$

The information is then transferred through all hidden layers $n \in \{1, \ldots, N\}$ according to

$$\boldsymbol{a}^n = \sigma(\boldsymbol{W}^n\boldsymbol{a}_{n-1} + \boldsymbol{b}^n) \tag{3}$$

and finally to the output layer with

$$\boldsymbol{a}^L = \sigma(\boldsymbol{W}^L\boldsymbol{a}^{L-1} + \boldsymbol{b}_L)\,. \tag{4}$$

Lastly, the class, y, is determined from the output layer as an integer

$$y = \arg\max_i a_i^L\,. \tag{5}$$

Often the entire algorithm is notationally reduced to a single function f or $\hat{k}$:

$$\boldsymbol{a}^L = f(\boldsymbol{x})\,, \tag{6}$$

$$y = \hat{k}(\boldsymbol{x}) = \arg\max_k f_k(\boldsymbol{x})\,. \tag{7}$$

Here $f(\boldsymbol{x})$ gives the output of the final layer given the input $\boldsymbol{x}$. $\hat{k}(\boldsymbol{x})$ returns the index of the final layer neuron with the highest value, that is, the label given to the image by the network.

To create a good classifier one must choose the right architecture and train it on a large dataset. The right architecture consists of choosing the density of hidden layers and the right activation functions. The weights and biases are initially random numbers and the network is simply guessing. However, the network is then trained using gradient descent. To initialize the training all weights and biases are joined into a single vector $\boldsymbol{\omega}$:

$$\boldsymbol{\omega} = \left(W_{0,0}^0, W_{0,1}^0, \ldots, b_{n_L-1}^L, b_{n_L}^L\right)\,.$$

The goal now is to derive how $\boldsymbol{\omega}$ should be altered to improve the accuracy. This is done by introducing a cost function that measures how poorly the algorithm currently performs at classifying. A simple cost function is the summed square of the error. Let $\boldsymbol{y}(\boldsymbol{x})$ be the correct output of the network,

$$\boldsymbol{y}(\boldsymbol{x}) = (0, \ldots, \underbrace{1}_{\text{index = correct label}}, \ldots, 0)\,,$$

and $\boldsymbol{a}^L$ be the current output of the system. The square error cost function is then

$$C = \frac{1}{2n}\sum_{\boldsymbol{x}} ||\boldsymbol{y}(\boldsymbol{x}) - \boldsymbol{a}^L(\boldsymbol{x})||^2 \tag{8}$$

where $\boldsymbol{x}$ is summed over a set of n training data-label sets. However, for many cases, other cost functions perform better, and one of the better ones is the *cross-entropy cost function*:

$$C = -\frac{1}{n}\sum_{\boldsymbol{x}}\sum_{i}\left[y_i \ln a_i^L(\boldsymbol{x}) + (1-y_i)\ln\left(1-a_i^L(\boldsymbol{x})\right)\right]. \quad (9)$$

Thus the goal of constructing a good classifier is reduced to minimizing the cost function. This is performed with *gradient descent*. The gradient of the cost function with respect to all weights and biases,

$$\nabla_{\boldsymbol{\omega}} C(\boldsymbol{x},\boldsymbol{\omega}) = \left(\frac{\partial C}{\partial W^0_{0,0}}, \frac{\partial C}{\partial W^0_{0,1}}, \ldots \right. \\ \left. \ldots, \frac{\partial C}{\partial b^L_{n_L-1}}, \frac{\partial C}{\partial b^L_{n_L}}\right), \quad (10)$$

is computed for a batch of n images. The gradients are averaged and then the weights and biases are altered according to

$$\boldsymbol{\omega} = \boldsymbol{\omega} - \frac{\eta}{N}\sum_{n=1}^{N}\nabla_{\boldsymbol{\omega}} C(\boldsymbol{x}_n,\boldsymbol{\omega}). \quad (11)$$

η is a scalar that affects the speed and accuracy of the training. Often the training set is randomly grouped into *batches* and (11) is repeated for each batch. The entire dataset is repeatedly trained on in a number of *epochs*.

The gradient is computed using the four *backpropagation equations*:

$$\boldsymbol{\delta}^L = \nabla_{\boldsymbol{a}^L} C \odot \sigma'(\boldsymbol{z}^L) \quad \text{(BP1)}$$

$$\boldsymbol{\delta}^l = \left(\left(\boldsymbol{W}^{l+1}\right)^T \boldsymbol{\delta}^{l+1}\right) \odot \sigma'(\boldsymbol{z}^l) \quad \text{(BP2)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \quad \text{(BP3)}$$

$$\frac{\partial C}{\partial W^l_{j,k}} = a^{l-1}_k \delta^l_j. \quad \text{(BP4)}$$

Here, $\odot$ is the *Hadamard product* which denotes the elementwise product of two vectors

$$[\boldsymbol{v}\odot\boldsymbol{u}]_i = v_i u_i$$

and $\boldsymbol{z}^l$ corresponds to $\boldsymbol{a}^l$ in (3) so that

$$\boldsymbol{a}^l = \sigma(\boldsymbol{z}^l).$$

The proof of the backpropagation equations is made using the chain rule and is available at [11, chap. 2].

B. Convolutional neural networks

A simple DNN has two fundamental issues: overfitting and dimensionality reduction. Overfitting occurs when a complex model adapts too well to the training data, and does not generalize to the test data. The dimensionality problem is due to that the two-dimensional image is reduced to a one-dimensional array and thus the network cannot identify shapes such as edges and corners, only individual pixels. To further improve the efficacy of a DNN one can implement it as a *convolutional neural network* (CNN). Again the idea is to mimic the way the human vision works and the two-dimensionality of the image is preserved in the network with convolutional layers. This helps the network create local perceptive fields and thus identify the fundamental features of the image.

A *convolutional layer* [13] has three dimensions: height, width and depth. The height and depth conserve the dimensions in the image; a neuron in a convolutional layer is only dependent on nearby neurons in the previous layer. In this way, the number of weights in the network is reduced which will reduce overfitting. The depths correspond to different filters that the network produces during training. Each filter may identify a specific feature of the image such as a horizontal edge, vertical edge or corner of a digit.

For the network to identify for example an eight it only needs to recognize two circles in the image and the locations are irrelevant, it can actually be harmful to know where the circles are since different hand-drawn eights can have circles at different locations. The accuracy of the filters thus needs to be reduced and this is done with a *sub-sampling* filter which will be implemented as a *max pooling* later in this study.

C. Architecture used in the study

During the study, six neural networks were trained. DNNs and CNNs were trained on the MNIST, fashion MNIST and CIFAR-10 datasets. It was then decided to discard the fashion MNIST networks, not because the results were uninteresting but rather that the complexity of the fashion MNIST data is a middle ground between MNIST and CIFAR-10 and it was more interesting analyzing the attack on the two extremes. Furthermore, the DNN classifier trained on CIFAR-10 was discarded since it performed very poorly with an accuracy of only 46.31 % on the test data.

Hence, adversarial attacks were performed on three classifiers: a DNN trained on MNIST described in Table I, a CNN trained on MNIST described in Table II and a CNN trained on CIFAR-10 described in Table III. A better CNN can be trained on CIFAR-10, for example, a *densely connected convolutional network* can be trained with an error rate of 5.19 % [2]. However, it was decided to use a simpler architecture of a convolutional network used in the original defensive distillation work [14] and further the paper describing the Carlini-Wagner method [9].

III. Adversarial samples

Given an image classifier, it was shown in 2014 that a correctly classified image could be changed in an almost imperceivable way to cause misclassification [4]. Such an image is called an *adversarial sample*. Since then many methods have been proposed to compute such samples. Initially, only *white-box attacks* were performed when the architecture and all weights were known. Then it was shown that adversarial samples are somewhat consistent between methods trained on the same task [7]. Thus *black-box attacks* could be performed when the architecture and weights were unknown. Here, a substitute network is trained to behave like the original and the unknown decision boundaries are approximated as the boundaries of the substitute network [15].

TABLE I
DNN CLASSIFIER ON THE MNIST DATASET. 99.62 % ACCURACY ON TRAINING DATA. 96.95 % ACCURACY ON TEST DATA. 28 618 TRAINABLE PARAMETERS.

Layer type (Activation)	Output shape
Flatten (None)	$28 \times 28 = 784$
Dense (ReLu)	32
Dense (ReLu)	32
Dense (ReLu)	32
Dense (ReLu)	32
Dense (Softmax)	10

TABLE II
CNN CLASSIFIER ON THE MNIST DATASET. 99.94 % ACCURACY ON TRAINING DATA. 99.36 % ACCURACY ON TEST DATA. 312 202 TRAINABLE PARAMETERS.

Layer type (Activation)	Output shape
Convolutional (ReLu)	$26 \times 26 \times 32$
Convolutional (ReLu)	$24 \times 24 \times 32$
Max pooling (None)	$12 \times 12 \times 32$
Convolutional (ReLu)	$10 \times 10 \times 64$
Convolutional (ReLu)	$8 \times 8 \times 64$
Max pooling (None)	$4 \times 4 \times 64$
Flatten (None)	1024
Dense (ReLu)	200
Dropout (None)	200
Dense (ReLu)	200
Dense (Softmax)	10

TABLE III
CNN CLASSIFIER ON THE CIFAR-10 DATASET. 98.96 % ACCURACY ON TRAINING DATA. 75.37 % ACCURACY ON TEST DATA. 1 147 978 TRAINABLE PARAMETERS.

Layer type (Activation)	Output shape
Convolutional (ReLu)	$30 \times 30 \times 64$
Convolutional (ReLu)	$28 \times 28 \times 64$
Max pooling (None)	$14 \times 14 \times 64$
Convolutional (ReLu)	$12 \times 12 \times 128$
Convolutional (ReLu)	$10 \times 10 \times 128$
Max pooling (None)	$5 \times 5 \times 128$
Flatten (None)	3200
Dense (ReLu)	256
Dropout (None)	256
Dense (ReLu)	256
Dense (Softmax)	10

The goal of many methods is to alter the image as little as possible. The perturbation to misclassification is conventionally measured in different *distance metrics* [9]. Here the L_p distance between original, $\boldsymbol{x}$, and adversarial image, $\boldsymbol{x}^*$ is measured. The distance is denoted $||\boldsymbol{x}-\boldsymbol{x}^*||_p$ with the p-norm defined as

$$||\boldsymbol{v}||_p = \left(\sum_i |v_i|^p \right)^{\frac{1}{p}} . \tag{12}$$

In this study, two methods are examined in four norms illustrated in Fig. 3.

L_0: The L_0 distance is not strictly a p-norm since $\lim_{p\to 0} \frac{1}{p} = \infty$. However, since

$$\lim_{p\to 0} |v_i|^p = \begin{cases} 0, & \forall v_i = 0 \\ 1, & \forall v_i \neq 0 \end{cases},$$

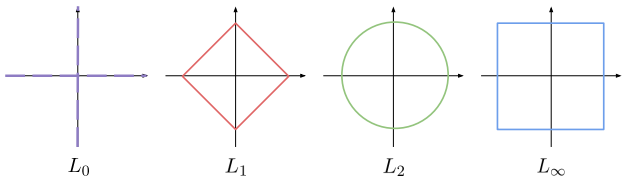


Fig. 3. Unit circles in the four norms L_0, L_1, L_2 and L_∞. All axes have equal scaling and all unit circles have the same radius. Since the L_0-norm only measures the number of nonzero elements, all points on a single axis have the same L_0-norm of 1.

$||\boldsymbol{x} - \boldsymbol{x}^*||_0$ is conventionally used to measure the number of altered pixels, regardless of the perturbed amount:

$$||\boldsymbol{x} - \boldsymbol{x}^*||_0 = \sum_i (x_i - x_i^*)^0$$
$$\text{where } 0^0 = 0 .$$

This norm is useful for adversarial machine learning outside the realm of image classification. For example, in a spam mail as few words as possible should be altered so that it is classified as genuine email.

L_1: The L_1-distance is the sum of all perturbations:

$$||\boldsymbol{x} - \boldsymbol{x}^*||_1 = \sum_i |x_i - x_i^*| .$$

L_2: The L_2-distance measures the standard *Euclidean norm*:

$$||\boldsymbol{x} - \boldsymbol{x}^*||_2 = \sqrt{\sum_i (x_i - x_i^*)^2} .$$

L_∞: Let a be the index of the largest absolute value of $\boldsymbol{v}$ in (12). When $p \to \infty$, $|v_a|^\infty$ will dominate over all other indexes $|v_i|^\infty, i \neq a$. The exponentiation $(\cdot)^{1/p}$ then returns $|v_a|$. Thus the L_∞-distance measures the largest change of a single pixel, regardless of the changes of all other pixels:

$$||\boldsymbol{x} - \boldsymbol{x}^*||_\infty = \max\left(|x_0 - x_0^*|, \ldots, |x_n - x_n^*|\right) .$$

Next, a *threat model* is to be defined. In the original FGSM study [5] and further studies using the basic iterative method [10], the goal is to perturb a set of images so that

$$||\boldsymbol{x} - \boldsymbol{x}^*||_\infty \leq \epsilon$$

for each image. With a small ϵ the change in the image will thus not be observable to a human. Then the error rate of the image classifier is measured on the adversarial set, with a high error rate implying a good adversarial method. The DeepFool study, [6], uses a different threat model. Here the goal is to approximate the smallest possible perturbation to cause misclassification:

$$\begin{aligned} \text{minimize} \quad & ||\boldsymbol{x} - \boldsymbol{x}^*||_2^2 \\ \text{such that} \quad & \hat{k}(\boldsymbol{x}^*) \neq \hat{k}(\boldsymbol{x}) . \end{aligned}$$

DeepFool, therefore, has near 100 % success rate since the iterative method is repeated until misclassification. In this study, a threat model similar to DeepFool is used. However, it is extended to be well defined in any of the four norms. For a

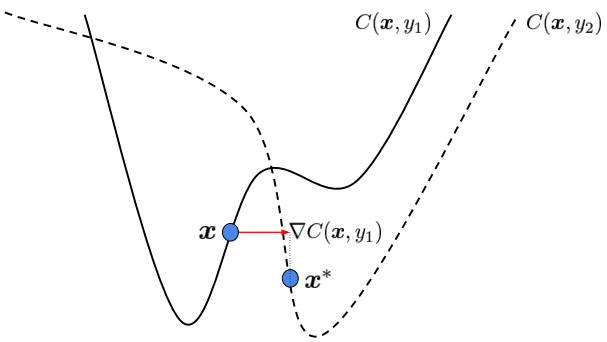


Fig. 4. Illustration of the FGSM attack. Let $\boldsymbol{x}$ be the input image and $y_1 = \hat{k}(\boldsymbol{x})$ be the label given to it by the network. The attack computes $\nabla C(\boldsymbol{x}, y_1)$ that indicates the direction that the image should be perturbed in, in order to increase the cost when the image is classified as y_1. With a sufficiently high cost, the network will give the image a cheaper label. Thus, the adversarial image $\boldsymbol{x}^*$ is classified as y_2 by the network.

given norm p, classifier $\hat{k} = \arg\max_i f_i$ as (6)-(7) and image $\boldsymbol{x}$ the goal of the two adversarial methods is to:

$$\begin{aligned} \text{minimize} \quad & ||\boldsymbol{x} - \boldsymbol{x}^*||_p \\ \text{such that} \quad & \hat{k}(\boldsymbol{x}^*) \neq \hat{k}(\boldsymbol{x}) \\ & \boldsymbol{x}^* \in [0,1]^n. \end{aligned} \tag{13}$$

The third row implies that the new image must have pixel values between zero and one. This is important since otherwise an adversarial image can easily be detected by a preprocessing stage, before even reaching the classifying network.

To compare different attacks, an appropriate metric needs to be defined. A commonly used one [6] is average *robustness*

$$\hat{\rho}_{adv}(f) = \frac{1}{|\mathfrak{D}|} \sum_{\boldsymbol{x} \in \mathfrak{D}} \frac{||\boldsymbol{x} - \boldsymbol{x}^*||_p}{||\boldsymbol{x}||_p} \tag{14}$$

where $\mathfrak{D}$ is the test dataset. However, for L_0, the normalization is often left out and only the number of changed pixels reported. Other relevant parameters are fooling rate, number of iterations until misclassification and running time until misclassification.

A. FGSM

The *iterative fast gradient method* (IFGM) accomplishes its attack with a method similar to when the neural network is trained with gradient descent (11). The foundation of the method, then called *fast gradient sign method* (FGSM), was first introduced with L_∞-distance metric and a single step in 2015 by I. Goodfellow et al [5]:

$$\boldsymbol{x}^* = \boldsymbol{x} + \epsilon \text{sign}\big[\nabla_{\boldsymbol{x}}\big(C(\boldsymbol{\omega}, \boldsymbol{x}, y)\big)\big] \,,$$

where the $C(\boldsymbol{\omega}, \boldsymbol{x}, y)$ is the cost of the current classification of $\boldsymbol{x}$ with the true label y and using the weights $\boldsymbol{\omega}$ in the network. The sign function returns the vector were all positive values are set to 1 and all negative values are set to -1. Since the weights are constant during the method $\boldsymbol{\omega}$ will from now on be left out. Furthermore, y will be changed to $y = \hat{k}(\boldsymbol{x})$. That is, y is the current label given to $\boldsymbol{x}$ by the network f despite the true label. The motivation behind the method is to increase the cost when classifying $\boldsymbol{x}^*$ as $y = \hat{k}(\boldsymbol{x})$ so that the network will simply give it another label with lower cost. An illustration of this is shown in Fig. 4. This was later implemented as the *basic iterative method* (BIM) [10] with clipping between each iteration:

$$\begin{aligned} \boldsymbol{x}_0^* &= \boldsymbol{x} \\ \boldsymbol{x}_{n+1}^* &= \text{clip}\Big[\boldsymbol{x}_n^* + \alpha \text{sign}\Big[\nabla_{\boldsymbol{x}}\big(C(\boldsymbol{\omega}, \boldsymbol{x}, y)\big)\Big]\Big] \,. \end{aligned} \tag{15}$$

The clip function performs a per-pixel clipping of the image so that all values are between zero and one:

$$\big[\text{clip}(\boldsymbol{x})\big]_i = \min\big(1, x_i, \max(0, x_i)\big) \,.$$

In this study, four versions of *iterative fast gradient method* (IFGM) are implemented in each of the four norms. Each method iterates until misclassification. In the first norm, L_2, the direction of the gradient is preserved. Therefore, instead of using the sign function in (15) the gradient is normalized in the L_2-norm:

$$\begin{aligned} \boldsymbol{x}_0^* &= \boldsymbol{x} \\ \boldsymbol{x}_{n+1}^* &= \text{clip}\left[\boldsymbol{x}_n^* + \alpha \frac{\nabla_{\boldsymbol{x}}\big(C(\boldsymbol{x}_n^*, y)\big)}{\big|\big|\nabla_{\boldsymbol{x}}\big(C(\boldsymbol{x}_n^*, y)\big)\big|\big|_2}\right] \,. \end{aligned} \tag{16}$$

IFGM in the L_∞-norm is identical to identical to the BIM method (15):

$$\begin{aligned} \boldsymbol{x}_0^* &= \boldsymbol{x} \\ \boldsymbol{x}_{n+1}^* &= \text{clip}\Big[\boldsymbol{x}_n^* + \alpha \text{sign}\Big[\nabla_{\boldsymbol{x}}\big(C(\boldsymbol{x}, y)\big)\Big]\Big] \end{aligned} \tag{17}$$

The gradient is signed since all values can be increased to the maximum value without increasing the L_∞-distance. The signed gradient will thus have a larger impact on the cost without a larger L_∞-step.

In the L_1-space the maximum argument in the gradient is found. The pixel with that index is then perturbed with α in the given direction:

$$\begin{aligned} \boldsymbol{x}_0^* &= \boldsymbol{x} \\ a &= \arg\max_k \big|\big[\nabla_{\boldsymbol{x}}\big(C(\boldsymbol{x}, y)\big)\big]_k\big| \end{aligned} \tag{18}$$

$$x_{n+1,a}^* = \text{clip}\left[x_{n,a}^* + \alpha \text{sign}\left[\frac{\partial C(\boldsymbol{x}, y)}{\partial x_a}\right]\right] \,. \tag{19}$$

Furthermore, a criterion is added so that nothing can be added to a pixel with the maximum value. Likewise, nothing can be subtracted from a pixel of value zero, instead the second largest value in (18) is used.

Lastly the fourth IFGM version creates adversarial samples with the least L_0-distance by considering the current pixel values in the image. The algorithm is shown in Algorithm 1. Consider an image with the first two pixels

$$\boldsymbol{x} = (0.1, 0.9, \dots)$$

and with the gradient

$$\nabla_{\boldsymbol{x}} C(\boldsymbol{x}, y) = (1, 2, \dots) \,.$$

With the L_1-method, (18)-(19), the second pixel should be increased. However, in the L_0-norm any pixel altered may as well be altered as much as possible to increase the cost.

Algorithm 1 L_0 IFGM

1: **input:** Image $\boldsymbol{x}$, classifier f.
2: **output:** Adversarial image $\boldsymbol{x}^*$.
3:
4: Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}, i \leftarrow 0$.
5: **while** $\hat{k}(\boldsymbol{x}_i) = \hat{k}(\boldsymbol{x})$ **do**
6: $\quad \boldsymbol{r}_i \leftarrow \nabla_{\boldsymbol{x}} C(\boldsymbol{x}, y)$
7: $\quad \boldsymbol{m} \leftarrow ((\mathbf{1} + \text{sign}[\boldsymbol{r}_i])/2 - \boldsymbol{x}) \odot \boldsymbol{r}_i$
8: $\quad a = \arg\max_i \mathbf{m}$
9: $\quad x_a \leftarrow (\text{sign}[r_a] + 1)/2$
10: $\quad i \leftarrow i + 1$
11: **end while**

Thus, the L_0-method should initially increase the first pixel to one before altering the second. This is implemented by multiplying each value in the gradient with the amount the corresponding pixel could be altered in that direction. This is stored in an impact-array, $\boldsymbol{m}$, consisting only of positive numbers and the maximum value is found. The corresponding pixel is then increased to one or zero depending on the sign of the gradient. This is repeated until misclassification. It should be noted that Algorithm 1 does not have 100 % success rate and sometimes get stuck changing a single pixel back and forth from zero to one. The accuracy can be significantly increased by only allowing a pixel to be altered once by setting

$$m_i = 0 \quad \forall i \in \{\text{indexes of previously altered pixels}\}$$

before row 8 in the algorithm.

In *targeted* IFGM a specified label, t, is set first and then an adversarial image $\boldsymbol{x}^*$ is found so that $t = \hat{k}(\boldsymbol{x}^*)$. This is made with gradient descent into a low cost for the targeted label, instead of gradient ascent to a high cost for the current. The iteration is stopped when the correct label is achieved. For example, targeted L_2-IFGM attack:

$$\boldsymbol{x}_0^* = \boldsymbol{x}$$
$$\boldsymbol{x}_{n+1}^* = \text{clip}\left[\boldsymbol{x}_n^* - \alpha \frac{\nabla_{\boldsymbol{x}}\big(C(\boldsymbol{x}_n^*, t)\big)}{\big|\big|\nabla_{\boldsymbol{x}}\big(C(\boldsymbol{x}_n^*, t)\big)\big|\big|_2}\right]. \qquad (20)$$

B. DeepFool

DeepFool [6] is a method of constructing adversarial images through the use of the gradient of the classifiers output[1]. The main idea is to, with the help of this gradient, construct a locally linearized decision boundary and through projection calculate the perturbation needed to reach the other side of this boundary. This is iterated until misclassification, and the complete algorithm is given in Algorithm 2. A sketch of what happens internally can be found in Fig. 5.

To understand how this works, consider a linear, affine classifier

$$f(\boldsymbol{x}) = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}\,.$$

Here an n-dimensional classified vector-image $\boldsymbol{x}$ lies within an n-dimensional polyhedron where all images have the same label. Due to the linearity of the system, the exact locations of the decision boundaries can be computed and an adversarial image $\boldsymbol{x}^*$ can easily be placed on the other side. A general neural network can be fooled by firstly approximating it as an affine classifier and then calculating where the closest decision boundary should be. The image is placed there but due to the nonlinearity of the network it is often placed still inside the polyhedron and further iterations are required. However, the fact that it converges so quickly anyway, suggests that the mean curvature of the decision boundaries is low.

The L_p-distance from the current image $\boldsymbol{x}_i$ with the label $\hat{k}(\boldsymbol{x})$ and the decision boundary to the region with another label k is approximated as

$$\frac{|f_k(\boldsymbol{x}_i) - f_{\hat{k}(\boldsymbol{x})}(\boldsymbol{x}_i)|}{||\nabla f_k(\boldsymbol{x}_i) - \nabla f_{\hat{k}(\boldsymbol{x})}(\boldsymbol{x}_i)||_q}$$

where $q = \frac{p}{p-1}$ is the dual norm to p. This is simply the perpendicular projection. This distance is calculated for each label $k \neq \hat{k}(\boldsymbol{x})$ and the shortest of these are found corresponding to label $\hat{l}$. The image is then updated to

$$\boldsymbol{x} = \boldsymbol{x} + \frac{|f'_{\hat{l}}|}{||\boldsymbol{w}'_{\hat{l}}||_q^q} |\boldsymbol{w}'_{\hat{l}}|^{q-1} \odot \text{sign}(\boldsymbol{w}')\,.^2 \qquad (21)$$

There is no guarantee that the method finds the smallest perturbation but it finds a good approximation. In fact, there are other methods finding smaller perturbations [9], but that require cleverly chosen hyperparameters.

In the original study, the method is implemented in the L_2 and L_∞-norm and without clipping, and finds that it is generally superior to FGSM [5]. A follow-up article by the same group [16] studies it in the L_0-norm, but notes that with clipping, it fails to converge.

In this study, DeepFool is instead compared to IFGM, always studied with clipping, and slightly modified to work in the L_0- and L_1-norms.

We also introduce a step-length, by replacing line 12 in Algorithm 2 with

$$\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \alpha \cdot \boldsymbol{r}_i\,.$$

The goal with this is to study if smaller perturbations can be achieved with smaller steps, or faster running times with larger steps, or if the exact distance to the linearized boundary is essential for convergence of the algorithm.

When clipping is implemented the 12$^{\text{th}}$ row in Algorithm 2 is replaced with

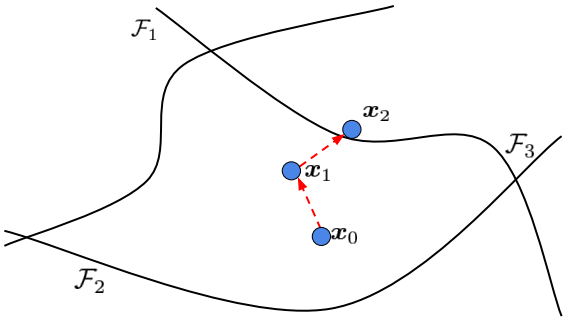
$$\boldsymbol{x}_{i+1} \leftarrow \text{clip}\left[\boldsymbol{x}_i + \boldsymbol{r}_i\right].$$

The method takes steps of varying sizes visible in row 11 in Algorithm 2. To make a fair comparison between IFGM and DF they both need to take steps of fixed length. Thus a *normalized DeepFoool* (NDF) is implemented. Here $\boldsymbol{r}_i$ is normalized in the current norm and multiplied with a scalar α:

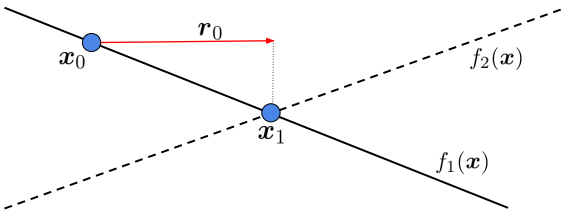
$$\boldsymbol{r}_i \leftarrow \alpha \frac{\boldsymbol{r}_i}{||\boldsymbol{r}_i||_p}\,.$$

[1] In practice the gradient is calculated based on $\boldsymbol{z}^L$ and not $\boldsymbol{a}^L = \text{softmax}(\boldsymbol{z}^L)$ That is, the computation is made on the final layer (4) before softmax is applied to it. This provides better numerical stability.

[2] The method is implemented with a slight overshoot: a number slightly larger than one is multiplied to the second term in the equation. The purpose here is to place $\boldsymbol{x}$ slightly beyond the decision boundary instead of slightly before, requiring further iterations.



(a) DeepFool tries to place $\boldsymbol{x}^*$ beyond the closest decision boundary of the neural network.



(b) In a linear classifier this could be done simply in one step by computing where f_2 is higher than $f_1 = \hat{k}(\boldsymbol{x}_0)$.

Fig. 5. Illustration of the DeepFool attack. Let $\boldsymbol{x}_0$ be the input image that currently lies in the centered boundary in (a). The DeepFool method approximated the shortest distance to place the image over the closest decision boundary ($\mathcal{F}_1$, $\mathcal{F}_2$ or $\mathcal{F}_3$) to cause misclassification. DeepFool approximates the boundaries as sides of a polyhedron and attempts each step to place it on the other side by projecting on the first Taylor expansion of f, illustrated in (b). However, due to the non-linear activation functions the method often requires further steps and isn't guaranteed to find the shortest distance which is shown in (a).

In the L_2-norm rows 10-11 in Algorithm 2 is replaced with

$$10: \quad \hat{l} \leftarrow \arg\min_{k \neq \hat{k}(\boldsymbol{x})} \frac{|f'_k|}{||\boldsymbol{w}'_k||_2}$$
$$11: \quad \boldsymbol{r}_i \leftarrow \frac{|f'_{\hat{l}}|}{||\boldsymbol{w}'_{\hat{l}}||_2} \boldsymbol{w}'_{\hat{l}}$$

and in the L_∞-norm they are replaced with

$$10: \quad \hat{l} \leftarrow \arg\min_{k \neq \hat{k}(\boldsymbol{x})} \frac{|f'_k|}{||\boldsymbol{w}'_k||_1}$$
$$11: \quad \boldsymbol{r}_i \leftarrow \frac{|f'_{\hat{l}}|}{||\boldsymbol{w}'_{\hat{l}}||_1} \text{sign}\left[\boldsymbol{w}'_{\hat{l}}\right].$$

In the L_1- and L_0-norm DeepFool is implemented in a similar fashion as IFGM, that is by changing the most significant pixel. This can be found from lines 10 and 11 in Algorithm 2 with In the L_1-norm rows 10-12 in Algorithm 2 is replaced with

$$10: \quad \hat{l} \leftarrow \arg\min_{k \neq \hat{k}(\boldsymbol{x})} \frac{|f'_k|}{||\boldsymbol{w}'_k||_\infty}$$
$$11: \quad a \leftarrow \arg\max |\boldsymbol{w}'_{\hat{l}}|$$
$$12: \quad x_{n+1}[a] \leftarrow x_n[a] + \frac{|f'_{\hat{l}}|}{w_{\hat{l}'}[a]}.$$

With clipping no pixels of value one can be increased or pixels of value zero be decreased. Instead the next highest value in row 11 above is used. In the L_0-implementation rows 10-11 in Algorithm 2 are the same as in the L_1-implementation, however, row 12 is replaced with rows 7-9 in Algorithm 1.

Targeted DF attacks are simple to implement given the original method. Instead of computing the closest condition

Algorithm 2 DeepFool [6]

1: **input:** Image $\boldsymbol{x}$, classifier f, norm p
2: **output:** Adversarial image $\boldsymbol{x}^*$.
3:
4: Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}, i \leftarrow 0$.
5: $q = \frac{p}{p-1}$
6: **while** $\hat{k}(\boldsymbol{x}_i) = \hat{k}(\boldsymbol{x})$ **do**
7: **for all** $k \neq \hat{k}(\boldsymbol{x})$ **do**
8: $\boldsymbol{w}'_k \leftarrow \nabla f_k(\boldsymbol{x}_i) - \nabla f_{\hat{k}(\boldsymbol{x})}(\boldsymbol{x}_i)$
9: $f'_k \leftarrow f_k(\boldsymbol{x}_i) - f_{\hat{k}(\boldsymbol{x})}(\boldsymbol{x}_i)$
10: **end for**
11: $\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(\boldsymbol{x})} \frac{|f'_k|}{||\boldsymbol{w}'_k||_q}$
12: $\boldsymbol{r}_i \leftarrow \frac{|f'_{\hat{l}}|}{||\boldsymbol{w}'_{\hat{l}}||_q^q} |\boldsymbol{w}'_{\hat{l}}|^{q-1} \odot \text{sign}(\boldsymbol{w}')$
13: $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$
14: $i \leftarrow i + 1$
15: **end while**
16: **return** $\boldsymbol{x}_i$

boundary $\hat{l}$ in row 10 in Algorithm 2, a target label is specified. More precise, with the targeted label t row 10 is replaced with

$$10: \quad \hat{l} \leftarrow t$$

for all given norms and the iteration is continued until the target label is achieved. This has been proposed in [17] and used as a subroutine, but to our knowledge, no literature reports an investigation of its performance.

IV. Experiments

A. Setup

The described algorithms were implemented using the neural network framework Tensorflow [18] and specifically the `tf.keras` submodule. To encourage reproducibility, the implementation is provided at [19] and to maintain comparability in running time, the experiments were run on an Early 2015 MacBook Pro with a 2.7 GHz i5-5257U CPU, similar to the hardware used in [6].

Two architectures are studied: a simple DNN with converging width, described in Table I, applied to the MNIST dataset, and a CNN very similar to the ones studied in other adversarial machine learning articles [10] and [4] described in Tables II and III for the MNIST and CIFAR-10 datasets respectively.

B. Clipping

The original DeepFool article [6] did not include an investigation on the impact of different regimes of clipping - after *each* iteration, after the *last* iteration or *none*. To investigate this, three parameters, fooling rate, percentage within bounds and number of iterations, were measured for two datasets and norms. The results are depicted in Fig. 6 and Table IV.

C. Introduction of step length in DeepFool

The original DeepFool-algorithm [6] has no parameterized step length, in fact, the main idea is to calculate the step as a projection onto the first order Taylor-approximation of

TABLE IV
THE EFFECT OF DIFFERENT CLIPPING REGIMES ON DEEPFOOL. CLIPPING CAN EITHER BE DONE NEVER (*none*), AFTER THE *last* ITERATION OR BETWEEN *each* ITERATION. THE FOOLING RATE IS THE NUMBER OF IMAGES PERCENTAGE OF IMAGES MISCLASSIFIED OF ALL 10000 IN THE TEST DATA. PERCENTAGE WITHIN DESCRIBES THE PERCENTAGE OF IMAGES WHERE ALL PIXELS LIE IN THE ALLOWED RANGE OF $[0, 1]$. NR OF ITERATIONS IS THE MEAN NUMBER OF ITERATIONS UNTIL THE ALGORITHMS TERMINATE.

Dataset and architecture	Norm	Clipping	Fooling rate %	Percentage within $[0, 1]$	Nr of iterations
MNIST DNN	L_2	None	100	0.0	2.225
		Last	0.51	100	2.225
		Each	100	100	17.8213
	L_∞	None	100	0	2.248
		Last	0.38	100	2.248
		Each	100	100	15.1803
CIFAR-10 CNN	L_2	None	100	54.99	2.956
		Last	79.77	100	2.956
		Each	100	100	3.0454
	L_∞	None	100	53.41	2.8785
		Last	76.46	100	2.8785
		Each	100	100	2.9602

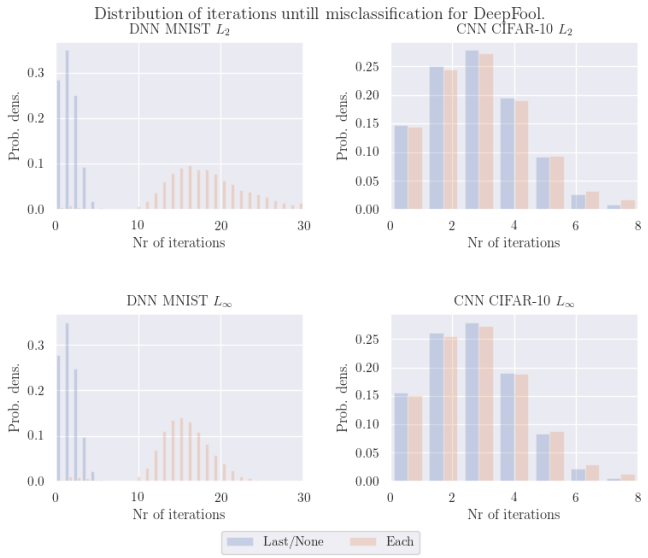


Fig. 6. Histogram distribution of the number of iterations required until misclassification for DeepFool applied to MNIST DNN and CIFAR-10 CNN for the L_2- and L_∞-norms.

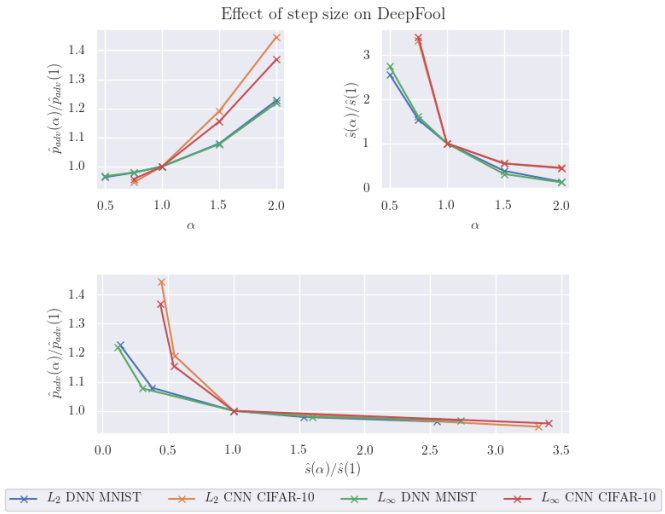


Fig. 7. The average perturbations $\hat{p}_{adv}$ and average number of steps required $\hat{s}$ evaluated for different step sizes α and for L_2- and L_∞-norms for MNIST and CIFAR-10, normalized to the values for $\alpha = 1$. An $\alpha < 1$ gives smaller perturbations for all datasets and architectures.

the decision boundary. Still, an idea could be to increase the magnitude of this step with a factor α, as this could potentially, for $\alpha < 1$ give smaller $\hat{p}_{adv}$, and for $\alpha > 1$ decrease the number of iterations and thus running time needed.

To test this, DeepFool is run with several step length for MNIST DNN, CIFAR-10 CNN in the norms L_2 and L_∞. The results can be seen in Fig. 7. Note that for comparability, the reported values are divided by the value for $\alpha = 1$.

D. Performance comparison

Some examples can be found in Fig. 1. Overall comparison of the performance is provided in Table V. This table does not contain the fooling rate, which was 100 % for DeepFool and normalized DeepFool. For IFGM, the gradient sometimes identically equaled the zero matrix, making it impossible to find an adversarial sample. This happened in up to 3 % of all cases, giving a lower bound of the fooling rate at 97%.

1) L_2- and L_∞-norm: To investigate the performance, normalized DeepFool and IFGM is run for different step lengths. For L_2 and MNIST, this is plotted in Fig. 8. For L_2 attacks on the CNN-CIFAR-10 classifier and L_∞ attacks on the DNN-MNIST and CNN-CIFAR-10 classifiers, the plots look very similar. Relevant values can be found in Table V.

2) L_1-norm: For the L_1-norm, a parameter study of the step size can be seen in Fig. 9 for CIFAR data. MNIST shows very similar results.

3) L_0-norm: For the L_0-norm, IFGM and DF have no step lengths. To investigate their performance, the algorithms were run on the three setups and the distribution of the number of pixels changed can be found in Fig. 10. Table VI summarizes key metrics of the algorithms and compares it to the same metrics in the L_1-norm. Here, the step length for IFGM was chosen to be comparatively large at $\alpha = 0.25$, to approximate the L_0 attack the best.

TABLE V
PERFORMANCE METRICS OF THE THREE ATTACKS. MATCHING MEANS THAT THE ALGORITHM IS EVALUATED FOR THE α WHERE IT HAS THE SAME VALUE AS ORDINARY DEEPFOOL. * STAR DENOTES THAT MATCHING COULD NOT BE REACHED.

Norm	Setup	$\hat{\rho}_{adv}$ DF	t DF	$\hat{\rho}_{adv}$ NDF matched t	$\hat{\rho}_{adv}$ IFGM matched t	t, NDF matched $\hat{\rho}_{adv}$	t, IFGM matched $\hat{\rho}_{adv}$
L_2	MNIST DNN	0.060	0.16	0.0597	0.0585	0.156	0.0426
	CIFAR-10 CNN	0.00498	0.301	0.0063	0.0045	0.76	0.079
L_∞	MNIST DNN	0.03725	0.141817	0.0369807	0.039938	0.12	*
	CIFAR-10 CNN	0.00522	0.277062	0.00637	0.00488533	0.352	0.07865

TABLE VI
PERFORMANCE OF OUR IMPACT-WEIGHTED L_0-VERSIONS OF DEEPFOOL AND IFGM, AND COMPARISON TO THE METRICS FOR THE L_1-VERSIONS. THE STEP SIZE α FOR IFGM IN THE L_1-VERSION IS 0.25. CP STANDS FOR CHANGED PIXELS AND IS REPORTED AS A MEAN VALUE. t DENOTES TIME AND IS ALSO REPORTED AS AN AVERAGE VALUE.

	L_0				L_1			
Setup	CP DF	t DF	CP IFGM	t IFGM	CP DF	t DF	CP IFGM	t IFGM
MNIST DNN	4.12	0.04	4.38	0.006	4.51	0.049	5.96	0.019
MNIST CNN	12.52	0.53	12.48	0.13	13.29	1.77	13.88	0.51
CIFAR-10 CNN	8.58	1.01	8.72	0.102	9.71	0.94	9.78	0.33

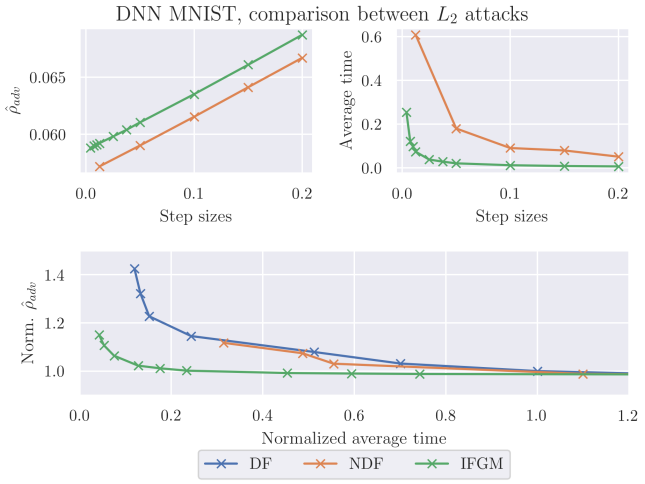


Fig. 8. The performance of L_2 attacks dependent on the step size. The upper left plots average relative image difference to the step size. Next, the average time taken to compute an adversarial sample is plotted to the steps size used. Lastly, the average relative image difference is plotted to the computational time taken which shows the time efficiency of the methods to produce a certain result.

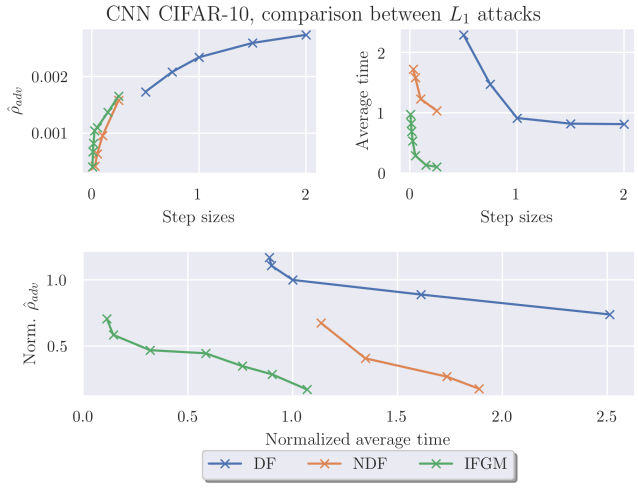


Fig. 9. The performance of L_1 attacks dependent on the step size. Note that the norm of the steps taken of DF isn't an order of magnitude larger than IFGM and NDF. Step sizes here for DF indicates the step size parameter multiplied with the last term in the right-hand side of (21).

E. Targeted attacks

A set of targeted L_0-attacks is displayed in Fig. 1 for both IFGM and DeepFool. The attacks are performed on the DNN-MNIST classifier where a number of each label is changed into each other label. For comparison, an untargeted attack is also performed.

In Fig. 12, $\hat{\rho}_{adv}$ is measured for a targeted attack in L_∞ for NDF and IFGM with the same step size, 0.001, on the DNN classifier on the MNIST dataset.

V. DISCUSSION

A. Is clipping important for DeepFool?

The original DeepFool article [6] did not include an investigation on the influence of clipping of the adversarial images, and reported values in the un-clipped case. This considered the L_2 and the L_∞-norm. In the L_1 norm, the SparseFool article [16] notes that the fooling rate is 100 % without clipping, around 5 % with clipping last and that the algorithm does not terminate with clipping at each iteration.

Our results, seen in Table IV show that for the L_2- and L_∞-norm, no clipping leads to that for MNIST, 100 % of images contain a pixel outside the allowed range, and for CIFAR-10, just above 50 %. For most applications, this figure is far too large to ignore, so some kind of clipping is necessary. However, with clipping after the last iteration, the fooling rate falls from 100 % to $\sim$ 0.5% and $\sim$ 75% for MNIST and CIFAR-10, respectively. These low figures force the usage of a harsher clipping, to be usable in applications, even though it should give a penalty in running time. To test this, the distribution of the number of iterations until misclassification

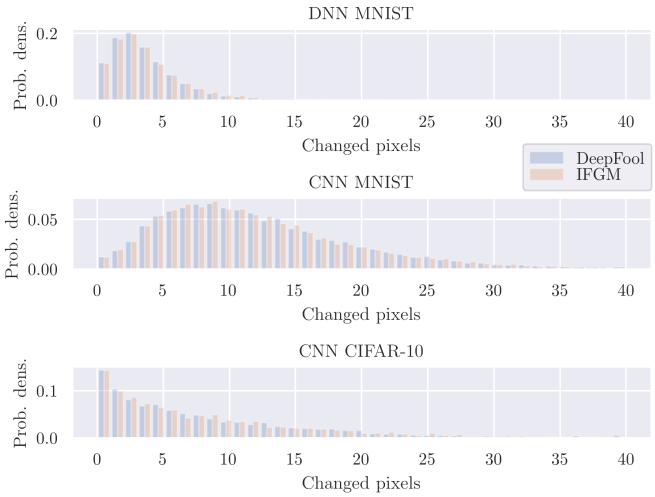


Fig. 10. Histogram distribution of changed pixels required to cause misclassification with the L_0 attacks.

is plotted in Fig. 6. For MNIST, the number of required iterations (and the running time), is increased by a factor of around 7, and for CIFAR-10, by 1.03.

For context, FGM-based methods have been shown to require clipping at each iteration in literature [5], [9], [10], so it comes as no surprise that our results show that DeepFool also requires it.

B. Step length in non-normalized DeepFool

Fig. 7 shows the effect of step size in a non-normalized version of DeepFool. Firstly, we can conclude that the introduction of α still gives a stable algorithm in the sense that it always terminates in the range $\alpha \in [0.5, 2]$. Secondly, the hypothesis that $\alpha < 1$ gives smaller $\hat{\rho}_{adv}$ at the expense of a larger amount of iterations needed, is true, and vice versa for $\alpha > 1$. However, using an $\alpha < 1$ clearly gives diminishing returns, as $\hat{\rho}_{adv}$ only decreases by around 5% while the running time increases with a factor of 3.5. Interestingly, in the lower trade-off graph, a clear distinction can be seen between MNIST DNN and CIFAR-10 CNN. It seems as if for more complex tasks, the running time cannot be decreased as easily as for simpler tasks.

C. Performance in L_2 and L_∞ norms

The performance metrics are summarized in Table V. Overall, there is little difference between the methods in regards to $\hat{\rho}_{adv}$, only up to around 5 %. This can be compared to state of the art Carlini-Wagner attack [9], which give 50 % smaller perturbations. However, this small difference proves to us that a normalized DeepFool is a valid attack. For the matching t and $\hat{\rho}_{adv}$, we note that normalized DeepFool performs slightly better than ordinary DeepFool on MNIST, but slightly worse on CIFAR-10. For IFGM, there is no such trend for $\hat{\rho}_{adv}$, but it is very clear that IFGM is much faster than DeepFool.

Thus, IFGM gives roughly the same perturbations as DeepFool, but requires an appropriate step size, which has to be set specifically for the dataset and architecture at hand. DeepFool, on the other hand, is parameterless. IFGM also has a second downside, of sometimes jamming due to a zero gradient, reducing the otherwise perfect fooling rate.

D. Performance in L_1 norm

For L_1-norm, as seen in Fig. 9, we can see that we can generate adversarial examples with DeepFool, by not allowing already extremal pixels to be changed. However, on the contrary to the results for L_2 and L_∞, the algorithms show big differences. Of the three (DeepFool, normalized DeepFool and IFGM) it is DeepFool that performs the worst. DeepFool has both a large $\hat{\rho}_{adv}$ and takes a long time. Normalized DeepFool performs much better with about half $\hat{\rho}_{adv}$ with the same amount of given time. IFGM performs best since it produces perturbations down to 15% of normal DeepFool in the same amount of time. The reason for this is not known, but it is very interesting.

E. Performance of our performed L_0 versions on DeepFool and IFGM

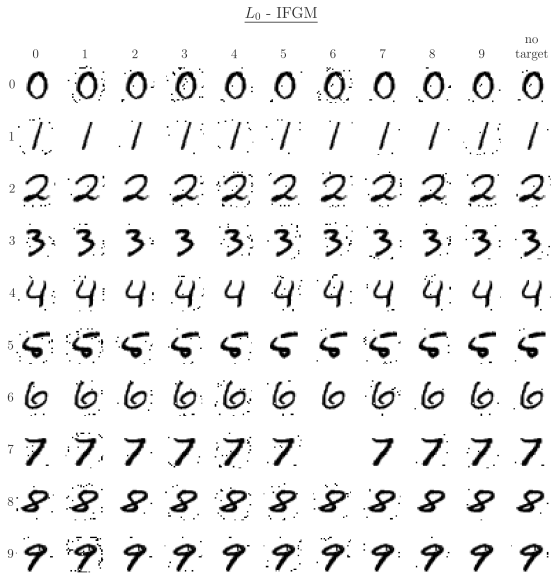
Our proposed L_0-versions of DeepFool and IFGM are essentially the same as the L_1-attack, modifying the most "important" pixel, with the difference that it now is modified to the extremal values and that the gradient is weighted according to how much the pixel can be changed. Table VI clearly shows that this reduces the mean number of pixels that need to be changed, so it seems to be a slight improvement, and not as large as we would have hoped. An interesting aspect is the distribution of the number of pixels, as seen in Fig. 10. Firstly, we see that DeepFool and FGM performance was almost indistinguishable. Secondly, the shape of the distributions are very different, and resemble Poisson distributions, but for CIFAR-10, with a too slow taper to the right. This could be an interesting topic for further investigations.

F. Targeted attacks

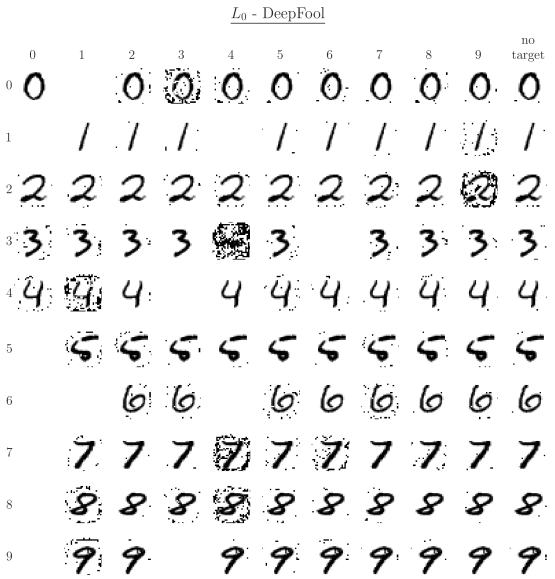
Through Fig. 11 and Fig. 12, we demonstrate that targeted versions are possible with both IFGM and DeepFool. To our knowledge, no performance investigation of targeted DeepFool can be found in literature, but it has been used as a subroutine in a more complex system [17]. However, as can quickly be noticed in Fig. 11, targeted DeepFool often fails to find a solution within a reasonable time, here seen as 10000 iterations, and many solutions are quite, visibly, bad. To quantify this, a similar experiment is done in the L_∞-norm, and reported in Fig. 12. Here, it can clearly be noted that $\hat{\rho}_{adv}$ is much smaller for IFGM than NDF. Interestingly, this is not the case for all pairs of starting and target labels, for example, a 1 is easily converted to a 6, 7 or 8, but very hard to convert to a 4.

G. Future work

For future work, similar experiments should be done on a harder task, for example, ImageNet, to see if the conclusions generalize. For a complete investigation, transferability of the attacks should also be studied for all datasets. This is since

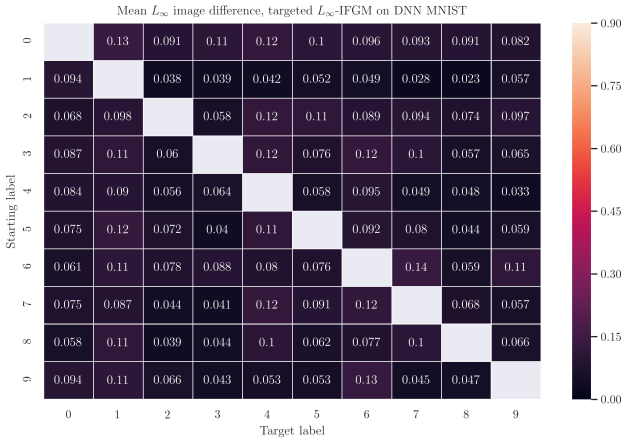


(a) Targeted L_0-IFGM attack on the DNN-MNIST classifier.

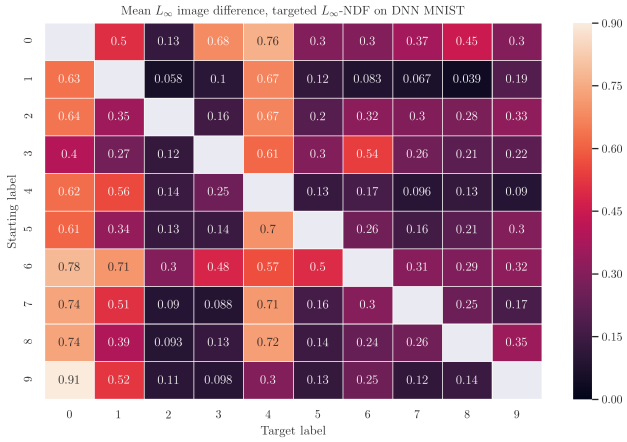


(b) Targeted L_0-DeepFool attack on the DNN-MNIST classifier.

Fig. 11. Examples of targeted L_0-attacks using the FMG (a) and DeepFool (b) methods. The sources are images of integers 0-9, ranging the ten rows. Eleven attacks are performed on each image for each method. The first ten attacks are targeted to each other label and shown in the first ten columns. The last column uses no target and simply tries to misclassify the image. The whited out images indicates unsuccessful targeted attacks after 10 000 iterations. The diagonal shows the original image, since an integer targeted to its own label isn't changed.



(a) Targeted L_∞-IFGM attack on the DNN-MNIST classifier.



(b) Targeted L_∞-DeepFool attack on the DNN-MNIST classifier.

Fig. 12. The mean, normalized image difference, $\hat{\rho}_{adv}$, measured for a targeted attack on the DNN-MNIST classifier in the L_∞ norm using IFGM and DeepFool attacks. Darker colors indicate smaller perturbations and blank tiles, trivial attacks, that is, with the same starting and target label.

results differ between the simple MNIST and more complex CIFAR-10. For example one can see that clipping is relevant when attacking both datasets but less so for CIFAR-10. Some other results could change with increasing complexity of the dataset.

In this paper, we have demonstrated that DeepFool and IFGM can work in a targeted fashion. A natural extension could be to do a quantitative investigation of $\hat{\rho}_{adv}$ and running time. A challenge with this is that this depends on starting and target label, which adds even more dimensions to the problem.

One way to speed up DeepFool, is to instead of looping over all possible classes, loop over only a subset of these. This is noted in the accompanying code to [6], which suggests looping over the top n classes sorted on initial classifier probability output. However, as far as we know, there exists no investigation on how this affects $\hat{\rho}_{adv}$.

A discovered problem with IFGM is that sometimes, the gradient is identically the zero matrix, resulting in that the iterations get stuck. Some initial experiments show that by perturbing the image with some very small noise, it is possible to get out of this situation, but this also requires further analysis.

In large, the experiments have shown that there is little difference between DeepFool and IFGM, other than the fact that no step length is needed for DeepFool. This suggests that ideas used for optimizing IFGM could also be applied to

DeepFool, for example by introducing momentum similarly to [20].

The fact that the introduction of a step length to non-normalized DeepFool produces such rich results, suggests that the first-order approximation of the decision boundary as a plane may not be enough. Instead, a second order approximation with a two-term Taylor series could be used.

VI. Conclusion

In this study, we examined the performance of two iterative adversarial attacks based on the gradient of the outputs (DeepFool) and the gradient of the cost functions (IFGM). In short, we find that their performance is nearly identical in the L_0, L_2 and L_∞ norms, but not in L_1. However, IFGM is faster, because it only calculates a single gradient, but has the downside that it requires a step length, which is dependent on dataset and architecture, while DeepFool works well in the parameterless version. More specifically, we find that

- Contrary to what [16] states, that without clipping, a significant portion of adversarial images have over- or undersaturated pixels, and that clipping at each iteration is important. However, this is less so for more advanced tasks.
- DeepFool can be tuned in the trade-off between perturbation size and running time with the introduction of a step length parameter.
- DeepFool can be normalized, without reducing its efficacy.
- Normalized DeepFool and IFGM produce very similar perturbations, but IFGM calculates fewer gradients and is up to an order of magnitude faster, but less so the smaller the perturbation is.
- The reason L_1 DF clipped at each iteration is stated not to work [16] is that it tries to change already extremal pixels. A modified L_1 DeepFool where these pixels are not allowed to change, works, but is much better when normalized.
- By using an impact-vector and changing pixels to their maximal values, L_1 attacks can be converted to better approximations of L_0 attacks in a simple manner, with a slight increase in performance.
- IFGM and DeepFool can be used in targeted versions, but DeepFool seems to have much worse performance than IFGM in that realm.
- In IFGM, the gradient sometimes becomes zero, giving a fooling rate $< 100\%$. This is not the case for DF, which can thus be seen as more reliable.

Acknowledgment

The authors wish to thank Ezgi Kormaz for her engaging supervision of the project as well as great reviews and reading material.

References

[1] G. T. Tsenov and V. M. Mladenov, "Speech recognition using neural networks," in *10th Symposium on Neural Network Applications in Electrical Engineering*, Sep. 2010, pp. 181–186.

[2] G. Huang, Z. Liu, L. v. d. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 2261–2269.

[3] S. Madisetty and M. S. Desarkar, "A neural network-based ensemble approach for spam detection in twitter," *IEEE Transactions on Computational Social Systems*, vol. 5, no. 4, pp. 973–984, Dec 2018.

[4] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations (ICLR)*, Dec. 2014.

[5] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations (ICLR)*, May 2015.

[6] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: A simple and accurate method to fool deep neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 2574–2582.

[7] S. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 86–94.

[8] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018, pp. 1625–1634.

[9] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 39–57.

[10] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *International Conference on Learning Representations (ICLR)*, April 2017.

[11] M. A. Nielson, *Neural Networks and Deep Learning*. Determination Press, 2015.

[12] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits. Accessed: 2019-04-16. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[13] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[14] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 582–597.

[15] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, April 2017, pp. 506–519.

[16] A. Modas, S. Moosavi-Dezfooli, and P. Frossard, "Sparsefool: a few pixels make a big difference," in *2019 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[17] C. Liao, H. Zhong, A. C. Squicciarini, S. Zhu, and D. J. Miller, "Backdoor embedding in convolutional neural network models via invisible perturbation," in *arxiv preprint*, 2018. [Online]. Available: https://arxiv.org/abs/1808.10307

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org.

[19] T. Liiv and A. Strömberg. (2019) Iterative gradient based adversarial attacks on neural network image classifiers. [Online]. Available: https://github.com/axelstr/Iterative_Gradient_Based_Adversarial_Attacks_on_Neural_Network_Image_Classifiers

[20] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," in *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018, pp. 9185–9193.