Objective Metrics and Gradient Descent Algorithms for Adversarial Examples in Machine Learning

Uyeong Jang
University of Wisconsin
Madison, Wisconsin
wjang@cs.wisc.edu

Xi Wu*
Google
wu.andrew.xi@gmail.com

Somesh Jha
University of Wisconsin
Madison, Wisconsin
jha@cs.wisc.edu

ABSTRACT

Fueled by massive amounts of data, models produced by machine-learning (ML) algorithms are being used in diverse domains where security is a concern, such as, automotive systems, finance, health-care, computer vision, speech recognition, natural-language processing, and malware detection. Of particular concern is use of ML in cyberphysical systems, such as driverless cars and aviation, where the presence of an adversary can cause serious consequences. In this paper we focus on attacks caused by adversarial samples, which are inputs crafted by adding small, often imperceptible, perturbations to force a ML model to misclassify. We present a simple gradient-descent based algorithm for finding adversarial samples, which performs well in comparison to existing algorithms. The second issue that this paper tackles is that of metrics. We present a novel metric based on few computer-vision algorithms for measuring the quality of adversarial samples.

KEYWORDS

Adversarial Examples, Machine Learning

1 INTRODUCTION

Massive amounts of data are currently being generated in domains such as health, finance, and computational science. Fueled by access to data, *machine learning (ML)* algorithms are also being used in these domains, for providing predictions of lifestyle choices [7], medical diagnoses [17], facial recognition [1], and more. However, many of these models that are produced by ML algorithms are being used in domains where security is a big concern – such as, automotive systems [26], finance [20], health-care [2], computer vision [21], speech recognition [14], natural-language processing [29], and cyber-security [8, 31]. Of particular concern is use of ML in

---

*Work was done while at University of Wisconsin

Figure 1: To humans, these two images appear to be the same. The image on the left is an ordinary image of a stop sign. The image on the right was produced by adding a small, precise perturbation that forces a particular image-classification DNN to classify it as a yield sign.

cyberphysical systems, where the presence of an adversary can cause serious consequences. For example, much of the technology behind autonomous and driver-less vehicle development is driven by machine learning [3, 4, 10]. *Deep Neural Networks (DNNs)* have also been used in airborne collision avoidance systems for unmanned aircraft (ACAS Xu) [18]. However, *in designing and deploying these algorithms in critical cyberphysical systems, the presence of an active adversary is often ignored.*

In this paper, we focus on attacks on outputs or models that are produced by machine-learning algorithms that occur *after training* or "external attacks", which are especially relevant to cyberphysical systems (e.g., for a driver-less car the ML-algorithm used for navigation has been already trained once the "car is on the road"). These attacks are more realistic, and are distinct from "insider attacks", such as attacks that poison the training data (see the paper [15] for a survey such attacks).

Specifically, we focus on attacks caused by *adversarial examples*, which are inputs crafted by adding small, often imperceptible, perturbations to force a trained ML model to misclassify. As a concrete example, consider a ML algorithm that is used to recognize street signs in a driver-less car, which takes the images, such as the one depicted in Figure 1, as input. While these two images may appear to be the same to humans, the image on the left [32] is an ordinary image of a stop sign while the right image was produced by adding a small, precisely crafted perturbation that forces a particular image-classifier to classify it as a yield sign. Here, the adversary could potentially use the altered image to cause the car to behave dangerously, if the car did not have additional fail-safes such as GPS-based maps of known stop-sign locations. As driver-less cars become more common, these attacks are of a grave concern.

Our paper makes contributions along two dimensions: a new algorithm for finding adversarial samples and better metrics for evaluating quality of adversarial samples. We summarize these contributions below.

Algorithms: Several algorithms for generating adversarial samples have been explored in the literature. Having a diverse suite of these algorithms is essential for understanding the nature of adversarial examples and also for systematically evaluating the robustness of defenses. The second point is underscored quite well in the following recent paper [6]. Intuitively, diverse algorithms for finding adversarial examples, exploit different limitations of a classifier and thus stress the classifier in a different manner. In this paper, we present a simple gradient-descent based algorithm for finding adversarial samples. Our algorithm is described in section 3 and an enhancement to our algorithm appears in the appendix. Even though our example is quite simple (although we discuss some enhancements to the basic algorithm), it performs well in comparison to existing algorithms. Our algorithm, NewtonFool, successfully finds small adversarial perturbations for all test images we use, but also it does so by significantly reducing the confidence probability. Detailed experimental results appear in section 4.

Metrics: The second issue that this paper tackles is the issue of metrics. Let us recall the adversary's goal: given an image I and a classifier F, the adversary wishes to find a "small" perturbation δ, such that $F(I)$ and $F(I + \delta)$ are different but I and $I + \delta$ "look the same" to a human observer (for targeted misclassification the label $F(I + \delta)$ should match the label that an adversary desires). The question is– how does one formalize "small" perturbation that is not perceptible to a human observer? Several papers have quantified "small perturbation" by using the number of pixels changed or the difference in the L_2 norm between I and $I + \delta$. On the other hand, in the computer-vision community several algorithms have been developed for tasks that humans perform quite easily, such as edge detection and segmentation. Our goal is to leverage these computer-vision algorithms to develop better metrics for to address the question given before. As a first step towards this challenging problem, we use three algorithms from the computer-vision literature (which is described in our background section 2) for this purpose, but recognize that this is a first step. Leveraging other computer-vision algorithms to develop even better metrics is left as future work.

Related work: Algorithms for generating adversarial examples is a very active area of research, and we will not provide a survey of all the algorithms. Three important related algorithms are described in the background section 2. However, interesting algorithms and observations about adversarial perturbations are constantly being discovered. For example, Kurakin, Goodfellow, and Bengio [22] show that even in physical-world scenarios (such as deployment of cyberphysical systems), machine-learning systems are vulnerable to adversarial examples. They demonstrate this by feeding adversarial images obtained from cell-phone camera to an ImageNet Inception classifier and measuring the classification accuracy of the system and discover that a large fraction of adversarial examples are classified incorrectly even when perceived through

the camera. Moosavi-Dezfooli et al. [24] propose a systematic algorithm for computing universal perturbations[1] In general, the area of analyzing of robustness of machine-learning algorithms is becoming a very important and several research communities have started working on related problems. For example, the automated-verification community has started developing verification techniques targeted for DNNs [16, 19].

2 BACKGROUND

This section describes the requisite background. We need Moore-Penrose pseudo-inverse of a matrix for our algorithm, which is described in section 2.1. Three techniques that are used in our metrics are described in the next three sub-sections. Formulation of the problem is discussed in section 2.5. Some existing algorithms for crafting adversarial examples are discussed in section 2.6. We conclude this section with a discussion section 2.7.

2.1 Moore-Penrose Pseudo-inverse

Given a $n \times m$ matrix A, a matrix A^+ is called it Moore-Penrose pseudo-inverse if it satisfies the following four conditions (A^T denotes the transpose of A):

(1) $AA^+A = A$
(2) $A^+AA^+ = A^+$
(3) $A^+A = (A^+A)^T$
(4) $AA^+ = (AA^+)^T$

For any matrix A, a Moore-Penrose pseudo-inverse exists and is unique [11]. Given an equation $A\mathbf{x} = \mathbf{b}$, $\mathbf{x_0} = A^+\mathbf{b}$ is the best approximate solution to $A\mathbf{x} = \mathbf{b}$ (i.e., for any vector $\mathbf{x}$ satisfying the equation, $\| A\mathbf{x_0} - b \|$ is less than or equal to $\| A\mathbf{x} - b \|$).

2.2 Edge Detectors

Given an image, edges are defined as the pixels whose value changes drastically from the values of its neighbor. The concept of edges has been used as a fundamental local feature in many computer-vision applications. The *Canny edge detector (CED)* [5] is a popular method used to detect edges, and is designed to satisfy the following desirable performance criteria: high true positive, low false positive, the distance between a detected edge and a real edge is small, and there is no duplicate detection of a single edge. In this section we will describe CED.
Preprocess – noise reduction. Most images contain random noise that can cause errors in edge detection. Therefore, filtering out noise is an essential preprocessing step to get stable results. Applying convolution with a Gaussian kernel, also know as Gaussian blur, is a common choice to smooth a given image. For any $n \in \mathbb{N}$, $(2n + 1) \times (2n + 1)$ Gaussian kernel is defined as follows.

$$K(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \text{ where } -n \leq x, y \leq n$$

The denoising is dependent on the choice of n and σ.
Computing the gradient. After the noise-reduction step, CED computes the intensity gradients, by convolving with

---

[1] A *single* vector, which when added to an image causes its label to change.

Sobel filters. For a given image I, the following shows an example convolutions of 3×3 Sobel filters, which has been used in our experiments (in the equations given below $*$ represents convolution).

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

G_x and G_y encodes the variation of intensity along x-axis and y-axis respectively, therefore we can compute the gradient magnitude and direction of each pixel using the following formula.

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

The angle θ is rounded to angles corresponding horizontal ($0°$), vertical ($90°$), and diagonal directions ($45°, 135°$).

Non-maximum suppression. The intensity gradient gives us enough information about the changes of the values over pixels, so edges can be computed by thresholding the intensity magnitude. However, as we prefer thin and clear boundary with no duplicated detection, Canny edge detector performs edge thinning, which is done by suppressing each gradient magnitude to zero unless it achieves the local maxima along the gradient direction. Specifically, for each pixel point, among its eight neighboring pixels, Canny edge detector chooses two neighbors to compare according to the rounded gradient direction θ.

- If $\theta = 0°$, choose the neighboring pixels at the east and west
- If $\theta = 45°$, choose the neighboring pixels at the north east and south west
- If $\theta = 90°$, choose the neighboring pixels at the north and south
- If $\theta = 135°$, choose the neighboring pixels at the north west and south east

These choices of pixels correspond to the direction perpendicular to the possible edge, and the Canny edge detector tries to detect a single edge achieving the most drastic change of pixel intensity along that direction. Therefore, it compares the magnitude of the pixel to its two neighbors along the gradient direction, and set the magnitude to be 0 when it is smaller than any magnitudes of its two neighbors.

Thresholding with hysteresis. Finally, the Canny edge detector thresholds gradients using hysteresis thresholding. In hysteresis thresholding, we first determine a strong edge (pixel with gradient bigger than θ_{high}) and a weak edge (pixel with gradient between θ_{low} and θ_{high}), while suppressing all non-edges (pixels with gradient smaller than θ_{low}). Then, the Canny edge detector checks the validity of each weak edge, based on its neighborhood. Weak edges with at least one strong edge neighbor will be detected as valid edges, while all the other weak edges will be suppressed.

The performance of Canny edge detector depends highly on the threshold parameters θ_{low} and θ_{high}, and those parameters should be adjusted according to the properties of input image. There are various heuristics to determine the thresholds, and we use the following heuristics in our experiments

- **MNIST:** While statistics over pixel values (e.g mean, median) are usually used to determine thresholds, pixel values in MNIST images are mostly 0, making such statistics unavailable. In this work, we empirically searched the proper values for thresholds, sufficiently high to be able to ignore small noise, and finally used $\theta_{low} = 300$ and $\theta_{high} = 2 \cdot \theta_{low}$.
- **GTSRB:** When distribution of pixel value varies, usually statistics over pixel value are used to adjust thresholds, because pixel gradient depends on overall brightness and contrast of image. Since those image properties varies in GTSRB images, we put thresholds as follows.

$$\theta_{low} = (1 - 0.33)\mu$$
$$\theta_{high} = (1 + 0.33)\mu$$

where μ is the mean of the values on pixels.

2.3 Fourier Transform

In signal processing, spectral analysis is a technique that transforms signals into functions with respect to frequency, rather than directly analyzing the signal on temporal or spatial domain. There are various mathematical operators (transforms) converting signals into spectra, and Fourier transform is one of the most popular operator among them. In this section we mainly discuss two dimensional Fourier transform as it is an operation on spatial domain where an image lies in and is commonly applied to image analysis.

Considering an image as a function f of intensity on two dimensional spatial domain, the Fourier transform F is written in the following form.

$$F(u,v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y)e^{-2\pi i(ux+vy)}dxdy$$

While this definition is written for continuous spatial domain, values in an image are sampled for a finite number of pixels. Therefore, for computational purposes, the corresponding transform on discrete two dimensional domain, or discrete Fourier transform, is used in most applications. For a function f of on a discrete grid of pixels, the discrete Fourier transform maps it to another function F on frequency domain as follows.

$$F(k,l) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \exp\left[-2\pi i \left(\frac{kx}{M} + \frac{ly}{N}\right)\right]$$

While naive computation of this formula requires quadratic time complexity, there are several efficient algorithms computing discrete Fourier transform with time complexity $O(n \log n)$, called *Fast Fourier Transform (FFT)*, and we use the two dimensional FFT in our analysis of perturbations.

Fourier transform on two dimensional spatial domain provides us spectra on two dimensional frequency (or spatial frequency) domain. These spectra describe the periodic structures across positions in space, providing us valuable information of features of an image. Specifically, low spatial frequency corresponds to the rough shape structure of the image, whereas spectrum on high spatial frequency part conveys detailed feature, such as sharp change of illumination and edges.

2.4 Histogram of Oriented Gradients

The *histogram of oriented gradients (HOG)* [9] is a feature descriptor, widely used for object detection. Simply, HOG descriptor is a concatenation of a number of locally normalized histograms. Each histogram contains local information about the intensity gradient directions, each local set of neighboring histograms is normalized to improve the overall accuracy. By concatenating those histogram vectors, HOG outputs a more compact description of the shape. For object detection, a machine learning algorithm is trained over HOG descriptors of training set images, to classify if any part of an input image has HOG descriptor should be labeled as "detected"

Computing the gradient. Similarly to the edge detector in 2.2, HOG starts from computing gradient for each pixel. While 3×3 Sobel filters are used in Canny edge detector, HOG applies the following simpler one dimensional filters to the input image.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} * I$$

$$G_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * I$$

Using the same formula used for Canny edge detector, HOG computes the gradient magnitude G and direction θ of each pixel. However, HOG does not round the angle θ, as it gives an important information for the next step.

Histogram construction. To construct histograms to be concatenated, HOG first divides the input image into smaller *cell*s, consisting of pixels (e.g. 8×8 pixels) then compute a histogram for each *cell*. Each histogram has several bins and each bin is assigned an angle. In [9], 9 bins corresponding to $0°, 20°, \ldots, 160°$ are used in each histogram.

Histograms are computed by the weighted vote for each pixels in a *cell* by voting the weighted magnitude to one or two bins. The weights are determined by the gradient direction θ, based on how close the angle is from its closest two bins.

Block normalization. Since the magnitude of gradients is highly dependent to the local illumination and contrast, each histogram should be locally normalized. *Block*, consisting of several number of cells (e.g. 2×2 cells), is the region that HOG applies the normalization to the histograms of the contained cells. While there are various way to normalize a single block, Dalal et.al.[9] reported that the performance is seldom influenced by the choice of normalization method, except for the case of using $L1$ normalization.

The resulting feature vector is dependent on the parameters introduced above: the number of pixels per cell, the number of cells per block, and the number of histogram bins. In our experiments, we use the following values (which are also used in [9]): 8×8 pixels per cell, 2×2 cells per block, and 9 bins for each histogram. For normalization method, we use $L2$ normalization.

2.5 Formulation

The adversarial goal is to take any input vector $\mathbf{x} \in \mathfrak{R}^n$ (vectors will be denoted in boldface) and produce a minimally altered version of $\mathbf{x}$, *adversarial sample* denoted by $\mathbf{x}^\star$, that has the property of being misclassified by a classifier $F : \mathfrak{R}^n \to C$. For our proposes, a classifier is a function from $\mathfrak{R}^n$ to C, where C is the set of class labels. Formally, speaking an adversary wishes to solve the following optimization problem:

$$\begin{aligned} \min_{\Delta \in \mathfrak{R}^n} \quad & \mu(\Delta) \\ \text{such that} \quad & F(\mathbf{x} + \Delta) \in C \\ & \Delta \cdot \mathbf{M} = 0 \end{aligned}$$

The various terms in the formulation are μ is a metric on $\mathfrak{R}^n$, $C \subseteq C$ is a subset of the labels, and $\mathbf{M}$ (called the *mask*) is a n-dimensional $0 - 1$ vector of size n. The objective function minimizes the metric μ on the perturbation Δ. Next we describe various constrains in the formulation.

- $F(\mathbf{x} + \Delta) \in C$
  The set C constrains the perturbed vector $\mathbf{x} + \Delta$ [2] to have the label (according to F) in the set C. For *misclassification* problems (the label of $\mathbf{x}$ and $\mathbf{x} + \Delta$ are different we have $C = C - \{F(\mathbf{x})\}$. For *targeted misclassification* we have $C = \{l\}$ (for $l \in C$), where l is the target that an attacker wants (e.g., the attacker wants l to correspond to a yield sign).

- $\Delta \cdot \mathbf{M} = 0$
  The vector M can be considered as a mask (i.e., an attacker can only perturb a dimension i if $M[i] = 0$), i.e., if $M[i] = 1$ then $\Delta[i]$ is forced to be 0.[3] Essentially the attacker can only perturb dimension i if the i-th component of M is 0, which means that δ lies in k-dimensional space where k is the number of non-zero entries in Δ.

- *Convexity*
  Notice that even if the metric μ is convex (e.g., μ is the l_2 norm), because of the constraint involving F the optimization problem is *not convex* (the constraint $\Delta \cdot \mathbf{M} = 0$ is convex). In general, solving convex optimization problems is more tractable non-convex optimization [25].

Generally, machine-learning algorithms use a loss function $\ell(F, \mathbf{x}, y)$ to penalize predictions that are "far away" from the true label $tl(\mathbf{x})$ of $\mathbf{x}$. For example, if we use $0 - 1$ loss function, then $\ell(F, \mathbf{x}, y) = \delta(F(\mathbf{x}), y)$, where $\delta(z, y)$ is equal to 1 iff $z = y$ (i.e., if $z \neq y$, then $\delta(z, y) = 0$). For notational convenience, we will write $L_F(\cdot)$ for $\ell(F, \cdot, \cdot)$, where $L_F(\mathbf{x}) = \ell(F, \mathbf{x}, tl(\mathbf{x}))$. Some classifiers $F(\mathbf{x})$ are of the form $\arg\max_l F_s(\mathbf{x})$ (i.e., the classifier F outputs the label with the maximum probability). For example, in a deep-neural network (DNN) that has a *softmax layer*, the output of the softmax layer

---

[2]The vectors are added component wise
[3]the i-the component of a vector $\mathbf{M}$ is written as $\mathbf{M}[i]$.

is a probability distribution over class labels (i.e., the probability of a label y intuitively means the belief that the classifier has in the example has label y). Throughout the paper, we sometimes refer to the function F_s as the softmax layer. In these case, we will consider the probability distribution corresponding to a classifier. Formally, let $c = |C|$ and F be a classifier, we let F_s be the function that maps $\mathbb{R}^n$ to $\mathbb{R}^c$ such that $\|F_s(\mathbf{x})\|_1 = 1$ for any $\mathbf{x}$ (i.e., F_s computes a probability vector). We denote $F_s^l(\mathbf{x})$ to be the probability of $F_s(\mathbf{x})$ at label l.

2.6 Some Existing Algorithms

In this section, we describe few existing algorithms. This section is not meant to be complete, but simply to give a "flavor" of the algorithms to facilitate the discussion.

Goodfellow et al. attack - This algorithm is also known as the *fast gradient sign method (FGSM)* [13]. The adversary crafts an adversarial sample $\mathbf{x}^\star = \mathbf{x} + \Delta$ for a given legitimate sample $\mathbf{x}$ by computing the following perturbation:

$$\Delta = \varepsilon \, \text{sign}(\nabla L_F(\mathbf{x})) \tag{1}$$

The gradient of the function L_F is computed with respect to $\mathbf{x}$ using sample $\mathbf{x}$ and label $y = tl(\mathbf{x})$ as inputs. Note that $\nabla L_F(\mathbf{x})$ is an n-dimensional vector and $\text{sign}(\nabla L_F(\mathbf{x}))$ is a n-dimensional vector whose i-th element is the sign of the $\nabla L_F(\mathbf{x})[i]$. The value of the *input variation parameter ε* factoring the sign matrix controls the perturbation's amplitude. Increasing its value, increases the likelihood of $\mathbf{x}^\star$ being misclassified by the classifier F but on the contrary makes adversarial samples easier to detect by humans.

Papernot et al. attack - This algorithm is suitable for targeted misclassification [28]. We refer to this attack as JSMA throughout the rest of the paper. To craft the perturbation Δ, components are sorted by decreasing adversarial saliency value. The adversarial saliency value $S(\mathbf{x}, t)[i]$ of component i for an adversarial target class t is defined as:

$$S(\mathbf{x}, t)[i] = \begin{cases} 0 \text{ if } \frac{\partial F_t}{\partial \mathbf{x}[i]}(\mathbf{x}) < 0 \text{ or } \sum_{j \neq t} \frac{\partial F_j}{\partial \mathbf{x}[i]}(\mathbf{x}) > 0 \\ \frac{\partial F_t}{\partial \mathbf{x}[i]}(\mathbf{x}) \left| \sum_{j \neq t} \frac{\partial F_j}{\partial \mathbf{x}[i]}(\mathbf{x}) \right| \text{ otherwise} \end{cases} \tag{2}$$

where matrix $J_F = \left[\frac{\partial F_j}{\partial \mathbf{x}[i]} \right]_{ij}$ is the Jacobian matrix for F_s. Input components i are added to perturbation Δ in order of decreasing adversarial saliency value $S(\mathbf{x}, t)[i]$ until the resulting adversarial sample $\mathbf{x}^\star = \mathbf{x} + \Delta$ achieves the target label t. The perturbation introduced for each selected input component can vary. Greater individual variations tend to reduce the number of components perturbed to achieve misclassification.

Deepfool. The recent work of Moosavi-Dezfooli et al. [23] try to achieve simultaneously the advantages of the methods in [33] and [12]: Being able to find small directions (competitive with [33]), and being fast (competitive with [12]). The result is the *DeepFool* algorithm, which is an iterative algorithm that, starting at $\mathbf{x}_0$, constructs $\mathbf{x}_1, \mathbf{x}_2, \ldots$. Suppose we are at $\mathbf{x}_i$, the algorithm first linearly approximates F_s^k for each $k \in C$ at $\mathbf{x}_i$:

$$F_s^k(\mathbf{x}) \approx F_s^k(\mathbf{x}_i) + \nabla F_s^k(\mathbf{x}_i) \cdot (\mathbf{x} - \mathbf{x}_i)$$

With these approximations, thus at $\mathbf{x}_i$ we want to find the direction $\mathbf{d}$ such that for some $l' \neq l$

$$F_s^{l'}(\mathbf{x}_i) + \nabla F_s^{l'}(\mathbf{x}_i) \cdot \mathbf{d} > F_s^l(\mathbf{x}_i) + \nabla F_s^l(\mathbf{x}_i) \cdot \mathbf{d}$$

In other words, at iteration i let P_i be the polyhedron that

$$P_i = \bigcap_{k \in C} \left\{ \mathbf{d} : F_s^k(\mathbf{x}_i) + \nabla F_s^k(\mathbf{x}_i) \cdot \mathbf{d} \right.$$
$$\left. \leq F_s^l(\mathbf{x}_i) + \nabla F_s^l(\mathbf{x}_i) \cdot \mathbf{d} \right\}$$

Therefore the $\mathbf{d}$ we are seeking for is $\text{dist}(\mathbf{x}_i, P_i^c)$, the distance of $\mathbf{x}_i$ to the complement of P_i is the smallest. Solving this problem *exactly* and iterating until a different label is obtained, results in the DeepFool algorithm.

2.7 Discussion

Several algorithms for crafting adversarial samples have appeared in the literature. These algorithms differ in three dimensions: the choice of the metric μ, the set of target labels $C \subseteq L$, and the mask M. None of these methods use a mask M, so we do not include it in the figure. We describe few attacks according to these dimensions in Figure 2 (please refer to the formulation of the problem at the beginning of the section).

3 OUR ALGORITHM

We now devise new algorithms for crafting adversarial perturbations. Our starting point is similar to Deepfool, and assumes that the classifier $F(\mathbf{x})$ is of the form $\arg \max_l F_s(\mathbf{x})$ and the softmax output F_s is available to the attacker. Suppose that $F(\mathbf{x}_0) = l \in C$, then $F_s^l(\mathbf{x}_0)$ is the largest probability in $F_s(\mathbf{x}_0)$. Note that F_s^l is a scalar function. Our method is to find a small $\mathbf{d}$ such that $F_s^l(\mathbf{x}_0 + \mathbf{d}) \approx 0$. We are implicitly assuming that there is a point $\mathbf{x}'$, *nearby* $\mathbf{x}$, that F "strongly believes" $\mathbf{x}'$ does *not* belong to class l (as the belief probability is "close to 0"), while it believes that $\mathbf{x}$ does. While our assumption is "aggressive", it turns out in our experiments this assumption works quite well in practice.

More specifically, with the above discussion, we now want to decrease the value of the function F_s^l as fast as possible to 0. Therefore, the problem is to solve the equation $F_s^l(\mathbf{x}) = 0$ starting with $\mathbf{x}_0$. The above condition can be easily generalized to the condition if the probability of label l goes below a specified threshold (i.e. $F_s^l(\mathbf{x}) < \epsilon$). We solve this problem based on Newton's method for solving nonlinear equations. Specifically, for $i = 0, 1, 2, \ldots$, suppose at step i we are at $\mathbf{x}_i$, we first approximate F_s^l at $\mathbf{x}_i$ using a linear function

$$F_s^l(\mathbf{x}) \approx F_s^l(\mathbf{x}_i) + \nabla F_s^l(\mathbf{x}_i) \cdot (\mathbf{x} - \mathbf{x}_i). \tag{3}$$

Let $\mathbf{d}_i = \mathbf{x} - \mathbf{x}_i$ be the perturbation to introduce at iteration i, and $p_i = F_s^l(\mathbf{x}_i)$ be the current belief probability of $\mathbf{x}_i$ as in class l. Assuming that we have not found the adversarial example, p_i must assume the largest value at the softmax layer, thus in particular $p_i \geq 1/|C|$. Finally, denote $\mathbf{g}_i = \nabla F_s^l(\mathbf{x}_i)$ be the gradient of F_s^l at $\mathbf{x}_i$. Then our goal is to solve the following linear system for $\mathbf{d}_i$:

$$p_i + \mathbf{g}_i \cdot \mathbf{d}_i = p_{i+1} \tag{4}$$

Method	C	Short Description
FGSM	$C - \{F(\mathbf{x})\}$	Adds a perturbation which is proportional to the sign of the gradient of the loss function at the image $\mathbf{x}$.
Papernot et al. JSMA	$\{l\}$	Constructs a saliency matrix $S(\mathbf{x}, t)$, where $\mathbf{x}$ is the image t is the target label. In each iteration, change the pixel according to the saliency matrix.
Deepfool	$C - \{F(\mathbf{x})\}$	In each iteration, tries to push the probabilities of other labels l' higher than the current label l.

Figure 2: The second column indicates whether the method is suited for targeted misclassification. The last column gives a short description of the method.

where $p_{i+1} < p_i$ is some appropriately chosen probability value we want to achieve in the next iteration. Denote $\delta_i = p_i - p_{i+1}$ be the decrease of probability. Now we are ready to describe the basic version of our new algorithm called *NewtonFool*.

Basic Version of NewtonFool. We start with a simple observation: Given p_{i+1}, the *minimal-norm solution* to (4) is $\mathbf{d}_i^* = \mathbf{g}_i^\dagger (p_{i+1} - p_i) = -\delta_i \, \mathbf{g}_i^\dagger$ where $\mathbf{g}_i^\dagger$ is the Moore-Penrose inverse of $\mathbf{g}_i$ (see the background section for a description of Moore-Penrose inverse). For rank-1 matrix $\mathbf{g}_i$, $\mathbf{g}_i^\dagger$ is precisely $\mathbf{g}_i / \| \mathbf{g}_i \|^2$, and so

$$\mathbf{d}_i^* = -\frac{\delta_i \, \mathbf{g}_i}{\| \mathbf{g}_i \|^2} \qquad (5)$$

Therefore the question left is how to pick p_{i+1}. We make two observations: First, it suffices that $p_{i+1} < 1/|C|$ in order to "fool" the classifier into a different class. Therefore we have an upper bound of δ_i, namely

$$\delta_i < p_i - 1/|C|.$$

Second, note that we want small perturbation, which means that $\| \mathbf{d}_i^* \|$ is small. This can be formalized as $\| \mathbf{d}_i^* \| \leq \eta \| \mathbf{x}_0 \|$ for some parameter $\eta \in (0, 1)$. Since $\| \mathbf{d}_i^* \| = \delta_i / \| \mathbf{g}_i \|$, this thus translates to

$$\delta_i \leq \eta \| \mathbf{x}_0 \| \| \mathbf{g}_i \|$$

Combining the two conditions above we thus have

$$\delta_i \leq \min \left\{ \eta \| \mathbf{x}_0 \| \| \mathbf{g}_i \|, p_i - \frac{1}{|C|} \right\}$$

We can thus pick

$$\delta_i^* = \min \{ \eta \| \mathbf{x}_0 \| \| \mathbf{g}_i \|, p_i - 1/|C| \}. \qquad (6)$$

Plugging this back into (5), we thus get direction

$$\mathbf{d}_i^* = -\delta_i^* \, \mathbf{g}_i / \| \mathbf{g}_i \|^2 \; .$$

This gives the algorithm shown in 1.

Astute readers may realize that this is nothing but gradient descent with step size $\delta_i^* / \| \nabla F_s^l(\mathbf{x}_i) \|^2$. However, there is an important difference: The step size of a typical gradient descent procedure is tuned on a complete heuristic basis. However, in our case, exploiting the structure of softmax layer and our assumption on the vulnerability of the neural network, the step size is *determined* once the intuitively sensible parameter η is fixed (which controls how small the perturbation we want). Note also that the step size changes over time according to

Algorithm 1

Input: $\mathbf{x}_0, \eta$, *maxIter*, a neural network F with a softmax layer F_s.
1: **function** NEWTONFOOLMINNORM($\mathbf{x}_0, \eta, maxIter, F$)
2:      $l \leftarrow F(\mathbf{x}_0)$
3:      $\mathbf{d} \leftarrow \mathbf{0}$
4:      **for** $i = 0, 1, 2, \ldots, maxIter - 1$ **do**
5:          $\delta_i^* \leftarrow \min \left\{ \eta \| \mathbf{x}_0 \| \| \nabla F_s^l(\mathbf{x}_i) \|, F_s^l(\mathbf{x}_i) - \frac{1}{|C|} \right\}$
6:          $\mathbf{d}_i^* \leftarrow -\frac{\delta_i^* \nabla F_s^l(\mathbf{x}_i)}{\| \nabla F_s^l(\mathbf{x}_i) \|^2}$
7:          $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \mathbf{d}_i^*$
8:          $\mathbf{d} \leftarrow \mathbf{d} + \mathbf{d}_i^*$
9:      **return** $\mathbf{d}$

(6): If initially $p_i - 1/|C|$ is too large, the step size is then determined by $\mathbf{g}_i$, the gradient vector we obtained on the current image $\mathbf{x}_i$. Otherwise, as p_i becomes closer and closer to $1/|C|$, we will instead adjust the step size according to $p_i - 1/|C|$.

3.1 Enhancements

Our basic algorithm attempts to drive down the probability of the label l with the maximum probability according F_s. We have extended this algorithm to consider multiple labels, but we have not implemented this enhancement. Our enhancement tries to "drive down" the probability of all labels in a set L_+ and "drive up" the probability of all labels in the set L_- (presumably an adversary wants an adversarial example whose label is in L_-). Our enhanced algorithm is described in the appendix A.

3.2 Possible Benefits in Convergence with Newton's Method

We now give some heuristic arguments regarding the benefits of using Newton's method. The main possible benefit is faster convergence to an adversarial example. At the high level, the main reason under the hood is actually that Newton's method is locally *quadratically convergent* to a zero x^* when solving a non-linear system of equations [30].

To start with, let us first consider an extreme case where, actually, the convergence of Newton's method does not apply directly. Suppose that in a reasonably small neighborhood $N(x, \delta)$ (under some norm), we have x^* so that $F_s^l(x^*) = 0$. That is, x^* is a *perfect* adversarial example where we believe

that it is not label l with probability 1. Note that in this case x^* is also a local minimal of F_s^l because F_s^l is non-negative. In this case, the gradient of F_s^l is however singular (0), so we cannot apply the convergence result of Newton's method to conclude fast convergence to x^*, to give that $F_s^l(x^*) = 0$. However, a crucial point now is that while we will not converge to x^* exactly, at some point in this iterative process, $F_s^l(x_k)$ will be *small enough* so that the network will no longer predict x_k as label l (which is the correct label), and thus gives an adversarial example. If we let k^* be the first iterate such that x_{k^*} is an adversarial example, our goal is thus to argue fast convergence to x_{k^*} (where the gradient exists but not singular).

Suppose now that $F_s^l(x_{k^*}) = p > 0$, then if we instead solve that $F_s^l(x) - p = 0$, then the zero point (for example x_{k^*}) has a non-singular gradient, and so the convergence result of Newton's method applies and says that we will converge quadratically to x_{k^*}. But this is exactly what NewtonFool algorithm is doing, modulo picking p: Starting with the hypothesis that nearby we have a adversarial point with very low confidence in l, we pick a *heuristic p*, and apply Newton's method to solve the equation so that the confidence hopefully decreases to p – If our guess is somewhat accurate, then we will quickly converge to an adversarial point. Of course, the crucial problem left is to pick the p – which we give some sufficient condition (such as $1/|C|$) to get guarantees.

4 EXPERIMENTS

In this section we present an empirical study comparing our algorithms with previous algorithms in producing adversarial examples. Our goals are to examine the tradeoff achieved by these algorithms in their (1) effectiveness in producing "indistinguishable" adversarial perturbations, and (2) efficiency in producing respective perturbations. More specifically, the following are three main questions we aim to answer:

(1) *How effective is NewtonFool in decreasing the confidence probability at the softmax layer for the correct label in order to produce an adversarial example?*
(2) *How is quality of the adversarial examples produced by NewtonFool compared with previous algorithms?*
(3) *How efficient is NewtonFool compared with previous algorithms?*

In summary, our findings are the following:

- (1)+(3) *NewtonFool achieves a better effectiveness-efficiency tradeoff than previous algorithms.* On one hand, NewtonFool is significantly faster than either DeepFool (up to **49X** faster) or JSMA (up to **800X** faster), and is only moderately slower than FGSM. This is not surprising since NewtonFool does not need to examine all classes for complex classification networks with many classes. On the other hand, under the *objective metric of Canny edge detection*, NewtonFool produces competitive and sometimes better adversarial examples when compared with DeepFool, and both of them produce typically significant better adversarial examples than JSMA and FGSM.
- (2) *NewtonFool is not only effective in producing good adversarial perturbations, but also significantly reduces*

the confidence probability of the correct class. Specifically, in our experiments not only NewtonFool successfully finds small adversarial perturbations for all test images we use, but also it does so by significantly reducing the confidence probability, typically from 0.8-0.9 to 0.2-0.4. While previous work have also shown that confidence probability can be significantly reduced with small adversarial perturbations, it is somewhat surprising that a *gradient descent* procedure, *constructed under an aggressive assumption on the vulnerabilities of deep neural networks against adversarial examples*, can achieve similar effects in such a uniform manner.

Two remarks are in order. First, we stress that, different from DeepFool and JSMA where these two algorithms are "best-effort" in the sense that they leverage first-order information from *all* classes in a classification network to construct adversarial examples, NewtonFool exploits an aggressive assumption that, "nearby" the original data point, there is another point where the confidence probability in the "correct class" is significantly lower. The exploitation of this assumption is similar to the structural assumption made by FGSM, yet NewtonFool gives significantly better adversarial examples than FGSM. Second, note that typically the *training* of a neural network is *gradient descent* on the hypothesis space (i.e., parameters of the network) with features (training data points) fixed. In this case, NewtonFool is "dual" in the sense that it is a *gradient descent* on the feature space (i.e., training features) with network parameters fixed. As a result of the above two points, we believe that the success of NewtonFool in producing adversarial examples gives a more explicit demonstration of vulnerabilities of deep neural networks against adversarial examples than previous algorithms.

In the rest of this section we give more details of our experiments. We begin by presenting experimental setup in Section 4.1. Then in Section 4.2 we provide experimental results on the effectiveness of NewtonFool. Finally in Section 4.3 and Section 4.4 we report perturbation quality and efficiency of NewtonFool compared with previous algorithms.

4.1 Experimental Setup

The starting point of our experiments is the cleverhans project by Papernot et al. [27], which implements FGSM, JSMA and a convolutional neural network (CNN) architecture for studying adversarial examples. We reuse their implementations of FGSM and JSMA, and the network architecture in our experiments to ensure fair comparisons. Our algorithms are implemented as extensions of cleverhans[4] using Keras and TensorFlow. We also ported the publicly available implementation of DeepFool [23] to cleverhans. All the experiments are done on a system with Intel Xeon E5-2680 2.50GHz CPUs (48-core), 64GB RAM, and CUDA acceleration using NVIDIA Tesla K40c.

Datasets. In our experiments we considered two datasets.

---

[4]cleverhans is a software library that provides standardized reference implementations of adversarial example construction techniques and adversarial training and can be found at https://github.com/tensorflow/cleverhans.

MNIST. MNIST dataset consists of 60,000 grayscale images of hand-written digit. There are 50,000 images for training and 10,000 images for testing. Each image is of size 28×28 pixels, and there are ten labels $\{0, \ldots, 9\}$ for classification.

GTSRB. GTSRB dataset consists of colored images of traffic signs, with 43 different classes. There are in total 39,209 images for training and 12,630 images for testing. In official GTSRB dataset, image sizes vary from 15×15 to 250×250 pixels, and we use the preprocessed dataset containing all images rescaled to size 32×32.

CNN Architecture. The CNN we used in experiments for both datasets is the network architecture defined in cleverhans. This networks has three consecutive convolutional layers with ReLu activation, followed by a fully connected layer outputting softmax function values for each classes. We train a CNN classifier achieving 94.14% accuracy for MNIST testset after 6-epoch training, and another CNN classifier achieving 86.12% accuracy for GTSRB testset after 100-epoch training.

Experimental Methodology. Since JSMA and DeepFool take too long to finish on the entire MNIST and GTSRB datasets, we use sampling based method for evaluation. More specifically, for MNIST we run ten independent experiments, where in each experiment we randomly choose 50 images, for each of the ten labels in $\{0, \ldots, 9\}$. We do similar things for GTSRB except that we choose 20 images at random for each of the 6 chosen classes among 43 possible classes. Those 6 classes were intentionally chosen to include diverse shapes, different colors, and varying complexity of embedded figures, of traffic signs.

With the sampled images, we then attack each CNN classifier using different adversarial perturbation algorithms , and evaluate the quality of perturbation and efficiency. Specifically:
(1) For NewtonFool, given data input $\mathbf{x}_0$ where it is classified as l, $F_s^l(\mathbf{x}_0) - F_s^l(\mathbf{x}_0 + \mathbf{d})$ in order to evaluate the effectiveness of NewtonFool in reducing confidence probability in class l.
(2) To evaluate the perturbation quality, we use three algorithms (Canny edge detector, FFT, and HOG) to evaluate the quality of the perturbed example.
(3) Finally we record running time of the attacking algorithm in order to evaluate efficiency.

Tuning of Different Adversarial Perturbation Algorithms. Finally, we describe how we tune different adversarial perturbation algorithms in order to ensure a fair comparison.

FGSM. To find the optimal value for ϵ minimizing the final perturbation Δ, we iteratively search from 0 to 10, increasing by 10^{-3}. We use the first ϵ achieving adversarial perturbation to generate the adversarial sample. Searching time for the best ϵ value is also counted towards the running time.

JSMA. Since target label l must be specified in JSMA, we try all possible targets, and then choose the best result that the adversarial example achieves the minimum number of pixel changes. For each trial, we use fixed parameters $\Upsilon = 14.5\%$, $\theta = 1$, which are valued used in [28]. We always increase the pixel values for perturbations. For all target labels, attack times are counted to the running time.

DeepFool. We fix $\eta = 0.02$, which is the value used in [23] to adjust perturbations in each step.

NewtonFool. We fix $\eta = 0.01$, which is the parameter used to determine the step size $\delta_i / \|\nabla F_s^l(\mathbf{x}_i)\|_2$ in gradient descent steps.

4.2 Effectiveness of NewtonFool

This section reports results in evaluating the effectiveness of NewtonFool. We do so by measuring (1) success rate of generating adversarial examples, and (2) changes in CNN classifier's confidence in its classification. Table 1 summarizes results for both. Specifically, column 5 and 6 gives the total number of images we use to test and the total number of successful attacks. In our experiments, NewtonFool achieves perfect success rate. Column 4 gives the success probability if ones makes a uniformly random guess, which is the value NewtonFool algorithm aims to achieve in theory. Column 2 and 3 gives results on the reduction of confidence before and after attacks, respectively, for both MNIST and GTSRB. We see that in practice while the confidence we end at is larger than $1/|C|$, NewtonFool still significantly reduce the confidence at the softmax layer, typically from "almost sure" (0.8-0.9) to "not sure" (0.2-0.4), while succeeding to produce adversarial perturbations.

4.3 Quality of Adversarial Perturbation

Now we evaluate the quality of perturbations using an *objective* metric: Recall that an objective metric measures the quality of perturbation *independent* of the optimization objective, and thus serves a better role in evaluating to what degree a perturbation is "indistinguishable.". Specifically, in our experiments we use the classic techniques of computer vision, as the objective metrics: *Canny edge detection*, *fast Fourier transform*, and *histogram of oriented gradients*.

Given an input image, Canny edge detection finds a wide range of edges in an image, which is a critical task in computer vision. We use Canny edge detection in the following way: We count the *number of edges* detected by the detector and use that as a metric for evaluation. Intuitively, an indistinguishable perturbation should maintain the number of edges after perturbation very close to the original image. Therefore, by measuring how close this count is to the count on the original image, we have a metric on the quality of the perturbation: Smaller the metric, better the perturbation. Note that there are many edge detection methods. We chose Canny edge detection since it is one of the most popular and reliable edge detection algorithms.

Discrete Fourier transform maps images to the two dimensional spartial domain, allowing us to analyse the perturbations according to their spectra. When analysing an image with its spectrum of spatial frequencies, higher frequency corresponds to feature details and abrupt change of values, whereas lower frequency corresponds to global shape information. As adversarial perturbations does not change the general shape but corrupt detailed features of the input image, measuring the size of the high frequency part of the spectrum is desirable as the metric for feature corruption. Therefore, we first

Dataset	Confidence before attack	Confidence after attack	$1/|C|$	Number of attacked samples	Number of successful attacks
MNIST	0.926 (0.135)	0.400 (0.065)	0.1	5000	5000
GTSRB	0.896 (0.201)	0.245 (0.102)	0.023	1200	1200

Table 1: Confidence reduction and success rate of NewtonFool. Column 2 and 3 gives results on the reduction of confidence before and after attacks. Column 4 gives the success probability if one makes a uniformly random guess, which is the value NewtonFool algorithm aims to achieve in theory. Column 5 and 6 gives the success rate results.

compute the Fourier transform of the perturbation, discard the values lies in the low frequency part, then measure the l_2 norm difference of the remaining part as a metric: Smaller metric implies that less feature corruption has been induced by the adversarial perturbation. For the low frequency part to be discarded, we chose the intersection of lower halves of the frequency domain along each dimensions (horizontal and vertical).

Object detection with HOG descriptor is done by sweeping a large image with a fixed size window, computing the HOG descriptor of the image restricetd in the window, and using a machine learning algorithm (e.g. SVM) to classify the descriptor vector as "detected" and "not detected". That is, if a perturbed image has HOG descriptor relatively close to the HOG descriptor of the original image, then the detecting algorithm is more likely to detect the perturbed image whenever the original image is detected. From this, we suggest another objective metric that measures the l_2 norm difference between two HOG descriptor vectors: One computed from the original image and the other from the perturbed image. The perturbations resulting smaller HOG vector difference are considered to have better quality.

In summary, in our experiments we find that: (1) Among all algorithms DeepFool and NewtonFool generally produce the best results, and typically are significantly better than FGSM and JSMA. (2) Further, DeepFool and NewtonFool are *stable* on both datasets and both of them give good results on both datasets. On the other hand, while FGSM performs relatively well on MNIST, it gives poor results on GTSRB, and JSMA performs the opposite: It performs well on GTSRB, but not on MNIST. (3) Finally, for all tests we have, NewtonFool gives competitive and sometimes better results than DeepFool. In the rest of this section we give detailed statistics. Results for HOG are provided in the appendix B.

Table 2 and Table 3 reports the number of edges we find on MNIST and GTSRB, respectively. Each column gives the results on the class represented by the image at the top, and each row presents the mean and standard deviation of the number of detected edges. Specifically, the first row gives the statistics on the original image, and the other rows gives the statistics on the adversarial examples found by different adversarial perturbation algorithms.

On MNIST (Table 2), FGSM, DeepFool and NewtonFool give similar results, where the statistics is very close to the statistics on the original images. DeepFool and NewtonFoolare especially close. On the other hand, we note that JSMA produces significantly more edges compared to other methods.

The result suggests that perturbations from JSMA can be more perceivable than perturbations produced by other algorithms.

On GTSRB dataset, the situation changes a little. Now FGSM produces the worst results among all algorithms (instead of JSMA), and it produces significantly more edges been detected on its produced adversarial examples. While the quality of adversarial examples produced by JSMA improves significantly, it is still slightly worse than those produced by DeepFool and NewtonFool in most cases (except for the last sign). Interestingly, NewtonFool gives the best result for all signs (again, except the last one).

Table 4 and Table 5 presents the distance on the domain of high spatial frequency we computed on MNIST and GTSRB, respectively. Again, each column gives the results on the class represented by the image at the top, and each row presents the mean and standard deviation of the hight frequency distances. For each row, we first generated adversarial perturbations using the algorithm of the row, computed the fast Fourier transform of the perturbation, then computed the l_2 norm of the high spatial frequency part.

On both of MNIST dataset and GTSRB dataset, DeepFool and NewtonFool produce very close results, achieving smaller values than the other two algorithms. On the other hand, the statistics on FGSM and JSMA are relatively worse. Especially, JSMA changes the high frequency part significantly more. From this result, perturbations from JSMA seem to corrupt more feature details than other algorithms. Remarkably, NewtonFool maintains the best result over all experiments on our Fourier transform metric.

4.4 Efficiency

In the final part of our experimental study we evaluate the efficiency of different algorithms. We measure the end-to-end time to produce adversarial examples. In short, we find that NewtonFool is substantially faster than either DeepFool (up to **49X**) or JSMA (up to **800X**), while being only moderately slower than FGSM. This is not surprising because NewtonFool does not need to examine first-order information for *all* classes of a complex classification network for many classes. However, it is somewhat surprising that, NewtonFool, a *gradient procedure* exploiting an *aggressive assumption on* the vulnerability of CNN, achieves substantially better efficiency while producing competitive and often times better adversarial examples. Due to space limitations detailed results are provided in the appendix B.

	0	1	2	3	4
Original image	1.71 (0.49)	1.07 (0.35)	1.36 (0.59)	1.17 (0.52)	1.22 (0.53)
FGSM	1.62 (0.54)	1.07 (0.34)	1.37 (0.66)	1.20 (0.55)	1.25 (0.57)
JSMA	2.34 (0.84)	2.39 (1.30)	2.00 (1.01)	1.69 (0.96)	1.67 (0.82)
DeepFool	1.60 (0.53)	1.06 (0.31)	1.34 (0.61)	1.17 (0.53)	1.26 (0.60)
NewtonFool	1.62 (0.54)	1.07 (0.34)	1.34 (0.61)	1.19 (0.55)	1.25 (0.59)

	5	6	7	8	9
Original image	1.22 (0.52)	1.78 (0.78)	1.08 (0.34)	2.23 (0.90)	1.56 (0.62)
FGSM	1.23 (0.51)	1.76 (0.78)	1.12 (0.38)	2.20 (0.90)	1.57 (0.68)
JSMA	1.97 (1.08)	2.47 (1.21)	1.87 (1.07)	2.95 (1.21)	2.21 (0.97)
DeepFool	1.23 (0.53)	1.75 (0.81)	1.11 (0.38)	2.14 (0.91)	1.53 (0.63)
NewtonFool	1.23 (0.55)	1.75 (0.81)	1.11 (0.38)	2.15 (0.91)	1.54 (0.64)

Table 2: MNIST: Results with Canny edge detection. FGSM, DeepFool and NewtonFool give close results, with NewtonFool being especially close with DeepFool. JSMA, on the other hand, produces significantly larger statistics with respect to the Canny edge detection metric.

Original image	13.73 (6.65)	17.85 (6.84)	11.69 (6.00)	18.14 (7.41)	10.37 (5.42)	13.66 (6.28)
FGSM	18.88 (7.71)	22.63 (7.76)	19.11 (7.56)	20.97 (8.01)	17.89 (7.92)	20.45 (9.46)
JSMA	15.04 (6.62)	18.54 (6.69)	12.95 (5.86)	19.18 (7.83)	11.20 (5.53)	14.71 (6.24)
DeepFool	14.84 (6.61)	18.82 (7.34)	12.96 (5.67)	18.40 (7.03)	11.16 (5.30)	15.06 (7.13)
NewtonFool	14.68 (6.88)	18.34 (7.11)	12.81 (5.73)	18.37 (7.25)	10.62 (5.20)	15.02 (7.08)

Table 3: GTSRB: Results with Canny edge detection. Now JSMA, DeepFool and NewtonFool give close results, while FGSM produces significantly worse results. NewtonFool gives the best results across all tests (except for the last sign).

	0	1	2	3	4
FGSM	20.50 (8.13)	13.01 (3.38)	15.96 (8.56)	13.56 (7.72)	11.95 (6.64)
JSMA	44.19 (7.29)	50.28 (8.37)	44.73 (8.53)	42.00 (9.24)	36.60 (6.52)
DeepFool	10.26 (3.57)	5.14 (1.13)	8.88 (4.33)	6.78 (3.49)	5.98 (2.73)
NewtonFool	9.57 (3.53)	4.62 (1.08)	8.26 (4.11)	6.17 (3.30)	5.39 (2.69)

	5	6	7	8	9
FGSM	12.57 (6.70)	15.33 (6.07)	15.79 (7.95)	11.99 (6.52)	10.43 (5.11)
JSMA	42.37 (9.04)	48.67 (9.42)	45.07 (8.69)	49.44 (11.69)	44.41 (9.35)
DeepFool	6.39 (3.11)	8.54 (3.17)	7.37 (3.09)	7.42 (3.82)	6.70 (3.01)
NewtonFool	5.95 (2.96)	7.83 (3.01)	6.60 (2.91)	6.76 (3.62)	5.96 (2.80)

Table 4: MNIST: Results with fast Fourier transform. DeepFool and NewtonFool give close results. FGSM and JSMA produce worse results with JSMA being significantly larger. NewtonFool produces the best results for all labels.

5 LIMITATIONS AND FUTURE WORK

We have not implemented the enhancement to our algorithm described in the appendix. In the future, we will implement the enhancement and compare its performance to our basic algorithm. The computer-vision research literature has several algorithms for analyzing images, such as segmentation, edge detection, and deblurring. In our evaluation we use one algorithm (i.e., edge detection) as a metric. Incorporating other computer-vision algorithms into a metric is a very interesting avenue for future work. For example let $\mu_1, \mu_2, \cdots, \mu_k$ be k metrics (e.g., based on number of pixels, edges, and segments). We could consider a weighted metric (i.e. the difference between two images x_1 and x_2 is $\sum_{i=1}^{k} w_i \mu_i(x_1, x_2)$). The question remains – what weights to choose? Perhaps mechanical turk

FGSM	46.86 (36.94)	37.50 (15.97)	44.34 (27.72)	37.81 (27.04)	15.12 (10.47)	50.84 (22.82)
JSMA	54.95 (28.17)	66.81 (18.04)	79.87 (23.89)	72.62 (29.64)	56.14 (15.56)	78.91 (24.02)
DeepFool	9.67 (7.18)	7.33 (2.70)	8.07 (3.96)	7.88 (6.43)	4.38 (1.21)	8.78 (4.41)
NewtonFool	9.02 (7.66)	6.16 (2.54)	6.84 (3.84)	6.93 (6.12)	3.49 (0.86)	7.95 (4.34)

Table 5: GTSRB: Results with fast Fourier transform. Again, DeepFool and NewtonFool give close results, while FGSM and JSMA produce worse results. Across all tests, NewtonFool achieves the best results.

studies can be used to train the appropriate weights. Moreover, the weighted metric $\sum_{i=1}^{k} w_i \mu_i(\mathbf{x}_1, \mathbf{x}_2)$ could also be incorporated into algorithms for constructing adversarial examples. All these directions are very important avenues for future work.

6 CONCLUSION

This paper presented a gradient-descent based algorithm for finding adversarial examples. We also presented edge detectors as a way for evaluating the quality adversarial examples generated by different algorithms. The research area of crafting adversarial examples is very active. On the other hand, metrics for evaluating the quality of adversarial examples has not been well studied. We believe that incorporating computer-vision algorithms into metrics is worthwhile goal and worthy of further research. Moreover, having a diverse set of algorithms for crafting adversarial examples is very important towards a thorough evaluation of proposed defenses.

ACKNOWLEDGMENTS

REFERENCES

[1] DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[2] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. 2015. Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning. *Nature biotechnology* (2015).

[3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. 2016. *End to End Learning for Self-Driving Cars.* Technical Report.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016). http://arxiv.org/abs/1604.07316.

[5] J Canny. 1986. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 8, 6 (June 1986), 679–698. https://doi.org/10.1109/TPAMI.1986.4767851

[6] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy*.

[7] Chih-Lin Chi, W. Nick Street, Jennifer G. Robinson, and Matthew A. Crawford. 2012. Individualized Patient-centered Lifestyle Recommendations: An Expert System for Communicating Patient Specific Cardiovascular Risk Information and Prioritizing Lifestyle Options. *J. of Biomedical Informatics* 45, 6 (Dec. 2012), 1164–1174.

[8] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. 2013. Large-scale malware classification using random projections and neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 3422–3426.

[9] Navneet Dalal and Bill Triggs. 2005. Histograms of Oriented Gradients for Human Detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CV PR'05) - Volume 1 - Volume 01 (CVPR '05)*. IEEE Computer Society, Washington, DC, USA, 886–893. https://doi.org/10.1109/CVPR.2005.177

[10] Nathan Eddy. 2016. AI, Machine Learning Drive Autonomous Vehicle Development. http://www.informationweek.com/big-data/big-data-analytics/ai-machine-learning-drive-autonomous-vehicle-development/d/d-id/1325906. (2016).

[11] Leslie Hogben (Editor). 2013. *Handbook of Linear Algebra*. Chapman and Hall/CRC.

[12] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and Harnessing Adversarial Examples. *CoRR* (2014).

[13] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proceedings of the 2015 International Conference on Learning Representations*. Computational and Biological Learning Society.

[14] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.

[15] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. 2011. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*. ACM, 43–58.

[16] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. 2017. Safety Verification of Deep Neural Networks.

[17] International Warfarin Pharmacogenetic Consortium. 2009. Estimation of the Warfarin Dose with Clinical and Pharmacogenetic Data. *New England Journal of Medicine* 360, 8 (2009), 753–764.

[18] K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. 2016. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*.

[19] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. 2017. An Efficient SMT Solver for Verifying Deep Neural Networks.

[20] Eric Knorr. 2015. How PayPal beats the bad guys with machine learning. http://www.infoworld.com/article/2907877/machine-learning/how-paypal-reduces-fraud-with-machine-learning.html. (2015).

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[22] A. Kurakin, I. J. Goodfellow, and S. Bengio. 2016. Adversarial Examples in the Physical world. (2016).

[23] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2015. DeepFool: a simple and accurate method to fool deep neural networks. *CoRR* (2015).

[24] Seyed Mohsen Moosavi Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2017. Universal adversarial perturbations. In *Proceedings of 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[25] Jorge Nocedal and Stephen Wright. 2006. *Numerical Optimization*. Springer.

[26] NVIDIA. 2015. NVIDIA Tegra Drive PX: Self-Driving Car Computer. (2015). http://www.nvidia.com/object/drive-px.html

[27] Nicolas Papernot, Ian Goodfellow, Ryan Sheatsley, Reuben Feinman, and Patrick McDaniel. 2016. cleverhans v1.0.0: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768* (2016).

[28] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*. arXiv preprint arXiv:1511.07528.

[29] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP 2014)* 12 (2014), 1532–1543.

[30] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. 2000. *Numerical mathematics*. p.307.

[31] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*. 611–626.

[32] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. 2012. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks* 0 (2012), –. https://doi.org/10.1016/j.neunet.2012.02.016

[33] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *CoRR* abs/1312.6199 (2013).

A EXTENSION

A.1 Problem Formulation

In general, given a point $\mathbf{x}_0$, we can consider two disjoint sets of labels L_+ and L_- as follows.

- $L_+ = \{l_1^+, \ldots, l_m^+\}$ is the set of m labels achieving high confidences $F_s^l(\mathbf{x}_0)$ of the classifier
- $L_- = \{l_1^-, \ldots, l_n^-\}$ is the set of n labels achieving low confidences $F_s^l(\mathbf{x}_0)$ of the classifier

The attacker tries to decrease (resp. increase) all F_s values of labels in L_+ (resp. in L_-). Specifically, for a given point $\mathbf{x}_0$ and a pair of parameters α, β, the attacker tries to find a sample $\mathbf{x}$ nearby $\mathbf{x}_0$ such that,

- For all label l in L_+, $F_s^l(\mathbf{x}) < \alpha$.
- For all label l in L_-, $F_s^l(\mathbf{x}) > \beta$.

Formally, the attack can be formulated with the following optimization problem.

$$
\begin{aligned}
\text{minimize} \quad & \|\mathbf{d}\| \\
\text{subject to} \quad & F_s^l(\mathbf{x}_0 + \mathbf{d}) < \alpha \text{ for all } l \in L_+ \\
& F_s^l(\mathbf{x}_0 + \mathbf{d}) > \beta \text{ for all } l \in L_-
\end{aligned}
$$

A.2 Generalizing the Solution of 3

Again, let $\mathbf{d}_i = \mathbf{x} - \mathbf{x}_i$ be the perturbation to introduce at iteration i, and let $p_i^{(l)} = F_s^l(\mathbf{x}_i)$ be the current belief probability of $\mathbf{x}_i$ as in class l. Similarly, denote $\mathbf{g}_i^{(l)} = \nabla F_s^l(\mathbf{x}_i)$ be the gradient of F_s^l at $\mathbf{x}_i$. Finally, denote $\delta_i^{(l)} = p_i^{(l)} - p_{i+1}^{(l)}$ be the decrease of probability in class l.

From equation 4, for each label $l \in L_+ \cup L_-$, our perturbation $\mathbf{d}_i$ should satisfy the following equation,

$$
\mathbf{g}_i^{(l)} \cdot \mathbf{d}_i = p_{i+1}^{(l)} - p_i^{(l)} = -\delta_i^{(l)} \tag{7}
$$

In other words, $\mathbf{d}_i$ is a solution to a linear system consisting of $(m + n)$ equations. For notational convenience, we assign numbers $1, 2, \ldots, m + n$ to each label $l \in L_+ \cup L_-$. Then, we can define a matrix G_i whose jth row is $\mathbf{g}_i^{(j)}$, and a vector δ_i whose jth element is $-\delta_i^{(j)}$, for $j = 1, 2, \ldots, m + n$.

$$
G_i = \begin{pmatrix} \mathbf{g}_i^{(1)} \\ \mathbf{g}_i^{(2)} \\ \vdots \\ \mathbf{g}_i^{(m+n)} \end{pmatrix} \quad \delta_i = \begin{bmatrix} -\delta_i^{(1)} \\ -\delta_i^{(2)} \\ \vdots \\ -\delta_i^{(m+n)} \end{bmatrix} \tag{8}
$$

Now, we have the following linear system determining $\mathbf{d}_i$.

$$
G_i \cdot \mathbf{d}_i = \delta_i \tag{9}
$$

In usual, $\mathbf{g}_i^{(j)}$'s are gradient vectors in a high dimensional vector space, and $(m + n)$ is much smaller than the dimension. Therefore, the linear system (9) is underdetermined. For an underdetermined linear system, if G_i is of full rank and $\delta_i^{(j)}$'s are fixed for all j, then the min-norm solution $\mathbf{d}_i^*$ can be computed as,

$$
\mathbf{d}_i^* = G_i^\dagger \cdot \delta_i \tag{10}
$$

where $G_i^\dagger$ is the Moore-Penrose pseudoinverse of G_i. This solution $\mathbf{d}_i^*$ satisfies all constraints of the form (7) for each label l, and achieves the minimum norm among vectors in the solution set. In this section, we assume that G_i is always of full rank, and the solution to the other case will be discussed at §A.3. As long as G_i is of full rank, $G_i^\dagger$ can be defined, so the only remaining task to compute (10) is to determine each values in δ_i.

A.2.1 Trivial Constraints on δ_i. For any lable $l \in L_+ \cup L_-$, its corresponding $\delta_i^{(l)}$ should satisfy the following condition.

- If $l \in L_+$, we want to decrease $p_i^{(l)}$, so $\delta_i^{(l)} = p_i^{(l)} - p_{i+1}^{(l)} \geq 0$
- If $l \in L_-$, we want to increase $p_i^{(l)}$, so $\delta_i^{(l)} = p_i^{(l)} - p_{i+1}^{(l)} \leq 0$

Also,

- If $l \in L_+$, it suffices to get $p_{i+1}^{(l)} < \alpha$, so we get an upper bound of the decrease $\delta_i^{(l)}$,

$$
\delta_i^{(l)} \leq \max\{0, p_i^{(l)} - \alpha\}
$$

- If $l \in L_-$, it suffices to get $p_{i+1}^{(l)} > \beta$, so we get an upper bound of the increase $-\delta_i^{(l)}$,

$$-\delta_i^{(l)} \leq \max\{0, \beta - p_i^{(l)}\}$$

Note that the max fucntion is used to ensure no changes for the labels which the adversarial goal is already achieved. Therefore, for labels that allow more changes, we have the following ranges that $\delta_i^{(l)}$'s can be adjusted.

$$\begin{cases} 0 \leq \delta_i^{(l)} \leq p_i^{(l)} - \alpha & \text{for } l \in L_+ \\ p_i^{(l)} - \beta \leq \delta_i^{(l)} \leq 0 & \text{for } l \in L_- \end{cases} \tag{11}$$

A.2.2 Enforcing the Norm Bound of $\mathbf{d}_i^*$. In this section, we consider how to ensure small perturbation in each iteration, i.e. $\| \mathbf{d}_i^* \| \leq \eta \| \mathbf{x}_0 \|$. From the minimal-norm solution $\mathbf{d}_i^* = G_i^\dagger \cdot \delta_i$, we have the following inequality.

$$\| \mathbf{d}_i \| = \| G_i^\dagger \cdot \delta_i \| \leq \| G_i^\dagger \| \| \delta_i \| \tag{12}$$

Note that if $G_i = \mathbf{g}_i^{(1)}$ has only one row and $\delta_i = -\delta_i^{(1)}$ is a scalar value, then equality is satisfied, so we have $\| \mathbf{d}_i \| = \frac{|\delta_i|}{\| \mathbf{g}_i \|}$, which corresponds to the norm of the previous minimal-norm solution 5. However, the equality may not hold for (12) in general.

Let $G_i = U\Sigma V^\top$ be the singular value decomposition of G_i. It is known that $G_i^\dagger = V\Sigma^\dagger U^\top$, so the 2-norm of $G_i^\dagger$ is $\| G_i^\dagger \| = \| \Sigma^\dagger \| = \frac{1}{\sigma_{\min}}$ where $\sigma_{\min}$ is the smallest singluar value of G_i. Combining with (12),

$$\| \mathbf{d}_i \| \leq \frac{\| \delta_i \|}{\sigma_{\min}}$$

To enforce the norm bound of $\mathbf{d}^*$, we can set the 2-norm constraint of δ_i as follows.

$$\| \delta_i \| \leq \eta \sigma_{\min} \| \mathbf{x}_0 \|$$

To get constraints for individual $\delta_i^{(j)}$, we strengthen the restriction further using the relationship between 2-norm and ∞-norm.

$$\| \delta_i \| \leq \sqrt{|L_+ \cup L_-|} \max_{l \in L_+ \cup L_-} |\delta_i^{(l)}| \leq \eta \sigma_{\min} \| \mathbf{x}_0 \|$$

With $|L_+ \cup L_-| = m + n$, we get the following constraints that ensures the norm bound of $\mathbf{d}_i^*$.

$$|\delta_i^{(l)}| \leq \frac{\eta \sigma_{\min} \| \mathbf{x}_0 \|}{\sqrt{m + n}} \quad \text{for each label } l \in L_+ \cup L_- \tag{13}$$

Combining the bounds (11) and (13), we get the following rules to set $\delta_i^{(l)}$ for label l at iteration i. These rules are designed to maximize the amount of possible change, i.e. $|\delta_i^{(l)}|$, while satisfying (11) and (13).

- If $l \in L_+$,
  - If $p_i^{(l)} < \alpha$, we don't need to decrease this probability. Set $\delta_i^{(l)} = 0$
  - If not, choose $\delta_i^{(l)} = \min\{p_i^{(l)} - \alpha, \frac{\eta \sigma_{\min} \| \mathbf{x}_0 \|}{\sqrt{m+n}}\}$
- If $l \in L_-$,
  - if $p_i^{(l)} > \beta$, we don't need to increase this probability. Set $\delta_i^{(l)} = 0$

- If not, choose $\delta_i^{(l)} = \max\{p_i^{(l)} - \beta, -\frac{\eta \sigma_{\min} \| \mathbf{x}_0 \|}{\sqrt{m+n}}\}$

A.3 What if G_i has some linearly dependent rows?

In usual, G_i is not guaranteed to be a full-rank matrix, and may contain linearly dependent rows, making it impossible to define $G_i^\dagger$. However, we can control the values of $\delta_i^{(l)}$ to resolve the dependency. In this section, we will present how to remove dependencies by setting $\delta_i^{(l)}$.

Without loss of generality, assume we have two dependent rows in G_i, correnponding to labels l_1 and l_2. Then, there is a nonzero constant c such that $\mathbf{g}_i^{(l_1)} = c \cdot \mathbf{g}_i^{(l_2)}$. One naïve choice to remove dependecy is just setting $\delta_i^{(l_1)} = \delta_i^{(l_2)} = 0$. Then, the constraints $\mathbf{g}_i^{(l_1)} \cdot \mathbf{d} = 0$ and $\mathbf{g}_i^{(l_2)} \cdot \mathbf{d} = 0$ become equivalent, thus removing one of the dependent rows from G_i does not change the solution set. Then, the algorithm proceeds with no changes of probabilities in those labels, hoping that $\mathbf{g}_{i+1}^{(l_1)}$ and $\mathbf{g}_{i+1}^{(l_2)}$ will become linearly independent at the next iteration.

The better solution is to make use of the constant factor c to define another constraint between $\delta_i^{(l_1)}$ and $\delta_i^{(l_2)}$. In previous setting of two dependent labels, if we set $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$, then the constraints $\mathbf{g}_i^{(l_1)} \cdot \mathbf{d} = \delta_i^{(l_1)}$ and $\mathbf{g}_i^{(l_2)} \cdot \mathbf{d} = \delta_i^{(l_2)}$ are equivalent upto multiplying constant c, so it is again safe to ignore one dependent row from G_i as it does not change the solution set.

Now, we summarize how to set $\delta_i^{(l)}$'s for two labels l_1 and l_2 to remove dependencies, while the added constraint is satisfied. There are several cases depending on the membership of labels l_1, l_2 and the sign of c.

(1) $l_1, l_2 \in L_+$ (resp. $l_1, l_2 \in L_-$)
  (a) If $c > 0$,
    - If $p_i^{(l_1)} < \alpha$ (resp. $p_i^{(l_1)} > \beta$), then $\delta_i^{(l_1)} = 0$ by our rules at §A.2.2. To satisfy the dependency condition $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$, we set $\delta_i^{(l_1)} = \delta_i^{(l_2)} = 0$
    - If $p_i^{(l_2)} < \alpha$ (resp. $p_i^{(l_2)} > \beta$), then $\delta_i^{(l_2)} = 0$ by our rules at §A.2.2. To satisfy the dependency condition $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$, we set $\delta_i^{(l_1)} = \delta_i^{(l_2)} = 0$
    - If both probabilities can be changed, we first set the $\delta_i^{(l)}$ with larger change $|\delta_i^{(l)}|$ possible while satisfying the constraint, then use $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$, to determine the other value.
  (b) If $c < 0$,
    - $\text{sign}(\delta_i^{(l_1)}) = \text{sign}(\delta_i^{(l_2)})$. To satisfy $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$ with $c < 0$, set $\delta_i^{(l_1)} = \delta_i^{(l_2)} = 0$
(2) $l_1 \in L_+, l_2 \in L_-$ (resp. $l_1 \in L_-, l_2 \in L_+$)
  (a) If $c > 0$,
    - $\text{sign}(\delta_i^{(l_1)}) \neq \text{sign}(\delta_i^{(l_2)})$. To satisfy $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$ with $c > 0$, set $\delta_i^{(l_1)} = \delta_i^{(l_2)} = 0$
  (b) If $c < 0$,
    - If $p_i^{(l_1)} < \alpha$ (resp. $p_i^{(l_1)} > \beta$), then $\delta_i^{(l_1)} = 0$ by our rules at §A.2.2. To satisfy the dependency condition $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$, we set $\delta_i^{(l_1)} = \delta_i^{(l_2)} = 0$

- If $p_i^{(l_2)} > \beta$ (resp. $p_i^{(l_2)} < \alpha$), then $\delta_i^{(l_2)} = 0$ by our rules at §A.2.2. To satisfy the dependency condition $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$, we set $\delta_i^{(l_1)} = \delta_i^{(l_2)} = 0$
- If both probabilities can be changed, we first set the $\delta_i^{(l)}$ with larger change $|\delta_i^{(l)}|$ possible while satisfying the constraint, then use $\delta_i^{(l_1)} = c \cdot \delta_i^{(l_2)}$, to determine the other value.

The following pseudocode incorporates the rules in §A.2.2 and §A.3.

Algorithm 2

Input: $\mathbf{x}_0, \eta, maxIter, \alpha, \beta, L_+, L_-$, a neural network F with a softmax layer F_s.

1: **function**   NEWTONFOOLGENERAL-IZED($\mathbf{x}_0, \eta, maxIter, \alpha, \beta, L_+, L_-, F$)
2:      $m \leftarrow |L_+|$
3:      $n \leftarrow |L_-|$
4:      $\mathbf{d} \leftarrow \mathbf{0}$
5:      **for** $i = 0, 1, 2, \dots, maxIter - 1$ **do**
6:          Compute $\mathbf{g}_i^{(l)} = \nabla F_s^l(\mathbf{x}_i)$ for all $l \in L_+ \cup L_-$
7:          Construct G_i as defined in (8)
8:          Remove dependencies of G_i using the tricks in §A.3, record added constraints on $\delta_i^{(l)}$'s
9:          Find the smallest singular value $\sigma_{\min}$ of G_i
10:         **for** $l \in L_+$ **do**
11:            **if** $p_i^{(l)} > \alpha$ **then**
12:              $\delta_i^{(l)} \leftarrow \min\{p_i^{(l)} - \alpha, \frac{\eta \sigma_{\min} \|\mathbf{x}_0\|}{\sqrt{m+n}}\}$
13:            **else**
14:              $\delta_i^{(l)} \leftarrow 0$
15:         **for** $l \in L_-$ **do**
16:            **if** $p_i^{(l)} < \beta$ **then**
17:              $\delta_i^{(l)} \leftarrow \max\{p_i^{(l)} - \beta, -\frac{\eta \sigma_{\min} \|\mathbf{x}_0\|}{\sqrt{m+n}}\}$
18:            **else**
19:              $\delta_i^{(l)} \leftarrow 0$
20:         Check if the constraints added at step 8 are all satisfied. If not, adjust one of two $\delta_i^{(l)}$, in the way that $|\delta_i^{(l)}|$ always decreases.
21:         $\mathbf{d}_i^* \leftarrow G_i^\dagger \cdot \delta_i$
22:         $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \mathbf{d}_i^*$
23:         $\mathbf{d} \leftarrow \mathbf{d} + \mathbf{d}_i^*$
24:      **return** $\mathbf{d}$

B ADDITIONAL EXPERIMENT

B.1 Results for HOG

Table 6 and Table 7 shows the distances between the HOG feature vectors of the original image and the perturbed image, on MNIST and GTSRB, respectively. Again, each column gives the results on the class represented by the image at the top, and each row presents the mean and standard deviation. To be specific, for each row, adversarial examples were generated by the designated algorithm, then their HOG feature vectors were compared to those of the corresponding orignal images.

On MNIST dataset, FGSM gives the best results, while Deep-Fool and NewtonFool give similar values in general with an exception of the second label (digit 1). JSMA produces bigger statistics except for the fifth label (digit 4), which implies that the perturbed image is less likely to be detected when the original image is detected correctly.

On GTSRB dataset, the situation changes entirely that now JSMA produces noticeably better results. Among the other three algorithms, NewtonFool gives the best results for all experiments, even achieving the smaller or equal results to that of JSMA for the last three labels. Similarly to the edge detection metric, FGSM becomes much worse for GTSRB dataset, whereas JSMA generates relatively better perturbations than its output on MNIST dataset.

B.2 Efficiency

We now give detailed statistics. Table 8 and Table 9 gives the running time results for MNIST and GTSRB, respectively. Each column corresponds to each class represented by the image at top, and each row presents the mean and standard deviation of the running time. For MNIST, (Table 8), NewtonFool is only moderately slower than FGSM (if one calculates the absolute difference in seconds). On the other hand, NewtonFool is substantially faster than DeepFool (up to **3X**) and JSMA (up to **6X**). The efficiency becomes more obvious on GTSRB dataset because it has significantly more classes. In this case, we observe that NewtonFool is up to **49X** faster than DeepFool, and is up to **800X** faster than JSMA.

	0	1	2	3	4
FGSM	0.47 (0.17)	0.50 (0.16)	0.44 (0.20)	0.38 (0.17)	0.40 (0.17)
JSMA	0.57 (0.17)	0.85 (0.18)	0.82 (0.29)	0.52 (0.17)	0.44 (0.14)
DeepFool	0.51 (0.19)	0.86 (0.38)	0.47 (0.21)	0.42 (0.18)	0.48 (0.20)
NewtonFool	0.49 (0.19)	0.83 (0.37)	0.46 (0.21)	0.40 (0.17)	0.47 (0.20)

	5	6	7	8	9
FGSM	0.39 (0.16)	0.40 (0.15)	0.46 (0.18)	0.37 (0.15)	0.34 (0.14)
JSMA	0.48 (0.16)	0.51 (0.17)	0.65 (0.21)	0.55 (0.19)	0.46 (0.16)
DeepFool	0.39 (0.16)	0.43 (0.16)	0.50 (0.19)	0.40 (0.15)	0.39 (0.16)
NewtonFool	0.38 (0.16)	0.42 (0.17)	0.47 (0.18)	0.38 (0.16)	0.37 (0.16)

Table 6: MNIST: Results with histogram of oriented gradients. FGSM, DeepFool and NewtonFool give close results, except for the second label (for digit 1) where JSMA, DeepFool and NewtonFool give close results. JSMA generally produces slightly larger statistics except for the fifth label (for digit 4).

FGSM	1.13 (0.57)	0.88 (0.21)	0.86 (0.20)	0.64 (0.29)	0.72 (0.28)	0.94 (0.23)
JSMA	0.42 (0.28)	0.57 (0.25)	0.69 (0.22)	0.48 (0.29)	0.99 (0.44)	0.87 (0.24)
DeepFool	0.96 (0.53)	0.85 (0.22)	0.96 (0.31)	0.56 (0.24)	0.96 (0.41)	0.89 (0.21)
NewtonFool	0.80 (0.48)	0.71 (0.18)	0.75 (0.21)	0.48 (0.19)	0.62 (0.23)	0.81 (0.19)

Table 7: GTSRB: Results with histogram of oriented gradients. Now JSMA gives better results than others. FGSM, DeepFool and NewtonFool produce slightly worse results, while NewtonFool performs better than the other methods but JSMA.

	0	1	2	3	4
FGSM	0.20 (0.02)	0.19 (0.02)	0.19 (0.03)	0.19 (0.03)	0.19 (0.02)
JSMA	8.72 (3.14)	4.21 (0.75)	8.29 (3.04)	7.18 (2.73)	6.47 (2.00)
DeepFool	5.90 (6.42)	3.98 (1.01)	5.46 (5.04)	4.30 (4.31)	3.46 (1.73)
NewtonFool	2.51 (1.02)	2.86 (0.96)	2.30 (1.29)	1.81 (1.06)	1.75 (0.96)

	5	6	7	8	9
FGSM	0.18 (0.02)	0.19 (0.02)	0.19 (0.02)	0.18 (0.02)	0.18 (0.02)
JSMA	6.76 (2.64)	7.39 (2.59)	6.44 (2.18)	8.07 (3.20)	6.46 (2.20)
DeepFool	5.48 (15.02)	7.64 (19.04)	4.89 (4.54)	3.76 (3.53)	4.35 (8.55)
NewtonFool	1.75 (0.96)	2.31 (0.93)	2.43 (1.26)	1.59 (0.92)	1.50 (0.73)

Table 8: MNIST statistics for the running time (unit: sec.)


FGSM	0.19 (0.03)	0.19 (0.02)	0.19 (0.03)	0.18 (0.03)	0.17 (0.02)	0.20 (0.02)
Papernot et al.	755.67 (879.74)	321.35 (289.29)	316.57 (257.51)	372.64 (279.62)	108.89 (59.59)	689.81 (511.29)
Deepfool	118.55 (1097.21)	24.93 (13.07)	34.99 (23.22)	28.19 (24.68)	13.57 (4.43)	41.78 (28.01)
NewtonFool	0.85 (0.66)	0.64 (0.29)	0.80 (0.36)	0.65 (0.41)	0.56 (0.30)	0.85 (0.42)

Table 9: GTSRB statistics for the running time (unit: sec.)