Adversarial-Playground: A Visualization Suite for Adversarial Sample Generation

Andrew Norton and Yanjun Qi
Department of Computer Science
University of Virginia
Charlottesville, VA 22904, USA
{apn4za, yanjun}@virginia.edu

Abstract

With growing interest in adversarial machine learning, it is important for machine learning practitioners and users to understand how their models may be attacked. We propose a web-based visualization tool, *Adversarial-Playground*, to demonstrate the efficacy of common adversarial methods against a deep neural network (DNN) model, built on top of the TensorFlow library. Adversarial-Playground provides users an efficient and effective experience in exploring techniques generating adversarial examples, which are inputs crafted by an adversary to fool a machine learning system. To enable Adversarial-Playground to generate quick and accurate responses for users, we use two primary tactics: (1) We propose a faster variant of the state-of-the-art Jacobian saliency map approach that maintains a comparable evasion rate. (2) Our visualization does not transmit the generated adversarial images to the client, but rather only the matrix describing the sample and the vector representing classification likelihoods [1].

1 Introduction

Though Deep Neural Networks (DNNs) have become an essential tool for many machine learning tasks, especially image classification [6], recent studies of *adversarial samples* show that intelligent attackers can force DNN models to misclassify examples by adding small and imperceptible modifications to pixel values on a regular test image [5, 10]. These maliciously generated adversarial examples are commonly crafted by using an optimization procedure to search for small, but effective, artificial perturbations (details in Section 2).

The goal of this paper is to introduce a visualization tool that enables deep learning practitioners and users to understand how their DNN models may be attacked. Investigating the behavior of machine learning systems in adversarial environments is an emerging topic at the junction between machine learning and computer security [2]. From the data scientists' perspective, machine learning models appear to be effective for many security tasks like malware classification [7] and face recognition [9]. However, it is important to realize that machine-learning techniques were not designed to withstand manipulations made by intelligent and adaptive adversaries. Unlike in applications of machine learning to other fields, security tasks involve adversaries that respond maliciously to the classifier [2]. Therefore we seek to provide an educational tool for visualizing adversarial examples generated by common evasion attacks and showing how these examples fool a state-of-the-art DNN model.

The proposed toolbox follows the spirit of the TensorFlow Playground–a web-based, educational tool for helping users understand how neural networks work [11]. TensorFlow Playground has been used in many classes as a pedagogical aid and helps the self-guided student learn more. The impact of this tool inspires us to visualize adversarial samples in a similar tool, which we call "Adversarial-Playground." It is a web-based visualization tool to assist researchers and students to understand and to compare the impact and efficacy of standard adversarial methods against deep learning. Adversarial-Playground provides quick and effective visualizations of adversarial examples through two strategies:

- Upon launch, Adversarial-Playground starts a lightweight Python websever hosting a collection of pages related to the visualization. The server-based nature allows remote hosting and the webserver may be hosted on any computer running TensorFlow. To speed up the response time, the visualization does not transmit adversarial examples in the image format to the client, but rather only

---

data matrix describing the sample and the vector of classification are transmitted. Then the graphics are rendered on the client-side. This design is different from the TensorFlow Playground, which was written using client-side technology–specifically, a highly optimized (and constrained) neural network class was written in Javascript.

- Most state-of-the-art algorithms generating adversarial examples suffer from slow performance due to expensive optimization and the large search space of image classification [3, 4]. This is less of an issue for studies focusing only on the effectiveness of adversarial attacks. However, the slow responses in visualizing adversarial examples will certainly impact users of our system. Therefore we introduce an improvement upon the current state-of-the-art Jacobian Saliency Map algorithm (JSMA) via a heuristic-based search space reduction. Instead of performing a costly search of all feature pairs (with quadratic runtime), we reduce the search to a linear runtime. Through empirical evaluations, the revised Faster version of JSMA (FJSMA) maintains the same evasion rate as the usual JSMA algorithm but executes far more quickly.

The rest of this paper is structured as follows. Section 2 reviews three types of adversarial examples generated by state-of-the-art methods. Section 3 introduces the system organization and software design of Adversarial-Playground. Section 4 presents the proposed FJSMA algorithm plus the empirical comparison with JSMA. Section 5 concludes the paper.

2 Background of Adversarial Examples

Studies of adversarial examples are part of the literature investigating the behavior of machine learning systems in adversarial environments. Relevant studies roughly fall into three types: (1) Poisoning attacks in which specially crafted samples are injected into the training of a learning model. (2) Privacy-aware machine learning to preserve the privacy of data such as differential privacy. (3) Evasion attacks are attacks in which the adversary's goal is to create inputs that are misclassified by a deployed target classifier. Most studies assume the adversary cannot influence the training data but instead finds "adversarial examples" to evade a trained classifier like DNN. This paper focuses on evasion attacks.

The goal of adversarial sample generation is to craft an input for a target classifier that is improperly classified, yet reveals only slight alteration from a human perspective. To formalize the extent of this modification, algorithms that generate adversarial samples work to minimize the difference between the "seed" image and

the resulting "evasive" sample on some selection of norm (distance) functions.

In some cases, it is important to the adversary to specify the "target" class of an evading sample – for example, the adversary may desire an image that looks like a "5" to be classified as a "7". This is referred to as a *targeted* approach. Conversely, if the adversary does not specify the desired class, the algorithm is considered to be *untargeted*.

Formally, let us denote $f : X \to C$ to be a classifier that maps the set of all possible inputs, X, some finite set of classes, C. Then, given a target class $y_t \in C$, a starting sample $x \in X$, and a norm $\| \cdot \|$, the goal of targeted adversarial sample generation is to find $x' \in X$ such that:

$$x' = \arg\min \left\{ \|x - x'\| : f(x') = y_t \right\} \qquad (1)$$

Similarly, in the untargeted case, the goal is to find x' such that:

$$x' = \arg\min \left\{ \|x - x'\| : f(x') \neq f(x) \right\} \qquad (2)$$

In this formalization, we see there are two key degrees of freedom in creating a new algorithm for evasive sample generation: targeted vs. untargeted generation, and the choice of norm. This provides a useful classification scheme for current adversarial sample generation algorithms, suggested by [3].

2.1 L^0 Norm

A simple way to determine the extent of the difference between two images is to count the number of pixels that differ between them. That is, if x is our original image, and $x' = x + r$ is the evading image (for some suitable value of r), then we can compute the L^0 distance between x and x' as following, where $[\cdot]$ is the Iverson bracket: [2]

$$\|r\|_0 = \sum_i [r_i \neq 0] \qquad (3)$$

Papernot, et al, suggested using the L^0 norm for evaluating the similarity of the initial sample and adversarial result [8]. Their approach computes the saliency map of a given input, then performs a combinatorial search to find two pixels to adjust.

This algorithm, and the fast gradient sign approach, was published in the Cleverhans package by Goodfellow, Papernot, and McDaniel [4]. The specifics of this algorithm will be discussed in more detail in Section 4.1.

---

[2]The Iverson bracket is defined as follows: $[P] = \begin{cases} 1 & P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$

2.2 L^2 Norm

A disadvantage of the L^0 norm is that it is not differentiable. Thus, from a theoretical standpoint, a more easily understood norm is the L^2 norm. This norm measures the standard Euclidean distance between two vectors; using the same notation as before, with x as the starting vector and $x' = x + r$ for the adversarial input, this norm is computed by the following:

$$\|r\|_2 = \left(\sum_i r_i^2 \right)^{1/2} \tag{4}$$

In their foundational paper on adversarial sample generation, Szegedy, et al, posed the issue as a convex optimization problem using the L^2 norm [10]. Then, this could be solved using the standard (albeit slow) method of box-constrained L-BFGS.

2.3 L^∞ Norm

A third commonly used norm in adversarial machine learning is the L^∞ norm, also called the *Chebyshev distance*. This measures the maximal change between two vectors along any single feature. That is if x is the starting vector and $x' = x + r$ is the adversarial input, the distance between them is computed by:

$$\|r\|_\infty = \max_i \{|r_i|\} \tag{5}$$

The Fast Gradient Sign method is commonly used due to its speed at generating evading inputs [5]. Unlike most prior approaches, which require iteratively changing the evasive sample away from its source class, FGS performs exactly one update step to obtain the evasive input. Essentially, the algorithm performs one step of gradient descent, but *away* from the loss function's minimum.

Formally, if x is our original input, $J(\theta, x, y)$ is the cost function for training the network, and x' is the evading input created by FGS, we have:

$$x' = x + \varepsilon \operatorname{sign}(\nabla_x J(\theta, x, y)) \tag{6}$$

The *attacking power*, ε, may be adjusted to fit the particular domain. Increasing ε increases the L^∞ distance between x and x', but it also increases the likelihood that the evading sample is misclassified. It is important to note that FGS is untargeted; this algorithm only cares about getting further away from the source class but does not specify any "target" class.
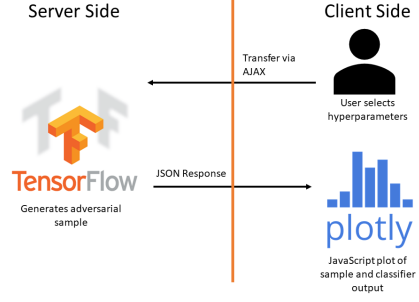

Figure 1: System Interaction

3 System Organization: Webserver with Client-side Visualization

As a web application, Adversarial-Playground splits the duties of visualization and computation between the client and server sides. Through the client, the user adjusts hyperparameters and submits their request for a generated sample to the server. Once the TensorFlow backend generates the adversarial image and classification likelihoods, the server returns this data to the client. Finally, this information is displayed graphically to the user through use of the `Plotly` JavaScript library (Figure 1).

When the application is accessed, the user is presented with the choice between several attack methods; some targeted and some untargeted. After selecting a method, the user may adjust model parameters and choose the source class (and target class, as applicable). When the user submits their selection of parameters, the server uses starts the creation of an adversarial sample against a pre-trained convolutional DNN model written with the TensorFlow library.

Using TensorFlow utilizes the GPU of the server to quickly return results to the client, even if the client is a lesser-powered machine than the server. Adversarial samples are generated in real time (rather than returning a precomputed result), so the user may experience a delay if the evasion method they selected is particularly slow. The resulting adversarial sample and likelihoods for each class are displayed to the user through client-side code.

3.1 Design Decisions

In creating our system, we made multiple design decisions. Here, we present the reasoning behind the three largest system-level decisions we made: building Adversarial Playground as a web-based application, utilizing both client- and server-side code, and rendering images with the client rather than the server.

Figure 2: Adversarial-Playground User Interface

(a) Adversarial and Original Sample

(b) Classification likelihoods
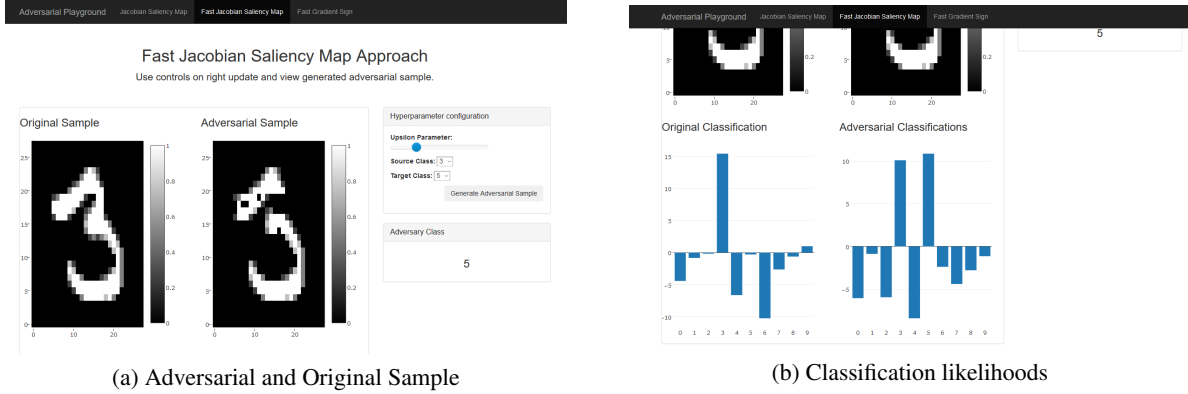
3.1.1 Web-based Framework

The first question we asked was whether we wanted a web-capable framework, or a strictly desktop application. This was a fairly simple design choice, as a web-based interface enables a large number of users to utilize the application without requiring an installation process on each computer. By eliminating an installation step, we encourage potential users who may be only casually interested in adversarial machine learning to explore what it is. This works towards the pedagogical goals of the software package.

3.1.2 Client-Server with Python back-end

Beyond just a web-based framework, we needed to decide if Adversarial Playground would be a client-only application, or if we would utilize server-side capabilities as well. The TensorFlow Playground was written entirely in Javascript and client-side technologies, allowing a lightweight server to host the service for many users. However, adversarial samples are usually generated on larger, deeper networks than those created by users of TensorFlow Playground, and this makes a JavaScript-only approach prohibitively expensive.

Instead, we chose to use a GPU-enabled server running Python with TensorFlow to generate the adversarial samples. In addition to a speed advantage, this allows our baseline deep neural network model to be replaced with a different one fairly simply. Likewise, adding new attack strategies is simpler, too, as TensorFlow has been used in many Adversarial Machine Learning research papers.

3.1.3 Client-side rendering

Once we decide to generate the samples on the server-side, it is tempting to generate the output images on the server as well. This leads us to the third design deci-

sion we made: client-side rendering of MNIST images and likelihood graphs. In our prototype stages, we used server-side rendering of these images using the Python library `matplotlib`, then loaded the image from the client directly.

However, this approach suffered from multiple disadvantages. Using only the default `matplotlib` utilities, we would have to write every generated image to disk and serve the unique URL to the user; this might be prohibitively expensive in terms of disk usage, depending on the number of users. Furthermore, each image was around 20 kilobytes, and waiting for the image to transfer added to the latency experienced by the user.

Fortunately, client-side rendering of images using `Plotly.JS` resolved both of these issues. Rather than transferring an entire PNG image with thousands of pixels, we send only the pixel values for the 28×28 MNIST images and the 10 values for classification likelihoods within the POST response to the client. This reduced the latency of each update significantly. Additionally, this eliminates the need for storing any produced image files on the server. As a further advantage, the Plotly framework allows interactive charts that display individual pixel values on mouseover, providing users with more detailed feedback.

3.2 MNIST Dataset

Adversarial-Playground uses the popular MNIST "hand-written digits" dataset for visualizing evasion attacks. This dataset contains 70,000 images of hand-written digits (0 through 9). Of these, 60,000 images are used as training data and the remaining 10,000 images are used for testing. Each image is 28×28 pixels, and each pixel is encoded as 8-bit grayscale. When the user selects "source class" for their sample, this selects an image from the testing set to be processed by the appropriate attack model.

We decided to only support the MNIST dataset for the visualization, as it is a common, but low-dimensional, data set for adversarial machine learning. Many adversarial machine learning papers work with some form of image data; since this is easy to visually process, we thought it would be best to work with either the MNIST, CIFAR, or ImageNet datasets. However, both ImageNet and CIFAR are much higher-dimensional than MNIST, which means generating adversarial samples is a more time-intensive process. To provide the user with a low-latency, yet still representative experience, we used the MNIST set.

3.3 Software Manual

The entirety of the code developed for this project is open-source on GitHub. In the interest of creating a high-quality, easy-to-use software package for demonstrating Adversarial Machine Learning, the following explains how to set up and use the package.

Setup The package is relatively minimal but requires a computer running TensorFlow 1.0 (or higher) with Python 3.5, the standard `SciPy` stack, and the Python package `Flask`. The code has been tested on both Windows and Linux operating systems.

To install, clone the GitHub repository and install the prerequisites via `pip3 -r install requirements.txt`. The pre-trained MNIST model is already stored in the GitHub repository; all that is needed to start the webapp is to run `python3 run.py`. Once the app is started, it will run on `localhost:9000`.

Usage To use the application, the user selects a model from the navigation bar at the top of the webapp. On the pane at right, the user should choose the desired value of Υ using the horizontal slider and the source/target class from the drop-down lists. This loads a sample from the MNIST testing set of the proper class into the selected adversarial sample generator (Figure 2a at right).

After the user has selected the desired parameters, they click "Update Model." This runs the adversarial algorithm in real-time to generate a possibly evasive sample. The sample is displayed in the primary pane to the left of the controls (Figure 2a at left). This sample is fed through the predictor, and then the likelihoods are normalized and displayed in bar charts below the samples (Figure 2b). Finally, the actual classification of generated sample is displayed below the controls at right.

4 Faster approach to JSMA

At the core of the Adversarial-Playground is a set of pre-implemented attack models. It was important to present the user with the choice between targeted and untargeted approaches, as well as a choice between models utilizing different norms. As such, we implemented saliency-map based approaches (one directly from Papernot et al [8], and a faster one of our own development) for our targeted methods and lightly modified the `cleverhans` implementation of the Fast Gradient Sign Method (FGS) for untargeted attacks.

As the FGS method is nearly identical to that found in `cleverhans`, we encourage the reader to consider [4] for implementation details. In the next two sections, we will review the details of the Jacobian Saliency Map Approach from Papernot's work [8] and our improvement, Fast Jacobian Saliency Map Apriori.

4.1 Jacobian Saliency Map Approach (JSMA)

The Jacobian Saliency Map Approach (JSMA) adjusts the starting input to maintain similarity based on the L^0. Applied to the MNIST model, the approach is as follows:

1. Compute the forward derivative of the classifier, $\nabla F(X)$.

2. Use the saliency map of the sample to determine two pixels to adjust.

3. Modify the two pixels and update the current sample.

4. Repeat until the adversarial sample, and the original input differs by at least Υ.

The first and last steps are fairly inexpensive to compute, while the primary computational difficulty is in using the saliency map to determine the pixels for adjustment. In their original paper, Papernot used Algorithm 1 for this selection process.

The key disadvantage of JSMA is that to determine which features to adjust, it must consider all pairs (p, q) of possible feature indices (see Algorithm 1). The loop in this routine must perform $\Theta(|\Gamma|^2)$ iterations, where $|\Gamma|$ is the feature size of each sample. When working on large feature sets, this becomes prohibitively expensive. By using a heuristic approximation of the JSMA algorithm, we achieve a faster runtime with comparable accuracy.

4.2 Fast Jacobian Saliency Map Apriori (FJSMA)

Our improved JSMA is inspired by the Apriori algorithm used in frequent set mining. The Apriori algorithm is a

Algorithm 1 Papernot's Saliency Map Feature Selection

$\nabla\mathbf{F}(\mathbf{X})$ is the forward derivative, Γ the features still in the search space, and t the target class

Input: $\nabla\mathbf{F}(\mathbf{X}), \Gamma, t$
1: **for** each pair $(p, q) \in \Gamma^2$, $p \neq q$ **do**
2:   $\alpha = \sum_{i=p,q} \frac{\partial\mathbf{F}_t(\mathbf{X})}{\partial\mathbf{X}_i}$
3:   $\beta = \sum_{i=p,q}\sum_{j\neq t} \frac{\partial\mathbf{F}_j(\mathbf{X})}{\partial\mathbf{X}_i}$
4:   **if** $\alpha < 0$ and $\beta > 0$ and $-\alpha \times \beta > \max$ **then**
5:    $p_1, p_2 \leftarrow p, q$
6:    $max \leftarrow -\alpha \times \beta$
7:   **end if**
8: **end for**
9: **return** p_1, p_2

Algorithm 2 Fast Jacobian Saliency Map Apriori Feature Selection

$\nabla\mathbf{F}(\mathbf{X}), \Gamma$, and t as in Algorithm 1, k is a small constant

Input: $\nabla\mathbf{F}(\mathbf{X}), \Gamma, t, k$
1: $K = \arg\mathrm{top}_{p\in\Gamma}\left(-\frac{\partial\mathbf{F}_t(\mathbf{X})}{\partial\mathbf{X}_p}; k\right)$
2: **for** each pair $(p, q) \in K \times \Gamma$, $p \neq q$ **do**
3:   $\alpha = \sum_{i=p,q} \frac{\partial\mathbf{F}_t(\mathbf{X})}{\partial\mathbf{X}_i}$
4:   $\beta = \sum_{i=p,q}\sum_{j\neq t} \frac{\partial\mathbf{F}_j(\mathbf{X})}{\partial\mathbf{X}_i}$
5:   **if** $\alpha < 0$ and $\beta > 0$ and $-\alpha \times \beta > \max$ **then**
6:    $p_1, p_2 \leftarrow p, q$
7:    $max \leftarrow -\alpha \times \beta$
8:   **end if**
9: **end for**
10: **return** p_1, p_2

fast, "bottom-up" approach to determining item sets with minimal support [1]. It achieves its speed through *a priori* elimination of certain suboptimal item sets. Similarly, it is reasonable to assume that some of the (p, q) pairs may be safely omitted from this loop.

Instead of exhaustively considering each feature pair (p, q), we rank the elements in the feature set Γ by the value of the Jacobian at that coordinate. (This is the contribution each element makes to α in Algorithm 1.) We then force the choice of p to be from the best k such features and allow q to be selected from the features remaining. Since this choice of p means its contribution to α is large, it is likely the product $-\alpha \times \beta$ will also be large.

Thus, we omit a large number of the (p, q) feature pairs through *a priori* knowledge derived from our heuristic. This alternative to Algorithm 1 is shown in Algorithm 2. If we denote $K = \arg\mathrm{top}_{p\in\Gamma}\left(-\frac{\partial\mathbf{F}_j(\mathbf{X})}{\partial\mathbf{X}_i}; k\right)$, where $\arg\mathrm{top}_{x\in A}\left(f(x); k\right)$ is the set consisting of the top k elements in A as ranked by f, then the loop in our Fast Jacobian Saliency Map Apriori (FJSMA) selection routine is $\Theta(|K| \cdot |\Gamma|)$, where $|K| \ll |\Gamma|$. Since determining the top k features can be done in $\Theta(|\Gamma|)$ time, this is a net improvement in asymptotic terms.

4.3 Empirical Results

The speed advantage of FJSMA as compared to JSMA is especially advantageous in the real-time environment of the Adversarial-Playground package. Low-latency generation of adversarial inputs provides a better user experience for this case, but also serves a practical purpose in real-world applications.

However, while the theory suggests that FJSMA will run faster than JSMA and it appears reasonable that they will generate evading samples at a similar rate, it remains

to demonstrate this. We compare an implementation of JSMA with our FJSMA implementation using a variety of values for k and Υ.

We ran both algorithms on the MNIST dataset and using the TensorFlow tutorial implementation of a deep convolutional neural network for MNIST Classification. To evaluate the performance of each approach, we compared the *evasion rate*–that is, the percentage of test samples that were successfully converted into evasive samples. This is the standard metric with which adversarial algorithms are evaluated. We want to ensure our FJSMA algorithm is not only faster than JSMA, but also evades the classifier at a similar rate.

After training the CNN network on the MNIST training set, we ran each of the three algorithms on all 10000 samples of the MNIST testing set to convert into evasive inputs. We varied the Υ parameter, and determined the evasion rate for each algorithm. For our FJSMA algorithm, we varied the value of k to be used; to better represent the performance for arbitrarily-sized feature sets, we set k in this experiment to be a percentage of the size of the feature set (in increments of 5%). Intuitively, this k value is a control on how tight of an approximation FJSMA is to JSMA; as k grows larger, we should expect the performance of the two approaches to converge to each other.

The results of this experiment are summarized in Figure 3a. As may be seen, the implementation of JSMA and the new FJSMA algorithm perform nearly equivalently for all tested values of Υ and k.

Additionally, we measured the wall clock time for each sample generation. The average time to form an evasive sample from the original, benign sample may be found in Figure 3b. As to be expected we see that our FJSMA approach greatly improves upon the speed of a similarly written JSMA, while maintaining nearly

Figure 3: Experiment Results

(a) Comparison of Evasion Rates

(b) Time for successful sample generation
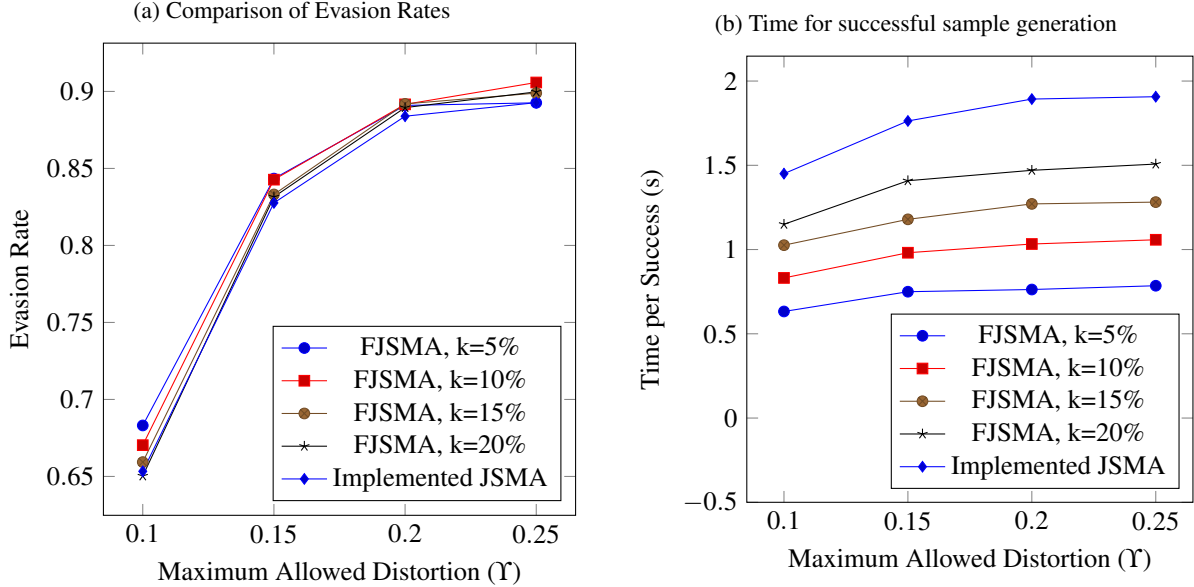
the same evasion rate.

5 Discussion and Future Work

Adversarial machine learning is a growing field of study, in which approaches to "evade" deep neural networks are developed. In this paper, we present a novel, web-based tool for visualizing the performance of evasion algorithms for deep neural networks. This helps both the researcher and the student understand and compare the impact of adversarial examples against DNN. Further, we provide an improvement to the Jacobian Saliency Map Approach (JSMA) from [8]. This improvement uses an *a priori* heuristic to reduce the search space significantly, therefore we name Fast Jacobian Saliency Map Apriori (FJSMA). FJSMA achieves a significant improvement in speed, while maintaining essentially the same evasion rate.

There are many directions in which this work may be developed. The most straightforward is to increase the variety of supported evasion methods. For example, the recent paper of Carlini et al [3] presents a new approach for each of the L^0, L^2, and L^∞ metrics; including these, and the original L^2-metric approach from [10], would be a reasonable step for comparing the performance of multiple evasion strategies.

However, to expand in this manner presents an additional issue of latency. To generate evading samples "on-demand," the adversarial algorithm must run quickly and these do algorithms take much longer to execute than the ones we selected for this work. A possible resolution is to pre-compute the results for the longer-running methods with a selected variety of hyperparameter values, then return these to the user. Another approach would be to attempt to apply the same greedy approximation technique used in FJSMA to the other attack algorithms, but this may not always be possible.

Another direction for development is to allow a choice of classifiers and datasets. Allowing the user to select from CIFAR, ImageNet, and MNIST data would highlight the similarities and differences between how a single attack method deals with different data. Similarly, providing the user with a choice of multiple pre-trained models – possibly with some hardened against attack through adversarial training – would further help distinguish artifacts of certain models from the performance of the base attacking method. These two extensions would help users more fully understand the behavior of an adversarial algorithm.

References

[1] AGRAWAL, R., AND SRIKANT, R. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), VLDB '94, Morgan Kaufmann Publishers Inc., pp. 487–499.

[2] BARRENO, M., NELSON, B., JOSEPH, A. D., AND TYGAR, J. The Security of Machine Learning. *Machine Learning 81*, 2 (2010), 121–148.

[3] CARLINI, N., AND WAGNER, D. Towards evaluating the robustness of neural networks. *CoRR abs/1608.04644* (2016).

[4] GOODFELLOW, I. J., PAPERNOT, N., AND MC-DANIEL, P. D. cleverhans v0.1: an adversarial machine learning library. *CoRR abs/1610.00768* (2016).

[5] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[6] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems* (2012), pp. 1097–1105.

[7] MICROSOFT CORPORATION. Microsoft Malware Competition Challenge. https://www.kaggle.com/c/malware-classification, 2015.

[8] PAPERNOT, N., MCDANIEL, P. D., JHA, S., FREDRIKSON, M., CELIK, Z. B., AND SWAMI, A. The limitations of deep learning in adversarial settings. *CoRR abs/1511.07528* (2015).

[9] PARKHI, O. M., VEDALDI, A., AND ZISSERMAN, A. Deep Face Recognition. In *British Machine Vision Conference* (2015).

[10] SZEGEDY, C., ZAREMBA, W., SUTSKEVER, I., BRUNA, J., ERHAN, D., GOODFELLOW, I. J., AND FERGUS, R. Intriguing properties of neural networks. *CoRR abs/1312.6199* (2013).

[11] YOSINSKI, J., CLUNE, J., NGUYEN, A. M., FUCHS, T. J., AND LIPSON, H. Understanding neural networks through deep visualization. *CoRR abs/1506.06579* (2015).