# 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 understanding of adversarial machine learning techniques, it is important for machine learning practitioners and users to understand how their models may be attacked. We propose a web-based visualization tool to demonstrate the efficacy of common adversarial methods against a sample MNIST model, built on top of the TensorFlow library. We also present a faster variant of the current Jacobian saliency map-based approach that maintains a comparable evasion rate.

## 1 INTRODUCTION

In recent years, Deep Neural Networks have become an essential tool for many machine learning tasks, especially image classification. However, this has given rise to the study of *adversarial samples*–carefully crafted inputs for a neural network that result in an incorrect classification.

In the spirit of the TensorFlow Playground, we present the Adversarial Playground: a web-based visualization tool to assist users in understanding the differences between common adversarial machine learning techniques. Upon launch, the application starts a lightweight Python websever hosting a collection of pages related to the visualization. The server-based nature allows remote hosting, but the webserver may be hosted on any computer running TensorFlow.

Further, we introduce an improvement upon the current state-of-the-art Jacobian Saliency Map algorithm via a heuristically-based search space reduction. Instead of performing a costly search of all feature pairs (with quadratic runtime), we constrain the possible values for one feature, reducing this to a linear runtime (with high constant factor). Our approach maintains the same evasion rate as the usual JSMA algorithm, but performs much faster.

## 2 REVIEW OF RESEARCH

The goal of adversarial machine learning is to craft an input for the classifier that is improperly classified, yet reveals only slight alteration from a human perspective. To formalize the extent of this alteration, algorithms for generating adversarial samples work to minimize the difference between the "source" image and the resulting "evasive" sample with respect to some norm.

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 a 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:

$$x' = \arg\min \{\|x - x'\| \, : \, f(x') = y_t\} \tag{1}$$

[*]https://www.apnorton.com
[†]https://www.cs.virginia.edu/yanjun

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

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

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 Carlini & Wagner (2016).

Table 1: Current techniques for adversarial sample generation

| **Norm** | **Targeted** | **Untargeted** |
|---|---|---|
| $L^0$ | JSMA, Carlini | |
| $L^2$ | LBFGS, Deepfool, and Carlini | |
| $L^\infty$ | Carlini | FGS |

## 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: [1]

$$\|r\|_0 = \sum_i [r_i \neq 0] \tag{3}$$

Papernot et al. (2015) suggested using the $L^0$ norm for evaluating the similarity of the initial sample and adversarial result. 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, were published in the Cleverhans package by Goodfellow et al. (2016). The specifics of this algorithm will be discussed in more detail in Section 3.2.1.

## 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. (2013) posed the issue as a convex optimization problem using the $L^2$ norm. Then, this could be solved using the standard (albeit slow) method of box-constrained L-BFGS.

## 2.3 $L^\infty$ Norm

A third commonly used norm in adversarial machine learning is the $L^\infty$ norm, also called the *Chebyshev distance*. This measures the maximal change between two vectors along any single feature.

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

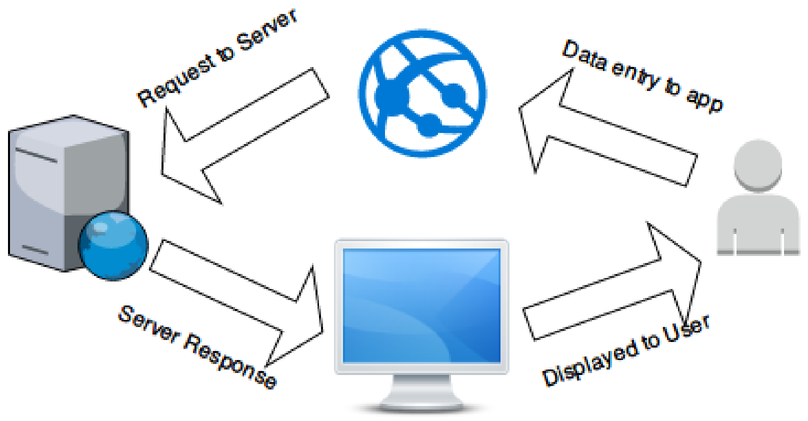


Figure 1: System Interaction

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 (Goodfellow et al. 2014) is commonly used due to its speed at generating evading inputs. 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 decent, 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:

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

The *attacking power*, $\epsilon$ may be adjusted to fit the particular domain. Increasing $\epsilon$ increases the $L^\infty$ 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 new, "target" class.

### 2.4 Web-based Visualization

Following the release of TensorFlow, Google released a project called TensorFlow Playground (Yosinski et al. 2015), a web-based, educational tool for understanding how neural networks work. This platform has been used in classes as a pedegogical aid, but also helps the self-guided student learn more.

The entire service is written using client-side technology; a highly optimized (and constrained) neural network class was written in Javascript. This was wrapped with a GUI, allowing users to specify initial weights and the network structure before training the model. The impact of this tool inspired the work we have done to perform adversarial sample visualization in a web-based tool.

## 3 Method

### 3.1 System Organization

As a web application, the Adversarial Playground splits the duties of visualization and computation between the client and server sides. The client's only responsibility is to render the data returned by the server and to provide the user with means to adjust model parameters and input. (Figure 1.)

When the application is accessed, the user is presented with the choice between several attack models; some targeted and some untargeted. After selecting a model, the user may adjust model parameters and choose the source class (and target class, as applicable). Once the user submits the request to generate an adversarial sample, control passes to the backend.

Once the user submits their selection of parameters, the server uses starts the creation of an adversarial sample against a pretrained MNIST model written with the TensorFlow library. This utilizes the GPU of the serving computer 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. Once the result is computed, images describing the resulting evasive sample and likelihoods for each class are returned.

## 3.2 Attack Models

At the core of the Adversarial Playground is a pair 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 two saliency-map based approaches (one from Papernot et al. 2015, 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 (see Table 2).

Table 2: Attack methods included in Adversarial Playground

| **Method** | **Norm** | **Targeted/untargeted** |
|---|---|---|
| Jacobian Saliency Map Approach | $L^0$ | Targeted |
| Fast Jacobian Saliency Map A-priori Approach | $L^0$ | Targeted |
| Fast Gradient Sign Method | $L^\infty$ | Untargeted |

As the FGS method is nearly identical to that found in `cleverhans`, we encourage the reader to consider Goodfellow et al. (2016) for implementation details. In the next two sections, we will review the details of the Jacobian Saliency Map Approach from Papernot et al. (2015) and our improvement, Fast Jacobian Saliency Map Apriori.

### 3.2.1 Jacobian Saliency Map Approach (JSMA)

As mentioned before, 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 original input differ by at least $\Upsilon$.

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. With large feature sets, this becomes prohibitively expensive. By using a heuristic approximation of the JSMA algorithm, we achieve a faster runtime with comparable accuracy.

### 3.2.2 Fast Jacobian Saliency Map Apriori (FJSMA)

In frequent set mining, the Apriori algorithm (Agrawal & Srikant 1994) is a fast, "bottom-up" approach to determining item sets with minimal support. It achieves its speed through *a priori* elimi-

**Algorithm 1 Papernot's Saliency Map Feature Selection**
$\nabla\mathbf{F}(\mathbf{X})$ is the forward derivative, $\Gamma$ 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 $p_1, p_2$
```

nation of certain suboptimal item sets. In a similar manner, 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 $\Gamma$ by the value of the Jacobian at that coordinate. (This is the contribution each element makes to $\alpha$ 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 $\alpha$ 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 $K = \arg\operatorname{top}_{p \in \Gamma} \left( -\frac{\partial \mathbf{F}_j(\mathbf{X})}{\partial \mathbf{X}_i};\ k \right)$, where $\arg\operatorname{top}_{x \in A} \left( f(x);\ k \right)$ 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.

**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: $K = \arg\operatorname{top}_{p \in \Gamma} \left( -\frac{\partial \mathbf{F}_j(\mathbf{X})}{\partial \mathbf{X}_i};\ k \right)$
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 $p_1, p_2$
```

## 4 Experiment

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.

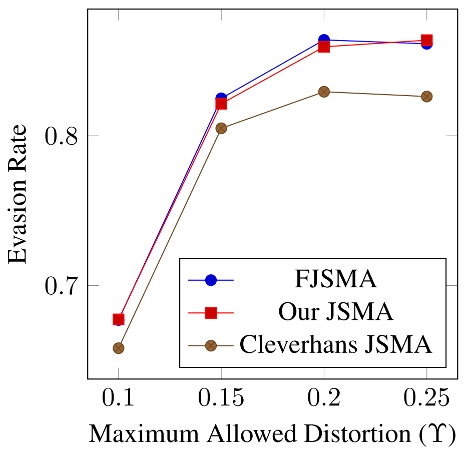
However, while the theory suggests that FJSMA will run faster than JSMA and it appears reasonable that they will generate evading samples at a similar rate, it remains to demonstrate this. We compare an implementation of a single-threaded JSMA with a singled-threaded FJSMA implementation, then compare both against the multithreaded implementation of JSMA found in Cleverhans (Goodfellow et al. 2016).

For evaluating the performance, we are running all three algorithms on the MNIST dataset and using the TensorFlow tutorial implementation of a deep convolutional neural network for MNIST Classification.

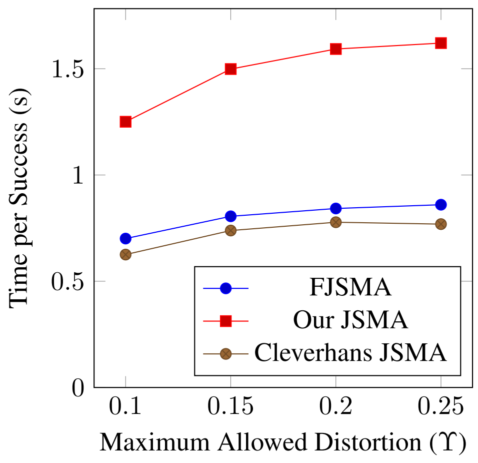
After training the network on the MNIST training set, we ran each of the three algorithms on 5000 of the testing samples to convert into evasive input. We varied the $\Upsilon$ parameter, and determined the *evasion rate* for each algorithm–that is, the percentage of testing samples that were successfully converted into evasive samples.

Figure 2: Experiment Results

(a) Comparison of Evasion Rates



(b) Time for successful sample generation



The results of this experiment are summarized in Figure 2a. As may be seen, our implementation of JSMA and the new FJSMA algorithm perform nearly equivalently for all tested values of $\Upsilon$. Furthermore, both implementations outperform the `cleverhans` implementation in terms of evasion rate.

Additionally, we measured the wall clock time for each sample generation. The average time to form a evasive sample from the original, benign sample may be found in Figure 2b. As to be expected, our implementation of JSMA is the slowest of the three compared algorithms. This is because our JSMA implementation is single-threaded, while the `cleverhans` implementation performs the search over the saliency map using all available processor cores. However, even the single-threaded FJSMA approach is competitive against the multi-threaded `cleverhans` attack. Thus, we see that our FJSMA approach greatly improves upon the speed of a single-threaded JSMA, while maintaining a similar evasion rate.

## 5 Software Manual

The entirety of the code developed for this project is open-source may be found 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.

### 5.1 Setup

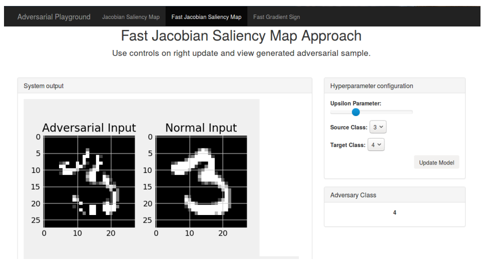
The package is fairly minimal, but needs requires a computer running TensorFlow 1.0 (or higher) with Python 3.5, the standard `SciPy` stack, and the Python package `Flask`.

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`.

### 5.2 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 select a desired value of $\Upsilon$ using the horizontal slider and the source/target class from the dropdowwn lists. This loads a sample from the MNIST testing set of the proper class into the selected adversarial sample generator (Figure 3a 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 on the primary pane to the left of the controls (Figure 3a at left). This sample is fed through the predictor, then the likelihoods are normalized and displayed in bar charts below the samples (Figure 3b). Finally, the actual classification of generated sample is displayed below the controls at right.



Fast Jacobian Saliency Map Approach

Use controls on right update and view generated adversarial sample.

Hyperparameter configuration

Upsilon Parameter:

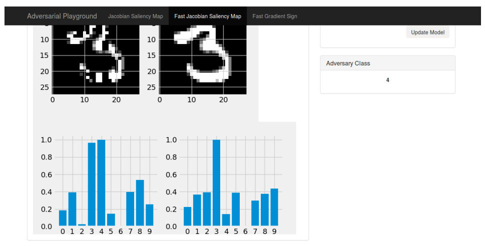
Source Class: 3

Target Class: 4

Adversary Class

4

(a) Adversarial and Original Sample



Adversary Class

4

(b) Classification likelihoods

Figure 3: Adversarial Playground User Interface

## 6 Discussion

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 using TensorFlow. This helps both the researcher and the student understand and compare the impact of various algorithms.

Further, we provide an improvement to the Jacobian Saliency Map Approach (JSMA) from Papernot et al. (2015). This improvement uses an *a priori* heuristic to reduce the search space significantly, so we propose the name Fast Jacobian Saliency Map Apriori (FJSMA). Using this algorithm causes a significant improvement in speed, while maintaining essentially the same evasion rate.

## References

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pp. 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8. URL `http://dl.acm.org/citation.cfm?id=645920.672836`.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. *CoRR*, abs/1608.04644, 2016. URL `http://arxiv.org/abs/1608.04644`.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Ian J. Goodfellow, Nicolas Papernot, and Patrick D. McDaniel. cleverhans v0.1: an adversarial machine learning library. *CoRR*, abs/1610.00768, 2016. URL `http://arxiv.org/abs/1610.00768`.

Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. *CoRR*, abs/1511.07528, 2015. URL `http://arxiv.org/abs/1511.07528`.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013. URL `http://arxiv.org/abs/1312.6199`.

Jason Yosinski, Jeff Clune, Anh Mai Nguyen, Thomas J. Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *CoRR*, abs/1506.06579, 2015. URL `http://arxiv.org/abs/1506.06579`.