Report 2: Generating adversarial samples for DNNs

Introduction: Deep neural networks(DNNs) has proven to be quite efficacious in machine learning tasks, with recent adoption in automation systems and cyberphysical systems. With increasing popularity, security threats follow. Adversaries subtly alter legitimate inputs (call input perturbation) to induce the trained model to produce erroneous outputs. Adversarial samples can be used to, for example, subvert fraud detection, bypass content filters or malware detection, or to mislead autonomous navigation systems. Adversarial sample transferability 1 is the property that some adversarial samples produced to mislead a specific model F can mislead other models - even if their architectures greatly differ. A practical impact of this property is that it leads to black box attacks.

Related Work:
In [4], it was discovered that several machine learning models including DNNs are vulnerable to adversarial samples. The cause of these adversarial examples was a mystery, and speculative explanations have suggested it is due to extreme nonlinearity of deep neural networks, perhaps combined with insufficient model averaging and insufficient regularization of the purely supervised learning problem until [2]. [2] explained the existence of adversarial examples in linear models and linear perturbation in non-linear models and enabled a fast way of generating adversarial samples. [4] also showed that

- Box-constrained L-BFGS can reliably find adversarial examples.
- On some datasets, such as ImageNet, the adversarial examples were so close to the original examples that the differences were indistinguishable to the human eye. The same adversarial example is often misclassified by a variety of classifiers with different architectures or trained on different subsets of the training data.
- Shallow softmax regression models are also vulnerable to adversarial examples.

Background:
A. Deep Neural Networks
A typical DNN architecture, graphically depicted in Fig. 1, consists of multiple successive layers of processing elements, or so-called "neurons". Each processing layer can be viewed as learning a different, more abstract representation of the original multidimensional input distribution. As a whole, a DNN can be viewed as a highly complex function that is capable of nonlinearly mapping original high-dimensional data points to a lower dimensional space. Starting from the input, computing the activations of each subsequent layer simply requires, at minimum, a matrix multiplication usually followed by summation with a bias vector. This process roughly models the process of a layer of neurons integrating the information received from the layer below (i.e., computing a pre-activation) before applying an elementwise activation function.

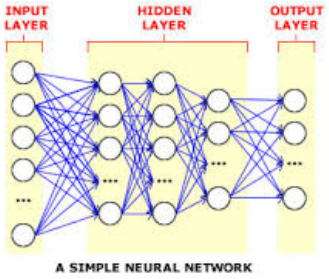

Fig 1. Neural Network architecture

During the learning phase of the model, the DNN's predictions are evaluated by comparing them with known target labels associated with the training samples (also known as the"ground truth"). Specifically, both predictions and labels are taken as the input to a selected cost function, such as cross-entropy. The DNN's parameters are then optimized with respect to this cost function using the method of steepest gradient descent, minimizing prediction errors on the training set. Parameter gradients are calculated using back-propagation of errors. Since the gradients of the weights represent their influence on the final cost, there have been multiple algorithms developed for finding optimal weights more accurately and efficiently.
More formally, given a training set (X, Y):
$\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where $x_i \in R^m$ is a data sample and $y_i \in R^k$ is data label, where, if categorical, is typically represented through a 1-of-k encoding. A standard neural network with a softmax output layer can be represented by the following function:
$f : R^m \rightarrow R^k$ -------- (1)
$x \rightarrow f(x)$.
The cost function for training a DNN as a (k-class) classifier is also defined as follows:
$L : R^m \times R^k \rightarrow R$ ------ (2)
$f(x) \times y \rightarrow L(f(x), y)$.

B. Adversarial samples
Adversarial samples are generated by computing the derivative of the cost function with respect to the network's input variables. The gradient of any input sample represents a direction vector in the model's high-dimensional input space. Along this direction, a small change in this input sample can cause the DNN to generate a completely different prediction result. This particular direction is important since it represents the most effective way one might compromise the optimization of the cost function. Discovering this particular direction is realized by transmitting the error gradients from output layer all the way back to the input layer via back-propagation. The gradient with respect to the input may then be applied to the input sample(s) to craft an adversarial example(s).
We now formally present the generation of adversarial samples by first defining the function g for generating any adversarial sample $\hat{x}$:
$g : R^m \times R^k \rightarrow R^m$ ----- (3)
$x \times y \rightarrow \hat{x}$,
where
$\hat{x} = \arg\max L(f(\hat{x}), y)$ s.t. $\| x - \hat{x} \|_p < \varepsilon$, ----- (4)
and $\|\cdot\|_p$ is p-norm: $\|v\|_p = (\sum_{i=1}^{n} |v_i|^p)^{1/p}$

Note that g is a one-to-one mapping, meaning that when there exist multiple $\hat{x}$ satisfying (4), only one is chosen. It should be noted that the constraint in (4) ensures that the distortion incurred by manipulation must be maintained at a small scale. Otherwise a significant distortion will leave the manipulation to be easily detected. Existing attacks [1], [2], [3], [4] consist of different

(a) (b) (c) (d)

Figure 2: The fast gradient sign method applied to logistic regression (where it is not an approximation, but truly the most damaging adversarial example in the max norm box). a) The weights of a logistic regression model trained on MNIST. b) The sign of the weights of a logistic regression model trained on MNIST. This is the optimal perturbation. Even though the model has low capacity and is fit well, this perturbation is not readily recognizable to a human observer as having anything to do with the relationship between 3s and 7s. c) MNIST 3s and 7s. The logistic regression model has a 1.6% error rate on the 3 versus 7 discrimination task on these examples. d) Fast gradient sign adversarial examples for the logistic regression model with = .25. The logistic regression model has an error rate of 99% on these examples.

approaches for generating adversarial samples. These approaches, ultimately, can all be described solving following optimization problem, which is a transformation of (4):

$\hat{x}$: min c $||x - ||\hat{x}$ − L(f ($\hat{x}$), y), ----- (6)

where c is some weight or coefficient. Here we refer to this type of attack as an optimal attack, and, for simplicity, denote $||x - \hat{x}||_p$ as the l_p distance.

Solving (6) is done much like solving for the weight coefficients of a standard DNN. Simply put, an attacker can use back-propagation and either gradient descent [2] or L-BFGS [1], [4]. In order to conduct the calculation, one first computes ∂^n L(f ($\hat{x}$)y) / ∂ $^n\hat{x}$and then use it for manipulating legitimate samples x. Note that here n = 1 if we use a first-order optimization method like gradient descent, or n = 2 if we utilize a second-order method such as the Newton–Raphson method. Since gradient descent requires an iterative calculation, it can be computationally expensive when the number of adversarial samples to be generated is large. To mitigate this problem, [2] proposed the fast gradient sign method to approximate this iterative calculation where one could use a single step of gradient descent, as follows:

$\hat{x}$ = x + ϵ · sign(∇L(f (x), y)), ------ (7)

where ϵ is set to control the scale of the distortions applied. Optimal attacks can be generalized to fast gradient sign attacks by selecting the l_∞ distance. In [4], the authors study an equivalent optimization problem to (6) by setting p = 2. Another type of attack, as described in [3], generates adversarial samples using a greedy approach to solve (6) and sets p = 0. More specifically, adversarial samples are crafted by iteratively perturbing one coordinate of x at a time. A coordinate is picked by determining by its influence on the final cost.

C. End-to-end Gradient Flow

In order to solve (6), previously introduced attacks must be able to calculate the n-th order derivative of L(f ($\hat{x}$), y) withrespect to $\hat{x}$. The calculation of ∂^n L(f ($\hat{x}$), y)/∂ $^n\hat{x}$requires the existence of an end-to-end gradient flow between the final objective and the input. More formally, given a DNN F 4 with cost function L, for any pair of input sample x and target label y, if ∂ n L(F (x), y)/∂x n can be computed by passing residual error ∂^n L(F(x), y)/∂ F(x)^n backwards to the input through intermediate layers of a (fully-differentiable) DNN, then we say there exists an end-to-end gradient flow. Existing attacks [1], [2], [4] generate adversarial sample $\hat{x}$ by exploiting the existence of an end-to-end gradient flow. As long as the DNN supports end-to-end gradient flow, it will always be susceptible to adversarial perturbation.

References:

[1] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks, 2016.

[2] Szegedy, Christian, Zaremba, Wojciech, Sutskever, Ilya, Bruna, Joan, Erhan, Dumitru, Goodfellow, Ian J., and Fergus, Rob. Intriguing properties of neural networks. 2014

[3] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings, 2016.

[4] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks, 2014.