THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

Gradient-based Adversarial Attacks to Deep Neural Networks in Limited Access Settings

by

Yash Sharma

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

May 8, 2018

Professor Sam Keene, Advisor

THE COOPER UNION FOR THE
ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Dean, School of Engineering Date

Prof. Sam Keene, Thesis Advisor Date

Acknowledgments

I would like to thank Dr. Sam Keene for his advisement. I would like to thank Dr. Pin-Yu Chen for his mentorship during my time at IBM Research and after, and for his willingness to answer all of my questions stemming from my constant confusion. I would lastly like to thank my family for their constant support throughout all of my endeavors, and their immense care for my well-being.

Abstract

Recent studies have highlighted the vulnerability of deep neural networks (DNNs) to adversarial examples - inputs formed by applying small but intentionally worst-case perturbations to examples from the dataset, such that the perturbed inputs result in the model outputting incorrect answers with high confidence. Gradient-based optimization is used to find said adversarial examples by jointly minimizing the perturbation while maximizing the probability that the generated example causes the target model to misclassify. This approach can be readily applied in the white-box case, where the attacker has complete access to the target model and thus can compute the gradients via backpropagation. We extend such approaches to the black-box case, where the attacker is only given query access and therefore incapable of directly computing the gradients. We introduce ZOO, which uses the finite difference method to estimate the gradients for optimization from the output scores. Furthermore, we also improve the state-of-the-art in the no-box case, where the attacker is not even capable of querying the target model. We introduce EAD, which incorporates L_1 minimization in order to encourage sparsity in the perturbation, hence generating more robust adversarial examples in the white-box case which can transfer to unseen models. Through experimental results attacking state-of-the-art models trained on the MNIST, CIFAR-10, and ImageNet datasets, we validate the effectiveness of the proposed attacks. In addition, we demonstrate that the proposed attacks can successfully attack recently proposed defenses in these limited access settings. We show that ZOO can succeed against the state-of-the-art ImageNet defense, Ensemble Adversarial Training, while EAD can succeed against the state-of-the-art MNIST defense, the Madry Defense Model, and input transformation defenses, such as Feature Squeezing.

Bibliographic Note

Portions of this thesis are based on prior peer-reviewed publications. ZOO was originally published in [1], and EAD was originally published in [2]. Finally, results using EAD to attack the Madry Defense Model [3] and Feature Squeezing [4] are currently under peer review.

Contents

1 Background

Much of this section is derived from the *Deep Learning Book* [5]. Please refer for further reading.

1.1 Gradient-based Optimization

Optimization refers to the task of either minimizing or maximizing some function $f(x)$ by altering x. Most optimization problems are usually phrased in terms of minimizing $f(x)$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(x)$.

The function we want to minimize or maximize is called the objective function. When minimizing, this function is also called the cost function, the loss function, or the error function. The altered value which minimizes or maximizes $f(x)$ is denoted as x^*.

1.1.1 First-order Optimization

Supposed we have a function $y = f(x)$, where both x and y are real numbers. The derivative of this function is denoted as $f'(x)$ or as $\frac{dy}{dx}$. The derivative $f'(x)$ gives the slope of $f(x)$ at the point x. In other words, it specifies how to scale a small change in the input in order to obtain the corresponding change in the output: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

The derivative is therefore useful for minimizing a function because it tells us how to change x in order to make a small improvement in y. We can thus reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative. This technique is called gradient descent [6], and is widely used for optimization in machine learning.

When $f'(x) = 0$, the derivative does not provide any information on which direction to move. Points where $f'(x) = 0$ are known as critical points or stationary points. A

local minimum is a point where $f(x)$ is lower than at all neighboring points, so it is no longer possible to decrease $f(x)$ by making infinitesimal steps. A local minimum is a point where $f(x)$ is higher than at all neighboring points, so it is no longer possible to increase $f(x)$ by making infinitesimal steps. Some critical points are neither maxima nor minima. These are known as saddle points, points which have neighbors where $f(x)$ is both higher and lower than that of the points themselves.

A point that obtains the absolute lowest value of $f(x)$ is a global minimum. It is possible for there to be only one global minimum or multiple global minima of the function. It is also possible for there to be local minima that are not globally optimal. In the context of machine learning, in particular deep learning, functions are often optimized which have many local minima that are not optimal, and many saddle points surrounded by very flat regions. Therefore, practitioners often settle for finding a value of f that is very low, but not necessarily minimal in the formal sense [5].

Functions which have multiple inputs are often optimized: $f : \mathbb{R}^n \to \mathbb{R}$. For such functions, we must make use of the concept of partial derivatives. The partial derivative $\frac{\partial}{\partial x_i} f(x)$ measures how f changes as only the variable x_i increases at point x. The gradient generalizes the notion of the derivative to the case where the derivative is with respect to a vector: the gradient of f is the vector containing all of the partial derivatives, denoted $\nabla_x f(x)$. Element i of the gradient is the partial derivative of f with respect to x_i. In multiple dimensions, critical points are points where every element of the gradient is equal to zero. As discussed, decreasing f by moving in the direction of the negative gradient is called gradient descent, otherwise known as steepest descent. The method proposes a new point as follows:

$$x' = x - \epsilon \nabla_x f(x) \tag{1}$$

where ϵ is the learning rate, a positive scalar determining the size of the step. ϵ is often set to a small constant, or found via a line search, where several values of ϵ are tested and one is chosen which results in the smallest objective function value. Optimization algorithms that use only the gradient, such as gradient descent, are called first-order optimization algorithms.

Gradient descent converges when every element of the gradient is zero (or, in practice, very close to zero). In some cases, we can avoid running the iterative algorithm by jumping directly to the critical point by solving the equation $\nabla_x f(x) = 0$ for x.

Although gradient descent is limited to optimization in continuous spaces, the general concept of repeatedly making a small move (that is approximately the best small move) towards better configurations can be generalized to discrete spaces. Ascending an objective function of discrete parameters is called hill climbing [7].

1.1.2 Second-order Optimization

Sometimes we need to find all of the partial derivatives of a function whose input and output are both vectors. The matrix containing all such partial derivatives is known as a Jacobian matrix. Specifically, if we have a function $f : \mathbb{R}^m \to \mathbb{R}^n$, then the Jacobian matrix $J \in \mathbb{R}^{nxm}$ of f is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(x)_i$.

We are also sometimes interested in a derivative of a derivative. This is known as a second derivative. For example, for a function $f : \mathbb{R}^n \to \mathbb{R}$, the derivative with respect to x_i of the derivative of f with respect to x_j is denoted as $\frac{\partial^2}{\partial x_i \partial x_j} f$. In a single dimension, we can denote $\frac{d^2}{dx^2} f$ by $f''(x)$. The second derivative tells us how the first derivative will change as we vary the input. This is important because it tells us whether a gradient step will cause as much of an improvement as we would expect based on the gradient alone.We can think of the second derivative as measuring curvature.

Suppose we have a quadratic function (many functions that arise in practice are not quadratic but can be approximated well as quadratic, at least locally). If such a function has a second derivative of zero, then there is no curvature. It is a perfectly flat line, and its value can be predicted using only the gradient. If the gradient is 1, then we can make a step of size ϵ along the negative gradient, and the cost function will decrease by ϵ. If the second derivative is negative, the function curves downward, so the cost function will actually decrease by more than ϵ. Finally, if the second derivative is positive, the function curves upward, so the cost function can decrease by less than ϵ. In summary, different forms of curvature affect the relationship between the value of the cost function predicted by the gradient and the true value.

When a function has multiple input dimensions, there are many second derivatives. These derivatives can be collected together into a matrix called the Hessian matrix. The Hessian matrix $H(f)(x)$ is defined such that $H(f)(x)_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(x)$. Equivalently, the Hessian is the Jacobian of the gradient.

The second derivative can be used to determine whether a critical point is a local maximum, a local minimum, or saddle point. Recall that on a critical point, $f'(x) = 0$. When the second derivative $f''(x) > 0$, the critical point x is a local minimum. Similarly, when $f''(x) < 0$, the critical point x is a local maximum. This is known as the second derivative test. Unfortunately, when $f''(x) = 0$, the test is inconclusive; x may be a saddle point, or a part of a flat region.

In multiple dimensions, there is a different second derivative for each direction at a single point. The condition number of the Hessian at this point measures how much the second derivatives differ from each other. The condition number can be computed by taking the ratio of the magnitude of the largest and smallest eigenvalue. When the Hessian has a poor condition number (large), gradient descent performs poorly. This is because in one direction, the derivative increases rapidly, while in another direction,

it increases slowly. Gradient descent is unaware of this change in the derivative so it does not know that it needs to explore preferentially in the direction where the derivative remains negative for longer. It also makes it difficult to choose a good step size. The step size must be small enough to avoid overshooting the minimum and going uphill in directions with strong positive curvature. This usually means that the step size is too small to make significant progress in other directions with less curvature.

This issue can be resolved by using information from the Hessian matrix to guide the search. The simplest method for doing so is known as Newton's method. Newton's method is based on using a second-order Taylor series expansion to approximate $f(x)$ near some point x_0. If we then solve for the critical point of this function, we obtain:

$$x^* = x_0 - H(f)(x_0)^{-1}\nabla_x f(x_0) \tag{2}$$

When f is a positive definite quadratic function, Newton's method consists of applying the above equation once to jump to the minimum of the function directly. When f is not truly quadratic but can be locally approximated as a positive definite quadratic, Newton's method consists of applying the above equation multiple times. Iteratively updating the approximation and jumping to the minimum of the approximation can reach the critical point much faster than gradient descent would. This is a useful property near a local minimum, but it can be a harmful property near a saddle point. Newton's method is only appropriate when the nearby critical point is a minimum (all the eigenvalues of the Hessian are positive), whereas gradient descent is not attracted to saddle points unless the gradient points toward them [5].

Optimization algorithms that use the Hessian matrix, such as Newton's method, are called second-order optimization algorithms [8].

The optimization algorithms employed in machine learning, and in particular deep learning, are applicable to a wide variety of functions, but come with almost no

guarantees. This is due to the fact that the family of functions used are non-convex (the Hessian is not positive semidefinite everywhere). Convex functions are well-behaved because they lack saddle points and all of their local minima are necessarily global minima. However, most problems in primarily deep learning are difficult to express in terms of convex optimization.

1.1.3 Constrained Optimization

Sometimes we wish not only to maximize or minimize a function $f(x)$ over all possible values of x. Instead we may wish to find the maximal or minimal value of $f(x)$ for values of x in some set S. This is known as constrained optimization. Points x that lie within the set S are called feasible points in constrained optimization terminology. For example, we often wish to find a solution that is small in some sense. A common approach in such situations is to impose a norm constraint, such as $\|x\| \leq 1$.

One simple approach to constrained optimization is simply to modify gradient descent taking the constraint into account. If we use a small constant step size ϵ, we can make gradient descent steps, then project the result back into S. If we use a line search, we can search only over step sizes ϵ that yield new x points that are feasible, or we can project each point on the line back into the constraint region. When possible, this method can be made more efficient by projecting the gradient into the tangent space of the feasible region before taking the step or beginning the line search [9].

A more sophisticated approach is to design a different, unconstrained optimization problem whose solution can be converted into a solution to the original, constrained optimization problem. The Karush-Kuhn Tucker (KKT) approach provides a very general solution to constrained optimization. Please refer to [10,11] for further detail.

1.2 Machine Learning

A machine learning algorithm is an algorithm that is able to learn from data. More precisely, a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [12].

1.2.1 Tasks

Machine learning allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. From a scientific and philosophical point of view, machine learning is interesting because developing our understanding of machine learning entails developing our understanding of the principles that underlie intelligence.

Machine learning tasks are usually described in terms of how the machine learning system should process an example. An example is a collection of features that have been quantitatively measured from some object or event that we want the machine learning system to process. We typically represent an example as a vector $x \in \mathbb{R}^n$ where each entry x_i of the vector is another feature. For example, the features of an image are usually the values of the pixels in the image.

Many kinds of tasks can be solved with machine learning. Some of the most common tasks include classification, where the computer program is asked to specify which of k categories some input belongs to, and regression, where the computer program is asked to predict a numerical value given some input. An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values), and the output is a numeric code identifying the object in the image. Object recognition is the same basic technology that allows computers to recognize faces [13], which can be used to automatically tag people in photo collections

and allow computers to interact more naturally with their users. An example of a regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premiums), or the prediction of future prices of securities. These kinds of predictions are also used in algorithmic trading.

1.2.2 Performance Measures

In order to evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure P is specific to the task T being carried out by the system.

1.2.2.1 Classification

For tasks such as classification, we often measure the accuracy of the model. Accuracy is just the proportion of examples for which the model produces the correct output. However when dealing with skewed datasets, when some classes are much more frequent than others, it is preferred to evaluate the performance of a classifier by looking at the confusion matrix. The general idea is to count the number of times instances of class A are classified as class B. Each row in a confusion matrix represents an actual class, while each column represents a predicted class. A perfect classifier would only have true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal.

Precision and recall give more concise quantitative metrics on the performance of the classifier. The precision is the accuracy of positive predictions, and is computed by dividing the number of true positives by the sum of the number of true positives and the number of false positives. A trivial way to have perfect precision is to make one single positive prediction and ensure it is correct. This would not be very useful since the classifier would ignore all but one positive instance. Due to this, precision is typically used alongside the recall metric, which is the ratio of positive instances

that are correctly detected by the classifier. It is computed by dividing the number of true positives by the sum of the number of true positives and the number of false negatives. It is often convenient to combine precision and recall into a single metric called the F_1 score, in particular to have a simple way to compare classifiers.

The F_1 score is the harmonic mean of precision and recall. Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F_1 score if both recall and precision are high. The F_1 score favors classifiers that have similar precision and recall. This is not necessarily the desired outcome, in some contexts precision is more significant, and in some contexts recall is more significant. For example, if the task was to detect videos that are safe for kids, a classifier which rejects many satisfactory videos (low recall) but keeps only safe ones (high precision) is preferred over a classifier which has a much higher recall but lets a few unsafe videos through. On the other hand, if the task was to detect shoplifters on surveillance images, low precision is manageable as long as the recall is high. Unfortunately, both precision and recall cannot be high: increasing precision reduces recall, and vice versa. This is called the precision/recall trade-off.

To understand this trade-off, consider a binary classifier making its classification decisions. For each instance, it computes a score based on a decision function, and if that score is greater than a threshold, it assigns the instance to the positive class, or else it assigns it to the negative class. Increasing the threshold increases precision but decreases recall. Conversely, lowering the threshold increases recall and reduces precision. Therefore, to select a good precision/recall trade-off, one can vary the threshold and plot precision directly against recall.

An alternative to the precision/recall curve is the receiver operating characteristic (ROC) curve. It is very similar, but instead of plotting precision versus recall, the

ROC curve plots the true positive rate (another term for recall) against the false positive rate. The FPR is the ratio of negative instances that are incorrectly classified as positive. It is equal to one minus the true negative rate, which is the ratio of negative instances that are correctly classified as negative. The TNR is also called specificity. Hence the ROC curve plots sensitivity (recall) versus 1-specificity. One way to compare classifiers is to measure the area under the curve (AUC). A perfect classifier will have an AUC equal to 1, whereas a purely random classifier will have an AUC equal to 0.5.

1.2.2.2 Regression

When choosing a performance metric for a regression task, the major question one asks is whether to penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes very large mistakes. If the first option is desired, the practitioner should choose to use the Root Mean Square Error (RMSE) criterion. If the second option is desired, the practitioner should choose to use the Mean Absolute Error (MAE).

Both the RMSE and the MAE are ways to measure the distance between two vectors: the vector of predictions and the vector of target values. Computing the root of a sum of squares (RMSE) corresponds to the Euclidean norm, or the L_2 norm. Computing the sum of absolutes (MAE) corresponds to the L_1 norm, or the Manhattan norm (because it measures the distance between two points in a city if one only travels along orthogonal city blocks). More generally, the L_p norm of a vector v containing n elements is defined as: $\|v\|_k = (|v_0|^k + |v_1|^k + ... + |v_n|^k)^{\frac{1}{k}}$. L_0 gives only the cardinality of the vector (the number of elements), and L_∞ gives the maximum absolute value in the vector.

The higher the norm index, the more it focuses on large values and neglects small ones. This is why the RMSE is more sensitive to outliers than the MAE. But when

outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

1.2.3 Types of Experience

Machine learning algorithms can be broadly categorized as unsupervised or supervised by what kind of experience they are allowed to have during the learning process.

Most learning algorithms can be understood as being allowed to experience an entire dataset. A dataset is a collection of many examples, where examples are also called data points. An example of a dataset is the classic Iris dataset [14], a collection of measurements of different parts of 150 iris plants. Each individual plant corresponds to one example. The features within each example are the measurements of each of the parts of the plant: the sepal length, sepal width, petal length and petal width. The dataset also records which species each plant belonged to. Three different species are represented in the dataset.

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. Some unsupervised learning algorithms perform clustering, which consists of dividing the dataset into clusters of similar examples. Another useful task is visualization, where the algorithm is fed a lot of complex and unlabeled data, and outputs a 2D or 3D representation of the data that can easily be plotted. These algorithms try to preserve as much structure as they can (e.g., trying to keep separate clusters in the input space from overlapping in the visualization), so the user can understand how the data is organized and perhaps identify unsuspected patterns. A related task is dimensionality reduction, in which the goal is to simplify the data without losing too much information. One simple way to do this is to merge several correlated features into one, more sophisticated algorithms exist as well.

Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target. Typical tasks performed in this case are classification and regression. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements.

Roughly speaking, unsupervised learning involves observing several examples of a random vector x, and attempting to implicitly or explicitly learn the probability distribution $p(x)$, or some interesting properties of that distribution, while supervised learning involves observing several examples of a random vector x and an associated value or vector y, and learning to predict y from x, usually by estimating $p(y|x)$. The term supervised learning originates from the view of the target y being provided by an instructor or teacher who shows the machine learning system what to do. In unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide.

Other variants of the learning paradigm are possible. For example, in semi- supervised learning, some examples include a supervision target but others do not. Some machine learning algorithms do not just experience a fixed dataset. For example, reinforcement learning algorithms interact with an environment, so there is a feedback loop between the learning system and its experiences. The learning system, called an agent in this context, can observe the environment, select and perform actions, and get rewards in return (or penalties in the form of negative rewards). It must then learn by itself the best strategy, called a policy, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

1.2.4 Generalization

The central challenge in machine learning is that we must perform well on new, previously unseen inputs, not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization.

Typically, when training a machine learning model, we have access to a training set, we can compute some error measure on the training set called the training error, and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the generalization error, also called the test error, to be low as well. The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

We typically estimate the generalization error of a machine learning model by measuring its performance on a test set of examples that were collected separately from the training set. The factors determining how well a machine learning algorithm will perform are is its ability to: 1) make the training error small, and 2) make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: underfitting and overfitting. Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its capacity. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set. Machine learning algorithms will generally perform best

when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task they may overfit.

Capacity is not determined only by the choice of model. The model specifies which family of functions the learning algorithm can choose from when varying the parameters in order to reduce a training objective. This is called the representational capacity of the model. In many cases, finding the best function within this family is a very difficult optimization problem. In practice, the learning algorithm does not actually find the best function, but merely one that significantly reduces the training error. These additional limitations, such as the imperfection of the optimization algorithm, mean that the learning algorithm's effective capacity may be less than the representational capacity of the model family.

While simpler functions are more likely to generalize (to have a small gap between training and test error) we must still choose a sufficiently complex hypothesis to achieve low training error. Typically, training error decreases until it asymptotes to the minimum possible error value as model capacity increases (assuming the error measure has a minimum value). Typically, generalization error has a U-shaped curve as a function of model capacity.

Non-parametric models reach the most extreme case of arbitrarily high capacity. Sometimes, non-parametric models are just theoretical abstractions (such as an algorithm that searches over all possible probability distributions) that cannot be implemented in practice. However, practical non-parametric models can be designed by making their complexity a function of the training set size. One example of such an algorithm is nearest neighbor regression. Unlike linear regression, a parametric model

which learns a function described by a parameter vector whose size is finite and fixed before any data is observed, the nearest neighbor regression model simply stores the X and y from the training set. When asked to classify a test point x, the model looks up the nearest entry in the training set and returns the associated regression target. If the algorithm is allowed to break ties by averaging the y_i values for all X_i that are tied for nearest, then this algorithm is able to achieve the minimum possible training error on any regression dataset.

The ideal model is an oracle that simply knows the true probability distribution that generates the data. Even such a model will still incur some error on many problems, because there may still be some noise in the distribution. The error incurred by an oracle making predictions from the true distribution $p(x, y)$ is called the Bayes error.

Training and generalization error vary as the size of the training set varies. Expected generalization error can never increase as the number of training examples increases. For non-parametric models, more data yields better generalization until the best possible error is achieved. Any fixed parametric model with less than optimal capacity will asymptote to an error value that exceeds the Bayes error. Note that it is possible for the model to have optimal capacity and yet still have a large gap between training and generalization error. In this situation, we may be able to reduce this gap by gathering more training examples.

1.2.4.1 Regularization

The no free lunch theorem [15] implies that we must design our machine learning algorithms to perform well on a specific task. We do so by building a set of preferences into the learning algorithm. When these preferences are aligned with the learning problems we ask the algorithm to solve, it performs better. One method of modifying a learning algorithm is to increase or decrease the model's representational capacity

by adding or removing functions from the hypothesis space of solutions the learning algorithm is able to choose.

The behavior of our algorithm is strongly affected not just by how large we make the set of functions allowed in its hypothesis space, but by the specific identity of those functions. For example, linear regression has a hypothesis space consisting of the set of linear functions of its input. These linear functions can be very useful for problems where the relationship between inputs and outputs truly is close to linear. They are less useful for problems that behave in a very nonlinear fashion. We can thus control the performance of our algorithms by choosing what kind of functions we allow them to draw solutions from, as well as by controlling the amount of these functions.

We can also give a learning algorithm a preference for one solution in its hypothesis space to another. This means that both functions are eligible, but one is preferred. The unpreferred solution will be chosen only if it fits the training data significantly better than the preferred solution.

For example, we can modify the training criterion for linear regression to include weight decay. To perform linear regression with weight decay, we minimize a sum $J(w)$ comprising both the mean squared error on the training and a criterion that expresses a preference for the weights to have a smaller squared L_2 norm. Specifically,

$$J(w) = MSE_{train} + \lambda w^T w \tag{3}$$

where λ is a value chosen ahead of time that controls the strength of our preference for smaller weights. When $\lambda = 0$, we impose no preference, and larger λ forces the weights to become smaller. Minimizing $J(w)$ results in a choice of weights that make a tradeoff between fitting the training data and being small. This gives us solutions that have a smaller slope, or put weight on fewer of the features.

More generally, we can regularize a model that learns a function $f(x; \theta)$ by adding a penalty called a regularizer to the cost function. In the case of weight decay, the

regularizer is $w^T w$. There are many other ways of expressing preferences for different solutions, both implicitly and explicitly.

Expressing preferences for one function over another is a more general way of controlling a model's capacity than including or excluding members from the hypothesis space. We can think of excluding a function from a hypothesis space as expressing an infinitely strong preference against that function. Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

1.2.4.2 Hyperparameters and Validation Sets

Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm. These settings are called hyperparameters. The values of hyperparameters are not adapted by the learning algorithm itself. The λ value used to control the strength of weight decay is an example of a hyperparameter.

Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because it is difficult to optimize. More frequently, the setting must be a hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies to all hyperparameters that control model capacity. If learned on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting.

To solve this problem, we need a validation set of examples that the training algorithm does not observe. As the test set is used to estimate the generalization error of a learner, it is important that the test examples are not used in any way to make choices about the model, including its hyperparameters. Therefore, we always construct the validation set from the training data. Specifically, we split the training data into two disjoint subsets. One of these subsets is used to learn the parameters. The other

subset is our validation set, used to estimate the generalization error during or after training, allowing for the hyperparameters to be updated accordingly. Typically, one uses about 80% of the training data for training and 20% for validation. After all hyperparameter optimization is complete, the generalization error may be estimated using the test set.

Dividing the dataset into a fixed training set and a fixed test set can be problematic if it results in the test set being small. A small test set implies statistical uncertainty around the estimated average test error, making it difficult to claim that algorithm A works better than algorithm B on the given task. When the dataset is too small, alternative procedures enable one to use all of the examples in the estimation of the mean test error, at the price of increased computational cost. These procedures are based on the idea of repeating the training and testing computation on different randomly chosen subsets or splits of the original dataset. The most common of these is the k-fold cross-validation procedure, in which a partition of the dataset is formed by splitting it into k non-overlapping subsets. The test error may then be estimated by taking the average test error across k trials. On trial i, the i-th subset of the data is used as the test set and the rest of the data is used as the training set.

1.2.5 Stochastic Gradient Descent

Gradient descent is used to minimize cost functions by updating the model parameters. However, a recurring problem in machine learning is that large training sets are necessary for good generalization, but large training sets are also more computationally expensive. The cost function used by a machine learning algorithm often decomposes as a sum over training examples of some per-example loss function. For these additive cost functions, the computational cost of gradient descent is linear in the number of examples. As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long.

The insight of stochastic gradient descent is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples. Specifically, on each step of the algorithm, we can sample a minibatch of examples drawn uniformly from the training set. The minibatch size m' is typically chosen to be a relatively small number of examples, ranging from 1 to a few hundred. Crucially, m' is usually held fixed as the training set size m grows. We may fit a training set with billions of examples using updates computed on only a hundred examples.

For a fixed model size, the cost per SGD update does not depend on the training set size m. In practice, we often use a larger model as the training set size increases, but we are not forced to do so. The number of updates required to reach convergence usually increases with training set size. However, as m approaches infinity, the model will eventually converge to its best possible test error before SGD has sampled every example in the training set. Increasing m further will not extend the amount of training time needed to reach the model's best possible test error. From this point of view, one can argue that the asymptotic cost of training a model with SGD is $O(1)$ as a function of m. Enhancements to SGD used in deep learning will be discussed in the subsequent section.

1.3 Deep Learning

The performance of simple machine learning algorithms depends heavily on the representation of the data they are given. This dependence on representations is a general phenomenon that appears throughout computer science and even daily life. In computer science, operations such as searching a collection of data can proceed exponentially faster if the collection is structured and indexed intelligently. People can easily perform arithmetic on Arabic numerals, but find arithmetic on Roman numerals much more time-consuming. It is not surprising that the choice of representation has an enormous effect on the performance of machine learning algorithms.

Many artificial intelligence tasks can be solved by designing the right set of features to extract for that task, then providing these features to a simple machine learning algorithm. For example, a useful feature for speaker identification from sound is an estimate of the size of the speaker's vocal tract. It therefore gives a strong clue as to whether the speaker is a man, woman, or child.

However, for many tasks, it is difficult to know what features should be extracted. For example, suppose that we would like to write a program to detect cars in photographs. We know that cars have wheels, so we might like to use the presence of a wheel as a feature. Unfortunately, it is difficult to describe exactly what a wheel looks like in terms of pixel values. A wheel has a simple geometric shape but its image may be complicated by shadows falling on the wheel, the sun glaring off the metal parts of the wheel, the fender of the car or an object in the foreground obscuring part of the wheel, and so on.

One solution to this problem is to use machine learning to discover not only the mapping from representation to output but also the representation itself. This approach is known as representation learning. Learned representations often result in much better performance than can be obtained with hand-designed representations. They also allow AI systems to rapidly adapt to new tasks, with minimal human intervention. A representation learning algorithm can discover a good set of features for a simple task in minutes, or a complex task in hours. Manually designing features for a complex task requires a great deal of human time and effort; it can take decades for an entire community of researchers.

When designing features or algorithms for learning features, our goal is usually to separate the factors of variation that explain the observed data. In this context, we use the word "factors" simply to refer to separate sources of influence; the factors are usually not combined by multiplication. Such factors are often not quantities that are

directly observed. Instead, they may exist either as unobserved objects or unobserved forces in the physical world that affect observable quantities. They may also exist as constructs in the human mind that provide useful simplifying explanations or inferred causes of the observed data. They can be thought of as concepts or abstractions that help us make sense of the rich variability in the data. When analyzing a speech recording, the factors of variation include the speaker's age, their sex, their accent and the words that they are speaking. When analyzing an image of a car, the factors of variation include the position of the car, its color, and the angle and brightness of the sun.

A major source of difficulty in many real-world artificial intelligence applications is that many of the factors of variation influence every single piece of data we are able to observe. The individual pixels in an image of a red car might be very close to black at night. The shape of the car's silhouette depends on the viewing angle. Most applications require us to disentangle the factors of variation and discard the ones that we do not care about.

Of course, it can be very difficult to extract such high-level, abstract features from raw data. Many of these factors of variation, such as a speaker's accent, can be identified only using sophisticated, nearly human-level understanding of the data. When it is nearly as difficult to obtain a representation as to solve the original problem, representation learning does not, at first glance, seem to help us.

Deep learning solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations. Deep learning allows the computer to build complex concepts out of simpler concepts. It achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones.

1.3.1 Motivation: The Curse of Dimensionality

Many machine learning problems become exceedingly difficult when the number of dimensions in the data is high. This phenomenon is known as the curse of dimensionality. Of particular concern is that the number of possible distinct configurations of a set of variables increases exponentially as the number of variables increases.

To understand the issue, let us consider that the input space is organized into a grid. We can describe low-dimensional space with a low number of grid cells that are mostly occupied by the data. When generalizing to a new data point, we can usually tell what to do simply by inspecting the training examples that lie in the same cell as the new input. If we wish to classify an example, we can return the most common class of training examples in the same cell. If we are doing regression we can average the target values observed over the examples in that cell. But what about the cells for which we have seen no example? Because in high-dimensional spaces the number of configurations is huge, much larger than our number of examples, a typical grid cell has no training example associated with it. How could we possibly say something meaningful about these new configurations? Many traditional machine learning algorithms simply assume that the output at a new point should be approximately the same as the output at the nearest training point. This assumption is called the smoothness prior or local constancy prior. This prior states that the function we learn should not change very much within a small region.

Many simpler algorithms rely exclusively on this prior to generalize well, and as a result they fail to scale to the statistical challenges involved in solving AI-level tasks. Deep learning introduces additional priors in order to reduce the generalization error on sophisticated tasks. The core idea in deep learning is that we assume that the data was generated by the composition of factors or features, potentially at multiple levels in a hierarchy. This apparently mild assumption allow an exponential gain in

the relationship between the number of examples and the number of regions that can be distinguished. The exponential advantages conferred by the use of deep, distributed representations counter the exponential challenges posed by the curse of dimensionality.

1.3.2 Feedforward Networks

The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation. These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y.

Feedforward neural networks are called networks because they are typically represented by composing together many different functions, commonly through a chain structure. The overall length of the chain gives the depth of the model, and each member of the chain is a layer. The learning algorithm must decide how to use these layers to best implement an approximation of f^*. Because the training data does not show the desired output for each of these layers, these layers are called hidden layers.

Finally, these networks are called neural because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the width of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value. The idea of using many layers of vector-valued representation is drawn from neuroscience. The choice of the functions $f_i(x)$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute.

Neural networks use an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an activation function, to describe the features: $h = g(W^T x + c)$, where W provides the weights of a linear transformation and c the biases. In modern neural networks, the default recommendation for the activation function is to use the rectified linear unit or ReLU [16–18] defined by $g(z) = \max\{0, z\}$. Generalizations of rectified linear units exist, such as leaky ReLU [19], parametric ReLU or PReLU [20], and maxout units [21].

1.3.3 Gradient-based Learning

Designing and training a neural network is not much different from training any other machine learning model with gradient descent. The largest difference is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value. Convex optimization converges starting from any initial parameters, while stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values.

1.3.3.1 Cost Functions

An important aspect of the design of a deep neural network is the choice of the cost function. In most cases, our parametric model defines a distribution $p(y|x;\theta)$ and we simply use the principle of maximum likelihood. This means we use the cross-entropy between the training data and the model's predictions as the cost function. The total cost function used to train a neural network will often combine one of the primary cost functions described here with a regularization term. The weight decay approach used

for linear models is also directly applicable to deep neural networks and is among the most popular regularization strategies. More advanced regularization strategies exist as well, such as dataset augmentation, early stopping, dropout [22], and adversarial training [23, 24].

The cross-entropy loss is given by,

$$J(\theta) = -\mathbb{E}_{x,y\sim p_{data}} \log p_{model}(y|x) \tag{4}$$

The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{model}$. For example, if $p_{model} = N(y; f(x;\theta), I)$, assumed to be Gaussian, we recover the mean square cost.

In order to train neural networks successfully, the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm. Functions that saturate (become very flat) undermine this objective because they make the gradient become very small. In many cases this happens because the activation functions used to produce the output of the hidden units or the output units saturate. The negative log-likelihood helps to avoid this problem for many models. Many output units involve an exp function that can saturate when its argument is very negative. The log function in the negative log-likelihood cost function undoes the exp of often used output units.

Alternative cost functions to the cross-entropy loss can be derived. For example, if we could train on infinitely many samples from the true data generating distribution, minimizing the mean squared error cost function gives a function that predicts the mean of y for each value of x. Likewise, in that scenario minimizing the mean absolute error cost function gives a function that predicts the median of y for each value of x. Unfortunately, mean squared error and mean absolute error often lead to poor results when used with gradient-based optimization. Some output units that saturate produce very small gradients when combined with these cost functions. This is the

major reason why the cross-entropy cost function is more popular than mean squared error or mean absolute error, even when it is not necessary to estimate an entire distribution $p(y|x)$.

1.3.3.2 Output Units

Linear Units One simple kind of output unit is an output unit based on an affine transformation with no nonlinearity. These are often just called linear units. Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms.

Sigmoid Units Many tasks require predicting the value of a binary variable y. Classification problems with two classes can be cast in this form. The maximum-likelihood approach is to define a Bernoulli distribution over y conditioned on x. A Bernoulli distribution is defined by just a single number. The neural net needs to predict only $P(y = 1|x)$. For this number to be a valid probability, it must lie in the interval [0,1]. To ensure there is always a strong gradient whenever the model has a wrong answer, the approach typically used is based on using sigmoid output units combined with maximum likelihood. When we use other loss functions, such as mean squared error, the loss can saturate anytime the logistic sigmoid function saturates. The gradient can shrink too small to be useful for learning whenever this happens, whether the model has the correct answer or the incorrect answer. For this reason, maximum likelihood is almost always the preferred approach to training sigmoid output units.

Softmax Units Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. This can

be seen as a generalization of the sigmoid function which was used to represent a probability distribution over a binary variable.

1.3.3.3 Backpropagation

When we use a feedforward neural network to accept an input x and produce an output $\hat{y}$, information flows forward through the network. The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{y}$. This is called forward propagation. During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$. The backpropagation algorithm [25] allows the information from the cost to then flow backwards through the network, in order to compute the gradient.

Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The backpropagation algorithm does so using a simple and inexpensive procedure.

The chain rule of calculus is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Backpropagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient. Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar. However, actually evaluating that expression in a computer introduces some extra considerations. Specifically, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient will need to choose whether to store these subexpressions or to recompute them several times. In some cases, computing the same subexpression twice would simply be wasteful. For complicated graphs, there can be exponentially many of these wasted computations, making a naive implementation of the chain rule infeasible. In other cases, computing the same subexpression

twice could be a valid way to reduce memory consumption at the cost of higher runtime.

The amount of computation required for performing backpropagation scales linearly with the number of edges in the computational graph, where the computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents) as well as performing one multiplication and one addition. The backpropagation algorithm is designed to reduce the number of common subexpressions without regard to memory. Specifically, it performs on the order of one Jacobian product per node in the graph. Backpropagation thus avoids the exponential explosion in repeated subexpressions.

Backpropagation computes the gradient with respect to each parent of node z by multiplying the current gradient by the Jacobian of the operation that produced z. We continue multiplying by Jacobians traveling backwards through the graph in this way until we reach x. For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

If the neural network cost function is roughly chain-structured, which is often true, backpropagation has $O(N)$ cost. This is far better than the naive approach, which might need to execute exponentially many nodes. To avoid recomputation, we can think of backpropagation as a table-filling algorithm that takes advantage of storing intermediate results. Each node in the graph has a corresponding slot in the table to store the gradient for that node. By filling in these table entries in order, backpropagation avoids repeating many common subexpressions. This table-filling strategy is commonly called dynamic programming.

The backpropagation algorithm is a special case of a broader class of techniques called reverse mode accumulation, which is an automatic differentiation strategy.

When the number of outputs of the graph is larger than the number of inputs, it is sometimes preferable to use another form of automatic differentiation called forward mode accumulation. This also avoids the need to store the values and gradients for the whole graph, trading off computational efficiency for memory.

1.3.3.4 Initialization Strategies

Training algorithms for deep learning models are usually iterative in nature and thus require the user to specify some initial point from which to begin the iterations. Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether. When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization as well.

Modern initialization strategies are simple and heuristic. Designing improved initialization strategies is a difficult task because neural network optimization is not yet well understood. Perhaps the only property known with complete certainty is that the initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way. Even if the model or training algorithm is capable of using stochasticity to compute different updates for different units (for example, if one trains with dropout), it is usually best to initialize each unit to compute a different function from all of the other units. This may help to

make sure that no input patterns are lost in the null space of forward propagation and no gradient patterns are lost in the null space of backpropagation. The goal of having each unit compute a different function motivates random initialization of the parameters.

Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly. We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The choice of Gaussian or uniform distribution does not seem to matter very much, but has not been exhaustively studied. The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

Larger initial weights will yield a stronger symmetry breaking effect, helping to avoid redundant units. They also help to avoid losing signal during forward or backpropagation through the linear component of each layer—larger values in the matrix result in larger outputs of matrix multiplication. Initial weights that are too large may, however, result in exploding values during forward propagation or backpropagation. Large weights may also result in extreme values that cause the activation function to saturate, causing complete loss of gradient through saturated units. These competing factors determine the ideal initial scale of the weights.

The perspectives of regularization and optimization can give very different insights into how we should initialize a network. The optimization perspective suggests that the weights should be large enough to propagate information success- fully, but some regularization concerns encourage making them smaller. The use of an optimization algorithm such as stochastic gradient descent that makes small incremental changes to the weights and tends to halt in areas that are nearer to the initial parameters (whether due to getting stuck in a region of low gradient, or triggering some early

stopping criterion based on overfitting) expresses a prior that the final parameters should be close to the initial parameters.

Some heuristics are available for choosing the initial scale of the weights. [26] suggest using a normalized initialization designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance. The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no nonlinearities. Real neural networks obviously violate this assumption, but many strategies designed for the linear model perform reasonably well on its nonlinear counterparts.

The approach for setting the biases must be coordinated with the approach for setting the weights. Setting the biases to zero is compatible with most weight initialization schemes.

1.3.3.5 Faster Optimizers

Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular. As discussed, it is possible to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data generating distribution.

A crucial parameter for the SGD algorithm is the learning rate. Previously, we have described SGD as using a fixed learning rate ϵ. In practice, it is necessary to gradually decrease the learning rate over time. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum. By comparison, the true gradient of the total cost function becomes small and then 0 when we approach and reach a minimum using batch gradient descent, so batch gradient descent can use a

fixed learning rate. In practice, it is common to decay the learning rate linearly until iteration τ, and after that it is common to leave ϵ constant.

The most important property of SGD and related minibatch or online gradient- based optimization is that computation time per update does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large. For a large enough dataset, SGD may converge to within some fixed tolerance of its final test set error before it has processed the entire training set. Despite this property, SGD can sometimes be quite slow, and thus faster optimizers have been developed.

Momentum The method of momentum [27] is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Formally, the momentum algorithm introduces a variable v that plays the role of velocity, it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient. The name momentum derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector v may also be regarded as the momentum of the particle. A hyperparameter α determines how quickly the contributions of previous gradients exponentially decay. The velocity v accumulates the gradient elements. The larger α is relative to ϵ, the more previous gradients affect the current direction.

Previously, the size of the step was simply the norm of the gradient multiplied by the learning rate. Now, the size of the step depends on how large and how aligned a sequence of gradients are. The step size is largest when many successive gradients point in exactly the same direction.

Nesterov Momentum [28] introduced a variant of the momentum algorithm that was inspired by Nesterov's accelerated gradient method [29]. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard method of momentum.

AdaGrad The AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values [30]. The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

In the context of convex optimization, the AdaGrad algorithm enjoys some desirable theoretical properties. However, empirically it has been found that, for training deep neural network models, the accumulation of squared gradients from the beginning of training can result in a premature and excessive decrease in the effective learning rate. AdaGrad performs well for some but not all deep learning models.

RMSProp The RMSProp algorithm [31] modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially

weighted moving average. AdaGrad is designed to converge rapidly when applied to a convex function. When applied to a non-convex function to train a neural network, the learning trajectory may pass through many different structures and eventually arrive at a region that is a locally convex bowl. AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure. RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

Adam Adam [32] is yet another adaptive learning rate optimization algorithm. Its name derives from the phrase "adaptive moments." In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin. RMSProp also incorporates an estimate of the (uncentered) second-order moment, however it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is gen-

erally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

In the current state of deep learning, Adam is the optimizer of choice.

1.3.4 Convolutional Networks

Convolutional networks [33] is a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

1.3.4.1 Convolution

Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant representations. Moreover, convolution provides a means for working with inputs of variable size.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means every output unit interacts with every input unit. Convolutional networks, however, typically have sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels.

This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime.

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation, it is still $O(k \times n)$, but it does further reduce the storage requirements of the model to k parameters. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called equivariance to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. When processing time series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later in

time. Similarly with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image. In some cases, we may not wish to share parameters across the entire image. For example, if we are processing images that are cropped to be centered on an individual's face, we probably want to extract different features at different locations, the part of the network processing the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin.

1.3.4.2 Pooling

A typical layer of a convolutional network consists of three stages. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage. In the third stage, we use a pooling function to modify the output of the layer further. A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood.

In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is. For example,

when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In other contexts, it is more important to preserve the location of a feature.

For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of an offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size.

1.3.4.3 Convolution and Pooling

One key insight is that convolution and pooling can cause underfitting. Like any prior, convolution and pooling are only useful when the assumptions made by the prior are reasonably accurate. If a task relies on preserving precise spatial information, then using pooling on all features can increase the training error. Some convolutional network architectures [34] are designed to use pooling on some channels but not on other channels, in order to get both highly invariant features and features that will not underfit when the translation invariance prior is incorrect. When a task involves incorporating information from very distant locations in the input, then the prior imposed by convolution may be inappropriate.

1.3.5 Recurrent Networks

Recurrent networks are a family of neural networks for processing sequential data. Much as a convolutional network is a neural network that is specialized for processing a grid of values X such as an image, a recurrent neural network is a neural network

that is specialized for processing a sequence of values $x_1, ..., x_n$. Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length. Parameter sharing makes it possible to extend and apply the model to examples of different forms (different lengths, here) and generalize across them. A recurrent neural network shares the same weights across several time steps.

One way to draw an RNN is as an unfolded computational graph, in which each component is represented by many different variables, with one variable per time step, representing the state of the component at that point in time. Computing the gradient through a recurrent neural network is straightforward, one simply applies the generalized back-propagation algorithm to the unrolled computational graph. No specialized algorithms are necessary. Gradients obtained by back-propagation may then be used with any general-purpose gradient-based techniques to train an RNN.

1.3.5.1 RNN Architectures

Bidirectional RNNs [35] combine an RNN that moves forward through time beginning from the start of the sequence with another RNN that moves backward through time beginning from the end of the sequence. This allows the output units to compute a representation that depends on both the past and the future but is most sensitive to the input values around time t, without having to specify a fixed-size window around t.

Encoder-decoder RNNs [36] can be trained to map an input sequence to an output sequence which is not necessarily of the same length. The idea is straightforward: (1) an encoder or reader or input RNN processes the input sequence. The encoder emits the context C, usually as a simple function of its final hidden state. (2) a decoder

or writer or output RNN is conditioned on that fixed-length vector to generate the output sequence $Y = (y_1, ..., y_n)$. One clear limitation of this architecture is when the context C output by the encoder RNN has a dimension that is too small to properly summarize a long sequence. This phenomenon was observed by [37] in the context of machine translation. They proposed to make C a variable-length sequence rather than a fixed-size vector. Additionally, they introduced an attention mechanism that learns to associate elements of the sequence C to elements of the output sequence.

1.3.5.2 LSTM and Other Gated RNNs

The basic problem of learning long-term dependencies is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones. The vanishing and exploding gradient problem for RNNs was independently discovered by separate researchers [38, 39]. One may hope that the problem can be avoided simply by staying in a region of parameter space where the gradients do not vanish or explode. Unfortunately, in order to store memories in a way that is robust to small perturbations, the RNN must enter a region of parameter space where gradients vanish. Specifically, whenever the model is able to represent long term dependencies, the gradient of a long term interaction has exponentially smaller magnitude than the gradient of a short term interaction. It does not mean that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies.

Gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode. The long short-term memory (LSTM) model's [40] core contribution is introducing self-loops to produce paths where the gradient can flow for long durations. By making the weight of the self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. Instead of a unit that simply applies an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity.

Gated recurrent units or GRUs [36] differ from LSTMs in that a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit.

2 Adversarial Examples

In many cases, neural networks have begun to reach human performance when evaluated on an i.i.d. test set. It is natural therefore to wonder whether these models have obtained a true human-level understanding of these tasks. In order to probe the level of understanding a network has of the underlying task, we can search for examples that the model misclassifies. [23] found that even neural networks that perform at human level accuracy have a nearly 100% error rate on examples that are intention-

ally constructed by using an optimization procedure to search for an input x' near a data point x such that the model output is very different at x'. In many cases, x' can be so similar to x that a human observer cannot tell the difference between the original example and the adversarial example, but the network can make highly different predictions.

The lack of robustness exhibited by deep neural networks (DNNs) to adversarial examples has raised serious concerns for security-critical applications, including traffic sign identification and malware detection, among others. Moreover, moving beyond the digital space, researchers have shown that these adversarial examples are still effective in the physical world at fooling DNNs [41, 42]. Due to the robustness and security implications, the means of crafting adversarial examples are called *attacks* to DNNs. In particular, *targeted attacks* aim to craft adversarial examples that are misclassified as specific target classes, and *untargeted attacks* aim to craft adversarial examples that are not classified as the original class. In addition to evaluating the robustness of DNNs, adversarial examples can be used as a regularization tool to train a robust model that is resilient to adversarial perturbations, known as *adversarial training* [23, 24, 43]. They have also been used in interpreting DNNs [44, 45].

Here we summarize related works on attacking and defending DNNs against adversarial examples.

2.1 Adversarial Attacks

2.1.1 White-box Attacks

Let $\mathbf{x}_0$ and $\mathbf{x}$ denote the original and adversarial examples, respectively, let l denote the correct label, and let t denote the target class to attack.

2.1.1.1 L-BFGS

Szegedy et al. [23] generated adversarial examples using box-constrained L-BFGS. Given an image $\mathbf{x}_0$, their method finds a different image $\mathbf{x}$ that is similar to $\mathbf{x}_0$ under L_2 distance, yet is labeled differently by the classifier. They model the problem as a constrained minimization problem:

$$\begin{aligned} &\text{minimize} \quad \|\mathbf{x}_0 - \mathbf{x}\|_2^2 \\ &\text{such that} \quad C(\mathbf{x}) = l \\ &\text{where} \quad \mathbf{x} \in [0,1]^p \end{aligned} \tag{5}$$

This problem can be very difficult to solve, however, so Szegedy et al. instead solve the following problem:

$$\begin{aligned} &\text{minimize} \quad \|\mathbf{x}_0 - \mathbf{x}\|_2^2 + c \cdot loss(\mathbf{x}, l) \\ &\text{such that} \quad \mathbf{x} \in [0,1]^p \end{aligned} \tag{6}$$

where $loss(\mathbf{x})$ is a function mapping an image to a positive real number quantifying the success of the attack. One common loss function to use is cross-entropy. Line search is performed to find the constant $c > 0$ that yields an adversarial example of minimum distance: in other words, we repeatedly solve this optimization problem for multiple values of c, adaptively updating c using bisection search or any other method for one-dimensional optimization. The box constraint is handled natively by the L-BFGS optimization algorithm.

2.1.1.2 Fast Gradient Methods

Fast gradient methods (FGM) use the gradient ∇J of the training loss J with respect to $\mathbf{x}_0$ for crafting adversarial examples [24]. For the L_∞ attack, the fast gradient sign method, $\mathbf{x}$ is crafted by

$$\mathbf{x} = \mathbf{x}_0 - \epsilon \cdot \text{sign}(\nabla J(\mathbf{x}_0, t)) \tag{7}$$

where ϵ specifies the L_∞ distortion between $\mathbf{x}$ and $\mathbf{x}_0$, and $\text{sign}(\nabla J)$ takes the sign of the gradient. Intuitively, for each pixel, the fast gradient sign method uses the gradient of the loss function to determine in which direction the pixel's intensity should be changed (whether it should be increased or decreased) to minimize the loss function; then, it shifts all pixels simultaneously. For L_1 and L_2 attacks, x is crafted by,

$$\mathbf{x} = \mathbf{x}_0 - \epsilon \frac{\nabla J(\mathbf{x}_0, t)}{\|\nabla J(\mathbf{x}_0, t)\|_q} \tag{8}$$

for $q = 1, 2$, where ϵ specifies the corresponding distortion. Untargeted attacks can be implemented in a similar fashion.

2.1.1.3 Iterative Fast Gradient Methods

Iterative fast gradient methods (I-FGM) were proposed in [46], which iteratively use FGM with a finer distortion, followed by an ϵ-ball clipping.In [43], PGD is introduced, where I-FGM is modified to incorporate random starts.

2.1.1.4 JSMA

Papernot et al. proposed a Jacobian-based saliency map algorithm (JSMA) optimized under L_0 distance for characterizing the input-output relation of DNNs [47]. It can be viewed as a greedy attack algorithm that iteratively modifies the most influential pixel for crafting adversarial examples.

2.1.1.5 DeepFool

DeepFool is an untargeted L_2 attack algorithm [48] based on the theory of projection to the closest separating hyperplane in classification. The authors construct DeepFool by imagining that the neural networks are totally linear, with a hyperplane separating each class from another. From this, they analytically derive the optimal solution to this simplified problem, and construct the adversarial example. Then, since neural

networks are not actually linear, they take a step towards that solution, and repeat the process a second time. The search terminates when a true adversarial example is found.

2.1.1.6 C&W Attack

Instead of leveraging the training loss, Carlini and Wagner designed an L_2-regularized loss function based on the logit layer representation in DNNs for crafting adversarial examples [49]. They rely on the initial formulation of adversarial examples presented in Szegedy et al. [23], and construct various loss functions better suited for optimization. Each of these loss functions outperform the cross-entropy loss used in [23], and the best objective function, which is utilized in the formulation, is the following,

$$f(\mathbf{x}, t) = \max\{\max_{j \neq t} [\mathbf{Logit}(\mathbf{x})]_j - [\mathbf{Logit}(\mathbf{x})]_t, -\kappa\} \tag{9}$$

where $\mathbf{Logit}(\mathbf{x}) = [[\mathbf{Logit}(\mathbf{x})]_1, \ldots, [\mathbf{Logit}(\mathbf{x})]_K] \in \mathbb{R}^K$ is the logit layer (the layer prior to the softmax layer) representation of $\mathbf{x}$ in the considered DNN, K is the number of classes for classification, and $\kappa \geq 0$ is a confidence parameter that guarantees a constant gap between $\max_{j \neq t} [\mathbf{Logit}(\mathbf{x})]_j$ and $[\mathbf{Logit}(\mathbf{x})]_t$. The loss function in (9) aims to render the label t the most probable class for $\mathbf{x}$, and the parameter κ controls the separation between t and the next most likely prediction among all classes other than t. For untargeted attacks, the loss function can be modified in a similar fashion.

The constant c in (6) is found via modified binary search. The Adam optimizer [32] is used, which does not natively support box constraints, thus Carlini and Wagner remove the box constraint by replacing $\mathbf{x}$ with $\frac{1+\tanh \mathbf{w}}{2}$, where $\mathbf{w} \in \mathbb{R}^p$. By using this change-of-variable, the optimization problem in (6) becomes an unconstrained minimization problem with $\mathbf{w}$ as an optimizer, and Adam can be applied for solving for the optimal $\mathbf{w}$ and obtain the corresponding adversarial example $\mathbf{x}$.

The C&W attack is considered to be one of the strongest attacks to DNNs, and is the algorithm we both build from and compare to.

2.1.2 Black-box Attacks

Due to its feasibility, the case where an attacker can have free access to the input and output of a targeted DNN while still being prohibited from performing backpropagation on the targeted DNN has been called a practical black-box attack setting for DNNs [49, 50]. Under this attack setting, existing attacking approaches tend to make use of the power of free query to train a *substitute model* [50], which is a representative substitute of the targeted DNN. The substitute model can then be attacked using any white-box attack techniques, and the generated adversarial images are used to attack the target DNN. The primary advantage of training a substitute model is its total transparency to an attacker, and hence essential attack procedures for DNNs, such as backpropagation for gradient computation, can be implemented on the substitute model for crafting adversarial examples. Moreover, since the substitute model is representative of a targeted DNN in terms of its classification rules, adversarial attacks to a substitute model are expected to be similar to attacking the corresponding targeted DNN. In other words, adversarial examples crafted from a substitute model can be highly transferable to the targeted DNN given the ability of querying the targeted DNN at will.

2.1.3 No-box Attacks

A no-box attack refers to the most challenging case where attacker is not only prohibited from performing backpropagation on the targeted DNN, but is unable to query any information from the targeted classifier for adversarial attacks. In this case, one cannot query the targeted DNN to train a substitute model, and must thus rely solely on the property of transferability.

In the context of adversarial attacks, *transferability* means that the adversarial examples generated from one model are also very likely to be misclassified by another model. One possible explanation of inherent attack transferability for DNNs lies in the findings that DNNs commonly have overwhelming generalization power and excessive linearity [23, 24]. Ensemble methods can be used for generating high-confidence adversarial examples for targeted transfer attacks [51]. More interestingly, the authors in [52] have shown that a carefully crafted universal perturbation to a set of natural images can lead to misclassification of all considered images with high probability, suggesting the possibility of attack transferability from one image to another. Further analysis and justification of a universal perturbation is given in [53].

2.2 Defenses to Adversarial Attacks

2.2.1 Adversarial Training

Adversarial training is training on adversarially perturbed examples from the training set [24]. Adversarial training discourages the highly sensitive locally linear behavior neural networks exhibit by encouraging the network to be locally constant in the neighborhood of the training data. This can be seen as a way of explicitly introducing a local constancy prior into supervised neural nets. The idea was first introduced in [23] but was not yet practical because of the high computation cost of generating adversarial examples. [24] showed how to generate adversarial examples inexpensively with the fast gradient sign method and made it computationally efficient to generate large batches of adversarial examples during the training process. After training to resist these cheap adversarial examples, the model is usually successfully able to resist new instances of the same kind of cheap adversarial example. However, if we then use expensive, iterative adversarial examples, like those in [49], the model is usually fooled.

Two defense methods attempt to successfully implement adversarial training, the Madry Defense Model [43] and Ensemble Adversarial Training [54]. The Madry defense model is a high capacity network trained against PGD, iterative FGM with random starts, which is deemed to be the strongest attack utilizing the local first-order information about the network. It is currently the state-of-the-art MNIST and CIFAR-10 defense. Ensemble adversarial training is a defense strategy which augments training data with perturbations transferred from other models, and was demonstrated to have strong robustness to transferred adversarial examples in the NIPS 2017 Competition on Adversarial Attacks and Defenses. It is the currently the state-of-the-art ImageNet defense. We demonstrate that these defenses can be successfully attacked using ZOO [1] and EAD [2].

2.2.2 Defensive Distillation

Defensive distillation [55] defends against adversarial perturbations by using the distillation technique in [56] to retrain the network with class probabilities predicted by the original network. It also introduces the temperature parameter T in the softmax layer for gradient masking. Gradient masking modifies the network to not give the attacker gradients to work on, an infinitesimal modification to the image causes negligible change in the output of the model. However, the attacker can train their own model, a smooth model that has a gradient, make adversarial examples for their model, and then deploy those adversarial examples successfully against the non-smooth gradient masked model.

In [49], the C&W attack was shown to break defensive distillation in the white-box case, using the unmasked logit layer representation, and in the black-box case, by generating transferable adversarial examples through increasing the margin hyperparameter κ. We demonstrate that EAD [2] can also break this defense.

2.2.3 Input Transformations

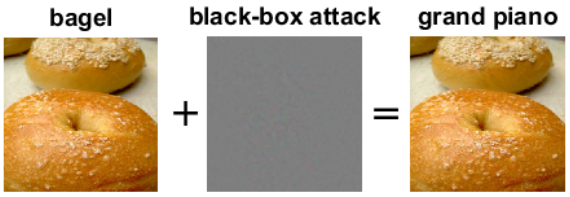
In recent work, attempts have been made to remove adversarial perturbations from the input. In [57], transformations based on image cropping and rescaling, bit-depth reduction, JPEG compression, total variance minimization, and image quilting were explored. These defenses were demonstrated to be surprisingly effective against existing attacks, and the strongest defenses were found to be total variance minimization and image quilting, due to their non-differentiable nature and inherent randomness. In [58], color bit-depth reduction and spatial smoothing were combined in a joint detection framework to achieve high detection rates against state-of-the-art attacks. We demonstrate that this joint detection method can be bypassed by EAD [2].

3 ZOO

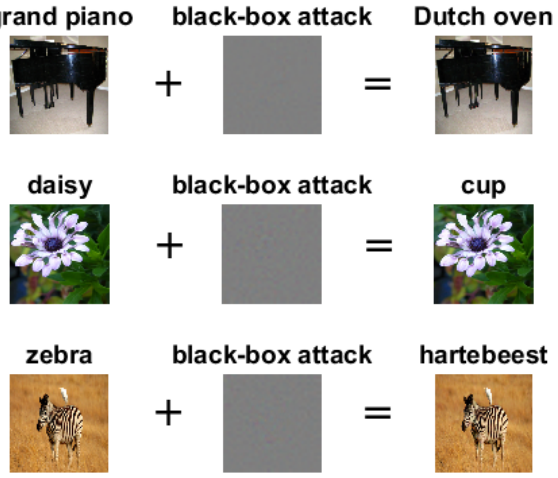
We show that a coordinate descent based method using only the zeroth order oracle (without gradient information) can effectively attack black-box DNNs. Comparing to the substitute model based black-box attack [50], our method significantly increases the success rate for adversarial attacks, and attains comparable performance to the state-of-the-art white-box attack (C&W attack).

In order to speed up the computational time and reduce number of queries for our black-box attacks to large-scale DNNs, we propose several techniques including attack-space dimension reduction, hierarchical attacks and importance sampling.

In addition to datasets of small image size (MNIST and CIFAR-10), we demonstrate the applicability of our black-box attack model to a large DNN - the Inception-v3 model [59] trained on ImageNet. Our attack is capable of crafting a successful adversarial image within a reasonable time, whereas the substitute model based black-



(a) a ZOO black-box targeted attack example



(b) ZOO black-box untargeted attack examples

Figure 1: Visual illustration of adversarial examples generated by applying our proposed black-box attack (ZOO) to sampled images from ImageNet. The columns from left to right are original images with correct labels, additive adversarial noise from our attack, and crafted adversarial images with misclassified labels.

box attack in [50] only shows success in small networks trained on MNIST and is hardly scalable to the case of ImageNet.

Finally, we attack ensemble adversarially trained networks, namely provided Inception-v3 [59] and Inception ResNet-v2 models [60][1].

3.1 Motivation

Under the black-box setting, the methodology of current attacks concentrates on training a substitute model and using it as a surrogate for adversarial attacks. In other words, a black-box attack is made possible by deploying a white-box attack

[1] `https://github.com/tensorflow/models/tree/master/research/adv_imagenet_models`

to the substitute model. Therefore, the effectiveness of such black-box adversarial attacks heavily depends on the attack transferability from the substitute model to the target model. Different from the existing approaches, we propose a black-box attack via zeroth order optimization techniques. More importantly, the proposed attack spares the need for training substitute models by enabling a "pseudo back propagation" on the target model. Consequently, our attack can be viewed "as if it was" a white-box attack to the target model, and its advantage over current black-box methods can be explained by the fact that it avoids any potential loss in transferability from a substitute model. The performance comparison between the existing methods and our proposed black-box attack will be discussed in the results section.

3.2 Algorithm

Zeroth order methods are derivative-free optimization methods, where only the zeroth order oracle (the objective function value $f(\mathbf{x})$ at any $\mathbf{x}$) is needed during optimization process. By evaluating the objective function values at two very close points $f(\mathbf{x}+h\mathbf{v})$ and $f(\mathbf{x}-h\mathbf{v})$ with a small h, a proper gradient along the direction vector $\mathbf{v}$ can be estimated. Then, classical optimization algorithms like gradient descent or coordinate descent can be applied using the estimated gradients. The convergence of these zeroth order methods has been proven in the optimization literature [61–63], and under mild assumptions (smoothness and Lipschitzian gradient) they can converge to a stationary point with an extra error term which is related to gradient estimation and vanishes when $h \to 0$.

Our proposed black-box attack to DNNs is cast as an optimization problem. It exploits the techniques from zeroth order optimization and therefore spares the need of training a substitute model for deploying adversarial attacks. Although it is intuitive to use zeroth order methods to attack a black-box DNN model, applying it naively can be impractical for large models.

For example, the Inception-v3 network [59] takes input images with a size of $299 \times 299 \times 3$, and thus has $p = 268,203$ variables (pixels) to optimize. To evaluate the estimated gradient of each pixel, we need to evaluate the model twice. To just obtain the estimated gradients of all pixels, $2p = 536,406$ evaluations are needed. For a model as large as Inception-v3, each evaluation can take tens of milliseconds on a single GPU, thus it is very expensive to even evaluate all gradients once. For targeted attacks, sometimes we need to run an iterative gradient descent with hundreds of iterations to generate an adversarial image, and it can be forbiddingly expensive to use zeroth order method in this case.

In the scenario of attacking black-box DNNs, especially when the image size is large (the variable to be optimized has a large number of coordinates), a single step of gradient descent can be very slow and inefficient, because it requires estimating the gradients of all coordinates to make a single update. Instead, we propose to use a coordinate descent method to iteratively optimize each coordinate (or a small batch of coordinates). By doing so, we can accelerate the attack process by efficiently updating coordinates after only a few gradient evaluations.

This idea is similar to DNN training for large datasets, where we usually apply stochastic gradient descent using only a small subset of training examples for efficient updates, instead of computing the full gradient using all examples to make a single update. Using coordinate descent, we update coordinates by small batches, instead of updating all coordinates in a single update as in gradient descent. Moreover, this allows us to further improve the efficiency of our algorithm by using carefully designed sampling strategy to optimize important pixels first.

3.2.1 Zeroth Order Stochastic Coordinate Descent

The attack formulation in [49] presumes a white-box attack because (i): the logit layer representation is a part of the internal state information of the DNN; and (ii) back-

propagation on the targeted DNN is required for solving the optimization problem. We amend our attack to the black-box setting by proposing the following approaches: (i) modify the loss function $f(\mathbf{x}, t)$ in (9) such that it only depends on the output F of a DNN and the desired class label t; and (ii) compute an *approximate* gradient using a finite difference method instead of actual back propagation on the targeted DNN, and solve the optimization problem via zeroth order optimization. We elucidate these two approaches below.

• **Loss function $f(\mathbf{x}, t)$ based on F:** Inspired by (9), we propose a new hinge-like loss function based on the output F of a DNN, which is defined as

$$f(\mathbf{x}, t) = \max\{\max_{i \neq t} \log[F(\mathbf{x})]_i - \log[F(\mathbf{x})]_t, -\kappa\}, \tag{10}$$

where $\kappa \geq 0$ and $\log 0$ is defined as $-\infty$. We note that $\log(\cdot)$ is a monotonic function such that for any $x, y \geq 0$, $\log y \geq \log x$ if and only if $y \geq x$. This implies that $\max_{i \neq t} \log[F(\mathbf{x})]_i - \log[F(\mathbf{x})]_t \leq 0$ means $\mathbf{x}$ attains the highest confidence score for class t. We find that the log operator is essential to our black-box attack since very often a well-trained DNN yields a skewed probability distribution on its output $F(\mathbf{x})$ such that the confidence score of one class significantly dominates the confidence scores of the other classes. The use of the log operator lessens the dominance effect while preserving the order of confidence scores due to monotonicity. Similar to (9), κ ensures a constant gap between $\max_{i \neq t} \log[F(\mathbf{x})]_i$ and $\log[F(\mathbf{x})]_t$.

For untargeted attacks, an adversarial attack is successful when $\mathbf{x}$ is classified as any class other than the original class label l. A similar loss function can be used (we drop the variable t for untargeted attacks):

$$f(\mathbf{x}) = \max\{\log[F(\mathbf{x})]_l - \max_{i \neq l} \log[F(\mathbf{x})]_i, -\kappa\}, \tag{11}$$

where l is the original class label for $\mathbf{x}$, and $\max_{i \neq l} \log[F(\mathbf{x})]_i$ represents the most probable predicted class other than l.

• **Zeroth order optimization on the loss function:** We discuss our optimization techniques for any general function f used for attacks.

We use the symmetric difference quotient [64] to estimate the gradient $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i}$ (defined as $\hat{g}_i$):

$$\hat{g}_i \coloneqq \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h}, \tag{12}$$

where h is a small constant (we set $h = 0.0001$ in all our experiments) and $\mathbf{e}_i$ is a standard basis vector with only the i-th component as 1. The estimation error (not including the error introduced by limited numerical precision) is in the order of $O(h^2)$. Although numerical accuracy is a concern, accurately estimating the gradient is usually not necessary for successful adversarial attacks. One example is FGSM, which only requires the sign (rather than the exact value) of the gradient to find adversarial examples. Therefore, even if our zeroth order estimations may not be very accurate, they suffice to achieve very high success rates, as we will show in our experiments.

For any $\mathbf{x} \in \mathbb{R}^p$, we need to evaluate the objective function $2p$ times to estimate gradients of all p coordinates. Interestingly, with just one more objective function evaluation, we can also obtain the coordinate-wise Hessian estimate (defined as $\hat{h}_i$):

$$\hat{h}_i \coloneqq \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_{ii}^2} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - 2f(\mathbf{x}) + f(\mathbf{x} - h\mathbf{e}_i)}{h^2}. \tag{13}$$

Remarkably, since $f(\mathbf{x})$ only needs to be evaluated once for all p coordinates, we can obtain the Hessian estimates without additional function evaluations.

It is worth noting that stochastic gradient descent and batch gradient descent are two most commonly used algorithms for training DNNs, and the C&W attack [49] also used gradient descent to attack a DNN in the white-box setting. Unfortunately, in the black-box setting, the network structure is unknown and the gradient computation via back propagation is prohibited. To tackle this problem, a naive solution is applying (12) to estimate gradient, which requires $2p$ objective function evaluations.

However, this naive solution is too expensive in practice. Even for an input image size of $64 \times 64 \times 3$, one full gradient descent step requires $24,576$ evaluations, and typically hundreds of iterations may be needed until convergence. To resolve this issue, we propose the following coordinate-wise update, which only requires 2 function evaluations for each step.

Algorithm 1 Stochastic Coordinate Descent

1: **while** not converged **do**
2: Randomly pick a coordinate $i \in \{1, \ldots, p\}$
3: Compute an update δ^* by approximately minimizing

$$\arg\min_{\delta} f(\mathbf{x} + \delta \mathbf{e}_i)$$

4: Update $\mathbf{x}_i \leftarrow \mathbf{x}_i + \delta^*$
5: **end while**

• **Stochastic coordinate descent:** Coordinate descent methods have been extensively studied in optimization literature [65]. At each iteration, one variable (coordinate) is chosen randomly and is updated by approximately minimizing the objective function along that coordinate (see Algorithm 1 for details). The most challenging part in Algorithm 1 is to compute the best coordinate update in step 3. After estimating the gradient and Hessian for $\mathbf{x}_i$, we can use any first or second order method to approximately find the best δ. In first-order methods, we found that Adam [32]'s update rule significantly outperforms vanilla gradient descent update and other variants in our experiments, so we propose to use a zeroth-order coordinate Adam, as described in Algorithm 2. We also use Newton's method with both estimated gradient and Hessian to update the chosen coordinate, as proposed in Algorithm 3. Note that when the Hessian is negative (indicating the objective function is concave along direction $\mathbf{x}_i$), we simply update $\mathbf{x}_i$ by its gradient. We will show the comparison of these two methods in the results section. Experimental results suggest coordinate-wise Adam is faster than Newton's method.

Algorithm 2 ZOO-ADAM: Zeroth Order Stochastic Coordinate Descent with Coordinate-wise ADAM

Require: Step size η, Adam states $M \in \mathbb{R}^p, v \in \mathbb{R}^p, T \in \mathbb{Z}^p$, Adam hyper-parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

$M \leftarrow \mathbf{0}, v \leftarrow \mathbf{0}, T \leftarrow \mathbf{0}$
while not converged **do**
  Randomly pick a coordinate $i \in \{1, \cdots, p\}$
  Estimate $\hat{g}_i$ using (12)
  $T_i \leftarrow T_i + 1$
  $M_i \leftarrow \beta_1 M_i + (1-\beta_1)\hat{g}_i, \quad v_i \leftarrow \beta_2 v_i + (1-\beta_2)\hat{g}_i^2$
  $\hat{M}_i = M_i/(1-\beta_1^{T_i}), \quad \hat{v}_i = v_i/(1-\beta_2^{T_i})$
  $\delta^* = -\eta \frac{\hat{M}_i}{\sqrt{\hat{v}_i}+\epsilon}$
  Update $\mathbf{x}_i \leftarrow \mathbf{x}_i + \delta^*$
end while

Algorithm 3 ZOO-Newton: Zeroth Order Stochastic Coordinate Descent with Coordinate-wise Newton's Method

Require: Step size η

while not converged **do**
  Randomly pick a coordinate $i \in \{1, \cdots, p\}$
  Estimate $\hat{g}_i$ and $\hat{h}_i$ using (12) and (13)
  if $\hat{h}_i \leq 0$ **then**
    $\delta^* \leftarrow -\eta \hat{g}_i$
  else
    $\delta^* \leftarrow -\eta \frac{\hat{g}_i}{\hat{h}_i}$
  end if
  Update $\mathbf{x}_i \leftarrow \mathbf{x}_i + \delta^*$
end while

Note that for algorithmic illustration we only update one coordinate for each iteration. In practice, to achieve the best efficiency of GPU, we usually evaluate the objective in batches, and thus a batch of $\hat{g}_i$ and $\hat{h}_i$ can be estimated. In our implementation we estimate $B = 128$ pixels' gradients and Hessians per iteration, and then update B coordinates in a single iteration.

3.2.2 Attack-space Dimension Reduction

We first define $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}_0$ and $\Delta\mathbf{x} \in \mathbb{R}^p$ to be the adversarial noise added to the original image $\mathbf{x}_0$. Our optimization procedure starts with $\Delta\mathbf{x} = 0$. For networks with a large input size p, optimizing over $\mathbb{R}^p$ (we call it *attack-space*) using zeroth order methods can be quite slow because we need to estimate a large number of gradients.

Instead of directly optimizing $\Delta\mathbf{x} \in \mathbb{R}^p$, we introduce a dimension reduction transformation $D(\mathbf{y})$ where $\mathbf{y} \in \mathbb{R}^m$, $range(D) \in \mathbb{R}^p$, and $m < p$. The transformation can be linear or non-linear. Then, we use $D(\mathbf{y})$ to replace $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}_0$ in the attack formulation:

$$\begin{aligned} &\text{minimize}_{\mathbf{y}} \ \|D(\mathbf{y})\|_2^2 + c \cdot f(\mathbf{x}_0 + D(\mathbf{y}), t) \\ &\text{subject to } \mathbf{x}_0 + D(\mathbf{y}) \in [0,1]^p. \end{aligned} \tag{14}$$

The use of $D(\mathbf{y})$ effectively reduces the dimension of attack-space from p to m. Note that we do not alter the dimension of an input image $\mathbf{x}$ but only reduce the permissible dimension of the adversarial noise. A convenient transformation is to define D to be the upscaling operator that resizes $\mathbf{y}$ as a size-p image, such as the bilinear interpolation method. For example, in the Inception-v3 network $\mathbf{y}$ can be a small adversarial noise image with dimension $m = 32 \times 32 \times 3$, while the original image dimension is $p = 299 \times 299 \times 3$. Other transformations like DCT (discrete cosine transformation) can also be used. We will show the effectiveness of this method in the results section.

3.2.3 Hierarchical Attack

When applying attack-space dimension reduction with a small m, although the attack-space is efficient to optimize using zeroth order methods, a valid attack might not be found due to the limited search space. Conversely, if a large m is used, a valid attack can be found in that space, but the optimization process may take a long time. Thus, for large images and difficult attacks, we propose to use a *hierarchical attack* scheme, where we use a series of transformations $D_1, D_2 \cdots$ with dimensions $m_1, m_2, \cdots$ to gradually increase m during the optimization process. In other words, at a specific iteration j (according to the dimension increasing schedule) we set $\mathbf{y}_j = D_i^{-1}(D_{i-1}(\mathbf{y}_{j-1}))$ to increase the dimension of $\mathbf{y}$ from m_{i-1} to m_i (D^{-1} denotes the inverse transformation of D).

For example, when using image scaling as the dimension reduction technique, D_1 upscales $\mathbf{y}$ from $m_1 = 32 \times 32 \times 3$ to $299 \times 299 \times 3$, and D_2 upscales $\mathbf{y}$ from $m_2 = 64 \times 64 \times 3$ to $299 \times 299 \times 3$. We start with $m_1 = 32 \times 32 \times 3$ variables to optimize with and use D_1 as the transformation, then after a certain number of iterations (when the decrease in the loss function is not apparent, indicating the need of a larger attack-space), we upscale $\mathbf{y}$ from $32 \times 32 \times 3$ to $64 \times 64 \times 3$, and use D_2 for the following iterations.

3.2.4 Optimize the Important Pixels First

One benefit of using coordinate descent is that we can choose which coordinates to update. Since estimating the gradient and Hessian for each pixel is expensive in the black-box setting, we propose to selectively update pixels by using *importance sampling*. For example, pixels in the corners or at the edges of an image are usually less important, whereas pixels near the main object can be crucial for a successful

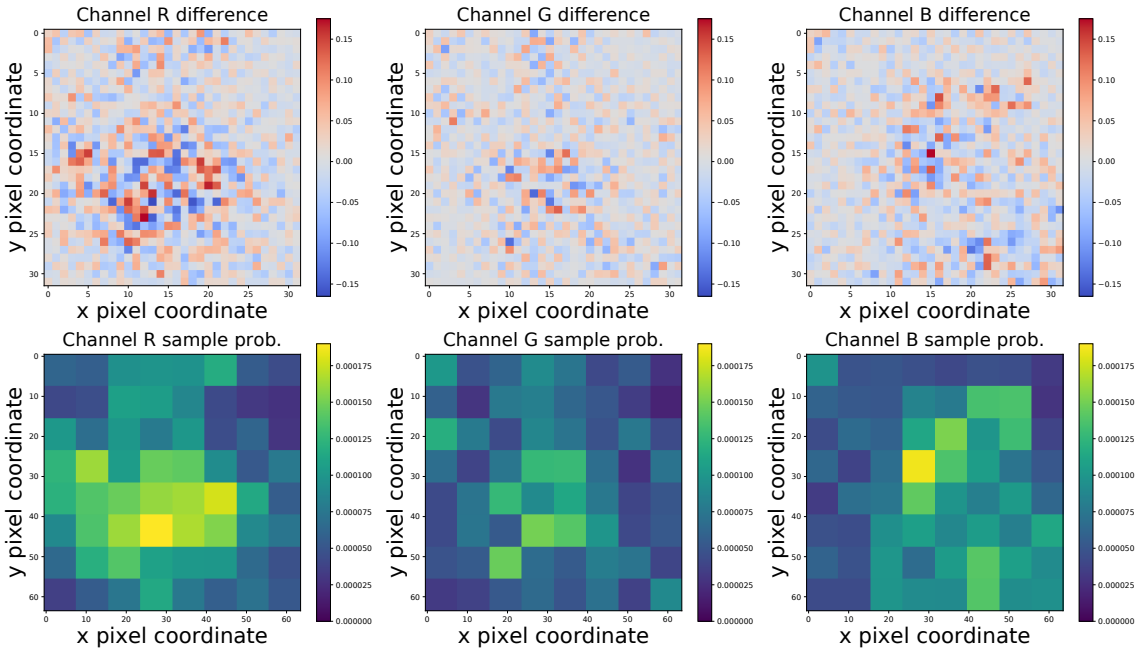


Figure 2: Attacking the bagel image in Figure 1 (a) with importance sampling. Top: Pixel values in certain parts of the bagel image have significant changes in RGB channels, and the changes in the R channel is more prominent than other channels. Here the attack-space is $32 \times 32 \times 3$. Although our targeted attack in this attack-space fails, its adversarial noise provides important clues to pixel importance. We use the noise from this attack-space to sample important pixels after we increase the dimension of attack-space to a larger dimension. Bottom: Importance sampling probability distribution for $64 \times 64 \times 3$ attack-space. The importance is computed by taking the absolute value of pixel value changes, running a 4×4 max-pooling for each channel, up-sampling to the dimension of $64 \times 64 \times 3$, and normalizing all values.

attack. Therefore, in the attack process we sample more pixels close to the main object indicated by the adversarial noise.

We propose to divide the image into 8×8 regions, and assign sampling probabilities according to how large the pixel values change in that region. We run a max pooling of the absolute pixel value changes in each region, up-sample to the desired dimension, and then normalize all values such that they sum up to 1. Every few iterations, we update these sampling probabilities according to the recent changes. In Figure 2, we show a practical example of pixel changes and how importance sampling probabilities are generated when attacking the bagel image in Figure 1 (a).

3.3 Experimental Results

3.3.1 Setup

We compare our attack (ZOO) with Carlini & Wagner's (C&W) white-box attack [49] and the substitute model based black-box attack [50]. We would like to show that our black-box attack can achieve similar success rate and distortion as the white-box C&W attack, and can significantly outperform the substitute model based black-box attack, while maintaining a reasonable attack time.

Our experimental setup is based on Carlini & Wagner's framework[2] with our Adam and Newton based zeroth order optimizer included. For substitute model based attack, we use the reference implementation (with necessary modifications) in CleverHans[3] for comparison. For experiments on MNIST and CIFAR, we use a Intel Xeon E5-2690v4 CPU with a single NVIDIA K80 GPU; for experiments on ImageNet, we use a AMD Ryzen 1600 CPU with a single NVIDIA GTX 1080 Ti GPU. For implementing zeroth order optimization, we use a batch size of $B = 128$; i.e., we evaluate 128 gradients and update 128 coordinates per iteration. In addition, we set $\kappa = 0$ unless specified.

3.3.2 MNIST and CIFAR-10

DNN Model. For MNIST and CIFAR-10, we use the same DNN model as in the C&W attack ([49], Table 1). For the substitute model based attack, we use the same DNN model for *both* the target model and the substitute model. If the architecture of a targeted DNN is unknown, black-box attacks based on substitute models will yield worse performance due to model mismatch.

[2] `https://github.com/carlini/nn_robust_attacks`

[3] `https://github.com/tensorflow/cleverhans/blob/master/tutorials/mnist_blackbox.py`

Target images. For targeted attacks, we randomly select 100 images from MNIST and CIFAR-10 test sets, and skip the original images misclassified by the target model. For each image, we apply targeted attacks to all 9 other classes, and thus there are 900 attacks in total. For untargeted attacks, we randomly select 200 images from the MNIST and CIFAR-10 test sets.

Parameter setting. For both our attack and the C&W attack, we run a binary search up to 9 times to find the best c (starting from 0.01), and terminate the optimization process early if the loss does not decrease for 100 iterations. We use the same step size $\eta = 0.01$ and Adam parameters $\beta_1 = 0.9, \beta_2 = 0.999$ for all methods. For the C&W attack, we run 1,000 iterations; for our attack, we run 3,000 iterations for MNIST and 1,000 iterations for CIFAR. Note that our algorithm updates far less variables because for each iteration we only update 128 pixels, whereas in the C&W attack all pixels are updated based on the full gradient in one iteration due to the white-box setting. Also, since the image size of MNIST and CIFAR-10 is small, we do not reduce the dimension of the attack-space or use hierarchical attack and importance sampling.

For training the substitute model, we use 150 hold-out images from the test set and run 5 Jacobian augmentation epochs, and set the augmentation parameter $\lambda = 0.1$. We implement FGSM and the C&W attack on the substitute model for both targeted and untargeted transfer attacks to the black-box DNN. For FGSM, the perturbation parameter $\epsilon = 0.4$, as it is shown to be effective in [50]. For the C&W attack, we use the same settings as the white-box C&W, except for setting $\kappa = 20$ for attack transferability and using 2,000 iterations.

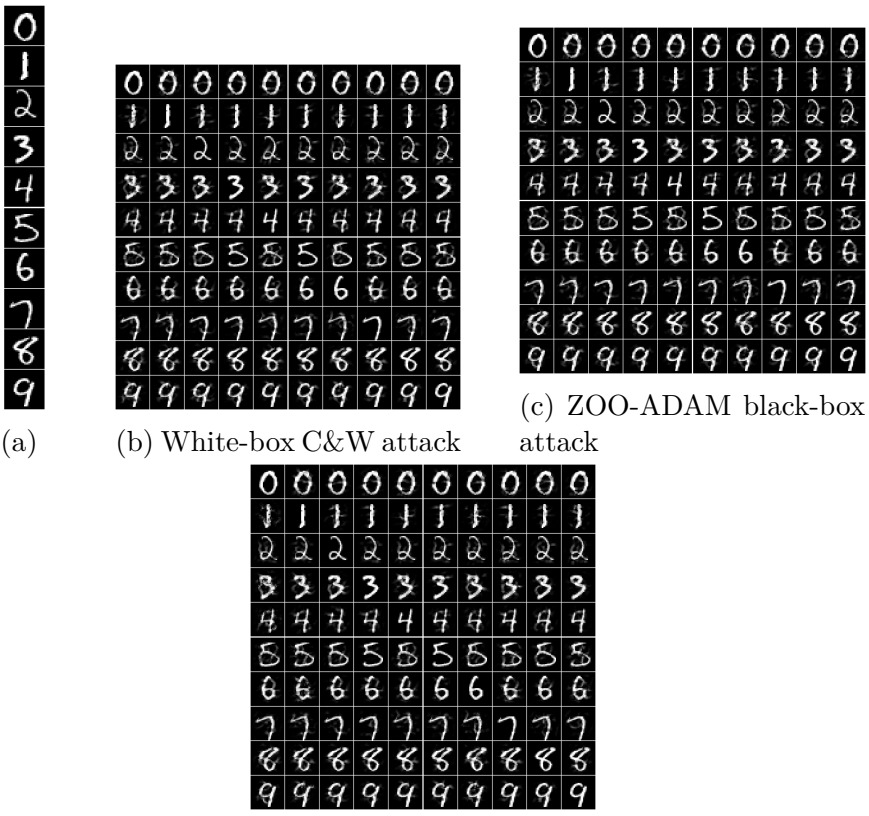
When attacking MNIST, we found that the change-of-variable via tanh can cause the estimated gradients to vanish due to limited numerical accuracy when pixel values are close to the boundary (0 or 1). As an alternative, we project the pixel values within

Table 1: MNIST and CIFAR-10 attack comparison: ZOO attains comparable success rate and L_2 distortion as the white-box C& W attack, and significantly outperforms the black-box substitute model attacks using FGSM (L_∞ attack) and the C&W attack [50]. The numbers in parentheses in Avg. Time field is the total time for training the substitute model. For FGSM we do not compare its L_2 with other methods because it is an L_∞ attack.

	MNIST					
	Untargeted			Targeted		
	Success Rate	Avg. L_2	Avg. Time (per attack)	Success Rate	Avg. L_2	Avg. Time (per attack)
White-box (C&W)	100 %	1.48066	0.48 min	100 %	2.00661	0.53 min
Black-box (Substitute Model + FGSM)	40.6 %	-	0.002 sec (+ 6.16 min)	7.48 %	-	0.002 sec (+ 6.16 min)
Black-box (Substitute Model + C&W)	33.3 %	3.6111	0.76 min (+ 6.16 min)	26.74 %	5.272	0.80 min (+ 6.16 min)
Proposed black-box (ZOO-ADAM)	100 %	1.49550	1.38 min	98.9 %	1.987068	1.62 min
Proposed black-box (ZOO-Newton)	100 %	1.51502	2.75 min	98.9 %	2.057264	2.06 min
	CIFAR-10					
	Untargeted			Targeted		
	Success Rate	Avg. L_2	Avg. Time (per attack)	Success Rate	Avg. L_2	Avg. Time (per attack)
White-box (C&W)	100 %	0.17980	0.20 min	100 %	0.37974	0.16 min
Black-box (Substitute Model + FGSM)	76.1 %	-	0.005 sec (+ 7.81 min)	11.48 %	-	0.005 sec (+ 7.81 min)
Black-box (Substitute Model + C&W)	25.3 %	2.9708	0.47 min (+ 7.81 min)	5.3 %	5.7439	0.49 min (+ 7.81 min)
Proposed Black-box (ZOO-ADAM)	100 %	0.19973	3.43 min	96.8 %	0.39879	3.95 min
Proposed Black-box (ZOO-Newton)	100 %	0.23554	4.41 min	97.0 %	0.54226	4.40 min

the box constraints after each update for MNIST (projected gradient descent). But for CIFAR-10, we find that using change-of-variable converges faster, as most pixels are not close to the boundary.

Results. As shown in Table 1, our proposed attack (ZOO) achieves nearly 100% success rate. Furthermore, the L_2 distortions are also close to the C&W attack, indicating our black-box adversarial images have similar quality as the white-box approach (Figures 3 and 4). Notably, our success rate is significantly higher than the substitute model based attacks, especially for targeted attacks, while maintaining reasonable average attack time. When transferring attacks from the substitute models to the target DNN, FGSM achieves better success rates in some experiments because it uses a relatively large $\epsilon = 0.4$ and introduces much more noise than the C&W attack. We also find that Adam usually works better than Newton's method in terms of computation time and L_2 distortion.



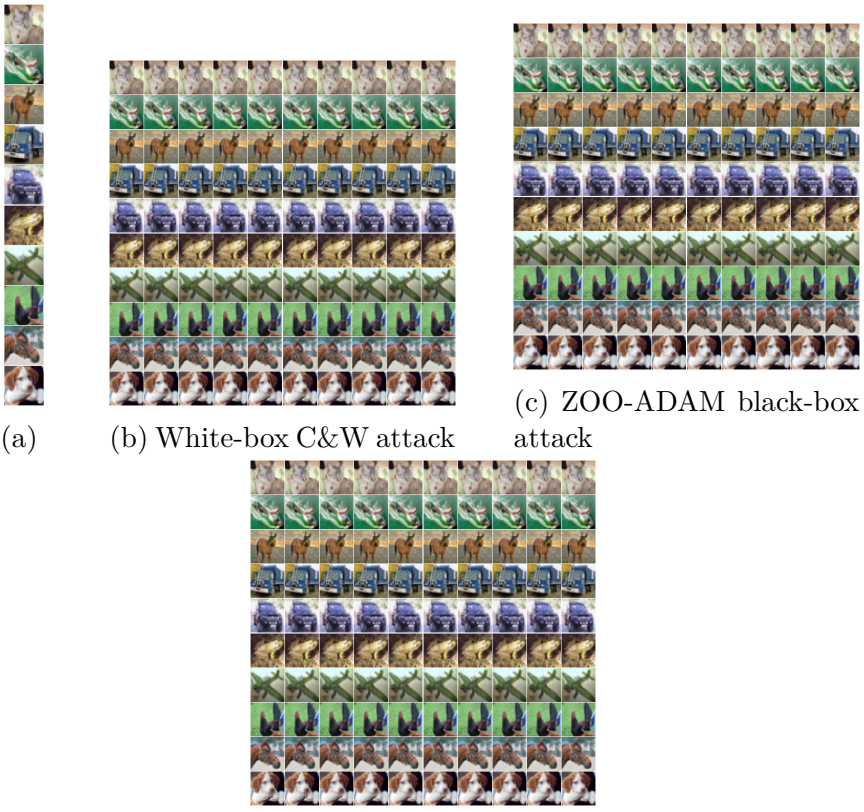
(a) (b) White-box C&W attack (c) ZOO-ADAM black-box attack

(d) ZOO-Newton black-box attack

Figure 3: Visual comparison of successful adversarial examples in MNIST. Each row displays crafted adversarial examples from the sampled images in (a). Each column in (b) to (d) indexes the targeted class for attack (digits 0 to 9).

3.3.3 Inception Network with ImageNet

Attacking a large black-box network like Inception-v3 [59] can be challenging due to its large attack-space and expensive model evaluation. Black-box attacks via substitute models become impractical in this case, as a substitute model with a large enough capacity relative to Inception-V3 is needed, and a tremendous amount of costly Jacobian data augmentation is needed to train this model. On the other hand, transfer attacks may suffer from lower success rate comparing to white-box attacks, especially for targeted attacks. Here we Here we apply the techniques proposed previously (di-



(a) (b) White-box C&W attack (c) ZOO-ADAM black-box attack (d) ZOO-Newton black-box attack

Figure 4: Visual comparison of successful adversarial examples in CIFAR-10. Each row displays crafted adversarial examples from the sampled images in (a). Each column in (b) to (d) indexes the targeted class for attack.

mension reduction, hierarchical attack, and importance sampling) to overcome the optimization difficulty toward effective and efficient black-box attacks.

- **Untargeted black-box attacks to Inception-v3.**

Examples. We use 150 images from the ImageNet test set for untargeted attacks. To justify the effectiveness of using attack-space dimension reduction, we exclude small images in the test set and ensure that all the original images are at least 299×299 in size. We also skip all images that are originally misclassified by Inception-v3.

Attack techniques and parameters. We use an attack-space of only $32 \times 32 \times 3$ (the original input space is $299 \times 299 \times 3$) and do not use hierarchical attack. We also set a

Table 2: Untargeted ImageNet attacks comparison. Substitute model based attack cannot easily scale to ImageNet.

	Success Rate	Avg. L_2
White-box (C&W)	100 %	0.37310
Proposed black-box (ZOO-ADAM)	88.9 %	1.19916
Black-box (Substitute Model)	N.A.	N.A.

hard limit of $1,500$ iterations for each attack, which takes about 20 minutes per attack in our setup. In fact, during $1,500$ iterations, only $1500 \times 128 = 192,000$ gradients are evaluated, which is even less than the total number of pixels ($299 \times 299 \times 3 = 268,203$) of the input image. We fix $c = 10$ in all Inception-v3 experiments, as it is too costly to do binary search in this case. For both C&W and our attacks, we use step size 0.002.

Results. We compare the success rate and average L_2 distortion between our ZOO attack and the C&W white-box attack in Table 2. Despite running only 1,500 iterations (within 20 minutes per image) and using a small attack-space ($32 \times 32 \times 3$), our black-box attack achieves about 90% success rate. The average L_2 distortion is about 3 times larger than the white-box attack, but our adversarial images are still visually indistinguishable (Figures 1). The success rate and distortion can be further improved if we run more iterations.

- **Targeted black-box attacks to Inception-v3.**

For Inception-v3, a targeted attack is much more difficult as there are 1000 classes, and a successful attack means one can manipulate the predicted probability of any specified class. However, we report that using our advanced attack techniques, $20,000$ iterations (each with $B = 128$ pixel updates) are sufficient for a *hard* targeted attack.

Example. We select an image (Figure 1 (a)) for which our untargeted attack failed, i.e., we cannot even find an *untargeted* attack in the $32 \times 32 \times 3$ attack-space, indicating that this image is hard to attack. Inception-v3 classifies it as a "bagel" with 97.0% confidence, and other top-5 predictions include "guillotine", "pretzel", "Granny

Smith" and "dough" with 1.15%, 0.07%, 0.06% and 0.01% confidence. We deliberately make the attack even harder by choosing the target class as "grand piano", with original confidence of only 0.0006%.

Attack techniques. We use attack-space dimension reduction as well as hierarchical attack. We start from an attack-space of $32 \times 32 \times 3$, and increase it to $64 \times 64 \times 3$ and $128 \times 128 \times 3$ at iteration 2,000 and 10,000, respectively. We run the zeroth order Adam solver (Algorithm 2) with a total of 20,000 iterations, taking about 260 minutes in our setup. Also, when the attack space is greater than $32 \times 32 \times 3$, we incorporate importance sampling, and keep updating the sampling probability after each iteration.

Reset Adam states. We report an additional method to reduce the final distortion - reset the Adam solver's states when a first valid attack is found during the optimization process. The reason is as follows. The total loss consists of two parts: $l_1 \coloneqq c \cdot f(\mathbf{x}, t)$ and $l_2 \coloneqq \|\mathbf{x} - \mathbf{x}_0\|_2^2$. l_1 measures the difference between the original class probability P_{orig} and targeted class probability P_{target} as defined in (10). When $l_1 = 0$, $P_{\text{orig}} \leq P_{\text{target}}$, and a valid adversarial example is found. l_2 is the L_2 distortion. During the optimization process, we observe that before l_1 reaches 0, l_2 is likely to increase, i.e., adding more distortion and getting closer to the target class. After l_1 reaches 0 it cannot go below 0 because it is a hinge-like loss, and at this point the optimizer should try to reduce l_2 as much as possible while keeping P_{target} only slightly larger than P_{orig}. However, when we run coordinate-wise Adam, we found that even after l_1 reaches 0, the optimizer still tries to reduce P_{orig} and to increase P_{target}, and l_2 will not be decreased efficiently. We believe the reason is that the historical gradient statistics stored in Adam states are quite stale due to the large number of coordinates. Therefore, we simply reset the Adam states after l_1 reaches 0 for the first time in order to make the solver focus on decreasing l_2 afterwards.

Results. Figure 5 shows how the loss decreases versus iterations, with all techniques discussed above applied in red; other curves show the optimization process without a certain technique but all others included. The black curve decreases very slowly, suggesting hierarchical attack is extremely important in accelerating our attack, otherwise the large attack-space makes zeroth order methods infeasible. Importance sampling also makes a difference especially after iteration 10,000 – when the attack-space is increased to $128 \times 128 \times 3$; it helps us to find the first valid attack over 2,000 iterations earlier, thus leaving more time for reducing the distortion. The benefit of reseting Adam states is clearly shown in Table 3, where the final distortion and loss increase noticeably if we do not reset the states. The proposed ZOO attack succeeds in decreasing the probability of the original class by over 160x (from 97% to about 0.6%) while increasing the probability of the target class by over 1000x (from 0.0006% to over 0.6%, which is top-1) to achieve a successful attack. Furthermore, as shown in Figures 1, the crafted adversarial noise is almost negligible and indistinguishable by human eyes.

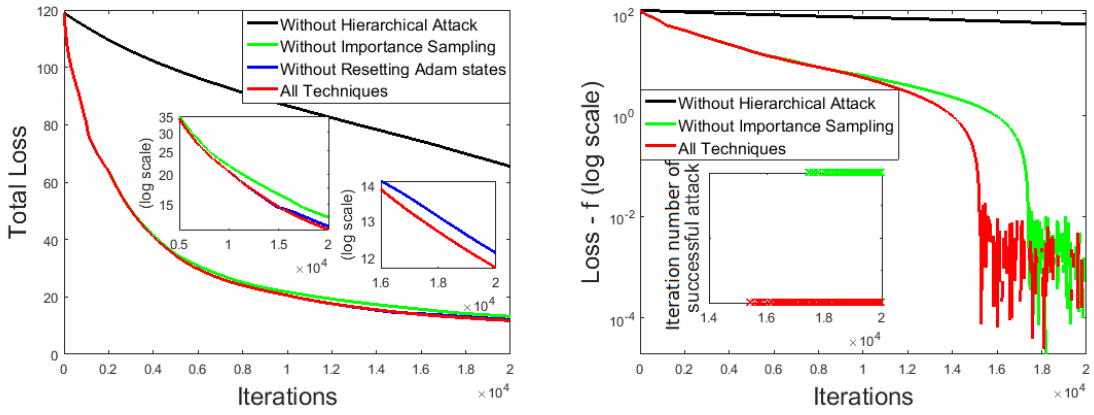


Figure 5: Left: total loss $\|\mathbf{x}-\mathbf{x}_0\|_2^2+c\cdot f(\mathbf{x},t)$ versus iterations. Right: $c\cdot f(\mathbf{x},t)$ versus iterations (log y-scale). When $c \cdot f(\mathbf{x},t)$ reaches 0, a valid attack is found. With all techniques applied, the first valid attack is found at iteration $15,227$. The optimizer then continues to minimize $\|\mathbf{x}-\mathbf{x}_0\|_2^2$ to reduce distortion. In the right figure we do not show the curve without resetting Adam states because we reset Adam states only when $c \cdot f(\mathbf{x},t)$ reaches 0 for the first time.

Table 3: Comparison of different attack techniques. "First Valid" indicates the iteration number where the first successful attack was found during the optimization process.

Black-box (ZOO-ADAM)	Success?	First Valid	Final L_2	Final Loss
All techniques	Yes	15,227	3.425	11.735
No Hierarchical Attack	No	-	-	62.439
No importance sampling	Yes	17,403	3.63486	13.216
No Adam state reset	Yes	15,227	3.47935	12.111

3.3.4 Ensemble Adversarial Training

We randomly sample 10 correctly classified examples from the ImageNet test set. We perform a targeted attack on the ensemble adversarially trained Inception-v3 and Inception ResNet-v2 models using the same configuration as presented in the previous section. We notice 100% success on both models, with the adversarial examples remaining visually imperceptible. We test as well with the "bagel" image in (Figure 1 (a)) and target class "grand piano", and are able to replicate the results in (Figure 1 (a)) against the ensemble adversarially trained networks.

4 EAD

We show that compared to the state-of-the-art L_2 and L_∞ attacks [46, 49], EAD can attain similar attack success rate when breaking undefended and defensively distilled DNNs [55]. More importantly, we find that L_1 attacks attain superior performance over L_2 and L_∞ attacks in transfer attacks and complement adversarial training. For the most difficult dataset (MNIST), EAD results in improved attack transferability from an undefended DNN to a defensively distilled DNN, achieving nearly 99% attack success rate. In addition, joint adversarial training with L_1 and L_2-based examples can further enhance the resilience of DNNs to adversarial perturbations.

We also find that L_1-based adversarial examples generated by EAD readily transfer in both the targeted and non-targeted cases to the Madry Defense Model, and despite the high L_∞ distortion, the visual distortion on the adversarial examples is minimal.

We also find that adversarial examples with minimal visual distortion generated by EAD can bypass the Feature Squeezing joint detection method.

These results suggest that EAD yields a distinct yet more effective set of adversarial examples, and is the state-of-the-art attack method in the no-box setting.

4.1 Motivation

In the literature, the similarity between original and adversarial examples has been measured by different distortion metrics. One commonly used distortion metric is the L_q norm, where $\|\mathbf{x}\|_q = (\sum_{i=1}^{p} |\mathbf{x}_i|^q)^{1/q}$ denotes the L_q norm of a p-dimensional vector $\mathbf{x} = [\mathbf{x}_1, \ldots, \mathbf{x}_p]$ for any $q \geq 1$. In particular, when crafting adversarial examples, the L_∞ distortion metric is used to evaluate the maximum variation in pixel value changes [24], while the L_2 distortion metric is used to improve the visual quality [49]. However, despite the fact that the L_1 norm is widely used in problems related to image denoising and restoration [66], as well as sparse recovery [67], L_1-based adversarial examples have not been rigorously explored. In the context of adversarial examples, L_1 distortion accounts for the total variation in the perturbation and serves as a popular convex surrogate function of the L_0 metric, which measures the number of modified pixels (i.e., sparsity) by the perturbation. To bridge this gap, we propose an attack algorithm based on elastic-net regularization, which we call **e**lastic-net **a**ttacks to **D**NNs (EAD). Elastic-net regularization is a linear mixture of L_1 and L_2 penalty functions, and it has been a standard tool for high-dimensional feature selection problems [68]. In the context of attacking DNNs, EAD generalizes the state-of-the-art attack proposed in [49] based on L_2 distortion, and is able to craft L_1-oriented adversarial examples that are fundamentally different from existing attack methods.

4.2 Algorithm

4.2.1 Elastic-net Regularization

Elastic-net regularization is a widely used technique in solving high-dimensional feature selection problems [68]. It can be viewed as a regularizer that linearly combines L_1 and L_2 penalty functions. In general, elastic-net regularization is used in the following minimization problem:

$$\text{minimize}_{\mathbf{z}\in\mathcal{Z}}\ f(\mathbf{z}) + \lambda_1\|\mathbf{z}\|_1 + \lambda_2\|\mathbf{z}\|_2^2, \tag{15}$$

where $\mathbf{z}$ is a vector of p optimization variables, $\mathcal{Z}$ indicates the set of feasible solutions, $f(\mathbf{z})$ denotes a loss function, $\|\mathbf{z}\|_q$ denotes the L_q norm of $\mathbf{z}$, and $\lambda_1, \lambda_2 \geq 0$ are the L_1 and L_2 regularization parameters, respectively. The term $\lambda_1\|\mathbf{z}\|_1 + \lambda_2\|\mathbf{z}\|_2^2$ in (15) is called the elastic-net regularizer of $\mathbf{z}$.

For standard regression problems, the loss function $f(\mathbf{z})$ is the mean squared error, the vector $\mathbf{z}$ represents the weights (coefficients) on the features, and the set $\mathcal{Z} = \mathbb{R}^p$. In particular, the elastic-net regularization in (15) degenerates to the LASSO formulation when $\lambda_2 = 0$, and becomes the ridge regression formulation when $\lambda_1 = 0$. It is shown in [68] that elastic-net regularization is able to select a group of highly correlated features, which overcomes the shortcoming of high-dimensional feature selection when solely using the LASSO or ridge regression techniques.

4.2.2 EAD Formulation

Inspired by the C&W attack [49], we adopt the same loss function f for crafting adversarial examples, as presented in (9). In addition to manipulating the prediction via the loss function in (9), introducing elastic-net regularization further encourages similarity to the original image when crafting adversarial examples. Our formulation of elastic-net attacks to DNNs (EAD) for crafting an adversarial example $(\mathbf{x}, t)$ with

respect to a labeled normal image $(\mathbf{x}_0, l)$ is as follows:

$$\begin{aligned} &\text{minimize}_{\mathbf{x}} \quad c \cdot f(\mathbf{x}, t) + \beta \|\mathbf{x} - \mathbf{x}_0\|_1 + \|\mathbf{x} - \mathbf{x}_0\|_2^2 \\ &\text{subject to} \quad \mathbf{x} \in [0,1]^p, \end{aligned} \tag{16}$$

where $f(\mathbf{x}, t)$ is as defined in (9), $c, \beta \geq 0$ are the regularization parameters of the loss function f and the L_1 penalty, respectively. Upon defining the perturbation of $\mathbf{x}$ relative to $\mathbf{x}_0$ as $\boldsymbol{\delta} = \mathbf{x} - \mathbf{x}_0$, the EAD formulation in (16) aims to find an adversarial example $\mathbf{x}$ that will be classified as the target class t while minimizing the distortion in $\boldsymbol{\delta}$ in terms of the elastic-net loss $\beta\|\boldsymbol{\delta}\|_1 + \|\boldsymbol{\delta}\|_2^2$, which is a linear combination of L_1 and L_2 distortion metrics between $\mathbf{x}$ and $\mathbf{x}_0$. Notably, the formulation of the C&W attack [49] becomes a special case of the EAD formulation in (16) when $\beta = 0$, which disregards the L_1 penalty on $\boldsymbol{\delta}$. However, the L_1 penalty is an intuitive regularizer for crafting adversarial examples, as $\|\boldsymbol{\delta}\|_1 = \sum_{i=1}^{p} |\boldsymbol{\delta}_i|$ represents the total variation of the perturbation, and is also a widely used surrogate function for promoting sparsity in the perturbation. As will be evident in the results section, including the L_1 penalty for the perturbation indeed yields a distinct set of adversarial examples, and it leads to improved attack transferability and complements adversarial learning.

4.2.3 EAD Algorithm

When solving the EAD formulation in (16) without the L_1 penalty (i.e., $\beta = 0$), Carlini and Wagner used a change-of-variable (COV) approach via the tanh transformation on $\mathbf{x}$ in order to remove the box constraint $\mathbf{x} \in [0,1]^p$ [49]. When $\beta > 0$, we find that the same COV approach is not effective in solving (16), since the corresponding adversarial examples are insensitive to the changes in β (see the results section for details). Since the L_1 penalty is a non-differentiable yet smooth function, the failure of the COV approach in solving (16) can be explained by its inefficiency in subgradient-based optimization problems [69].

To efficiently solve the EAD formulation in (16) for crafting adversarial examples, we propose to use the iterative shrinkage-thresholding algorithm (ISTA) [70]. ISTA can be viewed as a regular first-order optimization algorithm with an additional shrinkage-thresholding step on each iteration. In particular, let $g(\mathbf{x}) = c \cdot f(\mathbf{x}) + \|\mathbf{x} - \mathbf{x}_0\|_2^2$ and let $\nabla g(\mathbf{x})$ be the numerical gradient of $g(\mathbf{x})$ computed by the DNN. At the $k+1$-th iteration, the adversarial example $\mathbf{x}^{(k+1)}$ of $\mathbf{x}_0$ is computed by

$$\mathbf{x}^{(k+1)} = S_\beta(\mathbf{x}^{(k)} - \alpha_k \nabla g(\mathbf{x}^{(k)})) \tag{17}$$

where α_k denotes the step size at the $k+1$-th iteration, and $S_\beta : \mathbb{R}^p \mapsto \mathbb{R}^p$ is an element-wise projected shrinkage-thresholding function, which is defined as

$$[S_\beta(\mathbf{z})]_i = \begin{cases} \min\{\mathbf{z}_i - \beta, 1\}, & \text{if } \mathbf{z}_i - \mathbf{x}_{0i} > \beta; \\ \mathbf{x}_{0i}, & \text{if } |\mathbf{z}_i - \mathbf{x}_{0i}| \leq \beta; \\ \max\{\mathbf{z}_i + \beta, 0\}, & \text{if } \mathbf{z}_i - \mathbf{x}_{0i} < -\beta, \end{cases} \tag{18}$$

for any $i \in \{1, \ldots, p\}$. If $|\mathbf{z}_i - \mathbf{x}_{0i}| > \beta$, it shrinks the element $\mathbf{z}_i$ by β and projects the resulting element to the feasible box constraint between 0 and 1. On the other hand, if $|\mathbf{z}_i - \mathbf{x}_{0i}| \leq \beta$, it thresholds $\mathbf{z}_i$ by setting $[S_\beta(\mathbf{z})]_i = \mathbf{x}_{0i}$. Notably, since $g(\mathbf{x})$ is the attack objective function of the C&W method [49], the ISTA operation in (17) can be viewed as a robust version of the C&W method that shrinks a pixel value of the adversarial example if the deviation to the original image is greater than β, and keeps a pixel value unchanged if the deviation is less than β.

4.2.3.1 Proof of Optimality of ISTA for Solving EAD

Since the L_1 penalty $\beta\|\mathbf{x} - \mathbf{x}_0\|_1$ in (15) is a non-differentiable yet smooth function, we use the proximal gradient method [71] for solving the EAD formulation in (15). Define $\Phi_{\mathcal{Z}}(\mathbf{z})$ to be the indicator function of an interval $\mathcal{Z}$ such that $\Phi_{\mathcal{Z}}(\mathbf{z}) = 0$ if $\mathbf{z} \in \mathcal{Z}$ and $\Phi_{\mathcal{Z}}(\mathbf{z}) = \infty$ if $\mathbf{z} \notin \mathcal{Z}$. Using $\Phi_{\mathcal{Z}}(\mathbf{z})$, the EAD formulation in (15) can be rewritten as

$$\text{minimize}_{\mathbf{x} \in \mathbb{R}^p} \quad g(\mathbf{x}) + \beta\|\mathbf{x} - \mathbf{x}_0\|_1 + \Phi_{[0,1]^p}(\mathbf{x}), \tag{19}$$

where $g(\mathbf{x}) = c \cdot f(\mathbf{x}, t) + \|\mathbf{x} - \mathbf{x}_0\|_2^2$. The proximal operator $\text{Prox}(\mathbf{x})$ of $\beta\|\mathbf{x} - \mathbf{x}_0\|_1$ constrained to $\mathbf{x} \in [0,1]^p$ is

$$\begin{aligned}\text{Prox}(\mathbf{x}) &= \arg\min_{\mathbf{z}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{z} - \mathbf{x}\|_2^2 + \beta\|\mathbf{z} - \mathbf{x}_0\|_1 + \Phi_{[0,1]^p}(\mathbf{z}) \\ &= \arg\min_{\mathbf{z}\in[0,1]^p} \frac{1}{2}\|\mathbf{z} - \mathbf{x}\|_2^2 + \beta\|\mathbf{z} - \mathbf{x}_0\|_1 \\ &= S_\beta(\mathbf{x}), \end{aligned} \tag{20}$$

where the mapping function S_β is defined in (18). Consequently, using (20), the proximal gradient algorithm for solving (16) is iterated by

$$\mathbf{x}^{(k+1)} = \text{Prox}(\mathbf{x}^{(k)} - \alpha_k \nabla g(\mathbf{x}^{(k)})) \tag{21}$$

$$= S_\beta(\mathbf{x}^{(k)} - \alpha_k \nabla g(\mathbf{x}^{(k)})), \tag{22}$$

which completes the proof.

4.2.3.2 Implementation

Our EAD algorithm for crafting adversarial examples is summarized in Algorithm 4. For computational efficiency, a fast ISTA (FISTA) for EAD is implemented, which yields the optimal convergence rate for first-order optimization methods [70]. The slack vector $\mathbf{y}^{(k)}$ in Algorithm 4 incorporates the momentum in $\mathbf{x}^{(k)}$ for acceleration. In the experiments, we set the initial learning rate $\alpha_0 = 0.01$ with a square-root decay factor in k. During the EAD iterations, the iterate $\mathbf{x}^{(k)}$ is considered as a successful adversarial example of $\mathbf{x}_0$ if the model predicts its most likely class to be the target class t. The final adversarial example $\mathbf{x}$ is selected from all successful examples based on distortion metrics. In this paper we consider two decision rules for selecting $\mathbf{x}$: the least elastic-net (EN) and L_1 distortions relative to $\mathbf{x}_0$. The influence of β, κ and the decision rules on EAD will be investigated in the following section.

Algorithm 4 Elastic-Net Attacks to DNNs (EAD)

Input: original labeled image $(\mathbf{x}_0, t_0)$, target attack class t, attack transferability parameter κ, L_1 regularization parameter β, step size α_k, # of iterations I
Output: adversarial example $\mathbf{x}$
Initialization: $\mathbf{x}^{(0)} = \mathbf{y}^{(0)} = \mathbf{x}_0$
for $k = 0$ to $I - 1$ **do**
$\mathbf{x}^{(k+1)} = S_\beta(\mathbf{y}^{(k)} - \alpha_k \nabla g(\mathbf{y}^{(k)}))$
$\mathbf{y}^{(k+1)} = \mathbf{x}^{(k+1)} + \frac{k}{k+3}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})$
end for
Decision rule: determine $\mathbf{x}$ from successful examples in $\{\mathbf{x}^{(k)}\}_{k=1}^{I}$ (EN rule or L_1 rule).

4.3 Experimental Results

In this section, we compare the proposed EAD algorithm with the state-of-the-art attacks to DNNs on three image classification datasets - MNIST, CIFAR-10 and ImageNet. We would like to show that (i) EAD can attain attack performance similar to the C&W attack in breaking undefended and defensively distilled DNNs, since the C&W attack is a special case of EAD when $\beta = 0$; (ii) Comparing to existing L_1-based FGM and I-FGM methods, the adversarial examples using EAD can lead to significantly lower L_1 distortion and better attack success rate; (iii) The L_1-based adversarial examples crafted by EAD can achieve improved attack transferability and complement adversarial training.

4.3.1 Setup

We compare our attack (EAD) with Carlini & Wagner's (C&W) attack [49], the fast gradient method (FGM) attacks [24] using L_1, L_2, and L_∞ (FGSM), and the iterative fast gradient method (I-FGM) attacks [46] using L_1, L_2, and L_∞ (I-FGM).

Our experiment setup is based on Carlini and Wagner's framework[4]. For both the EAD and C&W attacks, we use the default setting[4], which implements 9 binary search steps on the regularization parameter c (starting from 0.001) and runs $I = 1000$

[4] `https://github.com/carlini/nn_robust_attacks`

iterations for each step with the initial learning rate $\alpha_0 = 0.01$. For finding successful adversarial examples, we use the reference optimizer[4] (Adam) for the C&W attack and implement the projected FISTA (Algorithm 4) with the square-root decaying learning rate for EAD. Similar to the C&W attack, the final adversarial example of EAD is selected by the least distorted example among all the successful examples. The sensitivity analysis of the L_1 parameter β and the effect of the decision rule on EAD will be investigated in the forthcoming results. Unless specified, we set the attack transferability parameter $\kappa = 0$ for both attacks.

We implemented FGM and I-FGM using the CleverHans package[5]. The best distortion parameter ϵ is determined by a fine-grained grid search - for each image, the smallest ϵ leading to a successful attack is reported. For I-FGM, we perform 10 FGM iterations (the default value) with ϵ-ball clipping. The distortion parameter ϵ' in each FGM iteration is set to be $\epsilon/10$, which has been shown to be an effective attack setting in [54].

The range and resolution of the grid search is presented in Table 4. The selected range for the grid search covers the reported distortion statistics of EAD and the C&W attack. The resolution of the grid search for FGM is selected such that it will generate 1000 candidates of adversarial examples during the grid search per input image. The resolution of the grid search for I-FGM is selected such that it will compute gradients for 10000 times in total (i.e., 1000 FGM operations $\times$ 10 iterations) during the grid search per input image, which is more than the total number of gradients (9000) computed by EAD and the C&W attack.

The image classifiers for MNIST and CIFAR-10 are trained based on the DNN models provided by Carlini and Wagner[4]. The image classifier for ImageNet is the Inception-v3 model [59]. For MNIST and CIFAR-10, 1000 correctly classified images are randomly selected from the test sets to attack an incorrect class label. For ImageNet,

[5] `https://github.com/tensorflow/cleverhans/blob/master/cleverhans/attacks.py`

Table 4: Range and resolution of the grid search for finding the optimal distortion parameter ϵ for FGM and I-FGM.

	Grid Search	
Method	Range	Resolution
FGM-L_∞	$[10^{-3}, 1]$	10^{-3}
FGM-L_1	$[1, 10^3]$	1
FGM-L_2	$[10^{-2}, 10]$	10^{-2}
I-FGM-L_∞	$[10^{-3}, 1]$	10^{-3}
I-FGM-L_1	$[1, 10^3]$	1
I-FGM-L_2	$[10^{-2}, 10]$	10^{-2}

100 correctly classified images and 9 incorrect classes are randomly selected to attack. All experiments are conducted on a machine with an Intel E5-2690 v3 CPU, 40 GB RAM and a single NVIDIA K80 GPU.

4.3.2 Evaluation Metrics

Following the attack evaluation criterion in [49], we report the attack success rate and distortion of the adversarial examples from each method. The attack success rate (ASR) is defined as the percentage of adversarial examples that are classified as the target class (which is different from the original class). The average L_1, L_2 and L_∞ distortion metrics of successful adversarial examples are also reported. In particular, the ASR and distortion of the following attack settings are considered:

Best case: The least difficult attack among targeted attacks to all incorrect class labels in terms of distortion.

Average case: The targeted attack to a randomly selected incorrect class label.

Worst case: The most difficult attack among targeted attacks to all incorrect class labels in terms of distortion.

4.3.3 Sensitivity Analysis and Decision Rule for EAD

We verify the necessity of using Algorithm 4 for solving the elastic-net regularized attack formulation in (16) by comparing it to a naive change-of-variable (COV) approach. In [49], Carlini and Wagner remove the box constraint $\mathbf{x} \in [0, 1]^p$ by replacing

$\mathbf{x}$ with $\frac{\mathbf{1}+\tanh \mathbf{w}}{2}$, where $\mathbf{w} \in \mathbb{R}^p$ and $\mathbf{1} \in \mathbb{R}^p$ is a vector of ones. The default Adam optimizer [32] is then used to solve $\mathbf{w}$ and obtain $\mathbf{x}$. We apply this COV approach to (16) and compare with EAD on MNIST with different orders of the L_1 regularization parameter β in Table 5. Although COV and EAD attain similar ASR, it is observed that COV is not effective in crafting L_1-based adversarial examples. Increasing β leads to less L_1-distorted adversarial examples for EAD, whereas the distortion (L_1, L_2 and L_∞) of COV is insensitive to changes in β. Similar insensitivity of COV on β is observed when one uses other optimizers such as AdaGrad, RMSProp or built-in SGD in TensorFlow [72]. We also note that the COV approach prohibits the use of ISTA due to the subsequent tanh term in the L_1 penalty.

The insensitivity of COV suggests that it is inadequate for elastic-net optimization, which can be explained by its inefficiency in subgradient-based optimization problems [69]. For EAD, we also find an interesting trade-off between L_1 and the other two distortion metrics - adversarial examples with smaller L_1 distortion tend to have larger L_2 and L_∞ distortions. This trade-off can be explained by the fact that increasing β further encourages sparsity in the perturbation, and hence results in increased L_2 and L_∞ distortion. Similar results are observed on CIFAR-10.

In Table 5, during the attack optimization process the final adversarial example is selected based on the elastic-net loss of all successful adversarial examples in $\{\mathbf{x}^{(k)}\}_{k=1}^{I}$, which we call the *elastic-net (EN) decision rule*. Alternatively, we can select the final adversarial example with the least L_1 distortion, which we call the L_1 *decision rule*. Figure 6 compares the ASR and average-case distortion of these two decision rules with different β on MNIST. Both decision rules yield 100% ASR for a wide range of β values. For the same β, the L_1 rule gives adversarial examples with less L_1 distortion than those given by the EN rule at the price of larger L_2 and L_∞ distortions. Similar trends are observed on CIFAR-10. In the following experiments, we will report the results of EAD with these two decision rules and set $\beta = 10^{-3}$, since on MNIST and

Table 5: Comparison of the change-of-variable (COV) approach and EAD (Algorithm 4) for solving the elastic-net formulation in (16) on MNIST. ASR means attack success rate (%). Although these two methods attain similar attack success rates, COV is not effective in crafting L_1-based adversarial examples. Increasing β leads to less L_1-distorted adversarial examples for EAD, whereas the distortion of COV is insensitive to changes in β.

		Best case				Average case				Worst case			
Optimization method	β	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞
COV	0	100	13.93	1.377	0.379	100	22.46	1.972	0.514	99.9	32.3	2.639	0.663
	10^{-5}	100	13.92	1.377	0.379	100	22.66	1.98	0.508	99.5	32.33	2.64	0.663
	10^{-4}	100	13.91	1.377	0.379	100	23.11	2.013	0.517	100	32.32	2.639	0.664
	10^{-3}	100	13.8	1.377	0.381	100	22.42	1.977	0.512	99.9	32.2	2.639	0.664
	10^{-2}	100	12.98	1.38	0.389	100	22.27	2.026	0.53	99.5	31.41	2.643	0.673
EAD (EN rule)	0	100	14.04	1.369	0.376	100	22.63	1.953	0.512	99.8	31.43	2.51	0.644
	10^{-5}	100	13.66	1.369	0.378	100	22.6	1.98	0.515	99.9	30.79	2.507	0.648
	10^{-4}	100	12.79	1.372	0.388	100	20.98	1.951	0.521	100	29.21	2.514	0.667
	10^{-3}	100	9.808	1.427	0.452	100	17.4	2.001	0.594	100	25.52	2.582	0.748
	10^{-2}	100	7.271	1.718	0.674	100	13.56	2.395	0.852	100	20.77	3.021	0.976

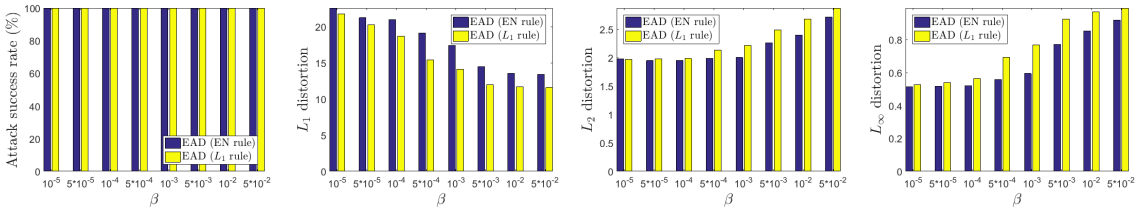


Figure 6: Comparison of EN and L_1 decision rules in EAD on MNIST with varying L_1 regularization parameter β (average case). Comparing to the EN rule, for the same β the L_1 rule attains less L_1 distortion but may incur more L_2 and L_∞ distortions.

CIFAR-10 this β value significantly reduces the L_1 distortion while having comparable L_2 and L_∞ distortions to the case of $\beta = 0$ (i.e., without L_1 regularization).

4.3.4 MNIST, CIFAR-10, and ImageNet

We compare EAD with the comparative methods in terms of attack success rate and different distortion metrics on attacking the considered DNNs trained on MNIST, CIFAR-10 and ImageNet. Table 6 summarizes their average-case performance. It is observed that FGM methods fail to yield successful adversarial examples (i.e., low ASR), and the corresponding distortion metrics are significantly larger than other methods. On the other hand, the C&W attack, I-FGM and EAD all lead to 100%

Table 6: Comparison of different attacks on MNIST, CIFAR-10 and ImageNet (average case). ASR means attack success rate (%). The distortion metrics are averaged over successful examples. EAD, the C&W attack, and I-FGM-L_∞ attain the least L_1, L_2, and L_∞ distorted adversarial examples, respectively.

	MNIST				CIFAR-10				ImageNet			
Attack method	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞
C&W (L_2)	**100**	22.46	**1.972**	0.514	**100**	13.62	**0.392**	0.044	**100**	232.2	**0.705**	0.03
FGM-L_1	39	53.5	4.186	0.782	48.8	51.97	1.48	0.152	1	61	0.187	0.007
FGM-L_2	34.6	39.15	3.284	0.747	42.8	39.5	1.157	0.136	1	2338	6.823	0.25
FGM-L_∞	42.5	127.2	6.09	0.296	52.3	127.81	2.373	0.047	3	3655	7.102	0.014
I-FGM-L_1	**100**	32.94	2.606	0.591	**100**	17.53	0.502	0.055	77	526.4	1.609	0.054
I-FGM-L_2	**100**	30.32	2.41	0.561	**100**	17.12	0.489	0.054	**100**	774.1	2.358	0.086
I-FGM-L_∞	**100**	71.39	3.472	**0.227**	**100**	33.3	0.68	**0.018**	**100**	864.2	2.079	**0.01**
EAD (EN rule)	**100**	**17.4**	2.001	0.594	**100**	**8.18**	0.502	0.097	**100**	**69.47**	1.563	0.238
EAD (L_1 rule)	**100**	**14.11**	2.211	0.768	**100**	**6.066**	0.613	0.17	**100**	**40.9**	1.598	0.293

attack success rate. Furthermore, EAD, the C&W method, and I-FGM-L_∞ attain the least L_1, L_2, and L_∞ distorted adversarial examples, respectively. We note that EAD significantly outperforms the existing L_1-based method (I-FGM-L_1). Compared to I-FGM-L_1, EAD with the EN decision rule reduces the L_1 distortion by roughly 47% on MNIST, 53% on CIFAR-10 and 87% on ImageNet. We also observe that EAD with the L_1 decision rule can further reduce the L_1 distortion but at the price of noticeable increase in the L_2 and L_∞ distortion metrics.

Notably, despite having large L_2 and L_∞ distortion metrics, the adversarial examples crafted by EAD with the L_1 rule can still attain 100% ASRs in all datasets, which implies the L_2 and L_∞ distortion metrics are insufficient for evaluating the robustness of neural networks. Moreover, the attack results in Table 6 suggest that EAD can yield a set of distinct adversarial examples that are fundamentally different from L_2 or L_∞ based examples. Similar to the C&W method and I-FGM, the adversarial examples from EAD are also visually indistinguishable.

4.3.5 Complementing Adversarial Training

To further validate the difference between L_1-based and L_2-based adversarial examples, we test their performance in adversarial training on MNIST. We randomly select

Table 7: Adversarial training using the C&W attack and EAD (L_1 rule) on MNIST. ASR means attack success rate. Incorporating L_1 examples complements adversarial training and enhances attack difficulty in terms of distortion.

Attack method	Adversarial training	Average case			
		ASR	L_1	L_2	L_∞
C&W (L_2)	None	100	22.46	1.972	0.514
	EAD	100	26.11	2.468	0.643
	C&W	100	24.97	2.47	0.684
	EAD + C&W	100	27.32	2.513	0.653
EAD (L_1 rule)	None	100	14.11	2.211	0.768
	EAD	100	17.04	2.653	0.86
	C&W	100	15.49	2.628	0.892
	EAD + C&W	100	16.83	2.66	0.87

1000 images from the training set and use the C&W attack and EAD (L_1 rule) to generate adversarial examples for all incorrect labels, leading to 9000 adversarial examples in total for each method. We then separately augment the original training set with these examples to retrain the network and test its robustness on the testing set, as summarized in Table 7. For adversarial training with any single method, although both attacks still attain a 100% success rate in the average case, the network is more tolerable to adversarial perturbations, as all distortion metrics increase significantly when compared to the null case. We also observe that joint adversarial training with EAD and the C&W method can further increase the L_1 and L_2 distortions against the C&W attack and the L_2 distortion against EAD, suggesting that the L_1-based examples crafted by EAD can complement adversarial training.

4.3.6 Breaking Defensive Distillation

In addition to breaking undefended DNNs via adversarial examples, here we show that EAD can also break defensively distilled DNNs. Defensive distillation [55] is a standard defense technique that retrains the network with class label probabilities predicted by the original network, soft labels, and introduces the temperature parameter T in the softmax layer to enhance its robustness to adversarial perturbations. Similar to the state-of-the-art attack (the C&W method), Figure 7 shows that EAD can attain 100% attack success rate for different values of T on MNIST

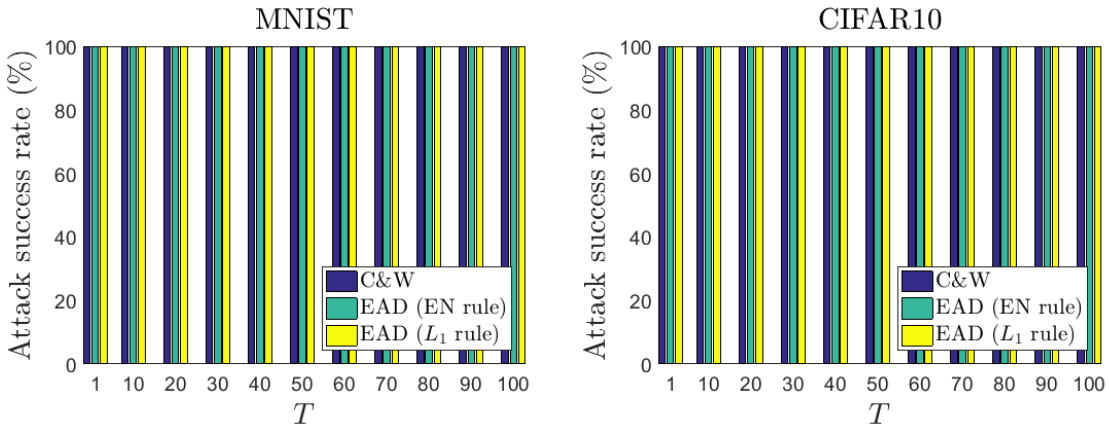


Figure 7: Attack success rate (average case) of the C&W method and EAD on MNIST and CIFAR-10 with respect to varying temperature parameter T for defensive distillation. Both methods can successfully break defensive distillation.

and CIFAR-10. Moreover, since the C&W attack formulation is a special case of the EAD formulation in (16) when $\beta = 0$, successfully breaking defensive distillation using EAD suggests new ways of crafting effective adversarial examples by varying the L_1 regularization parameter β.

4.3.7 Transfer Attacks in the No-box Setting

4.3.7.1 Defensive Distillation

It has been shown in [49] that the C&W attack can be made highly transferable from an undefended network to a defensively distilled network by tuning the confidence parameter κ in (9). Following [49], we adopt the same experiment setting for attack transferability on MNIST, as MNIST is the most difficult dataset to attack in terms of the average distortion per image pixel from Table 6.

Fixing κ, adversarial examples generated from the original (undefended) network are used to attack the defensively distilled network with the temperature parameter $T = 100$ [55]. The attack success rate (ASR) of EAD, the C&W method and I-FGM are shown in Figure 8. When $\kappa = 0$, all methods attain low ASR and hence do not produce transferable adversarial examples. The ASR of EAD and the C&W method

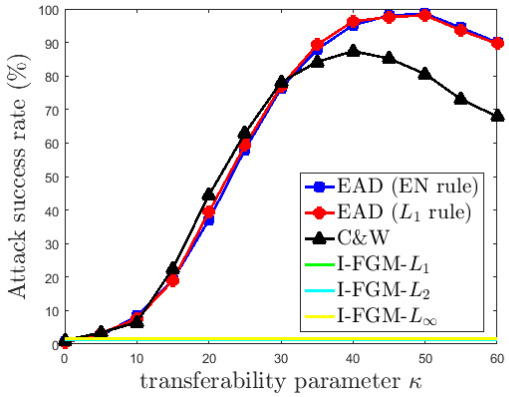


Figure 8: Attack transferability (average case) from the undefended network to the defensively distilled network on MNIST by varying κ. EAD can attain nearly 99% attack success rate (ASR) when $\kappa = 50$, whereas the top ASR of the C&W attack is nearly 88% when $\kappa = 40$.

improves when we set $\kappa > 0$, whereas I-FGM's ASR remains low (less than 2%) since the attack does not have such a parameter for transferability.

Notably, EAD can attain nearly 99% ASR when $\kappa = 50$, whereas the top ASR of the C&W method is nearly 88% when $\kappa = 40$. This implies improved attack transferability when using the adversarial examples crafted by EAD, which can be explained by the fact that the ISTA operation in (17) is a robust version of the C&W attack via shrinking and thresholding. We also find that setting κ too large may mitigate the ASR of transfer attacks for both EAD and the C&W method, as the optimizer may fail to find an adversarial example that minimizes the loss function f in (9) for large κ. The complete attack transferability results are given in Table 8.

4.3.7.2 Madry Defense Model

The Madry Defense Model is a sufficiently high capacity network trained against the strongest possible adversary, which they deem to be Projected Gradient Descent (PGD) starting from random perturbations around the natural examples [43]. For the MNIST model, 40 iterations of PGD were run, with a step size of 0.01. Gradient

Table 8: Comparison of attack transferability from the undefended network to the defensively distilled network ($T = 100$) on MNIST with varying transferability parameter κ. ASR means attack success rate (%). N.A. means not "not available" due to zero ASR. There is no κ parameter for I-FGM.

		Best case				Average case				Worst case			
Method	κ	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞
I-FGM-L_1	None	12.2	18.39	1.604	0.418	1.6	19	1.658	0.43	0	N.A.	N.A.	N.A.
I-FGM-L_2	None	9.8	17.77	1.537	0.395	1.3	17.25	1.533	0.408	0	N.A.	N.A.	N.A.
I-FGM-L_∞	None	14.7	46.38	2.311	0.145	1.7	48.3	2.44	0.158	0	N.A.	N.A.	N.A.
C&W (L_2)	0	5.4	11.13	1.103	0.338	1.1	10.16	1.033	0.343	0	N.A.	N.A.	N.A.
	5	16.6	15.58	1.491	0.424	3.4	17.35	1.615	0.46	0	N.A.	N.A.	N.A.
	10	42.2	21.94	2.033	0.525	6.5	21.97	2.001	0.527	0	N.A.	N.A.	N.A.
	15	74.2	27.65	2.491	0.603	22.6	32.54	2.869	0.671	0.4	56.93	4.628	0.843
	20	92.9	29.71	2.665	0.639	44.4	38.34	3.322	0.745	2.4	54.25	4.708	0.91
	25	98.7	30.12	2.719	0.664	62.9	45.41	3.837	0.805	10.9	71.22	5.946	0.972
	30	99.8	31.17	2.829	0.69	78.1	49.63	4.15	0.847	23	85.93	6.923	0.987
	35	100	33.27	3.012	0.727	84.2	55.56	4.583	0.886	30.5	105.9	8.072	0.993
	40	100	36.13	3.255	0.772	87.4	61.25	4.98	0.918	21	125.2	9.09	0.995
	45	100	39.86	3.553	0.818	85.2	67.82	5.43	0.936	7.4	146.9	10.21	0.996
	50	100	44.2	3.892	0.868	80.6	70.87	5.639	0.953	0.5	158.4	10.8	0.996
	55	100	49.37	4.284	0.907	73	76.77	6.034	0.969	0	N.A.	N.A.	N.A.
	60	100	54.97	4.703	0.937	67.9	82.07	6.395	0.976	0	N.A.	N.A.	N.A.
EAD (EN rule)	0	6	8.373	1.197	0.426	0.6	4.876	0.813	0.307	0	N.A.	N.A.	N.A.
	5	18.2	11.45	1.547	0.515	2.5	13.07	1.691	0.549	0	N.A.	N.A.	N.A.
	10	39.5	15.36	1.916	0.59	8.4	16.45	1.989	0.6	0	N.A.	N.A.	N.A.
	15	69.2	19.18	2.263	0.651	19.2	22.74	2.531	0.697	0.4	31.18	3.238	0.846
	20	89.5	21.98	2.519	0.692	37	28.36	2.99	0.778	1.8	39.91	3.951	0.897
	25	98.3	23.92	2.694	0.724	58	34.14	3.445	0.831	7.9	49.12	4.65	0.973
	30	99.9	25.52	2.838	0.748	76.3	40.2	3.909	0.884	23.7	59.9	5.404	0.993
	35	100	27.42	3.009	0.778	87.9	45.62	4.324	0.92	47.4	70.93	6.176	0.999
	40	100	30.23	3.248	0.814	95.2	52.33	4.805	0.945	71.3	83.19	6.981	1
	45	100	33.61	3.526	0.857	98	57.75	5.194	0.965	86.2	98.51	7.904	1
	50	100	37.59	3.843	0.899	98.6	66.22	5.758	0.978	87	115.7	8.851	1
	55	100	42.01	4.193	0.934	94.4	70.66	6.09	0.986	44.2	127	9.487	1
	60	100	46.7	4.562	0.961	90	75.59	6.419	0.992	13.3	140.35	10.3	1
EAD (L_1 rule)	0	6	6.392	1.431	0.628	0.5	6.57	1.565	0.678	0	N.A.	N.A.	N.A.
	5	19	8.914	1.807	0.728	3.2	9.717	1.884	0.738	0	N.A.	N.A.	N.A.
	10	40.6	12.16	2.154	0.773	7.5	13.74	2.27	0.8	0	N.A.	N.A.	N.A.
	15	70.5	15.39	2.481	0.809	19	18.12	2.689	0.865	0.3	23.15	3.024	0.884
	20	90	17.73	2.718	0.83	39.4	24.15	3.182	0.902	1.9	38.22	4.173	0.979
	25	98.6	19.71	2.897	0.851	59.3	30.33	3.652	0.933	7.9	45.74	4.818	0.997
	30	99.8	21.1	3.023	0.862	76.9	37.38	4.191	0.954	22.2	55.54	5.529	1
	35	100	23	3.186	0.882	89.3	41.13	4.468	0.968	46.8	66.76	6.256	1
	40	100	25.86	3.406	0.904	96.3	47.54	4.913	0.979	69.9	80.05	7.064	1
	45	100	29.4	3.665	0.931	97.6	55.16	5.399	0.988	85.8	96.05	7.94	1
	50	100	33.71	3.957	0.95	98.1	62.01	5.856	0.992	85.7	113.6	8.845	1
	55	100	38.09	4.293	0.971	93.6	65.79	6.112	0.995	43.8	126.4	9.519	1
	60	100	42.7	4.66	0.985	89.6	72.49	6.572	0.997	13	141.3	10.36	1

steps were taken in the L_∞ norm. The network was trained and evaluated against perturbations of size no greater than $\epsilon = 0.3$.

Table 9: Results of training the Madry Defense Model with a PGD adversary constrained under varying ϵ. 'Nat Test Accuracy' denotes accuracy on classifying original images. 'Adv Test Accuracy' denotes accuracy on classifying adversarial images generated by the same PGD adversary as was used for training.

Epsilon	Nat Test Accuracy	Adv Test Accuracy
0.1	99.38	95.53
0.2	99	92.65
0.3	98.16	91.14
0.4	11.35	11.35
0.5	11.35	11.35
0.6	11.35	11.35
0.7	10.28	10.28
0.8	11.35	11.35
0.9	11.35	11.35
1	11.35	11.35

Due to this, it is expected that the existing Madry Defense Model performs poorly against PGD attacks with $\epsilon > 0.3$. As suggested in [43], the PGD attack was run for 40 iterations, and to account for varying ϵ, the stepsize was set to $2\epsilon/40$. The adversarial retraining results are shown in Table 9. These results suggest that the Madry Defense Model can not be successfully adversarially trained using a PGD adversary with $\epsilon > 0.3$. This is understandable as with such large ϵ, the visual distortion is clearly perceptible.

We test the transferability properties of attacks in both the targeted case and non-targeted case against the adversarially trained model. We test the ability for attacks to generate transferable adversarial examples using an ensemble of undefended models, where the ensemble size is set to 3. We expect that if an adversarial example remains adversarial to multiple models, it is more likely to transfer to other unseen models [51, 73]. The undefended models we use are naturally trained networks of the same architecture as the defended model; the architecture was provided in the competition. For generating adversarial examples, we compare the optimization-based approach and the iterative fast gradient-based approach using EAD (with the EN decision rule) and PGD, respectively.

In our experiment, 1000 random samples from the MNIST test set were used. For the targeted case, a target class that is different from the original one was randomly

selected for each input image. The highest attack success rate (ASR) in both the targeted and non-targeted cases was yielded at $\beta = 0.01$. This was in fact the largest β tested, indicating the importance of minimizing the L_1 distortion for generating transferable adversarial examples. Furthermore, the improvement in ASR with increasing β was seen to be more significant at lower κ, indicating the importance of minimizing the L_1 distortion for generating transferable adversarial examples with minimal visual distortion. For PGD and I-FGM, ϵ was tuned from 0.1 to 1.0 at 0.1 increments.

In Table 10, the results for tuning κ for C&W and EAD at $\beta = 0.01$ are provided, and are presented with the results for PGD and I-FGM at the lowest ϵ values at which the highest ASR was yielded, for comparison. It is observed that in both the targeted case and the non-targeted case, EAD outperforms C&W at all κ. Furthermore, in the targeted case, at the optimal $\kappa = 50$, EAD's adversarial examples surprisingly have lower average L_2 distortion.

In the targeted case, EAD outperforms PGD and I-FGM at $\kappa = 30$ with much lower L_1 and L_2 distortion. In the non-targeted case, PGD and I-FGM yield similar ASR at lower L_∞ distortion. However, we argue that in the latter case the drastic increase in induced L_1 and L_2 distortion of PGD and I-FGM to generate said adversarial examples indicates greater visual distortion, and thus that the generated examples are less adversarial in nature.

We find that adversarial examples generated by PGD even at low ϵ such as 0.3, at which the attack performance is weak, have visually apparent noise. In Figure 9, adversarial examples generated by EAD are directly compared to those generated by PGD with similar average L_∞ distortion. EAD tuned to the optimal β (0.01) was used to generate adversarial examples with $\kappa = 10$. As can be seen in Table 10, the average L_∞ distortion under this configuration is 0.8. Therefore, adversarial examples

Table 10: Comparison of tuned PGD, I-FGM, C&W, and EAD adversarial examples at various confidence levels. ASR means attack success rate (%). The distortion metrics are averaged over successful examples.

		Targeted				Non-Targeted			
Attack Method	Confidence	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞
PGD	None	68.5	188.3	8.947	0.6	99.9	270.5	13.27	0.8
I-FGM	None	75.1	144.5	7.406	0.915	99.8	199.4	10.66	0.9
C&W	10	1.1	34.15	2.482	0.548	4.9	23.23	1.702	0.424
	30	69.4	68.14	4.864	0.871	71.3	51.04	3.698	0.756
	50	92.9	117.45	8.041	0.987	99.1	78.65	5.598	0.937
	70	34.8	169.7	10.88	0.994	99	119.4	8.097	0.99
EAD	10	27.4	25.79	3.209	0.876	39.9	19.19	2.636	0.8
	30	85.8	49.64	5.179	0.995	94.5	34.28	4.192	0.971
	50	98.5	93.46	7.711	1	99.6	57.68	5.839	0.999
	70	67.2	148.9	10.36	1	99.8	90.84	7.719	1

were generated by PGD with $\epsilon = 0.8$. Clearly, performing elastic-net minimization aids in minimizing visual distortion, even when the L_∞ distortion is large.

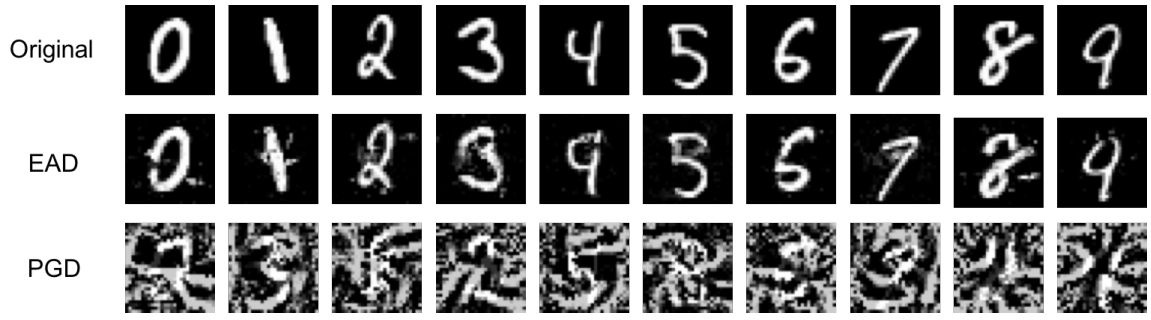


Figure 9: Visual illustration of adversarial examples crafted in the non-targeted case by EAD and PGD with similar average L_∞ distortion.

4.3.7.3 Feature Squeezing

Feature squeezing relies on applying input transformations to reduce the degrees of freedom available to an adversary by "squeezing" out unnecessary input features. The authors in [58] propose a detection method using such input transformations by relying on the intuition that if the original and squeezed inputs produce substantially different outputs from the model, the input is likely to be adversarial. By comparing the difference between predictions with a selected threshold value, the system is de-

Table 11: Comparison of PGD, C&W, and EAD results against the MNIST joint detector at various confidence levels. ASR means attack success rate (%). The distortion metrics are averaged over successful examples.

		Non-Targeted				Targeted							
						Next				LL			
Attack Method	Confidence	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞
PGD	None	100%	196.0	10.17	0.900	78%	169.8	8.225	0.881	67%	188.1	9.091	0.991
C&W	10	0%	21.05	1.962	0.568	0%	31.94	2.748	0.655	0%	37.78	3.207	0.732
	20	15%	27,21	2.472	0.665	10%	40.51	3.419	0.763	24%	47.86	3.977	0.820
	30	64%	34.30	3.019	0.754	67%	47.43	3.973	0.842	91%	59.56	4.811	0.888
	40	87%	42.04	3.590	0.831	97%	61.12	4.938	0.922	100%	72.88	5.715	0.939
EAD	10	24%	11.44	2.286	0.879	7%	19.69	3.114	0.942	7%	23.99	3.481	0.955
	20	80%	15.26	2.766	0.921	65%	26.80	3.752	0.964	78%	31.81	4.122	0.972
	30	95%	20.17	3.264	0.957	97%	35.50	4.449	0.983	93%	39.68	4.769	0.991
	40	97%	26.50	3.803	0.972	100%	44.75	5.114	0.992	100%	50.21	5.532	0.997

Table 12: Comparison of I-FGM, C&W, and EAD results against the CIFAR-10 joint detector at various confidence levels. ASR means attack success rate (%). The distortion metrics are averaged over successful examples.

		Non-Targeted				Targeted							
						Next				LL			
Attack Method	Confidence	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞	ASR	L_1	L_2	L_∞
I-FGM	None	100%	81.18	1.833	0.070	100%	212.0	4.979	0.299	100%	214.9	5.042	0.300
C&W	10	32%	10.51	0.274	0.033	0%	14.25	0.368	0.042	0%	17.36	0.445	0.049
	30	78%	28.80	0.712	0.073	51%	37.11	0.901	0.083	6%	41.51	1.006	0.093
	50	96%	59.32	1.416	0.130	98%	82.54	1.954	0.169	94%	90.17	2.129	0.179
	70	100%	120.2	2.827	0.243	100%	201.2	4.713	0.375	100%	212.2	4.962	0.403
EAD	10	46%	6.371	0.379	0.079	10%	8.187	0.508	0.109	0%	10.17	0.597	0.121
	30	78%	18.94	0.876	0.146	51%	25.98	1.090	0.166	23%	29.58	1.209	0.175
	50	94%	42.36	1.550	0.206	96%	62.90	2.094	0.247	90%	70.23	2.296	0.275
	70	100%	83.14	2.670	0.317	100%	157.9	4.466	0.477	100%	172.8	4.811	0.502

signed to output the correct prediction for legitimate examples and reject adversarial inputs. By combining multiple squeezers in a joint detection framework, the authors claim that the system can can successfully detect adversarial examples from eleven state-of-the-art methods [58].

Two types of feature squeezing were focused on by the authors in [58]: (i) reducing the color bit depth of images; and (ii) using smoothing (both local and non-local) to reduce the variation among pixels. For the detection method, the model's original prediction is compared with the prediction on the squeezed sample using the L_1 norm. As a defender typically does not know the exact attack method, multiple feature squeezers are combined by outputting the maximum distance. The threshold

Figure 10: Randomly selected set of non-targeted MNIST adversarial examples generated by EAD. First row: Original, Subsequent rows: $\kappa = \{10, 20, 30\}$.

Figure 11: Randomly selected set of non-targeted CIFAR-10 adversarial examples generated by EAD. First row: Original, Subsequent rows: $\kappa = \{10, 30, 50\}$.

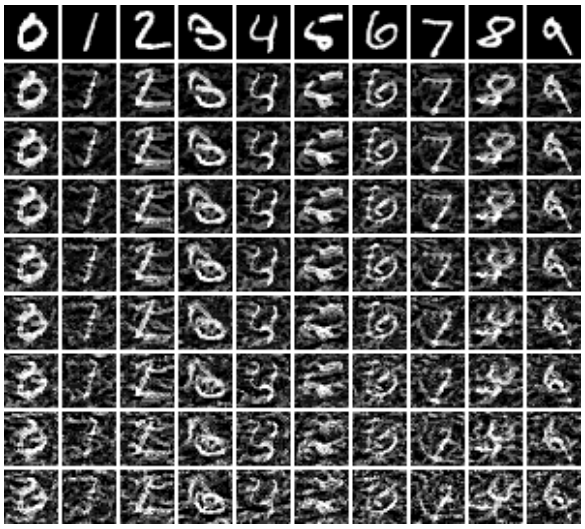


Figure 12: Randomly selected set of non-targeted MNIST adversarial examples generated by I-FGM.
First row: Original, Subsequent rows: $\epsilon = \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8.0.9, 1.0\}$.

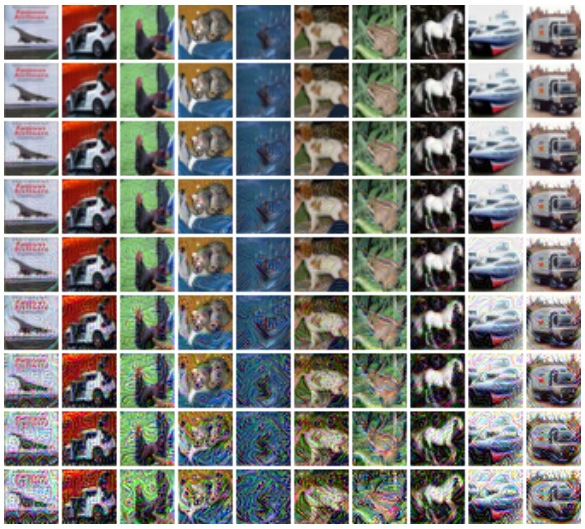


Figure 13: Randomly selected set of non-targeted CIFAR-10 adversarial examples generated by I-FGM.
First row: Original, Subsequent rows: $\epsilon = \{0.008, 0.04, 0.07, 0.1, 0.2, 0.3, 0.4, 0.5\}$.

is selected targeting a false positive rate below 5% by choosing a threshold that is exceeded by no more than 5% of legitimate samples.

For MNIST, the joint detector consists of a 1-bit depth squeezer with 2x2 median smoothing. For CIFAR-10, the joint detector consists of a 5-bit depth squeezer with 2x2 median smoothing and a non-local means filter with a 13x13 search window, 3x3 patch size, and a Gaussian kernel bandwidth size of 2. We use the same thresholds as used in [58]. We generate adversarial examples using the EAD [2] (with the EN decision rule) and I-FGM [46] attacks.

We randomly sample 100 images from the MNIST and CIFAR-10 test sets. For each dataset, we use the same pre-trained state-of-the-art models as used in [58]. We generate adversarial examples in the non-targeted case, force network to misclassify, and in the targeted case, force network to misclassify to a target class t. As done in [58], we try two different targets, the *Next* class ($t = label + 1$ mod $\#$ *of classes*) and the least-likely class (LL).

The generated adversarial examples are tested against the proposed MNIST and CIFAR-10 joint detection configurations. In Tables 11 and 12, the results of tuning κ for C&W and EAD are provided, and are presented with the results for I-FGM at the lowest ϵ value at which the highest attack success rate (ASR) was yielded, against the MNIST and CIFAR-10 joint detectors, respectively. In all cases, EAD outperforms the C&W L_2 attack, particularly at lower confidence levels, indicating the importance of minimizing the L_1 distortion for generating robust adversarial examples with minimal visual distortion. Specifically, we find that with enough κ, each attack is able to achieve near 100% ASR against the joint detectors.

In Figure 10, non-targeted MNIST adversarial examples generated by EAD are shown at $\kappa = \{10, 20, 30\}$. In Figure 11, non-targeted CIFAR-10 adversarial examples generated by EAD are shown at $\kappa = \{10, 30, 50\}$. In Figure 12, non-targeted MNIST adversarial examples generated by I-FGM are shown at $\epsilon = \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8.0.9, 1.0\}$. In Figure 13, non-targeted CIFAR-10 adversarial examples generated by I-FGM are shown at $\epsilon = \{0.008, 0.04, 0.07, 0.1, 0.2, 0.3, 0.4, 0.5\}$. These figures indicate that adversarial examples generated by EAD at high κ, which bypass the joint feature squeezing detector, have minimal visual distortion. This holds true for adversarial examples generated by I-FGM with high ϵ on CIFAR-10, but not on MNIST.

5 Conclusion

Gradient-based adversarial attacks have been readily applied in the white-box case, where the attacker has complete access to the target model and can compute the gradients via back-propagation. We have extended such approaches to the black-box case, where only query access is given to the attacker, with ZOO, which uses the finite difference method to estimate the gradients for optimization from the output scores. We also have improved the state-of-the-art in no-box attacks, where the attacker isn't even capable of querying the target model, with EAD, which incorporates L_1 minimization in order to encourage sparsity in the perturbation and hence generate more robust transferable adversarial examples. Through experimental results attacking state-of-the-art models trained on the MNIST, CIFAR-10, and ImageNet datasets, and state-of-the-art defenses for each dataset, we have validated the effectiveness of the proposed attacks.

Regarding future work, it is worth exploring whether gradient-free optimization strategies, like genetic algorithms, can be applied for generating adversarial examples in order to avoid the costly procedure of estimating the gradient. Furthermore, the black-box attack setting discussed in this work assumed the target model's output consists of each classes' confidence scores, however in many real-world scenarios, only the label is given. Formulating attacks which can find adversarial examples operating solely on label information is certainly of interest. Lastly, the experimental results presented here are extensive, however it is worth noting that in all experiments, solely the image domain has been considered. Work in other domains, such as text and speech, would aid in our understanding of the problem of adversarial examples.

References

[1] P. Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C. Hsieh, "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. ACM, 2017, pp. 15–26.

[2] P. Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C. Hsieh, "Ead: Elastic-net attacks to deep neural networks via adversarial examples," *arXiv preprint arXiv:1709.0414*, 2017.

[3] Y. Sharma and P. Y. Chen, "Attacking the madry defense model with l1-based adversarial examples," *arXiv preprint arXiv:1710.10733*, 2017.

[4] ——, "Bypassing feature squeezing by increasing adversary strength," *arXiv preprint arXiv:1803.09868*, 2018.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[6] A. Cauchy, "Methodes generales pour la resolution des syst'emes dequations simultanees," *C.R. Acad. Sci. Par., 25:536–538*, 1847.

[7] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.

[8] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York: Springer, 2006.

[9] J. B. Rosen, "The gradient projection method for nonlinear programming. part i. linear constraints," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 1, pp. 181–217, 1960. [Online]. Available: http://www.jstor.org/stable/2098960

[10] W. Karush, "Minima of Functions of Several Variables with Inequalities as Side Constraints," Master's thesis, Dept.˜of Mathematics, Univ.˜of Chicago, 1939.

[11] H. W. Kuhn and A. W. Tucker, "Nonlinear programming," in *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, 1950*. Berkeley and Los Angeles: University of California Press, 1951, pp. 481–492.

[12] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[13] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1701–1708. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2014.220

[14] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 7, pp. 179–188, 1936.

[15] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *Trans. Evol. Comp*, vol. 1, no. 1, pp. 67–82, Apr. 1997. [Online]. Available: http://dx.doi.org/10.1109/4235.585893

[16] K. Jarrett, K. Kavukcuoglu, and Y. Lecun, "What is the best multi-stage architecture for object recognition?" 2009.

[17] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. USA: Omnipress, 2010, pp. 807–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104322.3104425

[18] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 315–323. [Online]. Available: http://proceedings.mlr.press/v15/glorot11a.html

[19] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ser. ICCV '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1026–1034. [Online]. Available: http://dx.doi.org/10.1109/ICCV.2015.123

[21] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, pp. III–1319–III–1327. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042817.3043084

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2627435.2670313

[23] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, and I. Goodfellow, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[24] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1," D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds. Cambridge, MA, USA: MIT Press, 1986, ch. Learning Internal Representations by Error Propagation, pp. 318–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=104279.104293

[26] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[27] B. Polyak, "Some methods of speeding up the convergence of iteration methods," vol. 4, pp. 1–17, 12 1964.

[28] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, pp. III–1139–III–1147. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042817.3043064

[29] Y. Nesterov, "A method of solving a convex programming problem with convergence rate O(1/sqr(k))," *Soviet Mathematics Doklady*, vol. 27, pp. 372–376, 1983. [Online]. Available: http://www.core.ucl.ac.be/\~{}nesterov/Research/Papers/DAN83.pdf

[30] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[31] G. Hinton, N. Srivastava, and K. Swersky, "Lecture 6a overview of mini–batch gradient descent," 2012, https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[32] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[34] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions." CVPR, 2015.

[35] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *Trans. Sig. Proc.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1109/78.650093

[36] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014, cite arxiv:1406.1078Comment: EMNLP 2014. [Online]. Available: http://arxiv.org/abs/1406.1078

[37] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[38] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München," 1991.

[39] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994. [Online]. Available: http://dx.doi.org/10.1109/72.279181

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[41] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[42] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing robust adversarial examples," *arXiv preprint arXiv:1707.07397*, 2017.

[43] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[44] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," *ICML; arXiv preprint arXiv:1703.04730*, 2017.

[45] Y. Dong, H. Su, J. Zhu, and F. Bao, "Towards interpretable deep neural networks by leveraging adversarial examples," *arXiv preprint arXiv:1708.05493*, 2017.

[46] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," *ICLR'17; arXiv preprint arXiv:1611.01236*, 2016.

[47] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," *arXiv preprint arXiv:1511.07528*, 2016.

[48] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, no. EPFL-CONF-218057, 2016.

[49] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," *arXiv preprint arXiv:1608.04644*, 2017.

[50] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 506–519.

[51] Y. Liu, X. Chen, C. Liu, and D. Song, "Delving into transferable adversarial examples and black-box attacks," *arXiv preprint arXiv:1611.02770*, 2016.

[52] S. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," *arXiv preprint arXiv:1610.08401*, 2016.

[53] S. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, P. Frossard, and S. Soatto, "Analysis of universal adversarial perturbations," *arXiv preprint arXiv:1705.09554*, 2017.

[54] F. Tramèr, A. Kurakin, N. Papernot, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," *arXiv preprint arXiv:1705.07204*, 2017.

[55] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," *arXiv preprint arXiv:1511.04508*, 2016.

[56] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[57] C. Guo, M. Rana, and L. van der Maaten, "Countering adversarial images using input transformations," *arXiv preprint arXiv:1711.00117*, 2017.

[58] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint arXiv:1704.01155*, 2017.

[59] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[60] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *ICLR 2016 Workshop*, 2016. [Online]. Available: https://arxiv.org/abs/1602.07261

[61] Y. Nesterov *et al.*, "Random gradient-free minimization of convex functions," Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), Tech. Rep., 2011.

[62] S. Ghadimi and G. Lan, "Stochastic first-and zeroth-order methods for nonconvex stochastic programming," *SIAM Journal on Optimization*, vol. 23, no. 4, pp. 2341–2368, 2013.

[63] X. Lian, H. Zhang, C.-J. Hsieh, Y. Huang, and J. Liu, "A comprehensive linear speedup analysis for asynchronous stochastic parallel optimization from zeroth-order to first-order," in *Advances in Neural Information Processing Systems*, 2016, pp. 3054–3062.

[64] P. D. Lax and M. S. Terrell, *Calculus with applications*. Springer, 2014.

[65] D. P. Bertsekas, *Nonlinear programming*.

[66] H. Fu, M. K. Ng, M. Nikolova, and J. L. Barlow, "Efficient minimization methods of mixed l2-l1 and l1-l1 norms for image restoration," *SIAM Journal on Scientific computing 27(6):1881–1902*, 2006.

[67] E. J. Candès and M. B. Wakin, "An introduction to compressive sampling," *IEEE signal processing magazine 25(2):21–30*, 2008.

[68] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 67(2):301–320.*, 2005.

[69] J. Duchi and Y. Singer, "Efficient online and batch learning using forward backward splitting," *Journal of Machine Learning Research 10(Dec):2899–2934*, 2009.

[70] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM journal on imaging sciences 2(1):183–202.*, 2009.

[71] N. Parikh and S. Boyd, "Proximal algorithms," *Foundations and Trends in Optimization 1(3):127–239.*, 2009.

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

[73] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016.

A Code Sample

The remaining pages contain a representative sample of the code used for the presented experiments, detailing the ZOO and EAD (with the EN decision rule) Attack Algorithms.

A.1 ZOO Attack

```
## l2_attack_black.py -- attack a black-box network optimizing for
↪  l_2 distance
##
import sys
import os
import tensorflow as tf
import numpy as np
import scipy.misc
from numba import jit
import math
import time

BINARY_SEARCH_STEPS = 1  # number of times to adjust the constant
↪  with binary search
MAX_ITERATIONS = 10000   # number of iterations to perform gradient
↪  descent
ABORT_EARLY = True       # if we stop improving, abort gradient descent
↪  early
LEARNING_RATE = 2e-3     # larger values converge faster to less
↪  accurate results
TARGETED = True          # should we target one specific class? or
↪  just be wrong?
CONFIDENCE = 0           # how strong the adversarial example should be
INITIAL_CONST = 0.5      # the initial constant c to pick as a first
↪  guess

@jit(nopython=True)
def coordinate_ADAM(losses, indice, grad, hess, batch_size, mt_arr,
↪  vt_arr, real_modifier, up, down, lr, adam_epoch, beta1, beta2,
↪  proj):
    # indice = np.array(range(0, 3*299*299), dtype = np.int32)
    for i in range(batch_size):
        grad[i] = (losses[i*2+1] - losses[i*2+2]) / 0.0002
    # true_grads = self.sess.run(self.grad_op,
    ↪  feed_dict={self.modifier: self.real_modifier})
```

```
        # true_grads, losses, l2s, scores, nimgs =
        ↪  self.sess.run([self.grad_op, self.loss, self.l2dist,
        ↪  self.output, self.newimg], feed_dict={self.modifier:
        ↪  self.real_modifier})
        # grad = true_grads[0].reshape(-1)[indice]
        # print(grad, true_grads[0].reshape(-1)[indice])
        # self.real_modifier.reshape(-1)[indice] -= self.LEARNING_RATE *
        ↪  grad
        # self.real_modifier -= self.LEARNING_RATE * true_grads[0]
        # ADAM update
        mt = mt_arr[indice]
        mt = beta1 * mt + (1 - beta1) * grad
        mt_arr[indice] = mt
        vt = vt_arr[indice]
        vt = beta2 * vt + (1 - beta2) * (grad * grad)
        vt_arr[indice] = vt
        # epoch is an array; for each index we can have a different epoch
        ↪  number
        epoch = adam_epoch[indice]
        corr = (np.sqrt(1 - np.power(beta2,epoch))) / (1 - np.power(beta1,
        ↪  epoch))
        m = real_modifier.reshape(-1)
        old_val = m[indice]
        old_val -= lr * corr * mt / (np.sqrt(vt) + 1e-8)
        # set it back to [-0.5, +0.5] region
        if proj:
            old_val = np.maximum(np.minimum(old_val, up[indice]),
            ↪  down[indice])
        # print(grad)
        # print(old_val - m[indice])
        m[indice] = old_val
        adam_epoch[indice] = epoch + 1

    @jit(nopython=True)
    def coordinate_Newton(losses, indice, grad, hess, batch_size, mt_arr,
    ↪  vt_arr, real_modifier, up, down, lr, adam_epoch, beta1, beta2,
    ↪  proj):
        # def sign(x):
        #     return np.piecewise(x, [x < 0, x >= 0], [-1, 1])
        cur_loss = losses[0]
```

```
    for i in range(batch_size):
        grad[i] = (losses[i*2+1] - losses[i*2+2]) / 0.0002
        hess[i] = (losses[i*2+1] - 2 * cur_loss + losses[i*2+2]) /
        ↪  (0.0001 * 0.0001)
    # print("New epoch:")
    # print('grad', grad)
    # print('hess', hess)
    # hess[hess < 0] = 1.0
    # hess[np.abs(hess) < 0.1] = sign(hess[np.abs(hess) < 0.1]) * 0.1
    # negative hessian cannot provide second order information, just
    ↪  do a gradient descent
    hess[hess < 0] = 1.0
    # hessian too small, could be numerical problems
    hess[hess < 0.1] = 0.1
    # print(hess)
    m = real_modifier.reshape(-1)
    old_val = m[indice]
    old_val -= lr * grad / hess
    # set it back to [-0.5, +0.5] region
    if proj:
        old_val = np.maximum(np.minimum(old_val, up[indice]),
        ↪  down[indice])
    # print('delta', old_val - m[indice])
    m[indice] = old_val
    # print(m[indice])

@jit(nopython=True)
def coordinate_Newton_ADAM(losses, indice, grad, hess, batch_size,
↪  mt_arr, vt_arr, real_modifier, up, down, lr, adam_epoch, beta1,
↪  beta2, proj):
    cur_loss = losses[0]
    for i in range(batch_size):
        grad[i] = (losses[i*2+1] - losses[i*2+2]) / 0.0002
        hess[i] = (losses[i*2+1] - 2 * cur_loss + losses[i*2+2]) /
        ↪  (0.0001 * 0.0001)
    # print("New epoch:")
    # print(grad)
    # print(hess)
    # positive hessian, using newton's method
    hess_indice = (hess >= 0)
```

```
# print(hess_indice)
# negative hessian, using ADAM
adam_indice = (hess < 0)
# print(adam_indice)
# print(sum(hess_indice), sum(adam_indice))
hess[hess < 0] = 1.0
hess[hess < 0.1] = 0.1
# hess[np.abs(hess) < 0.1] = sign(hess[np.abs(hess) < 0.1]) * 0.1
# print(adam_indice)
# Newton's Method
m = real_modifier.reshape(-1)
old_val = m[indice[hess_indice]]
old_val -= lr * grad[hess_indice] / hess[hess_indice]
# set it back to [-0.5, +0.5] region
if proj:
    old_val = np.maximum(np.minimum(old_val,
    ↪  up[indice[hess_indice]]), down[indice[hess_indice]])
m[indice[hess_indice]] = old_val
# ADMM
mt = mt_arr[indice]
mt = beta1 * mt + (1 - beta1) * grad
mt_arr[indice] = mt
vt = vt_arr[indice]
vt = beta2 * vt + (1 - beta2) * (grad * grad)
vt_arr[indice] = vt
# epoch is an array; for each index we can have a different epoch
↪  number
epoch = adam_epoch[indice]
corr = (np.sqrt(1 - np.power(beta2,epoch[adam_indice]))) / (1 -
↪  np.power(beta1, epoch[adam_indice]))
old_val = m[indice[adam_indice]]
old_val -= lr * corr * mt[adam_indice] / (np.sqrt(vt[adam_indice])
↪  + 1e-8)
# old_val -= lr * grad[adam_indice]
# set it back to [-0.5, +0.5] region
if proj:
    old_val = np.maximum(np.minimum(old_val,
    ↪  up[indice[adam_indice]]), down[indice[adam_indice]])
m[indice[adam_indice]] = old_val
adam_epoch[indice] = epoch + 1
```

```
    # print(m[indice])

class BlackBoxL2:
    def __init__(self, sess, model, batch_size=1, confidence =
    ↪  CONFIDENCE,
                 targeted = TARGETED, learning_rate = LEARNING_RATE,
                 binary_search_steps = BINARY_SEARCH_STEPS,
                 ↪  max_iterations = MAX_ITERATIONS, print_every =
                 ↪  100, early_stop_iters = 0,
                 abort_early = ABORT_EARLY,
                 initial_const = INITIAL_CONST,
                 use_log = False, use_tanh = True, use_resize = False,
                 ↪  adam_beta1 = 0.9, adam_beta2 = 0.999,
                 ↪  reset_adam_after_found = False,
                 solver = "adam", save_ckpts = "", load_checkpoint =
                 ↪  "", start_iter = 0,
                 init_size = 32, use_importance = True):
        """
        The L_2 optimized attack.

        This attack is the most efficient and should be used as the
↪   primary
        attack to evaluate potential defenses.

        Returns adversarial examples for the supplied model.

        confidence: Confidence of adversarial examples: higher
↪   produces examples
              that are farther away, but more strongly classified as
↪   adversarial.
        batch_size: Number of gradient evaluations to run
↪   simultaneously.
        targeted: True if we should perform a targetted attack, False
↪   otherwise.
        learning_rate: The learning rate for the attack algorithm.
↪   Smaller values
              produce better results but are slower to converge.
        binary_search_steps: The number of times we perform binary
↪   search to
```

```
        find the optimal tradeoff-constant between distance and
↪ confidence.
        max_iterations: The maximum number of iterations. Larger
↪ values are more
        accurate; setting too small will require a large learning
↪ rate and will
        produce poor results.
        abort_early: If true, allows early aborts if gradient descent
↪ gets stuck.
        initial_const: The initial tradeoff-constant to use to tune
↪ the relative
        importance of distance and confidence. If
↪ binary_search_steps is large,
        the initial constant is not important.
        """

        image_size, num_channels, num_labels = model.image_size,
        ↪  model.num_channels, model.num_labels
        self.model = model
        self.sess = sess
        self.TARGETED = targeted
        self.LEARNING_RATE = learning_rate
        self.MAX_ITERATIONS = max_iterations
        self.print_every = print_every
        self.early_stop_iters = early_stop_iters if early_stop_iters
        ↪  != 0 else max_iterations // 10
        print("early stop:", self.early_stop_iters)
        self.BINARY_SEARCH_STEPS = binary_search_steps
        self.ABORT_EARLY = abort_early
        self.CONFIDENCE = confidence
        self.initial_const = initial_const
        self.start_iter = start_iter
        self.batch_size = batch_size
        self.num_channels = num_channels
        self.resize_init_size = init_size
        self.use_importance = use_importance
        if use_resize:
            self.small_x = self.resize_init_size
            self.small_y = self.resize_init_size
        else:
```

```
        self.small_x = image_size
        self.small_y = image_size

    self.use_tanh = use_tanh
    self.use_resize = use_resize
    self.save_ckpts = save_ckpts
    if save_ckpts:
        os.system("mkdir -p {}".format(save_ckpts))

    self.repeat = binary_search_steps >= 10

    # each batch has a different modifier value (see below) to
    ↪  evaluate
    # small_shape = (None,self.small_x,self.small_y,num_channels)
    shape = (None,image_size,image_size,num_channels)
    single_shape = (image_size, image_size, num_channels)
   small_single_shape = (self.small_x, self.small_y, num_channels)

    # the variable we're going to optimize over
    # support multiple batches
   # support any size image, will be resized to model native size
    if self.use_resize:
        self.modifier = tf.placeholder(tf.float32, shape=(None,
        ↪  None, None, None))
        # scaled up image
        self.scaled_modifier =
        ↪  tf.image.resize_images(self.modifier, [image_size,
        ↪  image_size])
        # operator used for resizing image
        self.resize_size_x = tf.placeholder(tf.int32)
        self.resize_size_y = tf.placeholder(tf.int32)
        self.resize_input = tf.placeholder(tf.float32, shape=(1,
        ↪  None, None, None))
        self.resize_op = tf.image.resize_images(self.resize_input,
        ↪  [self.resize_size_x, self.resize_size_y])
    else:
        self.modifier = tf.placeholder(tf.float32, shape=(None,
        ↪  image_size, image_size, num_channels))
        # no resize
        self.scaled_modifier = self.modifier
```

```
        # the real variable, initialized to 0
        self.load_checkpoint = load_checkpoint
        if load_checkpoint:
            # if checkpoint is incorrect reshape will fail
            print("Using checkpint", load_checkpoint)
            self.real_modifier = np.load(load_checkpoint).reshape((1,)
            ↪  + small_single_shape)
        else:
            self.real_modifier = np.zeros((1,) + small_single_shape,
            ↪  dtype=np.float32)
        # self.real_modifier = np.random.randn(image_size * image_size
        ↪  * num_channels).astype(np.float32).reshape((1,) +
        ↪  single_shape)
        # self.real_modifier /= np.linalg.norm(self.real_modifier)
        # these are variables to be more efficient in sending data to
        ↪  tf
        # we only work on 1 image at once; the batch is for evaluation
        ↪  loss at different modifiers
        self.timg = tf.Variable(np.zeros(single_shape),
        ↪  dtype=tf.float32)
        self.tlab = tf.Variable(np.zeros(num_labels), dtype=tf.float32)
        self.const = tf.Variable(0.0, dtype=tf.float32)

        # and here's what we use to assign them
        self.assign_timg = tf.placeholder(tf.float32, single_shape)
        self.assign_tlab = tf.placeholder(tf.float32, num_labels)
        self.assign_const = tf.placeholder(tf.float32)

        # the resulting image, tanh'd to keep bounded from -0.5 to 0.5
        # broadcast self.timg to every dimension of modifier
        if use_tanh:
            self.newimg = tf.tanh(self.scaled_modifier + self.timg)/2
        else:
            self.newimg = self.scaled_modifier + self.timg

        # prediction BEFORE-SOFTMAX of the model
        # now we have output at #batch_size different modifiers
        # the output should have shape (batch_size, num_labels)
        self.output = model.predict(self.newimg)
```

```
# distance to the input data
if use_tanh:
    self.l2dist = tf.reduce_sum(tf.square(self.newimg-tf.tanh
    ↪  (self.timg)/2),
    ↪  [1,2,3])
else:
    self.l2dist = tf.reduce_sum(tf.square(self.newimg -
    ↪  self.timg), [1,2,3])

# compute the probability of the label class versus the
↪  maximum other
# self.tlab * self.output selects the Z value of real class
# because self.tlab is an one-hot vector
# the reduce_sum removes extra zeros, now get a vector of size
↪  #batch_size
self.real = tf.reduce_sum((self.tlab)*self.output,1)
# (1-self.tlab)*self.output gets all Z values for other
↪  classes
# Because soft Z values are negative, it is possible that all
↪  Z values are less than 0
# and we mistakenly select the real class as the max. So we
↪  minus 10000 for real class
self.other = tf.reduce_max((1-self.tlab)*self.output -
↪  (self.tlab*10000),1)

# If self.targeted is true, then the targets represents the
↪  target labels.
# If self.targeted is false, then targets are the original
↪  class labels.
if self.TARGETED:
    if use_log:
        # loss1 = - tf.log(self.real)
        loss1 = tf.maximum(0.0, tf.log(self.other + 1e-30) -
        ↪  tf.log(self.real + 1e-30))
    else:
        # if targetted, optimize for making the other class
        ↪  (real) most likely
        loss1 = tf.maximum(0.0,
        ↪  self.other-self.real+self.CONFIDENCE)
else:
```

```
        if use_log:
            # loss1 = tf.log(self.real)
            loss1 = tf.maximum(0.0, tf.log(self.real + 1e-30) -
            ↪  tf.log(self.other + 1e-30))
        else:
            # if untargeted, optimize for making this class least
            ↪  likely.
            loss1 = tf.maximum(0.0,
            ↪  self.real-self.other+self.CONFIDENCE)

    # sum up the losses (output is a vector of #batch_size)
    self.loss2 = self.l2dist
    self.loss1 = self.const*loss1
    self.loss = self.loss1+self.loss2

    # these are the variables to initialize when we run
    self.setup = []
    self.setup.append(self.timg.assign(self.assign_timg))
    self.setup.append(self.tlab.assign(self.assign_tlab))
    self.setup.append(self.const.assign(self.assign_const))

    # prepare the list of all valid variables
    var_size = self.small_x * self.small_y * num_channels
    self.use_var_len = var_size
    self.var_list = np.array(range(0, self.use_var_len), dtype =
    ↪  np.int32)
    self.used_var_list = np.zeros(var_size, dtype = np.int32)
    self.sample_prob = np.ones(var_size, dtype = np.float32) /
    ↪  var_size

    # upper and lower bounds for the modifier
    self.modifier_up = np.zeros(var_size, dtype = np.float32)
    self.modifier_down = np.zeros(var_size, dtype = np.float32)

    # random permutation for coordinate update
    self.perm = np.random.permutation(var_size)
    self.perm_index = 0

    # ADAM status
    self.mt = np.zeros(var_size, dtype = np.float32)
```

```
        self.vt = np.zeros(var_size, dtype = np.float32)
        # self.beta1 = 0.8
        # self.beta2 = 0.99
        self.beta1 = adam_beta1
        self.beta2 = adam_beta2
        self.reset_adam_after_found = reset_adam_after_found
        self.adam_epoch = np.ones(var_size, dtype = np.int32)
        self.stage = 0
        # variables used during optimization process
        self.grad = np.zeros(batch_size, dtype = np.float32)
        self.hess = np.zeros(batch_size, dtype = np.float32)
        # for testing
        self.grad_op = tf.gradients(self.loss, self.modifier)
        # compile numba function
        # self.coordinate_ADAM_numba = jit(coordinate_ADAM, nopython
        ↪  = True)
        # self.coordinate_ADAM_numba.recompile()
        # print(self.coordinate_ADAM_numba.inspect_llvm())
        # np.set_printoptions(threshold=np.nan)
        # set solver
        solver = solver.lower()
        self.solver_name = solver
        if solver == "adam":
            self.solver = coordinate_ADAM
        elif solver == "newton":
            self.solver = coordinate_Newton
        elif solver == "adam_newton":
            self.solver = coordinate_Newton_ADAM
        elif solver != "fake_zero":
            print("unknown solver", solver)
            self.solver = coordinate_ADAM
        print("Using", solver, "solver")

    def max_pooling(self, image, size):
        img_pool = np.copy(image)
        img_x = image.shape[0]
        img_y = image.shape[1]
        for i in range(0, img_x, size):
            for j in range(0, img_y, size):
```

```
                img_pool[i:i+size, j:j+size] = np.max(image[i:i+size,
                ↪  j:j+size])
        return img_pool

    def get_new_prob(self, prev_modifier, gen_double = False):
        prev_modifier = np.squeeze(prev_modifier)
        old_shape = prev_modifier.shape
        if gen_double:
            new_shape = (old_shape[0]*2, old_shape[1]*2, old_shape[2])
        else:
            new_shape = old_shape
        prob = np.empty(shape=new_shape, dtype = np.float32)
        for i in range(prev_modifier.shape[2]):
            image = np.abs(prev_modifier[:,:,i])
            image_pool = self.max_pooling(image, old_shape[0] // 8)
            if gen_double:
                prob[:,:,i] = scipy.misc.imresize(image_pool, 2.0,
                ↪  'nearest', mode = 'F')
            else:
                prob[:,:,i] = image_pool
        prob /= np.sum(prob)
        return prob


    def resize_img(self, small_x, small_y, reset_only = False):
        self.small_x = small_x
        self.small_y = small_y
        small_single_shape = (self.small_x, self.small_y,
        ↪  self.num_channels)
        if reset_only:
            self.real_modifier = np.zeros((1,) + small_single_shape,
            ↪  dtype=np.float32)
        else:
            # run the resize_op once to get the scaled image
            prev_modifier = np.copy(self.real_modifier)
            self.real_modifier = self.sess.run(self.resize_op,
            ↪  feed_dict={self.resize_size_x: self.small_x,
            ↪  self.resize_size_y: self.small_y, self.resize_input:
            ↪  self.real_modifier})
        # prepare the list of all valid variables
```

```
        var_size = self.small_x * self.small_y * self.num_channels
        self.use_var_len = var_size
        self.var_list = np.array(range(0, self.use_var_len), dtype =
        ↪  np.int32)
        # ADAM status
        self.mt = np.zeros(var_size, dtype = np.float32)
        self.vt = np.zeros(var_size, dtype = np.float32)
        self.adam_epoch = np.ones(var_size, dtype = np.int32)
        # update sample probability
        if reset_only:
            self.sample_prob = np.ones(var_size, dtype = np.float32) /
            ↪  var_size
        else:
            self.sample_prob = self.get_new_prob(prev_modifier, True)
            self.sample_prob = self.sample_prob.reshape(var_size)

    def fake_blackbox_optimizer(self):
        true_grads, losses, l2s, loss1, loss2, scores, nimgs =
        ↪  self.sess.run([self.grad_op, self.loss, self.l2dist,
        ↪  self.loss1, self.loss2, self.output, self.newimg],
        ↪  feed_dict={self.modifier: self.real_modifier})
        # ADAM update
        grad = true_grads[0].reshape(-1)
        # print(true_grads[0])
        epoch = self.adam_epoch[0]
        mt = self.beta1 * self.mt + (1 - self.beta1) * grad
        vt = self.beta2 * self.vt + (1 - self.beta2) * np.square(grad)
        corr = (math.sqrt(1 - self.beta2 ** epoch)) / (1 - self.beta1
        ↪  ** epoch)
        # print(grad.shape, mt.shape, vt.shape,
        ↪  self.real_modifier.shape)
        # m is a *view* of self.real_modifier
        m = self.real_modifier.reshape(-1)
        # this is in-place
        m -= self.LEARNING_RATE * corr * (mt / (np.sqrt(vt) + 1e-8))
        self.mt = mt
        self.vt = vt
        # m -= self.LEARNING_RATE * grad
        if not self.use_tanh:
```

```
            m_proj = np.maximum(np.minimum(m, self.modifier_up),
            ↪  self.modifier_down)
            np.copyto(m, m_proj)
        self.adam_epoch[0] = epoch + 1
        return losses[0], l2s[0], loss1[0], loss2[0], scores[0],
        ↪  nimgs[0]


    def blackbox_optimizer(self, iteration):
        # build new inputs, based on current variable value
        var = np.repeat(self.real_modifier, self.batch_size * 2 + 1,
        ↪  axis=0)
        var_size = self.real_modifier.size
        # print(s, "variables remaining")
        # var_indice = np.random.randint(0, self.var_list.size,
        ↪  size=self.batch_size)
        if self.use_importance:
            var_indice = np.random.choice(self.var_list.size,
            ↪  self.batch_size, replace=False, p = self.sample_prob)
        else:
            var_indice = np.random.choice(self.var_list.size,
            ↪  self.batch_size, replace=False)
        indice = self.var_list[var_indice]
        # indice = self.var_list
        # regenerate the permutations if we run out
        # if self.perm_index + self.batch_size >= var_size:
        #     self.perm = np.random.permutation(var_size)
        #     self.perm_index = 0
        # indice = self.perm[self.perm_index:self.perm_index +
        ↪  self.batch_size]
        # b[0] has the original modifier, b[1] has one index added
        ↪  0.0001
        for i in range(self.batch_size):
            var[i * 2 + 1].reshape(-1)[indice[i]] += 0.0001
            var[i * 2 + 2].reshape(-1)[indice[i]] -= 0.0001
        losses, l2s, loss1, loss2, scores, nimgs =
        ↪  self.sess.run([self.loss, self.l2dist, self.loss1,
        ↪  self.loss2, self.output, self.newimg],
        ↪  feed_dict={self.modifier: var})
```

```
# losses = self.sess.run(self.loss, feed_dict={self.modifier:
↪  var})
# t_grad = self.sess.run(self.grad_op,
↪  feed_dict={self.modifier: self.real_modifier})
# self.grad = t_grad[0].reshape(-1)
# true_grads = self.sess.run(self.grad_op,
↪  feed_dict={self.modifier: self.real_modifier})
# self.coordinate_ADAM_numba(losses, indice, self.grad,
↪  self.hess, self.batch_size, self.mt, self.vt,
↪  self.real_modifier, self.modifier_up,
↪  self.modifier_down, self.LEARNING_RATE, self.adam_epoch,
↪  self.beta1, self.beta2, not self.use_tanh)
# coordinate_ADAM(losses, indice, self.grad, self.hess,
↪  self.batch_size, self.mt, self.vt, self.real_modifier,
↪  self.modifier_up, self.modifier_down,
↪  self.LEARNING_RATE, self.adam_epoch, self.beta1,
↪  self.beta2, not self.use_tanh)
# coordinate_ADAM(losses, indice, self.grad, self.hess,
↪  self.batch_size, self.mt, self.vt, self.real_modifier,
↪  self.modifier_up, self.modifier_down,
↪  self.LEARNING_RATE, self.adam_epoch, self.beta1,
↪  self.beta2, not self.use_tanh, true_grads)
# coordinate_Newton(losses, indice, self.grad, self.hess,
↪  self.batch_size, self.mt, self.vt, self.real_modifier,
↪  self.modifier_up, self.modifier_down,
↪  self.LEARNING_RATE, self.adam_epoch, self.beta1,
↪  self.beta2, not self.use_tanh)
# coordinate_Newton_ADAM(losses, indice, self.grad,
↪  self.hess, self.batch_size, self.mt, self.vt,
↪  self.real_modifier, self.modifier_up,
↪  self.modifier_down, self.LEARNING_RATE, self.adam_epoch,
↪  self.beta1, self.beta2, not self.use_tanh)
self.solver(losses, indice, self.grad, self.hess,
↪  self.batch_size, self.mt, self.vt, self.real_modifier,
↪  self.modifier_up, self.modifier_down, self.LEARNING_RATE,
↪  self.adam_epoch, self.beta1, self.beta2, not
↪  self.use_tanh)
# adjust sample probability, sample around the points with
↪  large gradient
if self.save_ckpts:
```

```
            np.save('{}/iter{}'.format(self.save_ckpts, iteration),
             ↪  self.real_modifier)

        if self.real_modifier.shape[0] > self.resize_init_size:
            self.sample_prob = self.get_new_prob(self.real_modifier)
            # self.sample_prob = self.get_new_prob(tmp_mt.reshape(se
             ↪  lf.real_modifier.shape))
            self.sample_prob = self.sample_prob.reshape(var_size)

        # if the gradient is too small, do not optimize on this
         ↪  variable
        # self.var_list = np.delete(self.var_list,
         ↪  indice[np.abs(self.grad) < 5e-3])
        # reset the list every 10000 iterations
        # if iteration%200 == 0:
        #    print("{} variables remained at last
         ↪  stage".format(self.var_list.size))
        #    var_size = self.real_modifier.size
        #    self.var_list = np.array(range(0, var_size))
        return losses[0], l2s[0], loss1[0], loss2[0], scores[0],
         ↪  nimgs[0]
        # return losses[0]

    def attack(self, imgs, targets):
        """
        Perform the L_2 attack on the given images for the given
↪   targets.

        If self.targeted is true, then the targets represents the
↪   target labels.
        If self.targeted is false, then targets are the original
↪   class labels.
        """
        r = []
        print('go up to',len(imgs))
        # we can only run 1 image at a time, minibatches are used for
         ↪  gradient evaluation
        for i in range(0,len(imgs)):
            print('tick',i)
            r.extend(self.attack_batch(imgs[i], targets[i]))
```

```
    return np.array(r)

# only accepts 1 image at a time. Batch is used for gradient
↪  evaluations at different points
def attack_batch(self, img, lab):
    """
    Run the attack on a batch of images and labels.
    """
    def compare(x,y):
        if not isinstance(x, (float, int, np.int64)):
            x = np.copy(x)
            if self.TARGETED:
                x[y] -= self.CONFIDENCE
            else:
                x[y] += self.CONFIDENCE
            x = np.argmax(x)
        if self.TARGETED:
            return x == y
        else:
            return x != y

    # remove the extra batch dimension
    if len(img.shape) == 4:
        img = img[0]
    if len(lab.shape) == 2:
        lab = lab[0]
    # convert to tanh-space
    if self.use_tanh:
        img = np.arctanh(img*1.999999)

    # set the lower and upper bounds accordingly
    lower_bound = 0.0
    CONST = self.initial_const
    upper_bound = 1e10

    # convert img to float32 to avoid numba error
    img = img.astype(np.float32)

    # set the upper and lower bounds for the modifier
    if not self.use_tanh:
```

```
        self.modifier_up = 0.5 - img.reshape(-1)
        self.modifier_down = -0.5 - img.reshape(-1)

    # clear the modifier
    if not self.load_checkpoint:
        if self.use_resize:
            self.resize_img(self.resize_init_size,
             ↪  self.resize_init_size, True)
        else:
            self.real_modifier.fill(0.0)

    # the best l2, score, and image attack
    o_best_const = CONST
    o_bestl2 = 1e10
    o_bestscore = -1
    o_bestattack = img

    for outer_step in range(self.BINARY_SEARCH_STEPS):
        print(o_bestl2)

        bestl2 = 1e10
        bestscore = -1

        # The last iteration (if we run many steps) repeat the
         ↪  search once.
        if self.repeat == True and outer_step ==
         ↪  self.BINARY_SEARCH_STEPS-1:
            CONST = upper_bound

        # set the variables so that we don't have to send them
         ↪  over again
        self.sess.run(self.setup, {self.assign_timg: img,
                                   self.assign_tlab: lab,
                                   self.assign_const: CONST})

        # use the current best model
        # np.copyto(self.real_modifier, o_bestattack - img)
        # use the model left by last constant change

        prev = 1e6
```

```
train_timer = 0.0
last_loss1 = 1.0
if not self.load_checkpoint:
    if self.use_resize:
        self.resize_img(self.resize_init_size,
        ↪  self.resize_init_size, True)
    else:
        self.real_modifier.fill(0.0)
# reset ADAM status
self.mt.fill(0.0)
self.vt.fill(0.0)
self.adam_epoch.fill(1)
self.stage = 0
multiplier = 1
eval_costs = 0
if self.solver_name != "fake_zero":
    multiplier = 24
for iteration in range(self.start_iter,
↪  self.MAX_ITERATIONS):
    if self.use_resize:
        if iteration == 2000:
        # if iteration == 2000 // 24:
            self.resize_img(64,64)
        if iteration == 10000:
      # if iteration == 2000 // 24 + (10000 - 2000) // 96:
            self.resize_img(128,128)
        # if iteration == 200*30:
        # if iteration == 250 * multiplier:
        #     self.resize_img(256,256)
    # print out the losses every 10%
    if iteration%(self.print_every) == 0:
       # print(iteration,self.sess.run((self.loss,self.r
        ↪  eal,self.other,self.loss1,self.loss2),
        ↪  feed_dict={self.modifier:
        ↪  self.real_modifier}))
        loss, real, other, loss1, loss2 =
        ↪  self.sess.run((self.loss,self.real,self.other
        ↪  ,self.loss1,self.loss2),
        ↪  feed_dict={self.modifier: self.real_modifier})
```

```
            print("[STATS][L2] iter = {}, cost = {}, time =
             ↪ {:.3f}, size = {}, loss = {:.5g}, real =
             ↪ {:.5g}, other = {:.5g}, loss1 = {:.5g}, loss2
             ↪ = {:.5g}".format(iteration, eval_costs,
             ↪ train_timer, self.real_modifier.shape,
             ↪ loss[0], real[0], other[0], loss1[0],
             ↪ loss2[0]))
            sys.stdout.flush()
            # np.save('black_iter_{}'.format(iteration),
             ↪ self.real_modifier)

        attack_begin_time = time.time()
        # perform the attack
        if self.solver_name == "fake_zero":
            l, l2, loss1, loss2, score, nimg =
             ↪ self.fake_blackbox_optimizer()
        else:
            l, l2, loss1, loss2, score, nimg =
             ↪ self.blackbox_optimizer(iteration)
        # l = self.blackbox_optimizer(iteration)

        if self.solver_name == "fake_zero":
            eval_costs += np.prod(self.real_modifier.shape)
        else:
            eval_costs += self.batch_size

    # reset ADAM states when a valid example has been found
    if loss1 == 0.0 and last_loss1 != 0.0 and self.stage ==
     ↪ 0:
        # we have reached the fine tunning point
        # reset ADAM to avoid overshoot
        if self.reset_adam_after_found:
            self.mt.fill(0.0)
            self.vt.fill(0.0)
            self.adam_epoch.fill(1)
        self.stage = 1
    last_loss1 = loss1

    # check if we should abort search if we're getting
     ↪ nowhere.
```

```
        # if self.ABORT_EARLY and
        ↪  iteration%(self.MAX_ITERATIONS//10) == 0:
        if self.ABORT_EARLY and iteration %
        ↪  self.early_stop_iters == 0:
            if l > prev*.9999:
                print("Early stopping because there is no
                ↪  improvement")
                break
            prev = l

        # adjust the best result found so far
        # the best attack should have the target class with
        ↪  the largest value,
        # and has smallest l2 distance
        if l2 < bestl2 and compare(score, np.argmax(lab)):
            bestl2 = l2
            bestscore = np.argmax(score)
        if l2 < o_bestl2 and compare(score, np.argmax(lab)):
            # print a message if it is the first attack found
            if o_bestl2 == 1e10:
                print("[STATS][L3](First valid attack found!)
                ↪  iter = {}, cost = {}, time = {:.3f}, size
                ↪  = {}, loss = {:.5g}, loss1 = {:.5g}, loss2
                ↪  = {:.5g}, l2 = {:.5g}".format(iteration,
                ↪  eval_costs, train_timer,
                ↪  self.real_modifier.shape, l, loss1, loss2,
                ↪  l2))
                sys.stdout.flush()
            o_bestl2 = l2
            o_bestscore = np.argmax(score)
            o_bestattack = nimg
            o_best_const = CONST

        train_timer += time.time() - attack_begin_time

    # adjust the constant as needed
    if compare(bestscore, np.argmax(lab)) and bestscore != -1:
        # success, divide const by two
        print('old constant: ', CONST)
        upper_bound = min(upper_bound,CONST)
```

```
            if upper_bound < 1e9:
                CONST = (lower_bound + upper_bound)/2
            print('new constant: ', CONST)
        else:
            # failure, either multiply by 10 if no solution found
            ↪ yet
          #         or do binary search with the known upper bound
            print('old constant: ', CONST)
            lower_bound = max(lower_bound,CONST)
            if upper_bound < 1e9:
                CONST = (lower_bound + upper_bound)/2
            else:
                CONST *= 10
            print('new constant: ', CONST)

    # return the best solution found
    return o_bestattack, o_best_const
```

A.2 EAD Attack (EN)

```
## en_attack.py -- attack a network optimizing elastic-net distance
↪  with an en decision rule
##
import sys
import tensorflow as tf
import numpy as np

BINARY_SEARCH_STEPS = 9  # number of times to adjust the constant
↪  with binary search
MAX_ITERATIONS = 10000   # number of iterations to perform gradient
↪  descent
ABORT_EARLY = True       # if we stop improving, abort gradient
↪  descent early
LEARNING_RATE = 1e-2     # larger values converge faster to less
↪  accurate results
TARGETED = True          # should we target one specific class? or
↪  just be wrong?
CONFIDENCE = 0          # how strong the adversarial example should be
INITIAL_CONST = 1e-3     # the initial constant c to pick as a first
↪  guess
BETA = 1e-3              # Hyperparameter trading off L2 minimization
↪  for L1 minimization

class EADEN:
    def __init__(self, sess, model, batch_size=1, confidence =
    ↪  CONFIDENCE,
                 targeted = TARGETED, learning_rate = LEARNING_RATE,
                 binary_search_steps = BINARY_SEARCH_STEPS,
                  ↪  max_iterations = MAX_ITERATIONS,
                 abort_early = ABORT_EARLY,
                 initial_const = INITIAL_CONST, beta = BETA):
        """
        EAD with EN Decision Rule

        Returns adversarial examples for the supplied model.
        """
```

```
image_size, num_channels, num_labels = model.image_size,
↪  model.num_channels, model.num_labels
self.sess = sess
self.TARGETED = targeted
self.LEARNING_RATE = learning_rate
self.MAX_ITERATIONS = max_iterations
self.BINARY_SEARCH_STEPS = binary_search_steps
self.ABORT_EARLY = abort_early
self.CONFIDENCE = confidence
self.initial_const = initial_const
self.batch_size = batch_size
self.beta = beta
self.beta_t = tf.cast(self.beta, tf.float32)

self.repeat = binary_search_steps >= 10

shape = (batch_size,image_size,image_size,num_channels)

# these are variables to be more efficient in sending data to
↪  tf
self.timg = tf.Variable(np.zeros(shape), dtype=tf.float32)
self.newimg = tf.Variable(np.zeros(shape), dtype=tf.float32)
self.slack = tf.Variable(np.zeros(shape), dtype=tf.float32)
self.tlab = tf.Variable(np.zeros((batch_size,num_labels)),
↪  dtype=tf.float32)
self.const = tf.Variable(np.zeros(batch_size),
↪  dtype=tf.float32)

# and here's what we use to assign them
self.assign_timg = tf.placeholder(tf.float32, shape)
self.assign_newimg = tf.placeholder(tf.float32, shape)
self.assign_slack = tf.placeholder(tf.float32, shape)
self.assign_tlab = tf.placeholder(tf.float32,
↪  (batch_size,num_labels))
self.assign_const = tf.placeholder(tf.float32, [batch_size])

self.global_step = tf.Variable(0, trainable=False)
self.global_step_t = tf.cast(self.global_step, tf.float32)

"""Fast Iterative Soft Thresholding"""
```

```
"""--------------------------------"""
self.zt = tf.divide(self.global_step_t,
↪  self.global_step_t+tf.cast(3, tf.float32))

cond1 = tf.cast(tf.greater(tf.subtract(self.slack,
↪  self.timg),self.beta_t), tf.float32)
cond2 = tf.cast(tf.less_equal(tf.abs(tf.subtract(self.slack,s
↪  elf.timg)),self.beta_t),
↪  tf.float32)
cond3 = tf.cast(tf.less(tf.subtract(self.slack,
↪  self.timg),tf.negative(self.beta_t)), tf.float32)

upper = tf.minimum(tf.subtract(self.slack,self.beta_t),
↪  tf.cast(0.5, tf.float32))
lower = tf.maximum(tf.add(self.slack,self.beta_t),
↪  tf.cast(-0.5, tf.float32))

self.assign_newimg = tf.multiply(cond1,upper)+tf.multiply(con
↪  d2,self.timg)+tf.multiply(cond3,lower)
self.assign_slack = self.assign_newimg+tf.multiply(self.zt,
↪  self.assign_newimg-self.newimg)
self.setter = tf.assign(self.newimg, self.assign_newimg)
self.setter_y = tf.assign(self.slack, self.assign_slack)
"""--------------------------------"""
# prediction BEFORE-SOFTMAX of the model
self.output = model.predict(self.newimg)
self.output_y = model.predict(self.slack)

# distance to the input data
self.l2dist =
↪  tf.reduce_sum(tf.square(self.newimg-self.timg),[1,2,3])
self.l2dist_y =
↪  tf.reduce_sum(tf.square(self.slack-self.timg),[1,2,3])
self.l1dist =
↪  tf.reduce_sum(tf.abs(self.newimg-self.timg),[1,2,3])
self.l1dist_y =
↪  tf.reduce_sum(tf.abs(self.slack-self.timg),[1,2,3])
self.elasticdist = self.l2dist + tf.multiply(self.l1dist,
↪  self.beta_t)
```

```
self.elasticdist_y = self.l2dist_y +
↪  tf.multiply(self.l1dist_y, self.beta_t)

# compute the probability of the label class versus the
↪  maximum other
real = tf.reduce_sum((self.tlab)*self.output,1)
real_y = tf.reduce_sum((self.tlab)*self.output_y,1)
other = tf.reduce_max((1-self.tlab)*self.output -
↪  (self.tlab*10000),1)
other_y = tf.reduce_max((1-self.tlab)*self.output_y -
↪  (self.tlab*10000),1)
if self.TARGETED:
    # if targeted, optimize for making the other class most
    ↪  likely
    loss1 = tf.maximum(0.0, other-real+self.CONFIDENCE)
    loss1_y = tf.maximum(0.0, other_y-real_y+self.CONFIDENCE)
else:
    # if untargeted, optimize for making this class least
    ↪  likely.
    loss1 = tf.maximum(0.0, real-other+self.CONFIDENCE)
    loss1_y = tf.maximum(0.0, real_y-other_y+self.CONFIDENCE)

# sum up the losses
self.loss21 = tf.reduce_sum(self.l1dist)
self.loss21_y = tf.reduce_sum(self.l1dist_y)
self.loss2 = tf.reduce_sum(self.l2dist)
self.loss2_y = tf.reduce_sum(self.l2dist_y)
self.loss1 = tf.reduce_sum(self.const*loss1)
self.loss1_y = tf.reduce_sum(self.const*loss1_y)

self.loss_opt = self.loss1_y+self.loss2_y
self.loss =
↪  self.loss1+self.loss2+tf.multiply(self.beta_t,self.loss21)

self.learning_rate =
↪  tf.train.polynomial_decay(self.LEARNING_RATE,
↪  self.global_step, self.MAX_ITERATIONS, 0, power=0.5)
start_vars = set(x.name for x in tf.global_variables())
optimizer =
↪  tf.train.GradientDescentOptimizer(self.learning_rate)
```

```
        self.train = optimizer.minimize(self.loss_opt,
        ↪  var_list=[self.slack], global_step=self.global_step)
        end_vars = tf.global_variables()
        new_vars = [x for x in end_vars if x.name not in start_vars]

        # these are the variables to initialize when we run
        self.setup = []
        self.setup.append(self.timg.assign(self.assign_timg))
        self.setup.append(self.tlab.assign(self.assign_tlab))
        self.setup.append(self.const.assign(self.assign_const))

        self.init = tf.variables_initializer(var_list=[self.global_st⌋
        ↪  ep]+[self.slack]+[self.newimg]+new_vars)

    def attack(self, imgs, targets):
        """
        Perform the EAD attack on the given images for the given
↪   targets.

        If self.targeted is true, then the targets represents the
↪   target labels.
        If self.targeted is false, then targets are the original
↪   class labels.
        """
        r = []
        print('go up to',len(imgs))
        for i in range(0,len(imgs),self.batch_size):
            print('tick',i)
            r.extend(self.attack_batch(imgs[i:i+self.batch_size],
            ↪  targets[i:i+self.batch_size]))
        return np.array(r)

    def attack_batch(self, imgs, labs):
        """
        Run the attack on a batch of images and labels.
        """
        def compare(x,y):
            if not isinstance(x, (float, int, np.int64)):
                x = np.copy(x)
                if self.TARGETED:
```

```
                x[y] -= self.CONFIDENCE
            else:
                x[y] += self.CONFIDENCE
            x = np.argmax(x)
        if self.TARGETED:
            return x == y
        else:
            return x != y

    batch_size = self.batch_size

    # set the lower and upper bounds accordingly
    lower_bound = np.zeros(batch_size)
    CONST = np.ones(batch_size)*self.initial_const
    upper_bound = np.ones(batch_size)*1e10

    # the best l2, score, and image attack
    o_besten = [1e10]*batch_size
    o_bestscore = [-1]*batch_size
    o_bestattack = [np.zeros(imgs[0].shape)]*batch_size

    for outer_step in range(self.BINARY_SEARCH_STEPS):
        # completely reset adam's internal state.
        self.sess.run(self.init)
        batch = imgs[:batch_size]
        batchlab = labs[:batch_size]

        besten = [1e10]*batch_size
        bestscore = [-1]*batch_size

        # The last iteration (if we run many steps) repeat the
        ↪  search once.
        if self.repeat == True and outer_step ==
        ↪  self.BINARY_SEARCH_STEPS-1:
            CONST = upper_bound

        # set the variables so that we don't have to send them
        ↪  over again
        self.sess.run(self.setup, {self.assign_timg: batch,
                                   self.assign_tlab: batchlab,
```

```
                                    self.assign_const: CONST})
            self.sess.run(self.setter, feed_dict={self.assign_newimg:
            ↪  batch})
            self.sess.run(self.setter_y, feed_dict={self.assign_slack:
            ↪  batch})
            prev = 1e6
            for iteration in range(self.MAX_ITERATIONS):
                # perform the attack
                self.sess.run([self.train])
                self.sess.run([self.setter, self.setter_y])
                l, l2s, l1s, elastic, scores, nimg =
                ↪  self.sess.run([self.loss, self.l2dist,
                ↪  self.l1dist, self.elasticdist, self.output,
                ↪  self.newimg])



                # print out the losses every 10%
                """
                if iteration%(self.MAX_ITERATIONS//10) == 0:
                    print(iteration,self.sess.run((self.loss,self.lo
↪  ss1,self.loss2,self.loss21)))
                """
                # check if we should abort search if we're getting
                ↪  nowhere.
                if self.ABORT_EARLY and
                ↪  iteration%(self.MAX_ITERATIONS//10) == 0:
                    if l > prev*.9999:
                        break
                    prev = l

                # adjust the best result found so far
              for e,(en,sc,ii) in enumerate(zip(elastic,scores,nimg)):
                    if en < besten[e] and compare(sc,
                    ↪  np.argmax(batchlab[e])):
                        besten[e] = en
                        bestscore[e] = np.argmax(sc)
                    if en < o_besten[e] and compare(sc,
                    ↪  np.argmax(batchlab[e])):
                        o_besten[e] = en
```

```
                o_bestscore[e] = np.argmax(sc)
                o_bestattack[e] = ii

        # adjust the constant as needed
        for e in range(batch_size):
            if compare(bestscore[e], np.argmax(batchlab[e])) and
             ↪  bestscore[e] != -1:
                # success, divide const by two
                upper_bound[e] = min(upper_bound[e],CONST[e])
                if upper_bound[e] < 1e9:
                    CONST[e] = (lower_bound[e] + upper_bound[e])/2
            else:
                # failure, either multiply by 10 if no solution
                 ↪  found yet
                #          or do binary search with the known upper
                 ↪  bound
                lower_bound[e] = max(lower_bound[e],CONST[e])
                if upper_bound[e] < 1e9:
                    CONST[e] = (lower_bound[e] + upper_bound[e])/2
                else:
                    CONST[e] *= 10

    # return the best solution found
    o_besten = np.array(o_besten)
    return o_bestattack
```