**HOCHSCHULE**
**HANNOVER**
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
*Fakultät IV*
*Wirtschaft und*
*Informatik*

# Adversarial Examples in Deep Learning: Attacks, Defenses and Implementations

Lennard Heidrich

Bachelor-Thesis in the degree course "Media Design Computing"

June 25, 2021

**Author** Lennard Heidrich
1475283
lennard.heidrich@gmail.com

**First examiner:** Prof. Dr. Adrian Pigors
Department of Computer Science, Faculty IV I
Hochschule Hannover
adrian.pigors@hs-hannover.de

**Second examiner:** M.Sc. Jussi Salzwedel
Department of Computer Science, Faculty IV I
Hochschule Hannover
jussi.salzwedel@hs-hannover.de

## Declaration of Authorship

I hereby declare that this thesis, and the work presented in it are my own and has been generated by me as the result of my own original research. I confirm that:

- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Hanover, June 25, 2021 Signature

Abstract

Adversarial examples are malicious inputs to a machine learning model, crafted by attackers. In image recognition, they are indistinguishable from normal images, but they cause drastic prediction errors. This thesis focuses on introducing popular mechanisms to attack and defend deep neural networks and the implementation. In the beginning, the basics of machine learning and deep learning are introduced to make the readers understand, on what basis these attacks and defenses function. Further on, some of the most popular attack and defense mechanisms will be explained, followed by implementing a number of them in Python. Libraries which serve the purpose of attacking machine learning and deep learning models in Python language, are analysed and used to present the use cases and their performance.

# Contents

# 1 Introduction

Deep learning represents one of the most successful machine learning techniques today. Especially in computer vision, state-of-the-art Deep Learning based applications manage to deliver outstanding results in development branches such as image recognition, self-driving cars, and face recognition. Despite their groundbreaking results, these applications all seem to be susceptible to so-called "Adversarial examples". They represent images, which do not seem to be visually distinguished from the correctly classified originals. However, these images can cause big errors in machine- and deep learning applications. These errors can lead to devastating consequences in real-life Deep Learning applications. One of the most popular examples is the perturbed panda. By adding noise which is imperceptible to the human eye, the classification of an image changes from "panda" to "gibbon" with high confidence.

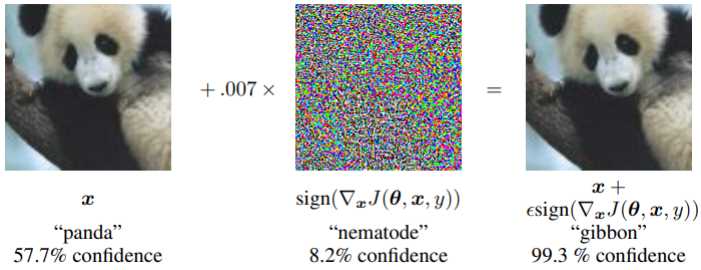



Figure 1.1: While the image is visually indifferent from the original, it is still being misclassified with high confidence.

The thesis provides an insight on how adversarial attack and defense mechanisms function. The basics of Machine Learning and Deep Learning are introduced as the basis of adversarial examples. Chapter 3 introduces some of the most popular attack and defense methods are presented. In chapter 4 a number of attack and defense methods are implemented with the help of selected Python libraries. For that purpose, a victim model is created and is subject to the attacks and defenses. Finally, the libraries are put into comparison to determine which one is best used for the task of attacking and defending the network.

# 2 Fundamentals of Machine Learning

## 2.1 Definition of Machine Learning

The overall task of artificial intelligence is to "automate intellectual tasks, normally performed by humans"[Cho17, P.4]. Machine learning is a part of artificial intelligence, or more generally, the "study of computer algorithms to improve through experience"[Machine Learning, Tom Mitchell 1997].

The emphasis lies on experience, which distinguishes machine learning from common hard-coded programming. T.M Mitchell described the problem as "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" [Mitchell, T. M.: Machine Learning. McGraw-Hill, New York, 1997].

## 2.2 Regression and Classification

The basic type of machine learning is logistic regression, where a statistical model outputs the probability of whether the input belongs to a certain discrete class [Gér17, p. 32]. There is also linear regression, where a model predicts a continuous value. A typical example would be the prediction of housing prices, where the price is determined by variables like location of the neighborhood, number of rooms, etc. Regression tasks mainly predict numerical values based on the task at hand. Classification, however, predicts discrete labels or classes[Cho17, P. 85]. After the computer has finished learning, it will regress or classify any new input, i.e., answers to questions based on what it has learned from the variable relationships. The final output is, therefore, a statistical model which is trained to generalize and to classify any new input. It is important to emphasize on generalization since a model that does not generalize is merely copying the input data[Cho17, P. 95].

To predict values, a machine learning model is trying to convert its training inputs into meaningful representations [Cho17, P. 6]. For instance, a machine learning model takes $x$ and $y$ coordinates as input and predicts whether the point is black or white based on it's position in the coordinate system(Fig. 2.1). A machine learning model attempts to

find a representation of the data which is meaningful for the classification of that point. [Cho17, P. 7]

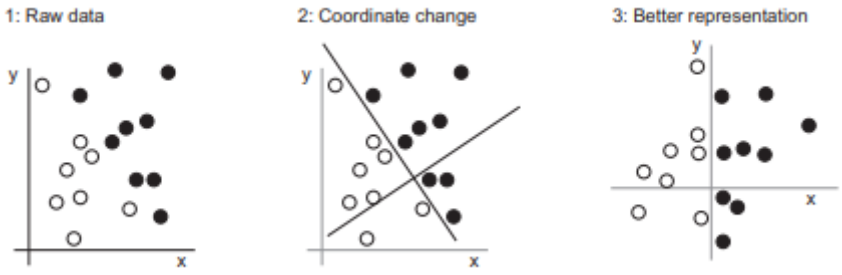



Figure 2.1: Classification of points [Cho17, P.7]

In the case of image classification, for example, a set of input data consists of images with corresponding labels/ targets of what each image is supposed to represent. The computer uses that data to learn the patterns and to output rules on how to map any unknown input to a target[Cho17, P.5].

### 2.2.1 Classifying an image

Classification tasks like in 2.1, use a set of variables $x$ to determine the class of the input. Each variable can be seen as an axis of a coordinate system and the values of these variables make up the coordinates. In image classification however, these variables $x$ are the pixels that make up an image. With each vector of pixels comes a corresponding vector of labels which represents the class of the image. Like the previous example, the model then learns meaningful representations on how to classify these inputs by drawing decision boundaries (see 2.1). More on image classification in the Convolutional Neural Networks section. This topic is further emphasised in chapter 4 regarding changing the pixels of an image to push them over the decision boundaries.

## 2.3 Supervised/ Unsupervised Learning

Supervised learning is the most commonly used method of learning among machine learning algorithms and is described, as a set of questions mapped to the respective answers, hence the name supervised [Cho17, P. 5]. Each input vector $x$ is labeled with a corresponding target $y$, that represents the class of $x$ its corresponding value. By labeling the dataset the algorithm is trained to create a statistical model. This statistical model

is then capable of mapping any unknown set of input to a certain class or predicting a numerical value based on the input. The previous example, which classified points as either black or white based on their position in the coordinate system, is a typical case of supervised learning (see 2.1). One of the most used applications of supervised learning is image classification, which is described in detail in 2.6.2.

Unsupervised learning does not rely on a labeled dataset. The algorithm is merely supplied with unlabelled data and it will figure out by itself, how to group it [Gér17, P. 33].

## 2.4 Evaluation

To train, evaluate, and test a machine learning model, three different datasets are required.

Training set: This is the data that the machine learning algorithm uses to learn about the data and is therefore used in the learning process.[Gér17, P. 61]. Since training requires the most data, the training set makes up for the majority of total data used.

Validation set: Each machine algorithm uses hyperparameters to configure the learning process. The validation set serves the purpose of testing out different values for these hyperparameters to find the best configuration and [Cho17, P. 97]

Testing set: Consist of a smaller fraction of the original dataset. Its purpose is to be fed into the trained model after the learning process is completed. Since this dataset is different from the training data and thus unknown to the algorithm, it must figure out itself based on what it has learned, how to predict on this new data [Cho17, P.97].

## 2.5 Overfitting

If a machine learning model performs well on training data but fails to generalize well on new data, it is overfitting [Gér17, p. 57]. The goal of a model is to generalize [Cho17, P.97]. Overfitting happens when the model is too complex in comparison to the noise and variation in the training data [Gér17, P. 58]. A model tends to therefore overfit when there is not enough data to train on. Gathering or generating more data should therefore solve the issue. Another option would be to simplify the model and its parameters [Gér17, P.57].

## 2.6 Deep learning

Deep learning is a different approach to machine learning (F. 2.2) inspired by the neural structure of the human brain [Gér17, P. 359]. The learning process is being realized by letting the algorithm learn through several, successive layers of neurons that compute different representations of the data. Each layer's output serves as the input of the next layer, therefore are all layers based on another. With each layer, the way the data is being represented becomes more important and more meaningful regarding the final prediction[Cho17, P. 8].

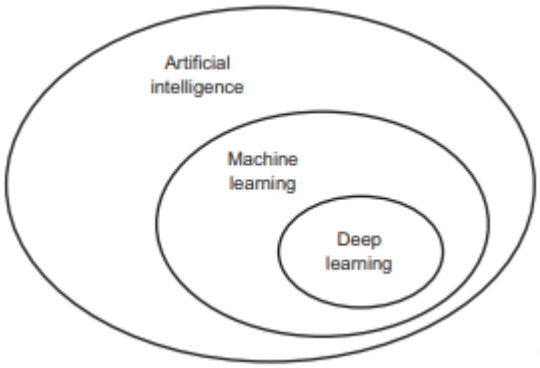



Figure 2.2: Deep learning in the context of machine learning and artificial intelligence

For example, if a model was trained to recognize handwritten digits, early layers in the network would deliver very general results about what number is being fed into the algorithm, while later layers would have more precise information about which number it is looking at (see 2.3) [Cho17, P.9]. These could be parts of a digit that are unique among all possible inputs. This uniqueness helps to make the final prediction of what number the initial input represents.

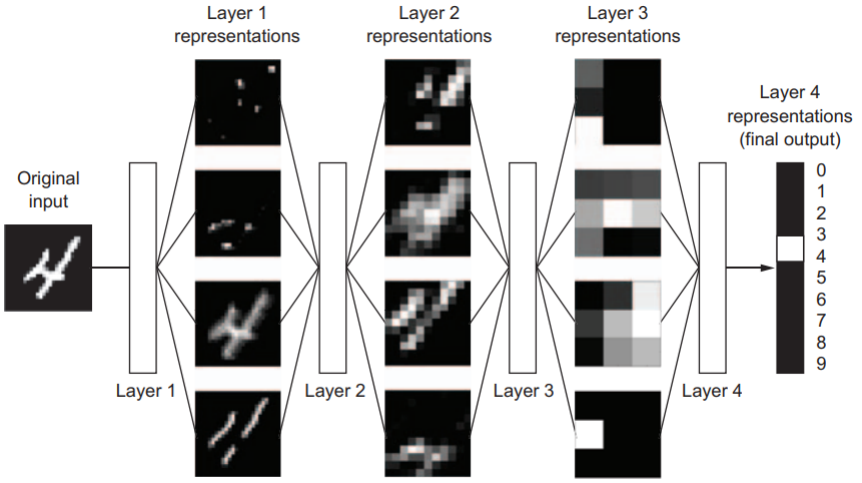



Figure 2.3: Representations becoming more and more meaningful for the classification task. [Cho17, P.9]

Deep learning is based on artificial neural networks [Gér17, P. 359]. Artificial neural networks start with some input that is being processed through the network and finally deliver an output. The most popular kind, the "feedforward neural network", processes the input forward through a stack of successive layers. In this method, the network requires the entire sequence of the input at once, and the length of the input is fixed while the input only flows forward through the network when processed[Cho17, P. 196]. A popular alternative is a recurrent network, which also has connections pointing backward to the input being able to vary in its length [Gér17, P. 648].

A setup that is heavily used today is the Multilayer Perceptron[Gér17, P. 359] since it covers a broad range of tasks that can be completed with its help. It consists of three categories of layers: input layer, hidden layers, and output layer. All layers within the network consist of neurons and are all connected through so-called weights [Gér17, P. 371]. Neurons are functions that output certain numerical values and are influenced by weights [Gér17, P. 365]. The unique feature of the MLP is that the neurons are densely connected through weights, meaning that each neuron of a layer is connected to all other neurons of the previous and following layer by weights(see 2.4). Except for the input layer, the value of each neuron of a layer is computed by calculating a weighted sum of the neurons of the previous layer and the weights connecting them to the current layer [Gér17, P.366]. Additionally, a bias b is being added to the calculation, to regulate the output. An **activation function** is applied to that weighted sum to determine how the output of a neuron is calculated [Cho17, P. 46] and to introduce some nonlinearity into the problem. The final output of a neuron is therefore called **activation**.

$$\text{ReLU}((x_1 w_1) + ... + (x_n w_n)... + b)$$

Without non-linearity, a chain of these linear transformations will result in just another linear transformation and there would be no difference between a single or a very deep stack of layers [Cho17, P. 72]. To solve this problem, activation functions are used. For hidden layers, one of the most popular functions is the Rectified Linear Unit (ReLU) function which maps any negative value to 0 while any positive value remains[Cho17, P. 72], [Gér17, p. 3764]. $x$ describes the input vector and $w$ describes a vector of weights. The MLP is a typical case of a feedforward architecture.

The input layer is the raw input to the network. How many neurons this layer has, depends on how many input values are fed into the network. For example, feeding a 3x3 grayscale image into a network would take 9 neurons in the input layer with each neuron carrying a brightness value.

Followed by the input layer comes the main configuration of the network, the hidden layers. The number of hidden layers and their size is not reliant on the input [Gér17, P. 371]. The weights are initially just random values that are being adjusted with each learning iteration [Cho17, P. 46]. These hidden layers often use ReLU as their activation function.

Finally, the output layer classifies the input by considering all weighted calculations. The output layer consists of several neurons, each representing a possible class. The activation function of the last layer in a neural network can vary between tasks. Sigmoid can be used for instance, to output a probability score between 0 and 1[Cho17, P. 71]. Softmax can be used to output a vector containing the probabilities for all classes [Gér17, P. 378]. After the output layer has been reached, a forward pass has been completed [Gér17, P. 373].

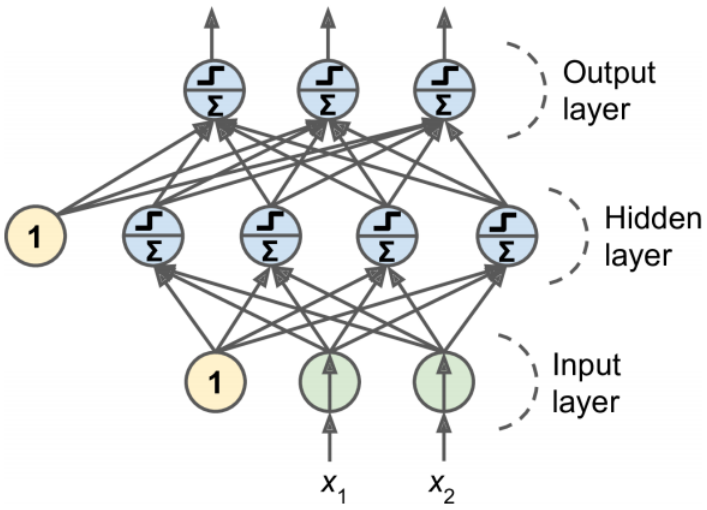



Figure 2.4: MLP architecture [Gér17, P. 371]

### 2.6.1 Loss Functions and Backpropagation

**Loss functions** are used at the end of a forward pass during the training as a feedback signal [Cho17, P. 22]. They calculate a distance score which measures how far off the model's prediction is from the true target. The lower the total cost, the higher the accuracy of the model. Which cost function to chose depends on the problem which is to solve[Cho17, P. 114].

- Binary cross-entropy: Binary classification, multiclass/ multilabel classification of a linear regression between 0 and 1. Used when sigmoid as output activation function.
- Multiclass/ single-label classification: Categorical cross-entropy. Used when softmax as output activation function.
- Regression: mean squared error (mse) . Most popular for regression problems [Cho17, P. 114].

These cost functions are being calculated for every batch of testing data that is being fed into the network. For each batch, the weights of the network are being tweaked a little, which changes the activations and lowers that cost. A method called **gradient descent** makes it possible to minimize the cost of a network by tweaking the weights of the network in a way that the function moves towards a minimum with each iteration [Gér17, P. 166]. This is achieved by calculating the gradient of a differentiable (loss)function $f$ [Cho17, P. 48]. To compute a gradient vector, all the partial derivatives of $f$ have to be calculated and put into a single vector [Gér17, P. 178]. A partial derivative is calculated for every dimension.

$$\nabla f(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x_1, x_2, \ldots, x_n) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x_1, x_2, \ldots, x_n) \end{bmatrix}$$

This vector finally gives information about the direction of the steepest ascent [Gér17, P. 177]. Since the loss is to be minimized, it should move towards a minimum, hence in the opposite direction(see 2.5). Initially, all weights are initialized randomly but with each iteration of gradient descent, the weights are tweaked a little so that the loss is being minimized[Gér17, p. 173],which gives them meaning regarding the problem. The step size, also called learning rate, determines how far gradient descent moves towards a minimum with each iteration. A step size too small could cause a very long learning process with many iterations, while a step size too large could cause the gradient to

descend to miss the minimum (Figure 2.5) [Gér17, P. 175]. If the gradient is 0, the local minimum has been found.

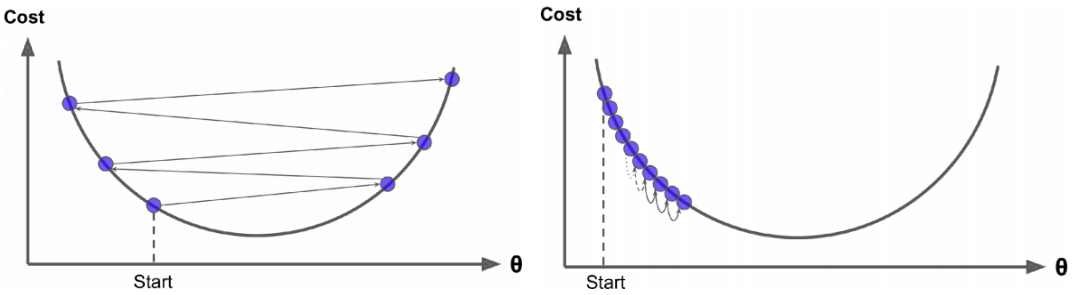



Figure 2.5: On the left, a step size too small and on the right a step size too large. [Gér17, P. 174]

Now, calculating all the derivatives for the gradient of every learning iteration and every input dimension is very time-consuming, therefore a more efficient way has to be introduced, called **backpropagation**. With the help of that algorithm, it is possible to automatically calculate the gradient concerning all the weights in the network in just one forward and one backward pass.[Gér17, P. 372] After the forward pass is completed, the algorithm measures, how much each of the output neurons contributed to the loss by applying a mathematical concept called chain rule [Cho17, P. 51]. Once finished, a gradient descend step towards the minimum is being performed, taking all the error values calculated by the backpropagation into consideration and tweaking the weights, respectively [Gér17, P. 373]. The automatic computation of gradients is also referred to as automatic differentiation or autodiff.

### 2.6.2 Convolutional Neural Networks

CNN's are key methods used in computer vision tasks involving artificial intelligence. Their advantage is, that they pick up on regional patterns of an image, while conventional neural networks only learn global patterns [Cho17, P. 122]. A neural network that classifies images has to therefore not only learn the features of the object that has to be recognized, but also their position in the image itself. This could lead to a misclassification just because the object was not at the position where the neural net expected it to be. Convolutional neural networks solve that issue.

The most important part of the network are the convolutional layers. Other than in densely connected networks, not every neuron is connected to every pixel in the input

image, which makes it possible to even feed large images into a network[Gér17, P. 580]. To implement that, a small filter is iterating over an image, connecting all pixels of the current step to one neuron of the following layer which downsamples the original image(see 2.6). The procedure is repeated for the following convolutional layers. How many pixels this filter travels with every iteration is determined by the stride value [Gér17, P. 582].

These filters can be seen as matrices, that contain numerical values. A filter that is supposed to pick up on the pattern of a diagonal line, for example, would have a 1 across its diagonal, while the other values remain 0 [Gér17, P. 583]. Previously, weights were described as scalar values that connect the neurons which influence the output. Convolutional neural networks use filters as sets of weights to connect neurons, with each value in a filter being considered a single weight [Gér17, P. 587]. For each iteration, it takes the filter to slide over a layer (see 2.7), a dot product of the filter and neurons is calculated. Like before, an activation function is applied to each dotproduct. All activations of a filter are written into a final matrix called a **feature map** [Gér17, P. 583]. This feature map, therefore, contains all activations calculated by a single filter. The following layer then contains a down-sampled version of the previous layer which has significant features based on the filters used (see 2.6).

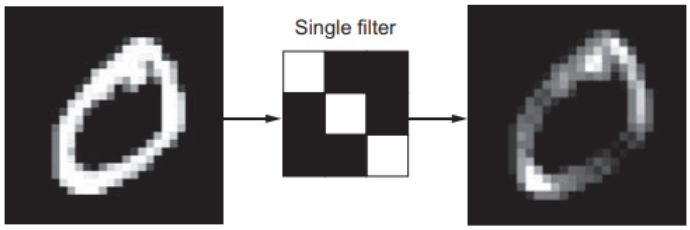



Figure 2.6: The resulting feature map after applying a vertical filter on the original image [Cho17, P. 147]

The mentioned filters contain high-level information in the beginning, like a diagonal line for instance. Filters learn patterns, which the algorithm is supposed to pick up on but ignore everything else besides that particular pattern (see 2.3)[Gér17, P. 583]. Initially, filters only represent very basic patterns like a diagonal line for example, but become more meaningful in layers located deep into the network[Cho17, P. 123]. When a convolutional neural network is supposed to classify images of cats and dogs, for instance, filters would initially only represent very basic features while deep layers would represent features like cat ears.

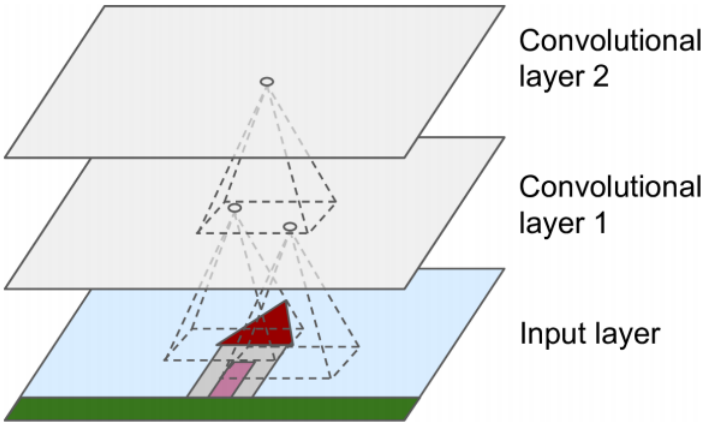



Figure 2.7: A filter iterating over different layers of a convolutional neural network. For each iteration, an activation is calculated which determines the value of a neuron in the following layer.[Gér17, P. 581]

Layers do not just output a single feature map per layer, but one for each filter used in a layer (see 2.8). The number of filters for each layer is custom. For each filter used, one feature map is created resulting in a stack of feature maps being the output of a layer[Gér17, P. 584]. Multiple trainable filters can be applied in a single stack of feature maps which makes it possible to detect more features[[Gér17, P. 585]. During the learning process, the model learns automatically, which filters are most useful, so they do not have to be set manually[Gér17, P. 584]. Though neurons, weight parameters, biases, and output differ between convolutional layers, the dimensions of the filters across all feature maps within a layer remain the same (Fig. 2.8). Since all neurons in a feature map use the same parameters (the same filter), the unique parameters of a model drastically decrease [Gér17, P. 585]. Images are usually not just black and white but use multiple channels for color representation. Initially, the input consists of 3 layers in the case of an RGB image but later are equal to the number of filters used in a layer [Cho17, P. 123].

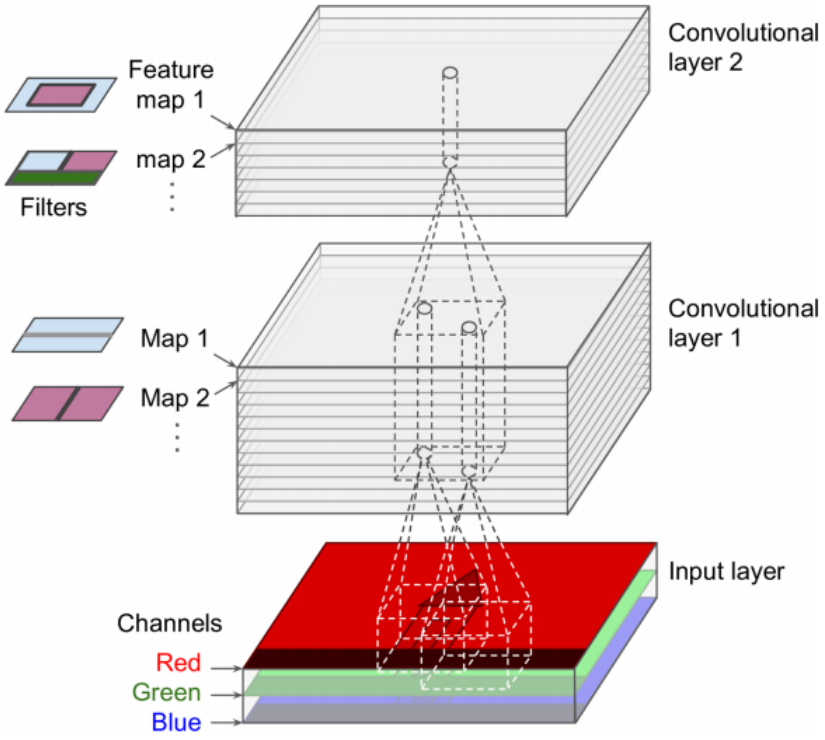



Figure 2.8: Multiple feature maps being extracted from a RGB image by 2 convolutional layers. Each layer takes the 3-Dimensional stack of feature maps as its next input. [Gér17, P. 586].

### 2.6.3 Pooling

The pooling operation is used to drastically downsample feature maps, to reduce the parameters and to make the network look at increasingly larger images. Other than with filters, the pooling operation does not calculate weighted sums but rather calculates a certain value from what a receptive field is observing[Cho17, P. 127]. For example, a receptive field is iterating over a layer, only saving the maximum value of every iteration to be saved in the resulting feature map. This particular case is called max-pooling[Cho17, P. 128]. Alternatives would be min- and average-pooling though max-pooling is especially useful since it preserves the most significant features of an image. Oftentimes, the receptive field of a pooling operation has 2x2 dimensions and iterates with a stride of 2, essentially downsampling the image by half.

### 2.6.4 The Architecture of a CNN

After the input image has been passed through several convolutional and pooling layers, it meets a couple of regular, densely connected layers like in a normal neural network. To pass a 3-dimensional output feature map into a regular layer, it needs to be flattened into a 1-dimensional vector. After the feature map has been flattened, it will be passed to the dense layer, followed by possibly more dense layers and finally a regular output layer that delivers, for example, a class probability [Cho17, P. 121].

## 2.7 Deep Learning Frameworks

The frameworks which are about to be introduced, use tensors at their very core to implement machine and deep learning tasks. Tensors are multidimensional arrays, which carry values of a certain datatype. These tensors are the most used datatype for machine learning problems [Cho17, P. 31]. For instance, a set of 10000 training images with image dimensions of 28x28, is converted into a tensor of (10000,28,28). Each image would then be a (28,28) tensor containing numerical values between 0 and 255. A 0-dimensional tensor is called a scalar, 1-dimensional, a vector, 2-dimensional matrix, and anything above just 3- or higher-dimensional[Cho17, p. 32]. Tensors have the following key attributes:

The number of axes: Also called the rank of a tensor. A 2-dimensional vector has 2 axes, a 3-dimensional has 3 axes, etc.

Shape: Describes how many dimensions the tensor has along each axis. For example, the tensor

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 2 & 3 \end{bmatrix}$$

has a shape of (3,3).

Datatype: All the values in a tensor have the same data type. Most commonly, float32, uint8, or float64[FC, P.32].

To create tensors, the Python library NumPy is used. It simplifies operations for complicated matrix and array operations.

Listing 2.1: Create a tensor with numpy

```
import numpy as np
x = np.array([[[1, 2, 3, 4, 5],
               [1, 2, 3, 4, 5],
               [1, 2, 3, 4, 5]],

              [[1, 2, 3, 4, 5],
               [1, 2, 3, 4, 5],
               [1, 2, 3, 4, 5]]])

Shape: (2,3,5)
```

### 2.7.1 Keras and TensorFlow

Tensorflow [Mar+15] is a symbolic math library that executes tensor manipulations to implement machine learning and deep learning functions. Its main purpose in deep learning is to implement solutions for auto differentiation. TensorFlow is used for research and production at Google and was developed by Google Brain.

Keras [Cho+15] primarily serves as an interface for an optimized tensor library in the background. Keras, therefore, does not deal with complicated tensor operations and calculations but wraps all of the underlying frameworks functions into a user-friendly API which makes it possible to create and train neural networks or CNN's in just a few lines of code. Keras used to be able to serve as an interface on top of many deep learning frameworks, but only TensorFlow is supported as of version 2.4.

### 2.7.2 PyTorch

PyTorch is an open source Deep Learning library which is based on the Torch library which is used for implementing deep learning algorithms. PyTorch is mainly used for tasks in computer vision and natural language processing. It is is mainly developed by the Facebook AI research lab. Although it mostly runs on Python, PyTorch also offers a C++ interface.

### 2.7.3 JAX

JAX [Bra+18] is Autograd and XLA brought together. Autograd is used for implementing backpropagation and gradient-based optimization. XLA, short for accelerated linear algebra is a compiler which can accelerate the performance of TensorFlow models without any source-code changes. JAX brings both libraries together by using XLA to

compile NumPy scripts which implement Autograd based functions. It was developed by Google to speed up research and to enable high performance machine learning.

# 3 Adversarial Examples

## 3.1 Introduction

Adversarial examples are created by an attacker to use as malicious input for a machine learning model, which causes misclassification. These inputs have been intentionally manipulated to cause prediction errors. Although this thesis is only going to focus on adversarial examples in computer vision, they are also a prominent topic in natural language processing and other fields of machine learning.

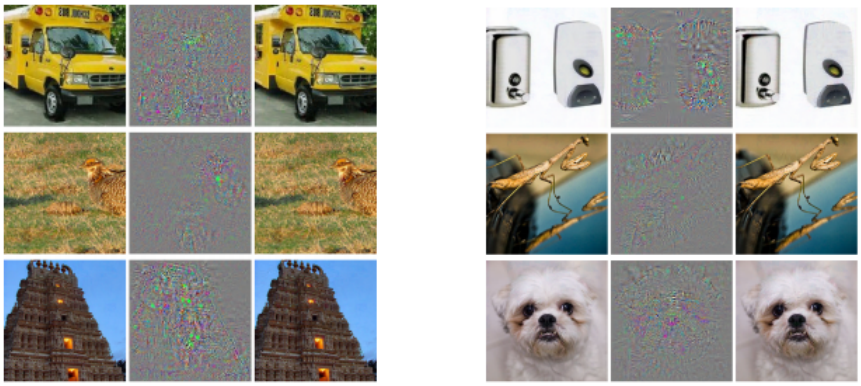


Figure 3.1: Original Images on the left, adversarial noise in the middle and the incorrectly classified image on the right [Sze+14, p. 6]

In computer vision, adversarial examples are created by adding a carefully calculated perturbation to an image that is specifically directed towards making a model misclassify the input. [GSS15, P. 1] These images look visually indifferent to the human eye from the original sample but lead to severe misclassification of the model. Images that were previously classified correctly by the model, are missing the correct target by a huge margin just by adding a small, imperceptible perturbation. Especially when it comes to real-world problems like automated driving, they pose a considerable threat that could lead to serious consequences [Yua+19, P. 1]. For example, the model could misclassify a stop sign in a way that it is going to be ignored, causing potential harm to the driver. In the following, several methods are being introduced to craft adversarial examples while

most of them follow the same goal: Calculate a perturbation which after being added to an image, leads to the loss of a network to be maximized. The perturbation is to be optimized in a way that it is as small and imperceptible as possible. Just any random perturbation added to an image would easily be detectable by humans, consequently, making adversarial examples trivial. By attempting to make the perturbation as small as possible while maintaining the goal of misclassification, this becomes an optimization problem.

### 3.1.1 Terminologies

For clarification, the terminology inspired by [Yua+19, P. 2] will be explained beforehand to make the following attack strategies easier to understand.

- $x$ : A clean input image without any perturbations.
- $r$ : Calculated perturbation.
- $x'$ : $x + r$.
- $l$ : Original label assigned to $x$
- $\theta$ : Parameters used in a deep learning model.
- $l'$ : Targeted class. The probability of that class is to be maximized with respect to x for incorrect classification.
- $f$ : Classifier that applies labels to any input $x$. A deep learning model in this case.
- $J$ : Cost to train a neural network.

With the explained terms, the problem above is more formally described as

$$\begin{aligned} &\text{minimize} \quad ||x - x'|| \\ s.t \quad &f(x') = l' \\ &f(x) = l \\ &l \neq l' \\ &x \in [0, 1] \end{aligned}$$

$||x - x'||$ denotes the distance between both images. The change of pixels has to be within some viable spectrum to be recognized. Usually, these values lie in between 0 – 255 but are often normalized to lie within 0-1. To measure the distance between the images in the input space, different vector norms are used [Yua+19, P. 3-4].

In the case of an **untargeted attack**, the problem is defined as

$$f(x) \neq f(x')$$

meaning that the output of the classifier for the clean and adversarial image has to be different. In comparison, a **targeted attack**, defined as

$$f(x) \ = l'$$

lays focus on changing the image in a way, that is being recognized as a chosen class.

In a **white box** scenario, the attacker knows and has access to all information of the model like the model architecture, training/ testing sets, hyperparameters, weights, activation functions, etc. This scenario is least likely to represent a real-world problem since most of the time, the attacker is a standard user without access to the required information.[AM18, P. 3]

A **black box** scenario represents the most realistic case of an attacker trying to attack a model. It is assumed, that the attack does not know the network or the training data. The only action the adversary can take is to observe how labels are being assigned to the input that they provided for the network themselves. This is a typical scenario where the attacker is a standard user who only has access to some interface but cannot access any backend functions making it more realistic than a white box scenario. [AM18, P. 2]

**Transferability** refers to the ability of adversarial examples, which were created based on the parameters of a certain classifier, to fool a classifier with different parameters and structures. These adversarial examples would then be universal for different model architectures [AM18, P. 3, Section 2].

### 3.1.2 Vectornorms

To calculate distances between datapoints, different vectornorms are used. These distances are important when the similarity between images is to be measured.

$\ell^2$ norm is also called Euclidean norm. The $\ell^2$ norm of a vector $||x||_2$ returns a positive value, describing its distance to the origin of the coordinate system. It is defined by:

$$x = (x_1, ..., x_n)$$
$$||x||_2 = \sqrt{x_1^2 + ... + x_n^2}$$

The $\ell^2$ distance $||x - y||_2$ of 2 vectors, however, describes the distance between them. It describes the shortest possible distance between them.

$$d(x, y) = ||x - y||_2$$

$\ell^0$ counts the number of non-zero elements within a vector. Taking the $\ell^0$ distance of 2 vectors, returns the number of dimensions in which both vectors do not contain equal values. The $\ell^0$ distance of the vectors (0,0) and (1,1) equals 2 because none of the dimensions match. [Yua+19, P. 5]

$$||x||_0 = |x_1|^0 \ + \ |x_2|^0 + \ ... \ + |x_n|^0$$

$\ell_\infty$ also called maximum norm, calculates the length of a vector by taking the largest absolute value within the vector. The $\ell_\infty$ distance can then be calculated with $d(x, y) = ||x - y||_\infty$.

$$||x||_\infty = \max\{|x_1|, ..., |x_n|\}$$

## 3.2 Finding Adversarial Examples

To find adversarial examples, the following formula describes the optimization problem at hand. An image $x$ from a data distribution $P_x$ is to be found, that maximizes the loss of the network in regards to $x$ [Mad+18, P. 3]. The procedure can also be called gradient ascent [Cho17, P. 326], since the loss is to be maximized other than with the counterpart, gradient descent which is used for training [Han20]. $P_x$ can be seen as a constraint which determines how distinguished the sought adversarial example is allowed to look [Mad+18, P. 3].

$$x' = \text{argmax}_{x \in P_x} J(\theta, x, l)$$

Adding any random noise to the image could eventually push the image towards a decision boundary but is likely not going to be an adversarial example, because the noise would be clearly perceptible (see 3.2) [Car19c]. It is therefore attempted to find a minimum perturbation, such that a decision boundary is crossed. Gradient ascent moves the image in a nonrandom direction such that the loss is maximized and the image crosses that boundary. To minimize the perturbation, $P_x$ is used as a contraint for how different the adversarial example is allowed to look and represents a set of allowed perturbations around $x$ [Mad+18, P. 3].

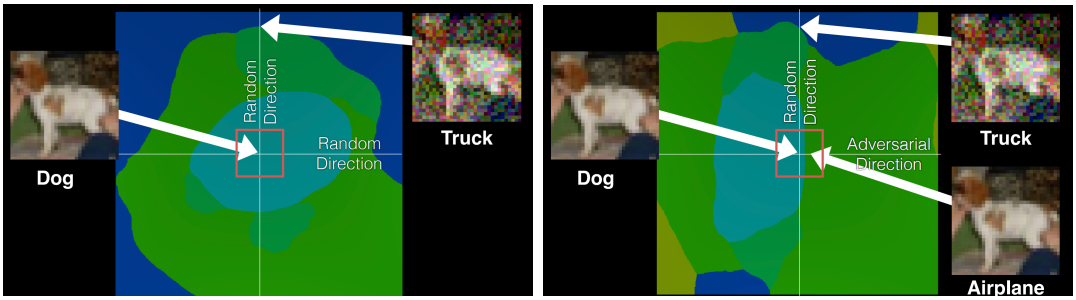



Figure 3.2: The classification of one image. Colors determine the classes and decision boundaries of the neural network.1. Image: Random noise changing the classification of the image from "Dog" to "Truck". 2. image: By going into the adversarial direction, the classification is changed and the perturbation is imperceptible. The red square can be interpreted as the radius $P_x$. Almost half of the space around $x$ is adversarial. [Car19b]

## 3.3 White-box attacks

### 3.3.1 L-BFGS Method

Szegedy et al. first presented the use of adversarial examples against neural networks in their 2014 paper "Intriguing properties of neural networks" [Sze+14], where they were the first ones to use adversarial examples to fool neural networks. By calculating a perturbation which is later added to the image, they managed to fool many state-of-the-art neural networks. To calculate the perturbation, the prediction error of the model on a particular input has to be maximized.

The assumption existed that a small perturbation $r$ added to an image $x$ should not change the classification of an image. By the concept of local generalization, an image that looks similar to $x$, should have the same class, since both images are visually similar to another. Adversarial examples proved, however, that the concept of local generalization does not work in all cases of noise or perturbations [Sze+14, P. 4].

The hypothesis at the time was, that adversarial examples do not reside within the radius around an image $x$, even though they are looking alike but reside all together somewhere within the input space, not necessarily close to the original sample. That makes deformations to the input of a model, that were originally used to improve the model's robustness, inefficient since all deformed inputs would merely lie close to the original training sample but not work against adversarial examples . These deformations would merely confirm the concept of local generalization[Sze+14, P. 5].

The perturbation $r$ is to be optimized such that the distance $||x - x'||_2$ between the original and the perturbed image is as small as possible while maintaining misclassification. Since $||r||_2$ is equal to $||x - x'||_2$, the distance can be minimized by minimizing $||r||_2$.

$$\text{minimize} \quad c\,||r||_2 + J(x + r, l) \quad \text{such that } x + r \in [0, 1]^m$$

For the approximation, the L-BFGS algorithm is used. The name of this method comes from the Broyden-Fletcher-Goldfarb-Shanno(BFGS) algorithm which is used for minimization in this case. By performing line-search, this algorithm runs until a minimum $c > 0$ has been found that satisfies $f(x + r) = l$. [Sze+14, P. 5]

Conducted experiments show the error induced by adversarial examples on different network architectures (see 3.3). The last line of the table in figure 3.3 shows the minimum distortion required to reach 0% prediction accuracy on the corresponding training data for the network the adversarial examples were based on. For each network, the table shows, the error caused on other network architectures based on its adversarial examples. [Sze+14, P. 7]

| | FC10($10^{-4}$) | FC10($10^{-2}$) | FC10(1) | FC100-100-10 | FC200-200-10 | AE400-10 | Av. distortion |
|---|---|---|---|---|---|---|---|
| FC10($10^{-4}$) | 100% | 11.7% | 22.7% | 2% | 3.9% | 2.7% | 0.062 |
| FC10($10^{-2}$) | 87.1% | 100% | 35.2% | 35.9% | 27.3% | 9.8% | 0.1 |
| FC10(1) | 71.9% | 76.2% | 100% | 48.1% | 47% | 34.4% | 0.14 |
| FC100-100-10 | 28.9% | 13.7% | 21.1% | 100% | 6.6% | 2% | 0.058 |
| FC200-200-10 | 38.2% | 14% | 23.8% | 20.3% | 100% | 2.7% | 0.065 |
| AE400-10 | 23.4% | 16% | 24.8% | 9.4% | 6.6% | 100% | 0.086 |
| Gaussian noise, stddev=0.1 | 5.0% | 10.1% | 18.3% | 0% | 0% | 0.8% | 0.1 |
| Gaussian noise, stddev=0.3 | 15.6% | 11.3% | 22.7% | 5% | 4.3% | 3.1% | 0.3 |

Figure 3.3: The effect of a network's adversarial examples on other networks [Sze+14, P. 7]

The results have proven that, even though the adversarial examples were created based on different architectures, they generalize well on other models too which makes them transferable. Although their discovery and their strong results, the origin and cause of adversarial examples were not found [GSS15, P. 1 ].

### 3.3.2 Fast Gradient Sign Method (FGSM)

Goodfellow et al. proposed in their 2014 paper "Explaining and Harnessing Adversarial Examples" that neural networks are especially vulnerable to adversarial examples due to their linear characteristics. Networks that use activation functions like ReLU, which work on a linear basis for every value greater than 0, are therefore somewhat vulnerable to adversarial examples [GSS15, P. 2]. The fast gradient sign method successfully uses the gradient of a models cost function to cause a linearization of the problem [GSS15, P. 3 ]. To execute the fast gradient method, the model parameters $\theta$ (i.e., weights), the input $x$ with its corresponding true labels $y$, and the loss of the network $J(\theta, x, l)$ are required to compute a perturbation $r$.

To compute the perturbation $r$, the parameter $\epsilon$ is being multiplied by the sign, formally described as:

$$sign(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

of the cost functions gradient $\nabla J$ to linearize $J$.[GSS15, P. 3 ].

$$r = \epsilon \cdot sign(\nabla_x J(\theta, x, l))$$

$\epsilon$ describes the magnitude of the calculated perturbation and often corresponds to the smallest pixel value in an image since a perturbation so small that it is not being acknowledged in a spectrum between 1-255(in most cases), would not have any effect on the classification [GSS15, P. 3].

The gradient of the cost function $\nabla_x J(\theta, x, l)$ gives information about in which direction the loss is increasing the steepest. By applying the sign function to every component of the gradient vector, each one is replaced by 1 or -1 in case the component does not equal 0. The perturbation vector $r$ which consists of small values with a magnitude $|r|_i \leq \epsilon$ is then added to the original image to increase the loss [Yua+19, P. 5]. Both, the original image $x$ and the perturbation $r$ can be seen as vectors being added together. Other than the previously described L-BFGS method, FGSM is an untargeted attack where a label cannot be chosen.

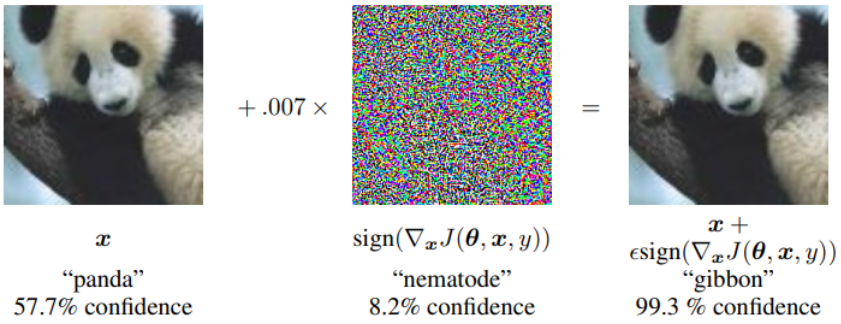



Figure 3.4: A perturbation crafted with FGSM. $\epsilon = 0.007$, a GoogLeNet example [GSS15, P. 3 ]

In comparison to the L-BFGS method, this variant is very efficient because FGSM calculates the perturbation in one step. The gradient required can be efficiently computed using the backpropagation algorithm which adds up to the efficiency of this method [GSS15, P. 3 ]. The impact of adversarial examples crafted by FGSM is high if used against an undefended network but does not uphold in case the attacked network has met necessary precautions [Mad+18, P. 5].

A variation of this procedure would be the Fast Gradient Value method in which the sign and $\epsilon$ are being left out of the formula to compute the raw value of the gradient. This causes the perturbation to be more significant overall since it is not being constraint by an $\epsilon$.

$$r = \nabla_x J(\theta, x, l)$$

It is also possible to alternate the method in a way, that it is targeted by maximizing the probability of a particular class. [Yua+19, P. 6 ]

$$x' = x - \epsilon \cdot sign(\nabla_x J(\theta, x, l'))$$

$l'$ is the class that is least likely to be picked by the model for classification. In this case, the method would be called One step least-like class method. [KGB17, P. 3]

According to Goodfellows' paper, adversarial examples can be found in specific directions within the input space. This goes against Szegedy et al.'s theory that they might reside all together somewhere in the input space [GSS15, P. 7-8].

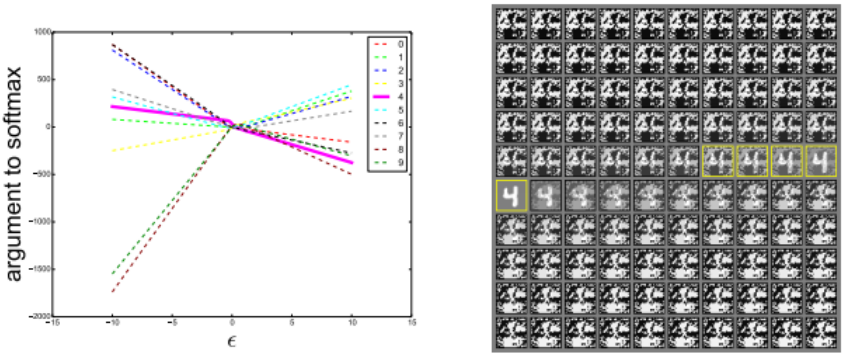



Figure 3.5: Graph showing the direction in which the classes spread out with different $\epsilon$. The correct class, in this case, is 4, as seen by the purple line. Right: Original image perturbed using FGSM using different values of $\epsilon$. Images in a yellow frame mark correctly classified images [GSS15, P. 8].

### 3.3.3 Projected Gradient Descent

In previous methods, adversarial examples were created in a single step in which the perturbation was created. Another approach is to tackle the problem iteratively, where FGSM is applied multiple times with small step sizes.

The previous definition of gradient ascent in input space attempts to find an adversarial example within an image distribution $P_x$. $P_x$ can therefore be seen as some radius around the original image $x$ in which the adversarial example has to reside [Han20]. The radius is defined as:

$$P_x = \{x' \mid ||x' - x||_\infty < \epsilon \}$$

Here, $P_x$ is a constraint that restricts the difference $||x' - x||_\infty$ of the clean image $x$ and the adversarial example $x'$ to be smaller than some constant radius $\epsilon$. The following formula iteratively calculates the perturbation in regard to $P_x$ [Mad+18, P. 4].

$$x^{t+1} = \Pi(\ x^t + \alpha\ sign(\nabla_{x^t}\ J(\theta, x^t, l'))$$

$\alpha$ denotes the chosen step size, in which gradient ascent travels towards regions with the highest loss, and the projector $\Pi$ projects $x$' back to the closest point to $x$ within the radius $P_x$ in case the algorithm steps outside of $P_x$ (see 3.6) [Han20]. The sign of the loss is therefore not multiplied with a scalar $\epsilon$, but with $\alpha$. The algorithm is being initialized at a random point (see 3.6) around $x$ within $\epsilon$ and will take iterative steps of size $\alpha$ towards the direction of the steepest ascent until the possible maximum loss has been found. To make sure that the found maximum loss is not a mere local maximum but the maximum loss achievable within $P_x$, the algorithm restarts a random number of times at random points $x'$. The authors discovered however, that even with a large number of restarts, areas with significantly higher loss could not be found [Mad+18, P. 7].

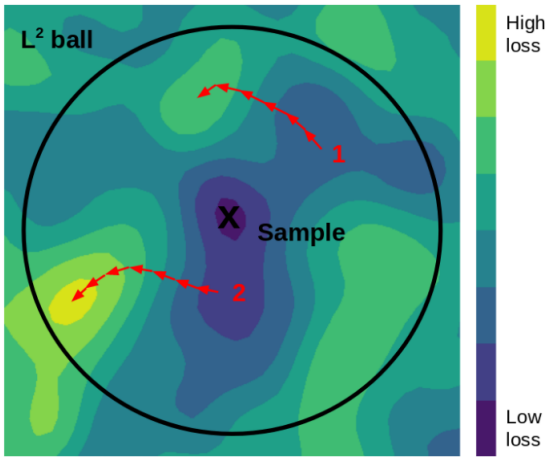



Figure 3.6: PGD restarting two times to find the maximum loss [Kna19]

Since this process is working iteratively towards an optimal result, it is possible to create more refined and universal perturbations than FGSM [Mad+18, P. 2]. The only downside is that projected gradient descent is computationally expensive since it uses FGSM in every iteration. FGSM alone is, therefore, more efficient since it only calculates one step but easier to defend against .

CIFAR10

| | Simple | Wide |
|---|---|---|
| Natural | 92.7% | 95.2% |
| FGSM | 27.5% | 32.7% |
| PGD | 0.8% | 3.5% |

(a) Standard training

| | Simple | Wide |
|---|---|---|
| Natural | 87.4% | 90.3% |
| FGSM | 90.9% | 95.1% |
| PGD | 0.0% | 0.0% |

(b) FGSM training

| | Simple | Wide |
|---|---|---|
| Natural | 79.4% | 87.3% |
| FGSM | 51.7% | 56.1% |
| PGD | 43.7% | 45.8% |

(c) PGD training

| Simple | Wide |
|---|---|
| 0.00357 | 0.00371 |
| 0.0115 | 0.00557 |
| 1.11 | 0.0218 |

(d) Training Loss

Figure 3.7: Performance of small/ wide versions of a CIFAR10 trained network on clean, FGSM, and PGD samples. It shows that adversarial examples created by PGD are immune to adversarial training based on FGSM examples. [Mad+18, P. 11]

### 3.3.4 DeepFool

With the help of DeepFool, it is possible to calculate more reliable and efficient perturbations than with previous methods [MFF16, P. 1]. DeepFools' main approach to cause misclassification is to manipulate an image in a way that it is just barely pushed over the decision boundary that determines its class. To achieve that, the authors of the DeepFool paper orthogonally project an image $x_0$ onto the separating hyperplane [MFF16, P. 3, Section 2].

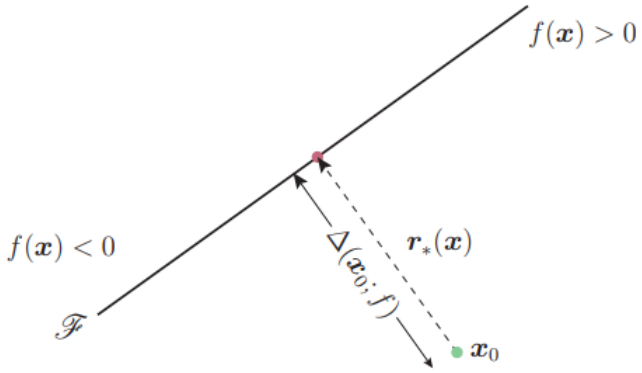



Figure 3.8: $x$ being projected onto the separating hyperplane. $r_*(x)$ = the calculated perturbation of $x_0$, $f(x) < 0$ and $f(x) > 0$ = sign functions of the prediction of $x$ [MFF16, P. 3]

On every iteration of the algorithm, a perturbation is added that brings $x_0$ closer to the hyperplane. The perturbation, thereby, is calculated using the formula for orthogonal vector projection [MFF16, P. 3, Section 2].

$$-\frac{f(x_i)}{||\nabla f(x_1)||_2^2}\nabla f(x_i)$$

$f(x_i)$ represents a vector of prediction values at the current iteration $x_i$ of $x$ which is to be projected on the gradient vector $\nabla f(x_i)$. $\nabla f(x_i)$ represents the orthogonal direction towards the hyperplane on which $x$ is supposed to be projected on. The algorithm applies the perturbation for as long as the sign of $f(x_i)$ does not change, in other words, until the classification changes. In some cases, $x_0$ lands directly on the plane, where it's suggested to add a very small vector $\eta$ to push it over [MFF16, P. 3, Section 2].

**Algorithm 1** DeepFool for binary classifiers

1: **input:** Image $\boldsymbol{x}$, classifier $f$.
2: **output:** Perturbation $\hat{\boldsymbol{r}}$.
3: Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
4: **while** $\text{sign}(f(\boldsymbol{x}_i)) = \text{sign}(f(\boldsymbol{x}_0))$ **do**
5: $\quad \boldsymbol{r}_i \leftarrow -\frac{f(\boldsymbol{x}_i)}{\|\nabla f(\boldsymbol{x}_i)\|_2^2} \nabla f(\boldsymbol{x}_i)$,
6: $\quad \boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$,
7: $\quad i \leftarrow i + 1$.
8: **end while**
9: **return** $\hat{\boldsymbol{r}} = \sum_i \boldsymbol{r}_i$.

Figure 3.9: The algorithm for DeepFool. While the classification does not change (line 4), create a perturbation that projects the image towards the separating hyperplane (line 5). Projections are summed up in every iteration(line 6). Ends when classification has changed. Returns a perturbation vector containing the sum of all projections (line 9). [MFF16, P. 3]

The multi-class variant of DeepFool additionally considers the closest plane to an image x, which it is to be projected on before it is finally pushed over. [MFF16, P. 3, Section 3]

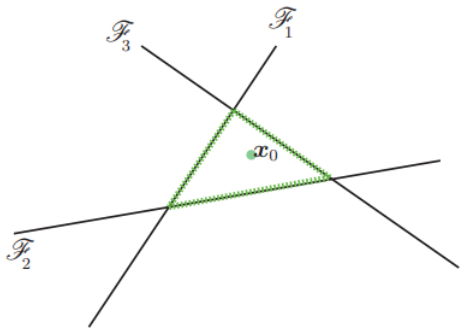



Figure 3.10: Multiclass Deepfool. $x_0$ is being projected onto the closest plane. [MFF16, P. 3]

DeepFool manages to create perturbations smaller than FGSM, making it harder to defend against them. It is therefore suggested by the authors to use adversarial examples created by DeepFool since they are more reliable. In terms of robustness assessment, it is proposed to use DeepFool examples in adversarial training due to their reliability and risk that other methods could give false statements about the robustness of a model [MFF16, P. 1].

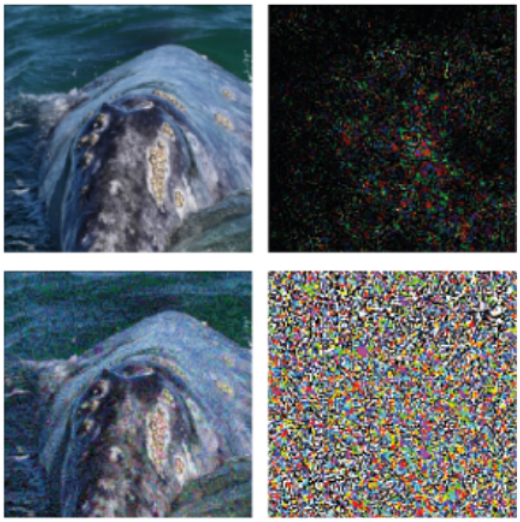


Figure 3.11: Top row: Perturbations created by DeepFool. Bottom row: Perturbations created by FGSM. [MFF16, P. 1]

When put into comparison, the adversarial examples created by DeepFool managed to lower the accuracy of target models further than FGSM. The following table shows the accuracies of 4 different deep learning models when supplied with FGSM and DeepFool images. The first 2 models were trained on MNIST images while the rest was trained of CIFAR10 [MFF16, P. 6].

| **Classifier** | DeepFool | Fast gradient sign |
|---|---|---|
| LeNet (MNIST) | 0.10 | 0.26 |
| FC500-150-10 (MNIST) | 0.04 | 0.11 |
| NIN (CIFAR-10) | 0.008 | 0.024 |
| LeNet (CIFAR-10) | 0.015 | 0.028 |

Figure 3.12: Model accuracy on FGSM and DeepFool examples. [MFF16, P. 6]

### 3.3.5 Carlini Wagner Attack

Nicholas Carlini and David Wagner proposed three adversarial attacks in which they have successfully broken many state-of-the-art defensive mechanisms against adversarial examples. Defensive distillation, for example, a defense mechanism that will be covered later, has improved the accuracy of selected models which have been attacked by other attack mechanisms from 0.5% to 95% [Pap+16, P. 1]. The Carlini-Wagner attack, however, managed to break that defense [CW17, P. 1].

Most types of attacks use gradient descent to find adversarial examples. In the case of an implemented defense, however, gradient descent has a problem with finding adversarial examples, because gradient descent works best with smooth downhill transitions into other classes, which is prevented (see 3.13). Since gradient descent fails to perform the task of finding adversarial examples due to the nonlinearity of the problem, the authors introduce a way to nullify defense mechanisms [Car19c]

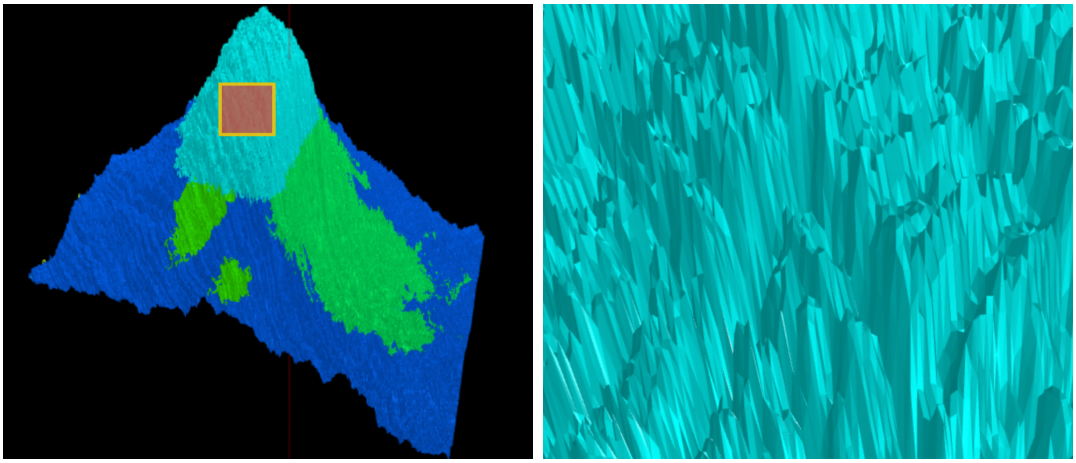


Figure 3.13: Loss surface of a neural network. Gradient descent would have a problem to descent into another class since a defense has been implemented. The height represents the confidence of the model and each color represents a different class.[Car19a]

An objective function $J$ is introduced, which shows, how close the sample image $x$ is to being classified as the desired target $l'$. Solving $J(x') = l'$ is very non-linear in its nature and therefore hard to solve. To tackle that, the new function $J$ is inserted into the original problem since it is better suited for for optimization [CW17, P. 6, Section V]. Here, the classifier is defined as $f$ and the predicted target as $f(x) = t$. The problem was originally defined as:

$$\text{minimize} \quad ||x' - x||$$

$$\text{such that } f(x') = l'$$
$$\text{and } x' \in [0,1]|$$

After the objective function is built into the original problem, the alternative formulation then looks as follows:

$$minimize\ ||x' - x||\ +\ c \cdot\ J(x')$$
$$such\ that\ x' \in [0,1]^m$$

$J(x') = l'$ is only valid, if and only if, $J(x') \leq 0$. $c$ is a constant chosen based on what value it best implements for $J(x') \leq 0$ and found by performing a binary search [Yua+19, P. 8, Section G]. Based on this, they have developed three attacks using the three norms in vector calculation $l_2$, $l_\infty$ and $l_0$ [CW17, P. 9-10].Different constraints used by the attacks smooth out the gradient descent, making it possible to find adversarial examples without getting stuck in extreme areas.

For every attack the authors proposed, they found better and more imperceptible adversarial examples than previously developed methods and they never encountered a case where an adversarial example could not be found [CW17, P. 15]. The developers of this attack method have proven with their developed strategy, that even defenses, which seem unbreakable at first can be overcome eventually which further emphasizes on the security risks in machine learning.

### 3.3.6 Universal Adversarial Perturbations

The authors of the paper "Universal adversarial perturbations" proposed a method, to calculate a single perturbation which causes misclassification when applied to a set of images. Other methods calculate perturbations individually for every image. This approach shows how well adversarial examples generalize not only over images but over different neural networks [Moo+17, P . 1].

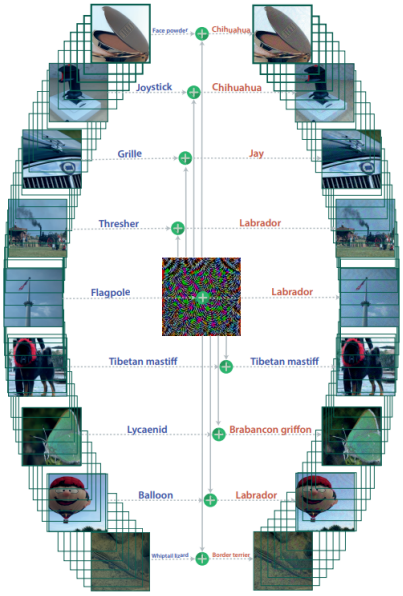



Figure 3.14: One perturbation misclassifying 8/9 images [Moo+17, P . 1]

The algorithm seeks to create a universal perturbation vector $v$, which consists of many small perturbations $r$, that cause misclassification of a classifier $f$ over a set of images $I$. Formally described as

$$f(x + v) \neq f(x)$$

which must be valid for the majority of $x \in I$. Additionally, there are two constraints following that formulation. A parameter $\xi$ which controls the magnitude of the perturbation as well as $\delta$, the parameter which controls the desired fooling rate, therefore the least number of images that have to be misclassified [Moo+17, P . 2].

$$||v|| \leq \xi$$
$$P(f(x') \neq f(x)) \geq 1 - \delta$$

$P$ represents the fooling probability of all perturbed images out of $I$ which has to be larger equal to the desired fooling rate. The algorithm iteratively runs over the distribution of images $I$ and gradually builds the perturbation by adding perturbations to the universal perturbation vector v which pushes individual images towards their decision boundary [Moo+17, P .2].

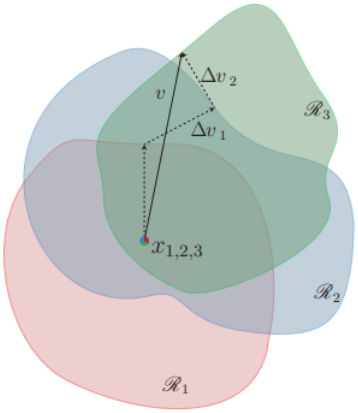



Figure 3.15: Adversarial perturbation being calculated iteratively by considering the three decision boundaries. Classes presented in colors, $x$ being 3 different images and $\Delta v_n$ being the change over time in the perturbation vector $v$. With every iteration, a perturbation is calculated and added to the universal perturbation vector $v$, such that the classification of the image is being pushed towards a decision boundary (different colored area). By doing that for every, image, $v$ manages to fool classifiers with several images but only one perturbation [Moo+17, P . 3].

While the fooling rate is not satisfied, the algorithm iterates over all images in $I$ for which the perturbation is to be calculated. While the classification of the current image is still the original one, a minimal perturbation $r$ is being added to the universal perturbation vector $v$ which fulfills the requirement $f(x') \neq f(x)$. The perturbations of each iteration are being summed at the end of the loop after being projected into the radius $\xi$ to control the magnitude. After the algorithm has iterated over all images and the desired fooling rate is met, the universal perturbation vector $v$ is a sum of all perturbations $r_i$ that managed to misclassify $I_i$ . How the radius is calculated, depends on the chosen vector norm. [Moo+17, P . 3]

Experiments with this method have shown, how well it generalizes over different datasets. The figure below shows the fooling rate of the perturbation tested on the initial set $X$, from which the perturbation was calculated and then the fooling rate of the same perturbation validated on a different dataset for several networks. Experiments were made for two different vector norms for which the authors have chosen $\xi$ to be 2000 and 10, respectively [Moo+17, P . 3-4].

| | | CaffeNet [8] | VGG-F [2] | VGG-16 [17] | VGG-19 [17] | GoogLeNet [18] | ResNet-152 [6] |
|---|---|---|---|---|---|---|---|
| $\ell_2$ | $X$ | 85.4% | 85.9% | 90.7% | 86.9% | 82.9% | 89.7% |
| | Val. | 85.6 | 87.0% | 90.3% | 84.5% | 82.0% | 88.5% |
| $\ell_\infty$ | $X$ | 93.1% | 93.8% | 78.5% | 77.8% | 80.8% | 85.4% |
| | Val. | 93.3% | 93.7% | 78.3% | 77.8% | 78.9% | 84.0% |

Figure 3.16: Fooling rate of universal adversarial perturbations on different networks[Moo+17, P. 4].

Further experiments have shown the transferability of universal perturbations on other networks that were not included in the creational process and have distinct datasets on which they have been trained on. The success is measured in the fooling rate.

| | VGG-F | CaffeNet | GoogLeNet | VGG-16 | VGG-19 | ResNet-152 |
|---|---|---|---|---|---|---|
| VGG-F | **93.7%** | 71.8% | 48.4% | 42.1% | 42.1% | 47.4 % |
| CaffeNet | 74.0% | **93.3%** | 47.7% | 39.9% | 39.9% | 48.0% |
| GoogLeNet | 46.2% | 43.8% | **78.9%** | 39.2% | 39.8% | 45.5% |
| VGG-16 | 63.4% | 55.8% | 56.5% | **78.3%** | 73.1% | 63.4% |
| VGG-19 | 64.0% | 57.2% | 53.6% | 73.5% | **77.8%** | 58.0% |
| ResNet-152 | 46.3% | 46.3% | 50.5% | 47.0% | 45.5% | **84.0%** |

Figure 3.17: 24 Comparison of how universal perturbations affect other networks. [Moo+17, P . 6]

With Universal Adversarial perturbations it is therefore not only possible to fool a classifier on a set distribution of images with just one perturbation but have also shown that the calculated perturbation is transferable to other Deep Learning models.

## 3.4 Black-box attacks

### 3.4.1 Approximating a Target Network

Since the attacker has no knowledge of the network, a substitute network is being constructed that is supposed to imitate the behavior of the original network. The goal is to create a network with similar decision boundaries as the original one. To approximate the model, the attacker first needs input that fits the task of the attacked model and could possibly be training data. Such a task could be the recognition of handwritten digits, so the attack would collect images of those in the same format. For that purpose, the attacker needs to collect training data and train a new model such that it imitates the decision boundaries of the original. [Pap+17, P. 3, Section 4]

The choice of the model architecture also depends on the task at hand. For vision-based tasks, a convolutional neural network would be the best choice . The attacker uses the selected images of his choice to be an input of the attacked model to extract the predicted labels. Once extracted, the new training set for the substitute model consists of the chosen images and the labels, which the attacked model previously predicted respectively. With the dataset, the substitute model is eventually trained. It is important not to query

the target model too many times to get predictions„ since it would make adversarial behavior detectable. [Pap+17, P. 4, Section 4.1] In order to increase the variation in the input and to better represent the decision boundaries of the attacked model. The initial training set is being augmented after it has been iterated through. The augmented set then serves as input for the next training iteration. For the augmented set, the attacker collects the labels again on which he repeats the training procedure for a set number of epochs[Pap+17, P. 4, Section 4.1 ].

After the surrogate model has been trained, any attack algorithm can be used to craft adversarial examples. [Pap+17, P.5, Section 4.2] Experiments have shown successful misclassification using the adversarial examples crafted with the fast gradient sign method based on their surrogate model, which were then used to attack other models [Pap+17, P . 10].

| | | Amazon | | Google | |
|---|---|---|---|---|---|
| Epochs | Queries | DNN | LR | DNN | LR |
| $\rho = 3$ | 800 | 87.44 | 96.19 | 84.50 | 88.94 |
| $\rho = 6$ | 6,400 | **96.78** | **96.43** | **97.17** | 92.05 |
| $\rho = 6^*$ | 2,000 | 95.68 | 95.83 | 91.57 | **97.72** |

Figure 3.18: 24 Test results of adversarial examples crafted based on surrogate models with FGSM. Attacked models were either Deep neural networks(DNN) or linear regression(LR) models. The queries describe, how many times the prediction of the target model had to be queued. [Pap+17, P . 10]

### 3.4.2 One Pixel Attack

This black-box method manages to fool classifiers by only changing one pixel in the entire image. For this, the authors proposed to use the concept of Differential Evolution [SVS19, P. 4, section B], where the algorithm iteratively finds the best candidate solution for the problem at hand . One candidate solution is a tuple of 5 values, $x, y$ for the coordinates of a pixel and RGB for the color values [SVS19, P. 4, section C]. Initially, 400 candidate solutions will be randomly generated and on each iteration, 400 more will be generated serving as children to their corresponding parent solution. If a child solution serves as a better fit for the problem as the parent solution, the parent will be replaced, otherwise, they will be discarded . The setup of the authors stopped at either 100 iterations or if a solution has been found, which causes the label of a target class to reach a confidence score of 90% and 5% on the actual target. To measure success, the values of the true label and the targeted label are compared [AM18, P. 4-5, Section 3.1.5].


Figure 3.19: Misclassification caused by the change of one pixel. Columns represent different networks with the original label in black, compared to the new label with its confidence score. [SVS19, P. 1]

The authors conducted several experiments showing major success attacks on multiple neural networks.

| | AllConv | NiN | VGG16 | BVLC |
|---|---|---|---|---|
| OriginAcc | 85.6% | 87.2% | 83.3% | 57.3% |
| Targeted | 19.82% | 23.15% | 16.48% | – |
| Non-targeted | 68.71% | 71.66% | 63.53% | 16.04% |
| Confidence | 79.40% | 75.02% | 67.67% | 22.91% |

Figure 3.20: Results of experiments conducted on several neural networks. Metrics are the original accuracy, whether the attack is targeted or not and the resulting confidence score[SVS19, P.6]

### 3.4.3 UPSET and ANGRI

These two attack methods, train neural networks on creating adversarial examples. In the case of UPSET, short for “Universal Perturbations for Steering to Exact Targets”, the network creates universal adversarial perturbations for every class in the target networks [Yua+19, P. 6, Section 3.1.9]. Each of these perturbations is designed to make the classifier classify any image to be of any of these classes. An image that was not classified to be of a certain class $l$ would then be classified as $l$ after the perturbation has been added .

The powerhouse of UPSET and ANGRI is the residual neural network used at the beginning of the training iteration, which takes a class $l$ as an input and creates an adversarial perturbation based on it [AM18, P. 6, Section 3.1.9]. Even though the perturbation can was only created from one class, it is universal for all other classes [Sar+17, P. 8]. The perturbation is added to the sample image $x$ to craft the adversarial example which is then used to calculate the loss of $m$ pre-trained classifiers $F$ in regards to the target label. UPSET then uses the fidelity loss and the misclassification loss of all attacked networks to calculate its own training loss. The loss of all classifiers is used as a penalty function for whenever the network does not predict the target class $l$ and the fidelity loss ensures that the input and output images look alike. Additionally, a weight $w$ is introduced to regulate the fooling- and the fidelity rate. The network is trained during multiple iterations of this process, to optimize the adversarial examples, the residual generating neural network outputs. Since UPSET only requires some clipping and addition to create adversarial examples, it works fast, while only requiring a target class as input [Sar+17, P.3, Section 3.3].

ANGRI, short for Antagonistic Network for Generating Rogue, works in a much similar way except that it does not create universal perturbations but creates perturbations based on the image the algorithm gets as input. It takes an image from the training distribution of the original network as input and transforms it in a way that it is being recognized as a target class $l$ [Sar+17, P. 3, Section 3.3].

According to the authors, ANGRI also performs well with noisy data while the performance of UPSET degrades. Since ANGRI has access to an input image, it can produce better and more imperceptible adversarial examples than UPSET. Nonetheless, both methods manage to create adversarial examples in a black box scenario while targeting multiple classifiers at once, which makes them better at generalizing [Sar+17, P. 9]. In experiments, the authors have trained several models on MNIST datasets which had initial accuracies between 97% and 99%. Half of the models were ResNet-style networks while the other half were convolutional networks [Sar+17, P. 6]. They documented the results for every classifier they targeted and how the perturbation created with the targeted classifier structure affects other networks.

| Victim | Trained on | | | |
|---|---|---|---|---|
| | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
| $C_1$ | **91%, 94%** | **48%, 50%** | 5%, 4% | 6%, 6% |
| $C_2$ | **50%, 47%** | **90%, 95%** | 8%, 5% | 11%, 14% |
| $C_3$ | 46%, 44% | 64%, 75% | **97%, 98%** | **67%, 75%** |
| $C_4$ | 40%, 36% | 61%, 71% | **39%, 31%** | **93%, 98%** |

Figure 3.21: Fooling rate of attacks on different networks. C1 and C2 are ResNet-style networks while C3 and C4 are convolutional networks. The first element of a row represent the fooling rate with UPSET and the second item the fooling rate with ANGRI. [Sar+17, P. 8]

The experiments show that the adversarial examples don't generalize well on different network structures. The comparison is drawn between ResNet-style networks and common convolutional neural networks. Even though the attacker has little information about the original network, just by training a network based on the outputs of other classifiers, the authors managed to create imperceptible adversarial examples to cause misclassifications in state-of-the-art neural networks. UPSET and ANGRI both manage to perturb images in a way that even though their visual appearance does not change to an extent that it is visible to the human eye, the classification by the model is far off the target.

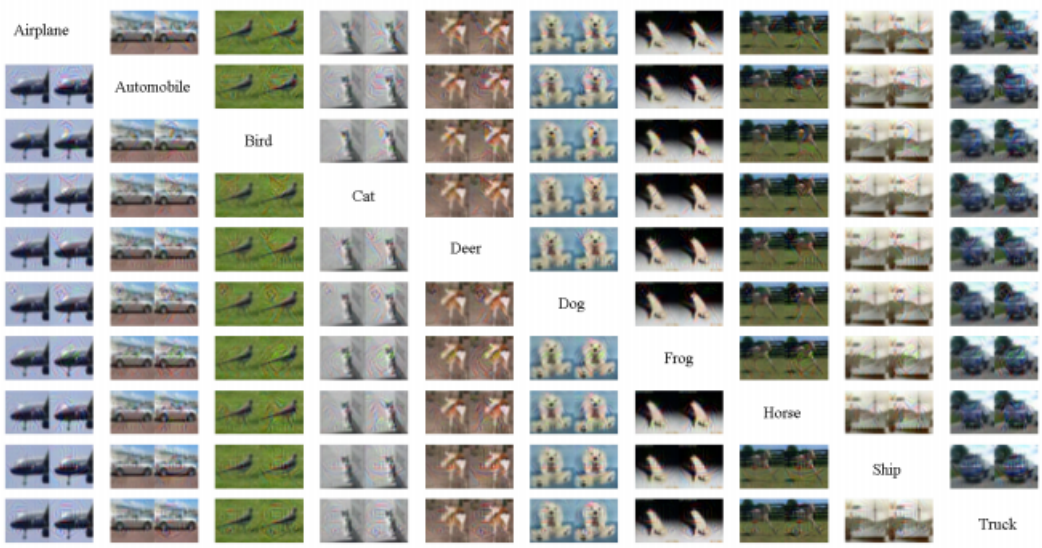



Figure 3.22: Rows represent target classes 0-9 and columns input classes 0-9. Left images are generated by ANGRI and right images by UPSET [Sar+17, P. 6 ]

## 3.5 Defending against Adversarial examples

In the following, several defense strategies will be introduced which attempt a defense against adversarial examples. The difference between the mechanisms is that some of them work pro-active, where the robustness of the network is improved before an attacker

creates adversarial examples, and others work reactive, where adversarial examples are detected after the network has been built [Yua+19, P. 14, Section IV].

### 3.5.1 Gradient Masking

Gradient masking is a technique used by several defensive mechanisms to make information inaccessible for an attacker. The goal is to deny any kind of gradient information which the attacker could use to estimate a model's gradient. To execute typical white-box attacks, the attacker needs full information on the gradients of the network. Hence, the attacker needs probabilities to know in what way to manipulate the image so that the class probabilities change. In terms of output, a model has two options. Either it outputs probability scores in a softmax label or it produces hard labels. By only producing hard labels, the attacker would not know how to manipulate the image since he is not aware of probability scores. For instance, the model would not have an output like "Dog, 95.6%" but only "Dog" which makes it harder for an attacker to estimate how to manipulate an image. If probabilities like 99% dog, 1% cat would be supplied, the attacker could easily estimate the gradients by measuring the small change of probabilities when perturbing the input [Goo+17].

Previously mentioned black-box attacks have shown, however, that not all information is denied to an attacker. The attacker can still train a surrogate network to imitate the targeted network. Since the surrogate network sufficiently supplied the attacker with its own gradient information, an attack can still be performed.

### 3.5.2 Adversarial Training

In the same paper in which Goodfellow et. Al has introduced the Fast Gradient Sign Method, they proposed a proactive strategy to make neural networks more robust to adversarial examples. By including images manipulated by the FGSM method next to clean training images into the training set but provide them with the correct label, the model learns how to correctly classify the manipulated images [GSS15, P. 4, Section 6]. However, this method only helps with defending against very particular adversarial examples but does not generalize well. As a consequence, the defense mechanism could be broken by merely changing the value of epsilon for the attack to a slightly higher value or by using iterative attacks. The increase of epsilon then increases the magnitude of the perturbation slightly . Since the model does not generalize well on FGSM crafted adversarial examples, it is not used to these changes and misclassifies them again. Iterative methods, like projected gradient descent, show higher success when used in adversarial training since they generalize better for different types of perturbations. It has been shown that the networks tend to overfit on adversarial examples when generated

by FGSM, which causes the network to have a higher accuracy on adversarial examples than on clean images. Consequentially, the network does not profit from any kind of robustness [Mad+18, P. 10].

### 3.5.3 Detect Adversarial Examples

This reactive strategy is a way to prevent a neural network from being fooled by training a second network on detecting adversarial examples. The detection network would then flag these images as adversarial for the main network, making sure that they will be removed from the classification process. If well trained, the detection network manages to generalize well between adversarial examples, making it very reliable in the detection task. In order to successfully fool the main network, the adversarial example would then also have to be able to fool the detection network [Goe20, P. 7]. Experiments of this defense mechanism have shown that the issue with FGSM perturbed images persists. They do not generalize well when the detection network is trained on FGSM samples but is supposed to detect other types of adversarial examples, like PGD for instance [Goe20, P. 7], [Yua+19, P. 14, Section C].

| Method | Baseline Accuracy | Flagging Adversarial Examples Accuracy |
|---|---|---|
| FGSM ($\varepsilon = 0.1$) | 23% | 83% |
| PGD | 20% | 60% |

Figure 3.23: The detection model has been trained on FGSM examples. Baseline accuracy denotes the accuracy without flagging and the last column shows how many adversarial examples could be flagged using the detection model for both FGSM and PGD. [Goe20, P. 7]

### 3.5.4 Defensive Distillation

Distillation was originally used to transfer knowledge from larger network architectures to smaller ones by feeding the probability scores of the first network into a second one [Pap+16, P. 1]. As an output layer, softmax is used which outputs a vector of probabilities which is distilled eventually. To distill the output of a network, the outputs of the softmax layer are divided by a temperature parameter[Yua+19, P. 14, Section A]. By choosing a small temperature parameter, the softmax layer will eventually change into a hard-labeled layer, where instead of probabilities, only the correct label has a 1

as its value while the rest remains at 0 whereas a high-temperature parameter increases the confidence of the distilled network. For the training of the first network, a high temperature is used which outputs high confidence scores on the softmax layer. [Yua+19, P. 14, Section A], [Pap+16, P. 5, Section C]. This method works reactive because it acts, before an attacker can potentially craft adversarial examples and attack it.

The original set of training images is labeled with hard labels, but while training a second network with the same architecture on the same set of images, the probability scores of the first network are being used as labels instead of hard labels. While training the second network, the temperature has the same value as when the original network was trained. The network is therefore trained on outputting probabilities rather than hard decisions. [Goo+17]

# 4 Libraries to create Adversarial Examples

There are several libraries developed in python to create adversarial examples and to benchmark trained models. All of these libraries are written in Python since it's the most commonly used programming language to implement machine learning algorithms. Initially, a victim model will be introduced. This Keras-built model will be subject to adversarial attacks in future pages and used to compare the attack methods as well as the libraries. Followed by the setup and the model, each library will be introduced with code examples on how to attack and potentially defend (if available) the victim model. For that purpose, outputs will be visualized. Finally, all libraries are put in comparison. To implement the code shown in the following code snippets, and to use the libraries presented, basic Python knowledge is required. However, the required Python knowledge can slightly vary between libraries.

## 4.1 Setup

To run scripts, either Google Colab or Anaconda [20] was used. Google Colab makes it possible to run code snippets in a web-based environment independent of local hardware. Scripts that needed more computational power were run in an Anaconda virtual environment on a local computer. Anaconda simplifies package management and offers many built-in data science tools. For both, Google Colab and the virtual environment, TensorFlow 2.5.0 and Python 3.8.8 were used. Additionally, the CUDA API by Nvidia was installed on the local machine to speed up training processes.

## 4.2 The Victim Model

To best represent the individual results of the libraries, all attacks and defenses will be tested on one Keras model. The model is trained on MNIST images, a dataset containing thousands of images of handwritten digits. MNIST is chosen for the presentation of results due to its low resolution of 28x28 images which makes the difference between

perturbed and clean images visible. The architecture and preprocessing steps were taken from [Cho17, P. 120].

### 4.2.1 Prerequisites

To create a model, the Keras library has to be imported, along with the dataset and NumPy.

Listing 4.1: Imports to build a Keras model

```
from keras import layers
from keras import models
from keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

import numpy as np
```

The data is provided by Keras and has to be split into a training and testing set first. Train_images contains 60000 images while the testing distribution only contains 10000. The tensors are reshaped and divided by 255 such that the pixel values lie in between 0 and 1 to fit into the network. Since this is a multiclass classification problem, categorical cross-entropy is used as the loss function which requires categorical labels. To_categorical turns the labels for each image into binary vectors where the position which is 1 represents the class. Instead of having a tensor of labels where each index has the format, i.e, $[3], [8] \ldots$, each index will be a vector of binary values $[0, 0, 1, 0 \ldots 0], [0, 0, 0 \ldots .0, 1, 0, 0]$. The length of each vector is equal to the number of classes. The training labels then contain 60000 binary vectors and the testing labels 10000, respectively. Each index of the image tensors corresponds to the index of the correct label in the label tensor.

Listing 4.2: Create a tensor with numpy

```
(train_images, train_labels), (test_images, test_labels) = mnist.
    load_data()
train_images = train_images.reshape((60000, 28, 28,1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28,1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

### 4.2.2 Building a Model

A sequential type model is used to build a classifier. Sequential type models follow a linear top to bottom structure and are used when every layer has exactly one input and output. Gradually, convolutional and max-pooling layers are added to the model. Convolutional layers need the number of filters, the filter dimensions, and an activation function as input. The initial convolutional layer requires the input dimensions. Since MNIST images all have 28x28 dimensions, the input is adjusted accordingly. Eventually, the convolutional layer is flattened and fed into a couple of dense layers where the final layer has a softmax activation function for the desired number of classes. The MNIST dataset contains images of the digits 0 – 9, therefore, 10 output classes are required.

Listing 4.3: Building the model architecture

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,
    28,1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Compile the model to configure how the network is being optimized and set the loss function during the learning process. Since images are assigned to multiple classes, categorical crossentropy is used as a loss function. One can also choose the metrics that should be observed.

Listing 4.4: Compiling the model

```
model.compile(optimizer='rmsprop',
    loss='categorical_crossentropy',
    metrics=['accuracy'])
```

### 4.2.3 Training

Finally, the model can be trained on the train images and labels. Epochs define the number of times the training data is being iterated [**Fc**] over and the batch size defines how much data the model is going to process at once since models never iterate over the entire dataset in a single step [Cho17, P. 34]. Afterwards, the test scores can be observed by evaluating the model.

Listing 4.5: Initialise the training process and evaluate

```
model.fit(train_images, train_labels, epochs=4, batch_size=32)
test_loss, test_acc = model.evaluate(test_images, test_labels)

print("Test␣loss:" ,test_loss)
print("Test␣acc:", test_acc)

Output:
Test loss: 0.0343...
Test acc: 0.9921...
```

For later use, the model can be saved to a location on the hard drive. By saving and loading in the model once it has been trained, the model does not need to be retrained whenever the script runs.

Listing 4.6: Save and load a model

```
model.save(path_to_model/CNN_model.h5)
model = keras.models.load_model(path_to_model/CNN_model.h5')
```

## 4.3 FoolBox

Foolbox, which is primarily developed by Jonas Rauber, a Ph.D. student at the University of Tübigen, is a library that lets the user generate state-of-the-art adversarial examples with a few lines of code. It was released in June 2016 and already supported many attack variants with its earliest version. The last major update released in 2020 where they rebuild the library on EagerPy instead of NumPy to increase performance. Supported machine learning frameworks are Tensorflow, PyTorch, and JAX [Rau+20].

Installation: !pip install foolbox

### 4.3.1 Prerequisites

At first, the foolbox library needs to be imported to use all its functions as well as the previously created Keras model.

Listing 4.7: Import FoolBox and load model

```
import foolbox as fb
model = keras.models.load_model('path_to_model/CNN_model.h5')
```

The original model needs to be passed to FoolBox type classifier, where the desired image bounds are specified. Since the training and testing images for the model have been normalized in a spectrum between 0 and 1 earlier by dividing the pixel values each by 255, the bounds can be set respectively.

Listing 4.8: Create FoolBox classifier with bounds

```
bounds = (0, 1)
fmodel = fb.TensorFlowModel(model, bounds=bounds)
```

### 4.3.2 Attacks with FoolBox

The test images will be used to create adversarial examples that are loaded in like before in code snippet 3. To fit into the attack methods offered by FoolBox, converting the images into eager tensors has to be done as an additional preprocessing step as well as casting the labels to int32.

Library and tutorials available at `https://github.com/bethgelab/foolbox`

Listing 4.9: Load sample images

```
test_labels = tf.convert_to_tensor(test_labels )
test_images = tf.convert_to_tensor(test_images )

test_labels = tf.dtypes.cast(test_labels, tf.int32)
```

An attack object requires the images, labels, the target model, and epsilon, which controls the magnitude of the perturbation, as parameters. After running the attack, the function returns 3 tensors that contain different information about the attack:

- The raw adversarial examples that are not subject to epsilon
- Adversarial examples with perturbations that are constrained to be no larger than epsilon
- A Tensor containing Boolean values, giving information which images turned out to be adversarial examples after the transformation and therefore caused misclassification.

Listing 4.10: Generate adversarial examples with FGSM, PGD, and DeepFool

```
raw_fgsm, clipped_fgsm, is_adv_fgsm = attack_fgsm(fmodel, test_images,
test_labels, epsilons=0.3)

raw_pgd, clipped_pgd, is_adv_pgd = attack_pgd(fmodel, test_images,
test_labels, epsilons=0.3)

raw_df, clipped_df, is_adv_df = attack_df(fmodel,test_images,
test_labels, epsilons=0.3)
```

To evaluate, how well these adversarial examples fool the classifier, the FoolBox accuracy function can be used.

Listing 4.11: Evaluate accuracies

```
print("clean␣acc:␣",fb.utils.accuracy(fmodel, test_images, test_labels)
    )
print("df␣acc:␣",fb.utils.accuracy(fmodel, clipped_df, test_labels))
...

clean acc:  0.9886999726295471
df acc:  0.1964000016450882
fgsm acc:  0.21050000190734863
Pgd acc:  0.00989999994635582
```

Library and tutorials available at `https://github.com/bethgelab/foolbox`

### 4.3.3 Visualization

In the following, a random image is drawn from the clipped image batches and the clean batch. They will be plotted side by side to visualize the change in look and classification. At first, the labels of the random image are needed. The clean and perturbed batches of images are passed to the model, to return a vector of predictions for each image. The pred_variables then contain the prediction of a single image, randomly drawn from the batch.

Listing 4.12: Drawing a random image from the data and create predictions

```
random_index = np.random.randint(len(images))

#prediction on clean and aversarial datasets
predictions_clean = model.predict(images)
predictions_df = model.predict(clipped_df)
...

#prediction of clean and adversarial images
pred_clean = np.argmax(predictions_clean[random_index])
pred_fgsm = np.argmax(predictions_fgsm[random_index])
...
```

By subtracting the clean image from the adversarial one, the perturbations $r$ can be visualized.

Listing 4.13: Perturbations only

```
r_df = tf.subtract(clipped_fgsm[random_index],test_images[random_index
    ])
...
```

The images can be plotted by reshaping them to fit into the plot. Initially, the images are tensors are 3 dimensional with the format (28,28,1), which cannot be plotted in a 2 dimensional plot. For plotting, matplotlib is used, a library for graph, and image visualization in Python.

Listing 4.14: Reshaping and plotting images

```
#Reshaping to fit into plot dimensions
plot_image_clean = np.reshape(clean_images[random_index], (28,28))
```

Library and tutorials available at https://github.com/bethgelab/foolbox

```
r_df = np.reshape(r_df, (28,28))
...

f, axarr = plt.subplots(2,4)
axarr[0][0].imshow(plot_image_clean, cmap='gray')
...

axarr[1][1].imshow(r_df, cmap='gray')
...

axarr[0][0].title.set_text("Clean: %i" % pred_clean)
...

plt.show()
```

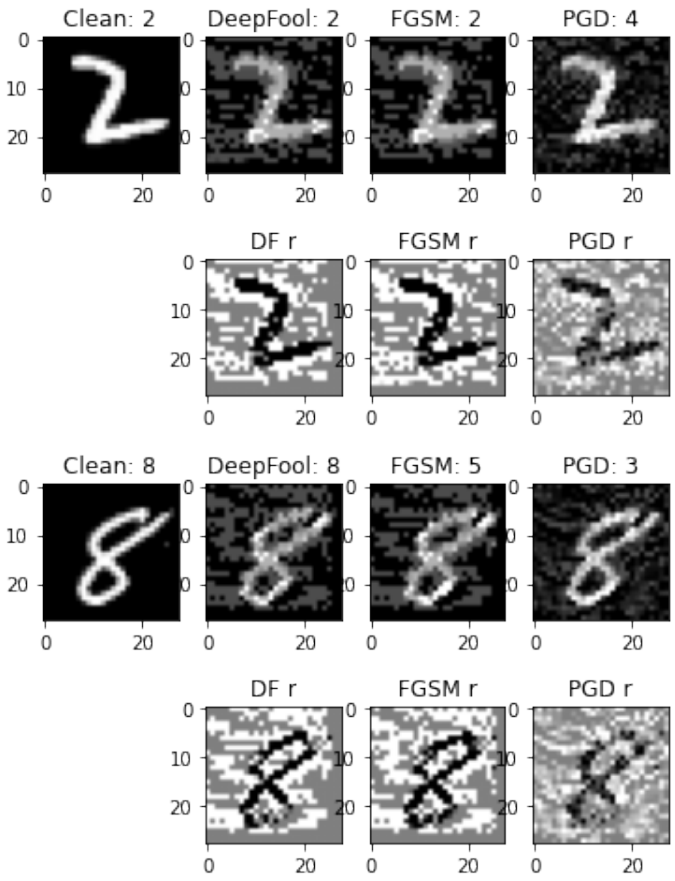



Figure 4.1: 2 examples of the plot above.

Library and tutorials available at `https://github.com/bethgelab/foolbox`

### 4.3.4 Multiple Epsilons

Instead of just assigning a single value to epsilon, the attack function can also take an array of epsilons as input. The attack then outputs image tensors for every value of epsilon. The following code shows how 5 values of epsilon are created in a spectrum between 0.0 and 0.3. The array is given to the attacks as input which they use to create the raw, clipped, and Boolean tensors.

To plot the overall accuracy of the model on the adversarial examples, the Boolean array can be used since it already gives information about whether the input was correctly classified or not. By taking the mean of the Booleans for every epsilon, the accuracy can be calculated.

Listing 4.15: Generating adversarial examples with multiple epsilon, calculating the accuracy for each epsilon and plotting the accuracies.

```
epsilons = np.linspace(0.0, 0.3, num=5)

raw_fgsm, clipped_fgsm, is_adv_fgsm = attack_fgsm(fmodel, test_images,
test_labels, epsilons=epsilons)
...

robust_accuracy_fgsm = 1 - is_adv_fgsm.float32().mean(axis=-1)
...

fig, axs = plt.subplots(1,3)
axs[0].plot(epsilons, robust_accuracy_fgsm.numpy())
...

axs[0].title.set_text("FGSM")
...
```

Library and tutorials available at `https://github.com/bethgelab/foolbox`

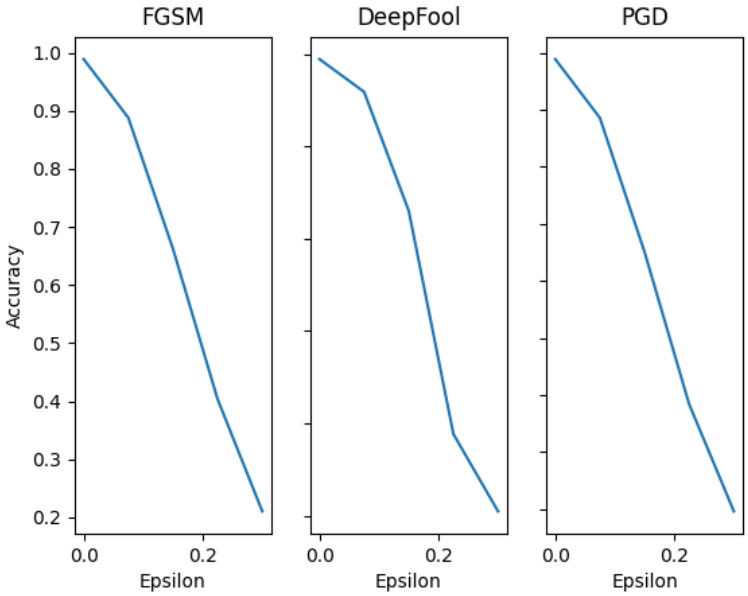



Figure 4.2: Accuracies for multiple epsilon of different attack methods.

### 4.3.5 Summary

Foolbox offers many utility functions to plot adversarial examples and to visualize perturbations, however, they could not be applied in this scenario because these functions did not support the used data format. The images require a lot preprocessing at first to apply the attack methods and an alternative Foolbox classifier is required. Though they could not be applied, the ability to plot the perturbations themselves, and to plot the robust accuracy for multiple epsilon is unique among the libraries. As of yet, Foolbox does not offer any implementations of known adversarial defenses.

Library and tutorials available at `https://github.com/bethgelab/foolbox`

## 4.4 AdvBox

AdvBox is a library that offers solutions for model benchmarking, without having to do any coding. The developers have written scripts that manage to benchmark models, just by executing a couple of command-line prompts. Other than previously introduced libraries, AdvBox uses PaddlePaddle a by Baidu developed deep learning framework. Although it is built on PaddlePaddle, it supports other frameworks like Tensorflow/ Keras and Pytorch. The library was released in 2018 but due to lacking support, a live functioning example cannot be presented. The developers present in several tutorials, however, how to generate a model with one of their scripts and then run an attack algorithm on it [Goo+20].

Installation: git clone https://github.com/baidu/AdvBox.git pip install -r requirements.txt from within the package directory. The library could not be tested, however, since the installation process kept failing due to outdated package and dependency errors. The setup and requirements documents that are needed for the installation were last updated 3 years ago.

### 4.4.1 Attacks with AdvBox

The first script generates, trains, and saves the model parameters to a certain directory.

Listing 4.16: Executing the model script in the command-line

```
python mnist_model.py
```

After that process is done, the attack script accesses the parameters and generates the adversarial examples.

Listing 4.17: Executing the attack script in the command-line

```
python mnist_tutorial_fgsm.py

Output:
attack success, original_label=4, adversarial_label=9, count=498
attack success, original_label=8, adversarial_label=3, count=499
attack success, original_label=6, adversarial_label=1, count=500
[TEST_DATASET]: fooling_count=394, total_count=500, fooling_rate
   =0.788000
fgsm attack done
```

Library and tutorials available at `https://github.com/advboxes/AdvBox`

The attack configuration can be changed in the attack script itself. After the attack is done, the command line prompts a report of the attack. The attack configuration can be changed in the attack script itself. After the attack is done, the command line prompts a report of the attack.

### 4.4.2 Conclusion

The approach of zero coding examples through the command-line is unique among the presented libraries, but the scripts and function often lack elaboration or only offer explanations in Chinese, which makes it difficult to create own implementations. Unfortunately, the library is not under development anymore and could therefore not be tested thoroughly due to failed package installations.

Library and tutorials available at `https://github.com/advboxes/AdvBox`

## 4.5 CleverHans

The CleverHans library is written and developed by some of the most notable authors of adversarial examples-based papers like Ian Goodfellow, Nicolas Papernot, or Nicholas Carlini. It contains several reference methods to create adversarial examples and benchmark models based on their training performance. The developers promise that by benchmarking models with their methods, high-accuracy statements about robustness can be made. The initial release by Ian Goodfellow in September 2016 only contained the fast gradient sign method. The library has since been under development with the most recent release, cleverhans 4.0. In the 4.0 version, CleverHans supports Tensorflow, JAX, and PyTorch while the attack methods for PyTorch have development priority. Installation: !pip install cleverhans [Pap+18].

### 4.5.1 Attacks with CleverHans

The following code chooses a random image from the test set. At first, the original label will be predicted by the model. The same image is then chosen to create an adversarial example by the fast gradient sign method and projected gradient descent. The methods require the model, the original image, epsilon, and the vector norm as inputs. Projected gradient descent additionally requires the step size and the number of steps performed. The clean image and adversarial examples are being compared in a plot with their corresponding labels.

Listing 4.18: Generating an adversarial example and plotting it

```
from cleverhans.tf2.attacks.fast_gradient_method import
fast_gradient_method
from cleverhans.tf2.attacks.projected_gradient_descent import
projected_gradient_descent

random_index = np.random.randint(10000)
image = test_images[random_index]

#pre-processing
image = np.expand_dims(image, axis=0)
pred_clean = model.predict(image)

#Create adversarial examples
x_pgd = projected_gradient_descent(model, image, 0.3, 0.01, 40, np.inf)
```

Library and tutorials available at https://github.com/cleverhans-lab/cleverhans

```
x_fgm = fast_gradient_method(model, image, 0.3, np.inf)

#prediction vectors
pred_fgsm = model(x_fgm)
pred_pgd = model(x_pgd)

#Create plot
f, axarr = plt.subplots(1,3)
axarr[0].imshow(np.reshape(image,(28,28)), cmap='gray')
...

axarr[0].title.set_text("Clean: %i" % np.argmax(pred_clean))
...
plt.show()
```

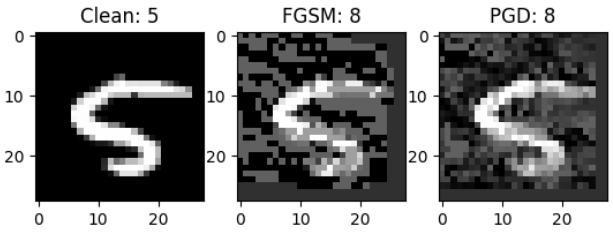



Figure 4.3: Output of above code example.

### 4.5.2 Defense with CleverHans

In their tutorials, the developers also offer solutions for how to implement adversarial training in Tensorflow. Since the training process is directly influenced by this procedure, native Tensorflow is used instead of the high-level API Keras. In the following code example, a network with the same layer architecture as the previously shown Keras model is trained on the MNIST dataset. During the training, however, the clean image

Library and tutorials available at https://github.com/cleverhans-lab/cleverhans

is swapped out with an adversarial example generated by projected gradient descent. The model is then evaluated on adversarial examples generated by the fast gradient sign method.

The model which is to fool is received from a custom-built function returning a sequentially created TensorFlow model. The following code snippet shows how the model is built. At first, the layers are being defined with their parameters and finally, the model is put together by the call method which returns it.

Listing 4.19: Building a model in TensorFlow

```
class Net(Model):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = Conv2D(32, (3, 3), activation="relu")
        ...
        self.flatten = Flatten()
        self.dense1 = Dense(64, activation="relu")
        self.dense2 = Dense(10)

    def call(self, x):
        x = self.conv1(x)
        ...
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)
```

After the model is defined, it needs to be trained on adversarial data. In the following, the train_step method is used to execute a single training iteration with a certain input image x and a label y. This method is essentially a way to customize the model.fit() function from the earlier Keras example. On a single training sample, the model predicts a vector of class probabilities and calculates the loss. It then computes all the gradients automatically using auto differentiation or autodiff. Auto differentiation is being implemented by using the gradient tape function. Finally, the function runs one step of gradient descent by applying the gradient.

While training epoch by epoch, the clean training examples x are being replaced with perturbed ones but the labels y remain true. Instead of a clean example, the train step function is now supplied with an adversarial example and performs gradient descent based on that. The following code snippet shows how the fast_gradient_method() function perturbs each training sample before it is used in the train step function. While

Library and tutorials available at https://github.com/cleverhans-lab/cleverhans

x is perturbed, y remains true which trains the model, to classify adversarial examples created by the fast gradient sign method correctly.

Listing 4.20: Adversarial training a model in TensorFlow

```
@tf.function
def train_step(x, y):
    #Record operations run during the forward pass
    with tf.GradientTape() as tape:
        predictions = model(x)
        loss = loss_object(y, predictions)

    #Gradient tape to automatically calculate gradients in respect to
       the
    #loss and trainable variables
    gradients = tape.gradient(loss, model.trainable_variables)

    #One step of gradient descent
    optimizer.apply_gradients(zip(gradients, model.trainable_variables)
       )
    train_loss(loss)

for epoch in range(4 #Custom chosen number of epochs):
    for (x, y) in data.train:
        #Replace clean example with perturbed one
        x =  fast_gradient_method(model, x, 0.3, np.inf)
        train_step(x, y)
```

Finally, the model can be evaluated on clean and adversarial examples. For each input $x$ and label $y$ in the test dataset, an adversarial example is created by using the fast_gradient_method function. The functions test_acc_clean, test_acc_fgsm, and test_acc_pgd are calculating the corresponding accuracies.

Listing 4.21: Computing accuracies based on adversarial examples crafted with the testing set

```
test_acc_clean = tf.metrics.SparseCategoricalAccuracy()
test_acc_fgsm = tf.metrics.SparseCategoricalAccuracy()
test_acc_pgd = tf.metrics.SparseCategoricalAccuracy()

for x, y in data.test:
```

Library and tutorials available at https://github.com/cleverhans-lab/cleverhans, Auto differentiation tutorial available at https://www.tensorflow.org/guide/autodiff, Model training tutorial available at https://www.tensorflow.org/guide/basic_training_loops

```
    y_pred = model(x)
    test_acc_clean(y, y_pred)

    x_fgm = fast_gradient_method(model, x, 0.3, np.inf)
    y_pred_fgm = model(x_fgm)
    test_acc_fgsm(y, y_pred_fgm)

    x_pgd = projected_gradient_descent(model, x, 0.3, 0.01, 40, np.inf)
    y_pred_pgd = model(x_pgd)
    test_acc_pgd(y, y_pred_pgd)


Output:
Clean acc: 98.02999877929688
FGSM acc:  89.33000183105469
PGD acc:  1.7100000381469727
```

The result shows that adversarial training based on the fast gradient sign method, helps to defend against its own adversarial examples but does not generalize well when trying to defend against other types of attacks as stated in [Mad+18, P. 10].

### 4.5.3 Summary

CleverHans offers basic implementations of the most popular adversarial attacks. Although implementing adversarial examples with CleverHans does not require many lines of code, the functions coming with CleverHans often require a lot of reshaping and pre-processing just to plot adversarial examples and to fit the clean samples into the attack functions. There are no set functions for adversarial defenses but the developers have code examples of an implementation for adversarial training. Besides the attack functions, no utility functions are given to plot images or calculate accuracies.

Library and tutorials available at https://github.com/cleverhans-lab/cleverhans

## 4.6 Adversarial Robustness Toolbox(ART)

ART, short for Adversarial-Robustness-Toolbox offers a wide range of adversarial attacks and defenses for developers to use. Other than the libraries presented thus far, they also offer solutions to create adversarial examples of other data types besides images, such as audio, video, etc. The library is mainly maintained by an IBM research team. The full list of contributors can be found in their papers or on GitHub [Nic+18]. Among all presented libraries, ART offers the largest amount of attacks and defenses due to the diversity of media to which their attacks can be applied. ART was initially released in January 2019 with popular attacks like the fast gradient sign method, Carlini & Wagner, Universal adversarial perturbations, and more [Nic+18]. Installation: !pip install adversarial-robustness-toolbox

### 4.6.1 Prerequisites

The adversarial examples will be created based on the testing data for the MNIST model. Therefore, the training and testing data needs to be reimported and preprocessed as it has been done in the preprocessing section of the victim model. Like FoolBox, ART requires its own type of classifier with specified image bounds, and therefore needs a wrapper class. That classifier is then used to create adversarial examples.

Listing 4.22: Calculating accuracies

```
from art.estimators.classification import KerasClassifier

model = keras.models.load_model('path_to_model/CNN_model.h5')
classifier = KerasClassifier(model=model,use_logits=False)
```

### 4.6.2 Attacks with ART

Similar to FoolBox, attack objects are required. They take the attack parameters, such as epsilon, and the classifier as input. The testing dataset is used to generate adversarial examples. As before, DeepFool, FGSM, and PGD are the attack methods. Adversarial examples can then be generated by calling the generate function of the respective attack and giving the images as input.

Library and tutorials available at `https://github.com/Trusted-AI/adversarial-robustness-toolbox`

Listing 4.23: Generating adversarial examples

```
from art.attacks.evasion import DeepFool
from art.attacks.evasion import FastGradientMethod
from art.attacks.evasion import ProjectedGradientDescent

images = test_images
df = DeepFool(classifier, epsilon=0.3)
fgsm = FastGradientMethod(classifier, eps = 0.3)
pgd = ProjectedGradientDescent(classifier, eps = 0.3 )

df_images = df.generate(x=images)
fgsm_images = fgsm.generate(x=images)
pgd_images = pgd.generate(x=images)
```

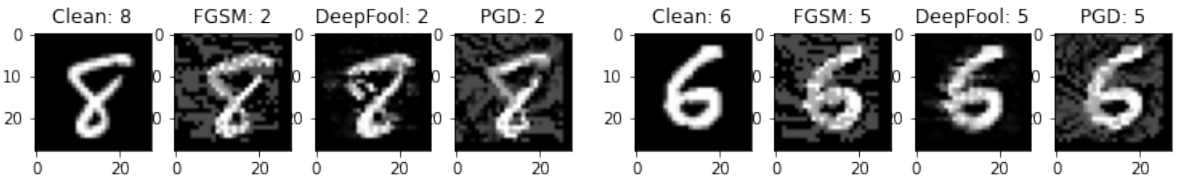
The Keras evaluate function can be used to calculate the accuracies of the attacks by calling it from the model.

Listing 4.24: Evaluating the model on the different data distributions

```
test_loss_fgsm, test_acc_fgsm = model.evaluate(fgsm_images,test_labels)
test_loss_df, test_acc_df = model.evaluate(df_images, test_labels)
test_loss_pgd, test_acc_pgd = model.evaluate(pgd_images, test_labels)

Output:
FGSM accuracy: 18.2999...%
PGD accuracy: 0.78999...%
DeepFool accuracy: 14.0300...%
```

To visualize the results, a random image has been drawn from each data distribution. The number on top shows the predicted class. The perturbations created by DeepFool are smaller than the ones created by the other 2 methods but still cause misclassifications. Perturbations created by FGSM are rougher compared to PGD and in one of the examples above, PGD manages to perturb the image in a way that the prediction is further off the actual target.




Library and tutorials available at `https://github.com/Trusted-AI/adversarial-robustness-toolbox`

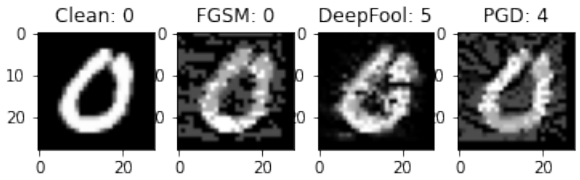



Figure 4.4: ART examples

### 4.6.3 Defense with ART

For adversarial training, the authors propose a slightly different approach than the previously shown example in CleverHans. Rather than swapping out every clean image with an adversarial example during training, an adversarial copy of the original training set is created, which contains adversarial examples based on all original training images. For the adversarial dataset, the fast gradient sign method is used. The adversarial training set is then appended to the end original one. Both the adversarial training set and the original one are of the same length. Since the indexes of the train images correspond to the indexes of the train labels, the labels are copied and appended at the end of the original label set, to create a list twice as long to make sure that the indexes of the adversarial train images correspond to the correct labels as well. The new training set can then be used to re-train the network on FGSM examples. The batch size and number of epochs are the same as in the original training process.

Listing 4.25: Train a model on an adversarial dataset

```
fgsm_train_images = fgsm.generate(train_images)
train_images_adv = np.append(train_images, pgd_images, axis=0)
train_labels_adv = np.append(train_labels, train_labels, axis=0)

classifier.fit(train_images_adv, train_labels_adv, batch_size=38,
    nb_epochs=4)

test_loss_adv, test_acc_adv = model.evaluate(x_test_adv, y_test)
print("FGSM␣accuracy:␣", test_acc_fgsm*100, "%" )

Output:
FGSM accuracy:  97.600...%
```

Library and tutorials available at `https://github.com/Trusted-AI/adversarial-robustness-toolbox`

ART is the only library of all presented to offer other defensive mechanisms than adversarial training like defensive distillation. To execute the defensive distillation mechanism, very much like the attack, a defense object has to be initiated. When given a training dataset and the original classifier to the defense object, it retrains the network and returns a more robust version of it. The function, therefore, requires a trained classifier, the batch size, and the number of epochs as parameters. The exact procedure is explained in the defensive distillation section above. Upon executing, the model will be retrained.

Listing 4.26: Train a model on an adversarial dataset

```
from art.defences.transformer.evasion import DefensiveDistillation

defense = DefensiveDistillation(model, batch_size=32, nb_epochs=8)
def_classifier = defense(x_train, classifier)
```

By default, the model has an 18% accuracy on FGSM test samples, but when defensive distillation was applied, the accuracy rose to 36%. Though the results are not as significant compared to adversarial training, defensive distillation does not require entirely new generated data to increase the accuracy of the model.

### 4.6.4 Summary

Out of all libraries, ART offers the most attack methods. It does not offer utility functions for plotting or accuracy but the attacks require very few lines of coding or preprocessing. Implementations in Keras are easy to write as well. ART does also offer attack strategies for other types of media such as video or mp3. Different from the other presented libraries, ART has many defensive mechanisms to protect models from adversarial examples.

Library and tutorials available at `https://github.com/Trusted-AI/adversarial-robustness-toolbox`

## 4.7 Comparison of the Libraries

At first, the libraries are being compared based on the attack and defense methods introduced. The following table will give an overview, which attacks and defenses are available for each library and which deep learning framework they support. An attack or defense only accounts as available, if it was applicable to the victim model.

| Method | FoolBox | AdvBox | ART | CleverHans |
|---|---|---|---|---|
| | | | | |
| Attacks | | | | |
| Fast Gradient Sign Method | ✓ | ✓ | ✓ | ✓ |
| Projected Gradient Descent | ✓ | ✓ | ✓ | ✓ |
| Carlini Wagner | ✓ | | ✓ | ✓ |
| DeepFool | ✓ | ✓ | ✓ | |
| Universal Adversarial Perturbation | | | ✓ | |
| One Pixel Attack | | | ✓ | |
| Basic Iterative Attack | | | ✓ | ✓ |
| LBFG-S Attack | | ✓ | | |
| | | | | |
| Defenses | | | | |
| Adversarial Training | | | ✓ | ✓ |
| Defensive Distillation | | | ✓ | |
| Flag Adversarial examples | | | | |
| | | | | |
| Supported Frameworks | | | | |
| Tensorflow | ✓ | ✓ | ✓ | ✓ |
| Keras | ✓ | ✓ | ✓ | ✓ |
| PyTorch | ✓ | ✓ | ✓ | ✓ |
| Scikit-learn | | | ✓ | |
| XGBoost | | | ✓ | |
| LightGBM | | | ✓ | |
| CatBoost | | | ✓ | |
| Gpy | | | ✓ | |
| PaddlePaddle | | ✓ | | |
| Caffe2 | | ✓ | | |
| MxNet | | ✓ | | |
| JAX | ✓ | | | ✓ |

In the following, the libraries are put into comparison. The metrics are required preprocessing, user support, utility functions for plot and accuracy, number of supported attack/ defense mechanisms, and whether the library is still under active development. Customer support and utility functions are graded between 1-5, with 1 being the worst

possible score and 5 the highest. Complexity follows a similar format. A score of 1 means that the libraries require little prior Python knowledge, while a score of 5 means that more advanced Python programming knowledge is required.

| Name | Complexity | Support | Utility | Attacks | Defenses | Active development |
|---|---|---|---|---|---|---|
| Foolbox | 4 | 3 | 4 | 26 | 0 | Yes |
| CleverHans | 3 | 5 | 1 | 7 | 1 | Yes |
| AdvBox | | 1 | | 6 | 5 | No |
| ART | 2 | 5 | 2 | 50+ | 29 | Yes |

FoolBox and CleverHans are best used for quickly generating adversarial examples. If the user just wants to quickly visualize adversarial results, these libraries are the best fit. It has to be noted that CleverHans and FoolBox require the most pre-existing Python programming knowledge to be used efficiently. This is due to the amount of preprocessing which needs to be done beforehand. For visualisation however, FoolBox is best used, since it offers the most utility functions out of all libraries. For basic benchmarking, CleverHans has the best implementations.

Due to technical difficulties, AdvBox could not be compared to the rest of the libraries in terms of attacking the victim model but was included regardless due to the unique approach to benchmarking Deep Learning models over the command line.

For advanced benchmarking on multiple adversarial attacks and defending models, ART is recommended since it offers the most attack and defense options. ART also supports the most Deep Learning frameworks which makes it somewhat universally applicable and up to the task. Due to the large number of attacks, defenses and supported frameworks, as well as the simplicity of usage, it is rated the highest among all libraries presented.

# 5 Conclusion

In this bachelor thesis, an overview of different adversarial attacks and defenses was given. To establish the basic knowledge required, machine learning and Deep Learning were described beforehand on a high-level. Eventually, some of the introduced attacks were implemented using different Python libraries. The libraries were put into comparison to determine, which one is best suited to attack and defend a deep learning model from adversarial examples. While working with these libraries, technical difficulties have been met, but nonetheless, it was possible to attack or defend a deep learning model with most of them. This thesis is therefore deemed successful since the victim model could be fooled with the majority of the presented libraries while creating an overview, which library is best suited for different tasks.

Even promising defense mechanisms like defensive distillation keep getting broken by new methods to generate adversarial examples, which raises the question if it is even possible to make a model resistant to adversarial examples. The fact that deep learning models are strongly susceptible to adversarial attacks shows that this field demands more research for the sake of machine learning security. Due to the fast advancement of Deep Learning applications for real world problems such as automated driving, security should have the utmost priority.

# Bibliography


[20] *Anaconda Software Distribution*. Version Vers. 2-2.4.0. 2020. URL: https://docs.anaconda.com/.

[AM18] Naveed Akhtar and A. Mian. "Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey". In: *IEEE Access* 6 (2018), pp. 14410–14430.

[Bra+18] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[Car19a] Nicholas Carlini. *Attacking Machine Learning: On the Security and Privacy of Neural Networks. RSA, 2019*. 2019. URL: https://nicholas.carlini.com/talks.

[Car19b] Nicholas Carlini. *Lessons Learned from Evaluating the Robustness of Neural Networks to Adversarial Examples. USENIX Security (invited talk), 2019*. 2019. URL: https://nicholas.carlini.com/talks.

[Car19c] Nicholas Carlini. *On Evaluating Adversarial Robustness CAMLIS (keynote), 2019*. 2019. URL: https://nicholas.carlini.com/talks.

[Cho+15] François Chollet et al. *Keras*. https://keras.io. 2015.

[Cho17] François Chollet. *Deep Learning with Python*. Manning, Nov. 2017. ISBN: 9781617294433.

[CW17] Nicholas Carlini and David A. Wagner. "Towards Evaluating the Robustness of Neural Networks". In: *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 39–57.

[Gér17] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 978-1491962299.

[Goe20] Ayush Goel. "An Empirical Review of Adversarial Defenses". In: *ArXiv* abs/2012.06332 (2020).

[Goo+17] Ian Goodfellow et al. *Attacking Machine Learning with Adversarial Examples*. Feb. 2017. URL: https://openai.com/blog/adversarial-example-research/.

[Goo+20] Dou Goodman et al. *Advbox: a toolbox to generate adversarial examples that fool neural networks*. 2020. arXiv: 2001.05574 [cs.LG].

[GSS15] I. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *CoRR* abs/1412.6572 (2015).

[Han20] Paul Hand. *Adversarial Examples for Deep Neural Networks*. June 2020. URL: https://khoury.northeastern.edu/home/hand/teaching/cs7150-summer-2020/index.html.

[KGB17] Alexey Kurakin, I. Goodfellow, and S. Bengio. "Adversarial Machine Learning at Scale". In: *ArXiv* abs/1611.01236 (2017).

[Kna19] Oscar Knagg. *Know your enemy How you can create and defend against adversarial attacks*. Jan. 2019. URL: https://towardsdatascience.com/know-your-enemy-7f7c5038bdf3.

[Mad+18] A. Madry et al. "Towards Deep Learning Models Resistant to Adversarial Attacks". In: *ArXiv* abs/1706.06083 (2018).

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[MFF16] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and P. Frossard. "DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 2574–2582.

[Moo+17] Seyed-Mohsen Moosavi-Dezfooli et al. "Universal Adversarial Perturbations". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 86–94.

[Nic+18] Maria-Irina Nicolae et al. "Adversarial Robustness Toolbox v1.2.0". In: *CoRR* 1807.01069 (2018). URL: https://arxiv.org/pdf/1807.01069.

[Pap+16] Nicolas Papernot et al. "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks". In: *2016 IEEE Symposium on Security and Privacy (SP)* (2016), pp. 582–597.

[Pap+17] Nicolas Papernot et al. "Practical Black-Box Attacks against Machine Learning". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017).

[Pap+18] Nicolas Papernot et al. "Technical Report on the CleverHans v2.1.0 Adversarial Examples Library". In: *arXiv preprint arXiv:1610.00768* (2018).

[Rau+20] Jonas Rauber et al. "Foolbox Native: Fast adversarial attacks to benchmark the robustness of machine learning models in PyTorch, TensorFlow, and JAX". In: *Journal of Open Source Software* 5.53 (2020), p. 2607. DOI: 10.21105/joss.02607. URL: https://doi.org/10.21105/joss.02607.

[Sar+17] Sayantan Sarkar et al. “UPSET and ANGRI : Breaking High Performance Image Classifiers”. In: *ArXiv* abs/1707.01159 (2017).

[SVS19] Jiawei Su, Danilo Vasconcellos Vargas, and K. Sakurai. “One Pixel Attack for Fooling Deep Neural Networks”. In: *IEEE Transactions on Evolutionary Computation* 23 (2019), pp. 828–841.

[Sze+14] Christian Szegedy et al. “Intriguing properties of neural networks”. In: *CoRR* abs/1312.6199 (2014).

[Yua+19] Xiaoyong Yuan et al. “Adversarial Examples: Attacks and Defenses for Deep Learning”. In: *IEEE Transactions on Neural Networks and Learning Systems* 30 (2019), pp. 2805–2824.