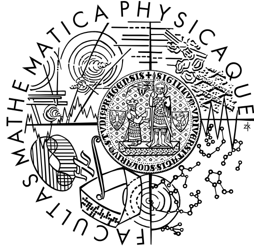

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Bc. Matěj Kocián

Adversarial Examples in Machine Learning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Informatics
Study branch: Artificial Intelligence

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

I would like to thank my supervisor, Mgr. Martin Pilát, Ph.D., for his kindness, valuable advice and encouragement. I would also like to thank my family and friends for their lasting support and for reading through drafts of this thesis and offering helpful comments.

Title: Adversarial Examples in Machine Learning

Author: Bc. Matěj Kocián

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Deep neural networks have been recently achieving high accuracy on many important tasks, most notably image classification. However, these models are not robust to slightly perturbed inputs known as adversarial examples. These can severely decrease the accuracy and thus endanger systems where such machine learning models are employed. We present a review of adversarial examples literature. Then we propose new defenses against adversarial examples: a network combining RBF units with convolution, which we evaluate on MNIST and get better accuracy than with an adversarially trained CNN, and input space discretization, which we evaluate on MNIST and ImageNet and obtain promising results. Finally, we explore a way of generating adversarial perturbation without access to the input to be perturbed.

Keywords: adversarial examples, machine learning, neural networks

Contents

Introduction **3**

1 Background **5**

1.1 Machine learning concepts . . . 5
1.1.1 Image classification . . . 6
1.2 Neural networks . . . 7
1.2.1 Training of neural networks . . . 10
1.2.2 Logistic regression . . . 13
1.2.3 Convolutional networks . . . 14
1.2.4 Radial Basis Function networks . . . 15
1.2.5 Autoencoder . . . 16
1.3 Other models . . . 16
1.3.1 Decision tree . . . 16
1.3.2 K-means . . . 17
1.3.3 Support Vector Machines . . . 17
1.3.4 Ensemble . . . 17

2 Adversarial Examples **18**

2.1 Adversarial examples generation . . . 18
2.1.1 Optimization-based methods . . . 19
2.1.2 Fast Gradient (Sign) Method . . . 20
2.1.3 Iterative FGSM . . . 21
2.1.4 Papernot's method . . . 22
2.2 Properties . . . 22
2.2.1 Transferability . . . 22
2.2.2 Robustness to transformations . . . 24
2.2.3 Detectability . . . 25
2.3 Defenses . . . 25
2.3.1 Adversarial training . . . 26
2.3.2 Denoising autoencoders and DCN . . . 28
2.3.3 Simple image transformations . . . 28
2.3.4 Gradient masking . . . 29
2.4 Black-box attacks on ML models . . . 29
2.4.1 Evolutionary generation of adversarial examples . . . 29
2.4.2 Substitution attacks . . . 31
2.4.3 Targeted substitution attacks . . . 32
2.5 Related research . . . 32
2.5.1 Fooling examples . . . 33
2.5.2 Generative Adversarial Networks . . . 33
2.6 Conclusion . . . 34

3 In search for new defenses and attacks **36**

3.1 RBF models on MNIST . . . 36
3.1.1 Method . . . 36
3.1.2 Results and discussion . . . 38

3.2 The effects of rounding on MNIST . . . 38
3.2.1 Method . . . 39
3.2.2 Results and discussion . . . 39
3.3 The effects of rounding on ImageNet . . . 43
3.3.1 Method . . . 43
3.3.2 Results and discussion . . . 43
3.4 Common adversarial perturbation . . . 53
3.4.1 Results and discussion . . . 53

Conclusion **55**

Bibliography **56**

List of Figures **61**

List of Abbreviations **62**

Introduction

Teaching computers to learn from experience has always been of great interest in artificial intelligence research. Known by the name of machine learning, the field has recently been successful (and achieved mainstream popularity) by reaching high, in some cases almost human-like accuracy on tasks ranging from computer vision to machine translation. This allowed many engineering problems to be solved and found applications in multiple industries.

However, Szegedy et al. [2013] showed that most of these very well performing models have easily exploitable weak spots. By changing the input data slightly – so slightly it would not affect the decision of a human classifying these data, and maybe even escape their attention – a potential attacker might be able to completely change the model's response. Such maliciously perturbed inputs are called adversarial examples.

If deployed against real world systems that use machine learning, such as autonomous cars, content filters or authentication systems, adversarial examples could endanger security of the users or grant the attacker improper access. It is therefore important to find ways to make machine learning models robust to adversarial perturbation, and also to discover new ways of creating adversarial examples, in order to be able to test our defenses against them.

It is unlikely that the problem of adversarial examples will be completely solved anytime soon. A perfect defense would amount to having a perfect model that would understand its input as well as a human would. We are still very far from this goal, but hopefully we can at least make some steps in the right direction.

The goals of this thesis are the following:

- To do a review of existing literature about adversarial examples. The paper by Szegedy et al. [2013] sparked relatively big interest in this phenomenon and it was soon followed by many others. There are multiple variables that comprise the scenario of attack and defense and a review is needed to help us orient ourselves in the plenty of work that has been done to explore the task properties.

- To find new defenses against adversarial examples. Many approaches have already been tried, but none of them solve the problem completely. We aim to introduce defenses that would be simple and at least partially effective, with the hope that they could be expanded with future work or combined with other existing approaches for greater effectiveness.

- To explore new methods of adversarial examples generation. We need to know what the adversary is able to do in order to estimate how much our machine learning models could be affected and, hopefully, to devise a suitable defense.

The structure of the thesis is as follows.

In Chapter 1, we describe the general machine learning tasks and commonly used machine learning models. We focus on image classification, because it is the

task that adversarial examples are most usually studied on, and convolutional neural networks, which are the state-of-the-art model type for this task and other computer vision problems.

In Chapter 2, we present a review of adversarial examples literature. We describe methods generating adversarial examples, their properties and classification, as well as approaches that have been used to defend machine learning models against them.

In Chapter 3, we propose new ways to make image classification models more robust to adversarial perturbation and evaluate them on adversarial examples generated from standard datasets by commonly used methods. We also explore the possibility of generating adversarial examples without access to the original example.

1. Background

In this chapter we introduce basic machine learning concepts and several models that appear in the works on adversarial examples or are used in the third chapter. We describe the task of image classification, which is most commonly targeted by adversarial examples, as well as some of the datasets that image classification is studied on.

1.1 Machine learning concepts

The machine learning tasks are divided mainly into *supervised* and *unsupervised* learning.

Unsupervised learning is generally concerned with describing the distribution of the observable data, without predicting any unobserved values. In this text, we are mostly interested in supervised learning.

Supervised learning generally tries to construct a model of the conditional probability $\mathrm{P}(Y|X)$ from given data. The random variable X may be called an *input*, a *predictor*, an *independent variable* or *features*, whereas Y is known as an *output*, a *response* or a *dependent variable* [Friedman et al., 2001]. Only a finite dataset sampled from the distribution $\mathrm{P}(X, Y)$ is available. This dataset is split into two parts, the *training set* that is used to train the model, and the *test set* which is used to estimate the model's expected population error. Sometimes a *validation set* is taken from the training set to estimate model's error on unseen data during training.

If Y is quantitative, e.g. the height of a person, the supervised machine learning task of predicting this quantity is called *regression*, whereas if Y is categorical, e.g. representing species of the Iris genus, the task of predicting this category (also *label* or *class*) is called *classification*.

The quality of a model is measured by an *error function* (*loss*, *cost*) evaluated on the test set (during training, evaluation on the training set may be used). For regression, a popular error function is the *mean squared error*

$$\mathrm{MSE}(f, x, y) = \frac{1}{n} \sum_{i=1}^{n} \left\| f(x^{(i)}) - y^{(i)} \right\|_2^2,$$

where $x^{(i)}$ and $y^{(i)}$ are the individual observations of (X, Y) and $f(x^{(i)})$ is the value of Y that the model predicted. MSE can be used for classification, too, as $f(x^{(i)})$ is often represented as an *one-hot vector*, i.e. a vector with 1 for the predicted class and zeros elsewhere. However, the *mean cross-entropy*

$$\frac{1}{n} \sum_{i=1}^{n} H(y^{(i)}, f(x^{(i)})) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j} -y_j^{(i)} \log f(y^{(i)})_j$$

is usually preferred over MSE for classification [Golik et al., 2013]. Another useful measure is the *accuracy* – the rate of correctly classified inputs or, conversely, the *error rate* – the rate of incorrectly classified inputs.

1.1.1 Image classification

Image classification is just a specific kind of classification. The inputs have the form of matrices of pixel intensities in the case of grayscale images and matrices of channel intensity vectors (i.e. tensors) in the case of color images. The channels usually account for the red, green and blue component of the RGB color space. For simplicity, we often assume that the intensities are real numbers from the interval $[0, 1]$. However, in practice, they have to be discretized, with 256 possible values (corresponding to one byte) per pixel channel being a common choice.

Datasets

Publicly available datasets of labeled images are very valuable for computer vision research. They alleviate researchers from having to collect and label their own images, which is tedious or costly work. They also serve as a common benchmark for model comparison.

The *Modified National Institute of Standards and Technology* (MNIST) database of handwritten digits was created by LeCun et al. [1990] from the larger NIST database. They rescaled the black and white NIST images to 28×28 pixels, thus introducing grayscale levels (see Figure 1.1 for examples). They also mixed new training and test sets, with 60 000 and 10 000 images, respectively, approximately half of them coming from the original training set and half from the original test set. The mixing was done because the NIST training and test set digits were written by different groups of people, the former being more easily recognizable. In 1998, LeCun et al. [1998] obtained 0.7% error rate on MNIST and since than, the achieved error rate dropped as low as 0.21% [Wan et al., 2013].

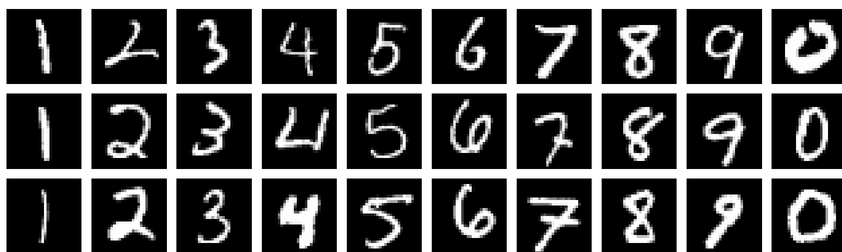

Figure 1.1: Examples of the MNIST digits.

The CIFAR10 and CIFAR100 datasets [Krizhevsky and Hinton, 2009] consist of 60 000 color images of size 32×32 pixels, divided into 10 and 100 classes, respectively. See Figure 1.2 for examples of the classes and corresponding images. The training and test set contain 50 000 and 10 000 images, respectively. The number of images belonging to each of the classes is the same. The state-of-the-art accuracy is about 0.95 on CIFAR10 [Springenberg et al., 2014] and about 0.72 on CIFAR 100 [Snoek et al., 2015].

The *ImageNet Large Scale Visual Recognition Challenge 2012* (ILSVRC 2012) image classification task dataset [Russakovsky et al., 2014] is much bigger in terms of the size and number of images as well as the number of classes. It is based on the *ImageNet* [Deng et al., 2009] – an image database organized according to the WordNet hierarchy [Miller, 1995]. ImageNet contains more than 14 million URLs of labeled images for more than 20 000 *synsets* (synonym sets) of WordNet.

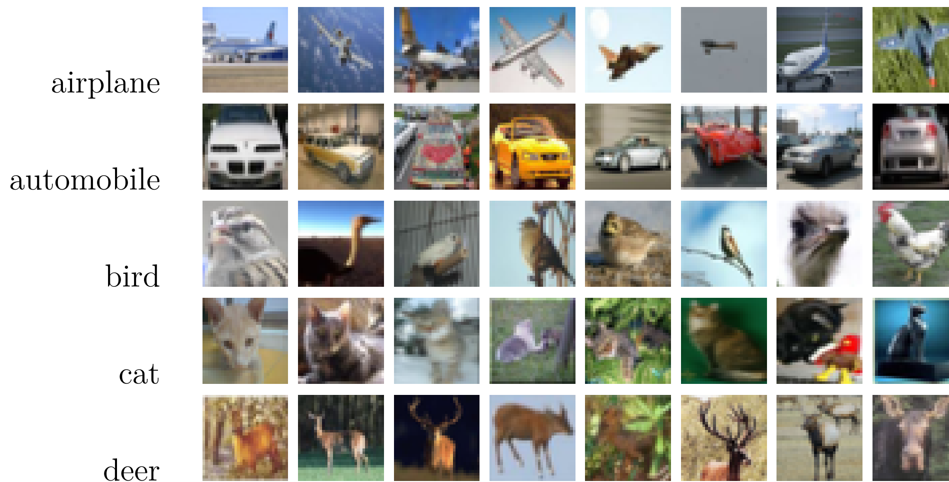

Figure 1.2: Examples of the CIFAR10/CIFAR100 images.

ILSVRC 2012 selected 1000 classes from the synsets such that no class is an ancestor of another class in the hierarchy of synsets. The width and height of the images vary, but generally they are in the order of hundreds of pixels. See Figure 1.3 for an example.

The ILSVRC 2012 training set contains more than 1.2 million images (at least 732 for each class) and the validation set contains 50 000 images (50 for each class). Their sizes are 138 GB and 6.3 GB, respectively. The test set of 100 000 images is not publicly available and is used during the annual competition to evaluate the submitted models.

The models are compared by measuring the top-5 error rate, i.e. the rate of images for which the correct class does not appear in the five most probable classes according to the model. This is done mainly because each image was labeled with only one "correct" class to keep the costs of labeling manageable, but there are often multiple objects in the images.

The competition has been running since 2010, but since 2012 the dataset has been kept the same. Krizhevsky et al. [2012] won the ILSVRC 2012 by a large margin of 9.8%, achieving 16.4% error rate. The subsequent editions of the challenge saw further decrease in top-5 error rates, going below 4% [He et al., 2016]. Note that Russakovsky et al. [2014] reported top-5 error of 5.1% for a human annotator, so trying to decrease the error of machine learning models even further may not be reasonable.

In the following sections we describe several machine learning models that have been targeted by adversarial examples in the literature. We focus on neural networks, which are most commonly targeted, probably because they have been achieving state-of-the-art results in image recognition for some time.

1.2 Neural networks

Neural networks are machine learning models consisting of connected nodes, each performing a simple computation based on its inputs, that together can represent much more complex relationships. Although originally inspired by networks of

monarch butterfly, . . .

Gordon setter

Border terrier

sax, saxophone

Australian terrier

potpie

toaster

kuvasz

passenger car, . . .

Figure 1.3: A random sample from the ILSVRC 2012 validation set. These nine images are all 500 pixel wide and 333 to 375 pixel high. The whole class name of the first one is "monarch, monarch butterfly, milkweed butterfly, Danaus plexippus" and of the last one "passenger car, coach, carriage".

interconnected neuron cells in human brain that gave them their name, generally they are designed for computational efficiency rather than biological plausibility. It can be more useful to think of them in terms of function composition.

The simplest case of a neural "network" is a *perceptron*, introduced by Rosenblatt [1958]. It consists of one node with d incoming connections (*dendrites*, *edges*) and one output (*axon*). The incoming connections have associated *weights* $w \in \mathbb{R}^d$ and *bias* (*threshold*) $b \in \mathbb{R}$. Input $x \in \mathbb{R}^d$ is classified as one of two classes depending on the output

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

See Figure 1.4 for an illustration of a perceptron.

The weights are fitted to the training data by the perceptron training algorithm [Rosenblatt, 1958].

Of course, this is a simple model, and can only learn to classify linearly separable data. To make it more capable, we could try to connect more nodes and

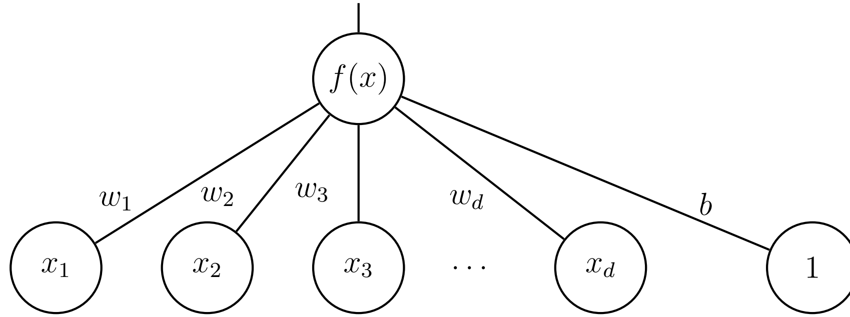

Figure 1.4: A perceptron with weights $w \in \mathbb{R}^d$ computing $w \cdot x + b$ for an input $x \in \mathbb{R}^d$. The elements of x can be thought of as nodes in an input layer and $w_1, \ldots, w_d$ as weights of the edges connecting the inputs to the perceptron. The bias b can be considered a weight of a connection from an input node which is always set to 1.

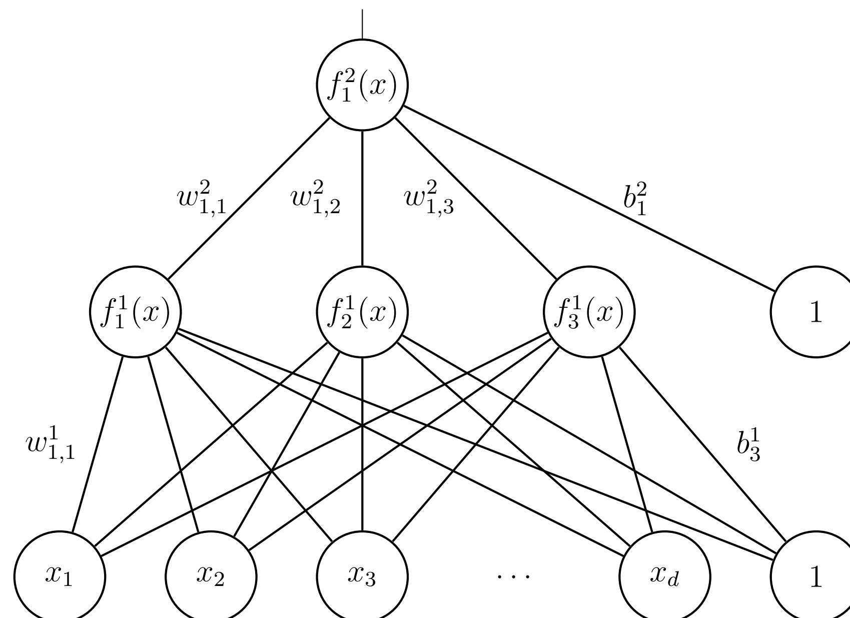

Figure 1.5: An example of a multilayer perceptron with one hidden layer (f_1^1, f_2^1, f_3^1). We omit most weight labels for space reasons.

create an actual neural network or a *multilayer perceptron* (MLP). Nodes from the second layer up would use the outputs of the previous layer as their input. The layers except for the last one (output) are called *hidden*, because we cannot observe their values when we use the network as a black box. However, without any further modifications, the MLP would be equivalent to a simple perceptron. Consider for example a two layer perceptron with three nodes f_1^1, f_2^1, f_3^1 in the first and one node f_1^2 in the second layer (see Figure 1.5).

Then the network's output is

$$
\begin{aligned}
f(x) &= f_1^2\left(\left(f_1^1(x), f_2^1(x), f_3^1(x)\right)\right) \\
&= w_1^2 \cdot \left(w_1^1 \cdot x + b_1^1, w_2^1 \cdot x + b_2^1, w_3^1 \cdot x + b_3^1\right) + b_1^2 \\
&= w_1^2 \cdot \left(\left(\sum_{i=1}^{d} w_{1,i}^1 x_i\right) + b_1^1, \left(\sum_{i=1}^{d} w_{2,i}^1 x_i\right) + b_2^1, \left(\sum_{i=1}^{d} w_{3,i}^1 x_i\right) + b_3^1\right) + b_1^2 \\
&= b_1^2 + \sum_{j=1}^{3} w_{1,j}^2 \left(b_j^1 + \sum_{i=1}^{d} w_{j,i}^1 x_i\right) \\
&= b_1^2 + \sum_{j=1}^{3} w_{1,j}^2 b_j^1 + \sum_{i=1}^{d} \sum_{j=1}^{3} w_{1,j}^2 w_{j,i}^1 x_i \\
&= \left(b_1^2 + \sum_{j=1}^{3} w_{1,j}^2 b_j^1\right) + \left(\sum_{j=1}^{3} w_{1,j}^2 w_j^1\right) \cdot x.
\end{aligned}
$$

This is equal to $b + w \cdot x$ for some $b \in \mathbb{R}$ and $w \in \mathbb{R}^d$, and hence the network is equivalent to a perceptron with weights w and bias b.

To make a neural network able to represent nonlinear functions, a nonlinear *activation function* is applied to the output of every node. The traditional activation function is the *logistic sigmoid*

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Other possibilities include the hyperbolic tangent or the rectified linear unit $\text{ReLU}(x) = \max(0, x)$, which has been popular recently [Goodfellow et al., 2016]. See the plot in Figure 1.6. Note that because neural networks are typically trained by gradient descent, the activation function should be differentiable at least almost everywhere.

With a non-constant, bounded and monotonically increasing activation function, neural networks can represent the approximation of any function $f : \mathbb{R}^d \to \mathbb{R}$ arbitrarily well, given they have (at least) two layers and enough nodes in the first layer [Hornik et al., 1989]. However, it took some time before an efficient algorithm for their training was devised.

1.2.1 Training of neural networks

When training neural networks, we are trying to minimize their error on training data (loss function). This is commonly done by using gradient descent – repeatedly computing the gradient of the loss function with respect to the network's weights, scaling it by a learning rate constant and subtracting it from the weights. Note that the loss function for a neural network is usually quite complex and non-convex, with a lot of local minima, and therefore gradient descent is not guaranteed to find the global minimum. Nevertheless, the method usually works well in practice.

Differentiating the loss function with respect to one weight is a relatively straightforward application of the chain rule of calculus:

$$\frac{\partial f(g(x))}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial f(g(x))}{\partial g(x)_j} \frac{\partial g(x)_j}{\partial x_i},$$

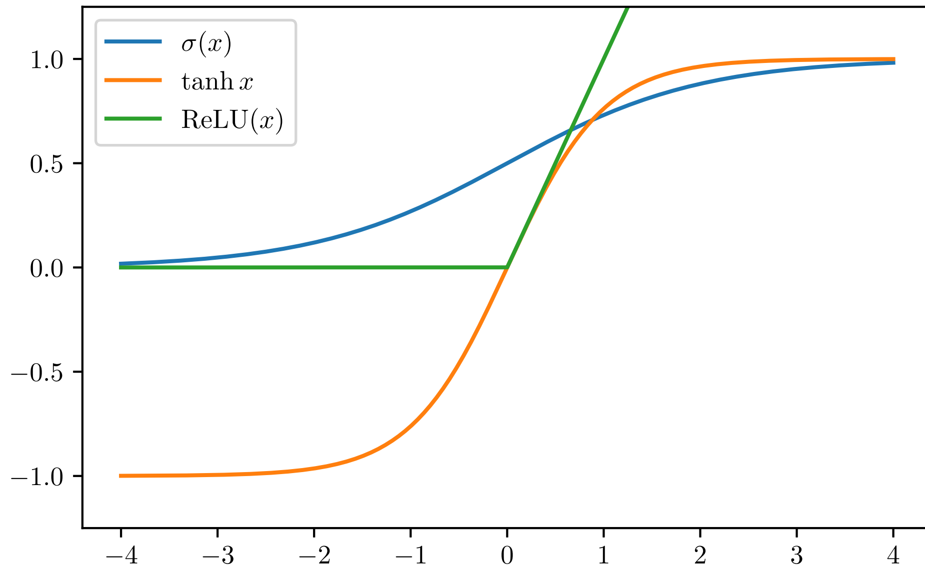

Figure 1.6: Plots of the three possibly most frequently used activation functions. Note how σ and tanh are almost constant (and therefore have gradient of almost zero) for most of $\mathbb{R}$.

for $g : \mathbb{R}^d \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. It can be computationally inefficient if done naively, though, as many shared subexpressions can arise that would be computed exponentially many times. A fast algorithm for gradient computation in neural networks is called *backpropagation (of errors)* [Rumelhart et al., 1986], as opposed to the *forward propagation* during computation of the network's output $f(x)$. The algorithm saves intermediate results in a way similar to dynamic programming.

Backpropagation

Let us consider a fully connected network with n layers of $d_1, \ldots, d_n$ neurons. The weight of the connection from the i-th neuron in the $(k-1)$-th layer to the j-th neuron in the k-th layer is denoted $w^k_{j,i}$. Each neuron in the k-th layer computes its value as a weighted sum of the outputs of the neurons in the previous layer and applies the activation function a:

$$f^k_j(x) = a\left(s^k_j(x)\right) = a\left(\sum_{i=1}^{d_{k-1}} w^k_{j,i} \cdot f^{k-1}_i(x)\right).$$

The loss function $J(x, \theta)$ depends on the training inputs x, which are fixed, and all the network parameters (weights) θ, which we are trying to set so that J is minimized. To do that, we initialize the weights to small random values (chiefly to break their symmetry; for discussion of initialization see Goodfellow et al. [2016]) and then repeatedly find the gradient $\nabla_\theta J(x, \theta)$ and move the weights in the opposite direction by a small step (i.e. we perform gradient descent).

We compute the derivatives with respect to the weights of the last (n-th) layer

using the chain rule:

$$\frac{\partial J(x,\theta)}{\partial w^n_{j,i}} = \frac{\partial J(x,\theta)}{\partial a(s^n_j(x))} \frac{\partial a(s^n_j(x))}{\partial s^n_j(x)} \frac{\partial s^n_j(x)}{\partial w^n_{j,i}}.$$

Because the activation function is being differentiated with respect to its argument and $s^n_j(x)$ is a sum with only one term depending on $w^n_{j,i}$, we have

$$\begin{aligned}\frac{\partial J(x,\theta)}{\partial w^n_{j,i}} &= \frac{\partial J(x,\theta)}{\partial a(s^n_j(x))} a'(s^n_j(x)) \frac{\partial}{\partial w^n_{j,i}} \sum_{k=1}^{d_{n-1}} w^n_{j,k} \cdot f^{n-1}_k(x) \\ &= \frac{\partial J(x,\theta)}{\partial a(s^n_j(x))} a'(s^n_j(x)) f^{n-1}_i(x).\end{aligned}$$

The derivative of the loss function depends on its realization, for MSE it would be

$$\frac{\partial J(x,\theta)}{\partial a(s^n_j(x))} = \frac{\partial}{\partial f^n_j(x)} \sum_{k=1}^{d_n} (y_k - f^n_k(x))^2 = 2\left(y_j - f^n_j(x)\right)$$

and for cross-entropy

$$\frac{\partial J(x,\theta)}{\partial a(s^n_j(x))} = \frac{\partial}{\partial f^n_j(x)} \sum_{k=1}^{d_n} -y_k \log f^n_k(x) = -\frac{y_j}{f^n_j}.$$

For the j-th node in the m-th layer we save

$$\delta^m_j = \frac{\partial J(x,\theta)}{\partial a(s^m_j(x))} a'(s^m_j(x)).$$

The gradient at hidden layers is computed similarly. The differentiation differs only in the first factor:

$$\begin{aligned}\frac{\partial J(x,\theta)}{\partial w^m_{j,i}} &= \left(\frac{\partial J(x,\theta)}{\partial a(s^m_j(x))}\right) \frac{\partial a(s^m_j(x))}{\partial s^m_j(x)} \frac{\partial s^m_j(x)}{\partial w^m_{j,i}} \\ &= \left(\sum_l \frac{\partial J(x,\theta)}{\partial s^{m+1}_l(x)} \frac{\partial s^{m+1}_l(x))}{\partial a(s^m_j(x))}\right) a'(s^m_j(x)) f^{m-1}_i(x) \\ &= \left(\sum_l \frac{\partial J(x,\theta)}{\partial s^{m+1}_l(x)} \frac{\partial}{\partial f^m_j(x)} \sum_r w^{m+1}_{l,r} f^m_r(x)\right) a'(s^m_j(x)) f^{m-1}_i(x) \\ &= \left(\sum_l \frac{\partial J(x,\theta)}{\partial s^{m+1}_l(x)} w^{m+1}_{l,j}\right) a'(s^m_j(x)) f^{m-1}_i(x) \\ &= \left(\sum_l \frac{\partial J(x,\theta)}{\partial a(s^{m+1}_l(x))} \frac{\partial a(s^{m+1}_l(x))}{\partial s^{m+1}_l(x))} w^{m+1}_{l,j}\right) a'(s^m_j(x)) f^{m-1}_i(x) \\ &= \left(\sum_l \delta^{m+1}_l w^{m+1}_{l,j}\right) a'(s^m_j(x)) f^{m-1}_i(x).\end{aligned}$$

Note that the first factor can be computed using the saved values in the following layer without looking further up. The computation of the whole derivative requires two other local quantities, namely $s^m_j(x)$ and $f^{m-1}_i(x)$. These should be saved during the forward pass. In the above, we did not explicitly account for bias updates; biases can be regarded as weights on connections to neurons that always output 1. Also note that the activation function could be different for every node and the above computation would still work.

Weight update

After the gradient $\nabla_\theta J(x, \theta)$ is computed, θ is updated. In its basic form the update rule of gradient descent looks like this

$$\theta := \theta - \lambda \nabla_\theta J(x, \theta),$$

where $\lambda > 0$ is called the *learning rate*. Many modifications have been proposed, mostly adding *momentum* to the θ moving through the space of parameters to avoid oscillations, and adapting the learning rate over time. Examples of popular optimizers include AdaGrad [Duchi et al., 2011] or Adam [Kingma and Ba, 2014].

Because the training sets are usually large and exact gradient computation would be unnecessarily slow, approximating it by computing a gradient on a smaller batch is often preferred. In theory, such *stochastic gradient descent* (SGD) requires the learning rate to approach zero to converge.

Regularization

Neural networks with a lot of nodes are complex machine learning models and as such can easily overfit to the training set, leading to decreased accuracy on the test set. To avoid overfitting, several regularization techniques can be employed. We can penalize big weights (L^2, L^1 regularization), augment the training set, inject noise, apply *dropout* (randomly set weights to zero during training [Srivastava et al., 2014]), stop the training early using a validation set, force some weights to share value (convolutional neural networks can be seen as an instance of parameter sharing) or use adversarial training, which we describe in the next chapter. For an overview of regularization methods, see Goodfellow et al. [2016].

1.2.2 Logistic regression

Logistic regression [Cox, 1958] is a machine learning model used for classification. Although it has been developed independently, logistic regression can be thought of as a simple neural network with one layer. In the case of classification into only two classes, it consists of one node with the sigmoid activation function:

$$f(x) = \sigma(w \cdot x + b).$$

The input is then classified as the first (positive) class if $f(x) > 0.5$ and as the second (negative) class otherwise. When classifying into multiple classes, the output layer becomes the *softmax* layer:

$$f(x)_i = \frac{\exp(w_i \cdot x + b_i)}{\sum_j \exp(w_j \cdot x + b_j)}.$$

The multiple class classification case is sometimes also called *multinomial* logistic regression. As the output elements are in the range $[0, 1]$ and sum to 1, they can be used to estimate the probabilities of the input belonging to the each of the classes. This estimated probability is called the *confidence* of the model. A softmax layer is often used as the final layer of a neural network classifier.

1.2.3 Convolutional networks

Most machine learning models, including the neural networks we have seen so far, do not take the order of input dimensions into account – they are permutation invariant. However, for some kinds of data, and especially for images, this ordering is intuitively very important. When an image is taken, it is usually not possible to ensure that the photographed object will be located at some exact pixel coordinates. If the dimensions were not ordered naturally, a small change in the object's location could result in a big perceived difference in the captured image data.

The *Neocognitron* [Fukushima and Miyake, 1982] was designed to exploit this structure of image data. It later inspired the creation of convolutional neural networks (CNNs) [LeCun et al., 1990], which are widely used today.

CNNs usually contain mainly two alternating types of layers – *convolutional* (which gave them their name) and *pooling*. The last few layers are usually fully connected.

Convolutional layer

A convolutional layer contains spatially arranged neurons. They are usually divided into multiple *channels* (*filters*). The channels contain neurons positioned in a rectangular grid. Each neuron looks only at a small rectangular group of neurons (a *patch*) in the previous layer. Each channel contains the same number of neurons covering the same patches as neurons in other channels, and, importantly, all neurons in a channel share weights. The rectangular patches covered by a channel are all of the same size and are themselves arranged in a rectangular grid, possibly overlapping with their neighboring patches. The distance between two neighboring patch centers is referred to as the *step size* (*stride*).

In another view, a convolutional layer is performing discrete 2D convolution (or, usually, cross-correlation).

A convolution of functions $f, g : \mathbb{R} \to \mathbb{R}$ is defined as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(a)g(x-a)\,\mathrm{d}a,$$

basically a moving sum of g weighted by a *kernel* f. In real world computations, the functions need to be discretized. In the case of images, we would do two-dimensional discrete convolution of image I and kernel $K : \mathbb{Z}^2 \to \mathbb{R}$:

$$(I * K)(i,j) = \sum_{i' \in \mathbb{Z}} \sum_{j' \in \mathbb{Z}} I(i-i', j-j')K(i',j').$$

A similar operation is the cross-correlation

$$(I * K)(i,j) = \sum_{i' \in \mathbb{Z}} \sum_{j' \in \mathbb{Z}} I(i+i', j+j')K(i',j'),$$

which differs only by having the kernel flipped and is equivalent to convolution for the purpose of learning the right kernel.

In practice, both the image and the kernel need to be nonzero only at a finite number of coordinates, so that we can evaluate the infinite summation by summing over the bounding rectangle only. To further reduce computational costs, we may

do steps of size $s > 1$ and perform a *downsampled* convolution of an image matrix $I \in \mathbb{R}^{h_1 \times w_1}$ by a kernel matrix $K \in \mathbb{R}^{h_2 \times w_2}$:

$$c(I, K, s)_{i,j} = C = \sum_{i'=1}^{h_2} \sum_{j'=1}^{w_2} I_{h_2+(i-1)\cdot s-i'+1, w_2+(j-1)\cdot s-j'+1} K_{i',j'},$$

where $C \in \mathbb{R}^{(\lfloor (h_1-h_2)/s \rfloor+1)\times(\lfloor (w_1-w_2)/s \rfloor+1)}$.

Because the convolution reduces the input width by $w_2 - 1$ (when $s = 1$; similarly for height), sometimes the input is padded with zeros so that the shape stays the same. This is handily called *same* convolution, as opposed to the *valid* case without padding.

A convolutional layer typically performs several convolutions with different kernels of the same size, each resulting in one channel. If already the input has multiple channels, three-dimensional convolution will be used.

The last step is then to apply some activation function to every scalar in every channel.

Pooling layer

A pooling layer combines several neighboring nodes in the previous layer into one. Maybe the most popular type is the *max-pooling* layer [Zhou and Chellappa, 1988] which returns the maximum value for each rectangular patch. Pooling is employed to make the network less sensitive to small translations of the input. Pooling layers are also useful in gradually reducing the representation size, so that the final fully connected layers (and the network in general) need less parameters that have to be trained.

Image classification convolutional networks

Convolution networks have been successfully applied to image classification tasks. They were used by LeCun et al. [1990] to recognize digits from the MNIST dataset and widely popularized when Krizhevsky et al. [2012] won ILSVRC 2012 by a large margin with their CNN architecture known as AlexNet. Subsequent editions of the challenge were also won by CNNs like GoogLeNet [Szegedy et al., 2015] or ResNet [He et al., 2016].

1.2.4 Radial Basis Function networks

Radial Basis Function (RBF) networks [Broomhead and Lowe, 1988] are neural networks that make use of the distances of the input from some chosen points in space called *centroids* (also *prototypes*). A basic RBF network with centroids $c_1, \ldots, c_m$ computes

$$f(x) = \sum_{i=1}^{m} w_i \cdot K_i(x, c_i),$$

where K_i are *kernel functions*. The most commonly used kernel function is the Gaussian

$$K_i(x, c) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{\|x - c\|_2^2}{2\sigma_i^2}\right).$$

As $K_i(x, c_i)$ is weighted by w_i, the coefficient $1/\sqrt{2\pi\sigma_i^2}$ can be incorporated into it. We also set $\beta_i = 1/(2\sigma_i^2)$ and get the simplified kernel

$$K_i(x, c) = \exp\left(-\beta_i \|x - c\|_2^2\right).$$

The training of RBF networks is usually done in two steps. First, the positions of the m centroids are chosen. They can be randomly sampled from the training examples, or the training examples can be clustered by the K-means algorithm (see Section 1.3.2) and the centers of the clusters then used as centroids of the RBF network. Note that this step is unsupervised unless the clustering is done independently on each class in the training data.

In the second step, the weights w_i need to be determined. When doing regression with a simple RBF network without any activation function, it is possible to set the weights by linear regression. Otherwise, if the RBF network is a part of some more complex neural network, they can be trained with SGD.

SGD can be also used to fine tune all the parameters in the network, including centroid coordinates, after the initial training.

Because the Gaussian goes quickly to zero with increasing distance, RBF networks can generally react only to inputs in the vicinity of the training data and have difficulties when generalizing in some direction is required.

1.2.5 Autoencoder

An *autoencoder* is a neural network that is trained so that it tries to reconstruct its input, i.e. the output layer has the same dimensionality as the input layer and the originally unlabeled training data are duplicated as the target values. Autoencoders are size constrained to not be able to copy the input perfectly, in particular, one of their layers is relatively small and its activation serves as the *representation* (code) of the input. Autoencoders can often learn useful properties of the data [Goodfellow et al., 2016].

1.3 Other models

1.3.1 Decision tree

A *decision tree* is a tree with one condition in each of its non-leaf nodes. Every condition acts on one feature x_i and splits the input space into several parts; there is one child node for each part. If the feature x_i is categorical, the condition splits the space into as many parts as there are categories of x_i. If x_i is continuous, some threshold t is chosen and the condition splits the space into two parts, one where $x_i < t$ and the other where $x_i \geq t$. Each leaf has one associated class (or a constant when doing regression). During inference, we travel from the root and in each node we evaluate the condition on our input x to decide into which child we should descend. When we reach a leaf, x gets classified as the corresponding class.

There are multiple methods of decision tree construction (training) such as CART [Breiman, 1984] or C4.5 [Quinlan, 1992].

1.3.2 K-means

K-means [Lloyd, 1982] is an unsupervised machine learning algorithm performing clustering. It divides the input space (and therefore also the training data) into K clusters, represented by their centroids (also prototypes). Each point of the input space belongs to the cluster represented by the nearest centroid. Note that the algorithm does not pick K; it has to be supplied by the user.

The algorithm is as follows. First, initial centroid positions $c_1, \ldots, c_K$ are chosen. A common approach is to sample them from the training data points. Then two steps are repeated until convergence: the assignment step

$$C_i \leftarrow \{x \in X : \|x - c_i\|_2 \leq \|x - c_l\|_2 \, \forall l \in \{1, \ldots, K\}\} \quad \forall i \in \{1, \ldots, K\},$$

where X are the training data, and the update step

$$c_i \leftarrow \frac{1}{|C_i|} \sum_{x \in C_i} x \quad \forall i \in \{1, \ldots, K\}.$$

The algorithm may be restarted several times with different random initializations and the final clustering chosen as the one with the lowest average distance from the training data to their respective centroid.

1.3.3 Support Vector Machines

Support Vector Machines (SVMs) [Cortes and Vapnik, 1995] are machine learning models that utilize so called "kernel trick" to transform the input space into a space with a lot more dimensions and then aim to construct an optimal separating hyperplane in the new space. Although they are primarily binary classifiers, multiclass classification can be performed with one SVM for each class.

1.3.4 Ensemble

An *ensemble* combines predictions of multiple models. In the case of classification, the predictions can be combined by voting; in the case of regression a weighted sum can be used. The models in the ensemble are often of the same type and can be constrained to remain simpler and to learn different aspects of the data. Common ensembling techniques include *bagging* and *boosting*. Decision trees are often used in ensembles, forming *random forests*. [Friedman et al., 2001]

2. Adversarial Examples

Deep neural networks (DNNs) have been recently achieving state-of-the-art performance on image classification datasets such as ImageNet [Deng et al., 2009, Goodfellow et al., 2014b], as has been demonstrated by Krizhevsky et al. [2012], Szegedy et al. [2015] and others. For example, He et al. [2016] have achieved 96.43% top-5 test accuracy with their ResNet architecture, which is difficult to exceed even for humans [Russakovsky et al., 2014].

But test set accuracy alone might not be a good measure of machine learning model robustness. Szegedy et al. [2013] discovered it is possible to force the network to misclassify an image by adding a very small perturbation to it – so small that a human would not notice the difference, implying the real class of the image stays the same. They termed these perturbed inputs *adversarial examples*.

The existence of adversarial examples poses a threat to the security of systems incorporating machine learning models. For example, autonomous cars could be forced to crash, intruders could confound face recognition software [Sharif et al., 2016] or other biometric authentication systems to be allowed improper access, or SPAM filters and other content filters could be bypassed. Moreover, adversarial examples can often be generated not only with access to the model's architecture and parameters, but also without it (such attacks would be called *white-box* and *black-box attacks*, respectively). Hence, secrecy alone does not solve the problem. The psychological aspect is also significant. Just as we see some opposition to machine learning systems whose reasoning is difficult to interpret, systems prone to misclassification of not visibly differing inputs might be met with distrust, even if their overall accuracy was superior to humans. Therefore, it is important to understand the weaknesses of our models and come up with ways to defend them against potential adversaries.

This chapter is organized as follows. In Section 2.1, we describe methods of adversarial examples generation that require access to the model architecture and parameters (white-box attacks). Then we discuss properties of adversarial examples in Section 2.2 and defenses that have been applied against them in Section 2.3. In Section 2.4, we look at methods generating adversarial examples without knowledge of the model architecture or parameters (black-box attacks). Finally, in Section 2.5, we describe some research directions which are somewhat related to adversarial examples.

2.1 Adversarial examples generation

In this section, we explore several methods of adversarial examples generation. All of them need access to the model's architecture and parameters, and hence are sometimes referred to as *white-box* attacks (as opposed to black-box attacks, which will be described in Section 2.4).

2.1.1 Optimization-based methods

Szegedy et al. [2013] generated adversarial examples by finding an approximate solution to the following optimization problem: Let $f : [0,1]^d \to \{1,\dots,k\}$ be a classifier mapping image pixel values to a discrete label set with associated loss function $J_f : [0,1]^d \times \{1,\dots,k\} \to \mathbb{R}^+$. For a given image $x \in [0,1]^d$ and target label $\tilde{y} \in \{1,\dots,k\}$, $\tilde{y} \neq f(x)$ minimize $\|\eta\|_2$ for $\eta \in \mathbb{R}^d$ subject to

$$f(x+\eta) = \tilde{y},$$

$$x+\eta \in [0,1]^d.$$

They used box-constrained L-BFGS for this problem and were able to successfully generate adversarial examples for each image and each network they tried (the accuracy on these examples was 0%). The networks were trained on the MNIST dataset [LeCun et al., 2010], ImageNet [Deng et al., 2009] and YouTube video frames dataset [Le et al., 2011]. Figure 2.1 shows some adversarial examples generated with this technique. Adding Gaussian noise was not sufficient to generate adversarial examples, as it caused images to be incorrectly classified only when severely distorted.

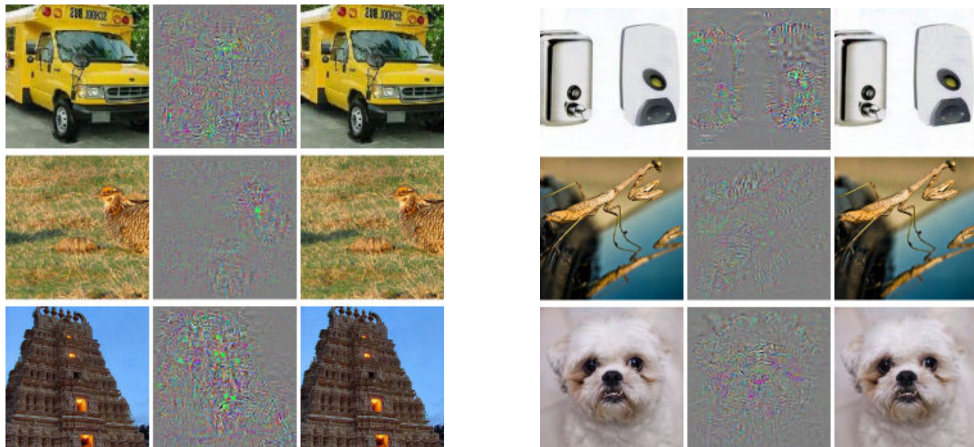

Figure 2.1: Adversarial examples generated for an ImageNet DNN. Left: a correctly classified image. Right: an adversarial example, incorrectly classified as *ostrich, Struthio camelus*. Middle: the difference magnified by 10. [Szegedy et al., 2013]

Liu et al. [2016] created targeted adversarial examples without the hard constraint for the class; instead, it was part of the minimized function

$$\lambda \cdot \|\eta\|_2 + J_f(x+\eta, \tilde{y}),$$

where λ is a parameter controlling the importance of the examples staying undetected. They optimized it using Adam optimizer [Kingma and Ba, 2014].

The disadvantage of optimization-based techniques is that they are relatively computation heavy (see Section 2.1.2 for a faster approach). On the other hand, they are usually able to generate closer, harder to detect adversarial examples while at the same time achieving higher misclassification rate [Kurakin et al., 2016a, Liu et al., 2016].

2.1.2 Fast Gradient (Sign) Method

While for some time it had been hypothesized [Szegedy et al., 2013] that the DNNs' vulnerability to adversarial examples is caused by their excessive non-linearity, Goodfellow et al. [2014b] tried to explain it the other way around – by approximating the DNN with a linear classifier.

For a linear model with a weight vector $w \in \mathbb{R}^d$, the answer to an adversarial input $\tilde{x} = x + \eta$, where $x \in \mathbb{R}^d$ is the original example and $\eta \in \mathbb{R}^d$ the adversarial perturbation, is

$$w^\top \tilde{x} = w^\top x + w^\top \eta.$$

The adversary wants to keep η small, so that the perturbation is not discovered. Let us consider the max-norm constraint $\|\eta\|_\infty < \varepsilon$ for some $\varepsilon > 0$. Then $w^\top \eta$ is maximized by assigning $\eta = \varepsilon \cdot \mathrm{sgn}(w)$:

$$w^\top \eta = \varepsilon \cdot w^\top \mathrm{sgn}(w) = \varepsilon \cdot \sum_{i=1}^{d} \mathrm{sgn}(w_i)\,|w_i|\,\mathrm{sgn}(w_i) = \varepsilon \cdot \sum_{i=1}^{d} |w_i| = \varepsilon \, \|w\|_1 \, .$$

Then, as $\|\eta\|_\infty$ is not affected by dimensionality d, and assuming the average size of elements of w does not change, $w^\top \eta$ will grow linearly with increasing d. For high-dimensional problems we can then make many small, imperceptible changes that sum up to one large change to the output.

Goodfellow et al. [2014b] propose the following method for generating adversarial examples for neural networks (and other non-linear models with loss function which is differentiable almost everywhere). We linearize the model by considering the gradient of its loss function J_f with respect to an input $x \in \mathbb{R}^d$ and then compute the adversarial perturbation η as before:

$$\eta = \varepsilon \cdot \mathrm{sgn}(\nabla_x J_f(x, y)),$$

where y is the ground truth label corresponding to x.

This method is called *fast gradient sign method* (FGSM). As the gradient can be computed using backpropagation, it is much more efficient than solving L-BFGS. It also reliably generates adversarial examples: Goodfellow et al. [2014b] found that using $\varepsilon = 0.25$ caused shallow softmax model trained on MNIST to misclassify 99.9% of adversarial inputs with average confidence of 79.3%. A deep convolutional network had an 87.15% error rate on adversarial examples generated from a preprocessed version of the CIFAR-10 test set. See Figures 2.2 and 2.3 for examples of adversarial inputs created by this method.

Note that in the above form, FGSM generates untargeted adversarial examples. If we want to misclassify x as a given $\tilde{y}$, it is simple to modify it accordingly:

$$\eta = -\varepsilon \cdot \mathrm{sgn}(\nabla_x J_f(x, \tilde{y})) \tag{2.1}$$

FGSM has showed that easily generated adversarial examples are not caused by the network's depth – shallow models such as logistic regression are also affected. [Goodfellow et al., 2014b] argue that deep neural networks are (at least theoretically) able to resist adversarial perturbations, as they are universal function approximators given enough units in a hidden layer Hornik et al. [1989]. Whether this can be achieved in practice remains to be seen.

Figure 2.2: An adversarial example generated by FGSM for an ImageNet DNN. Left: correctly classified image. Right: an adversarial example, incorrectly classified as *nematode* with confidence 99.3%. Middle: the gradient sign of the DNN's cost function with respect to the input. It was multiplied by $\varepsilon = 0.007$ and added to the original image to form the adversarial example. [Goodfellow et al., 2014b]

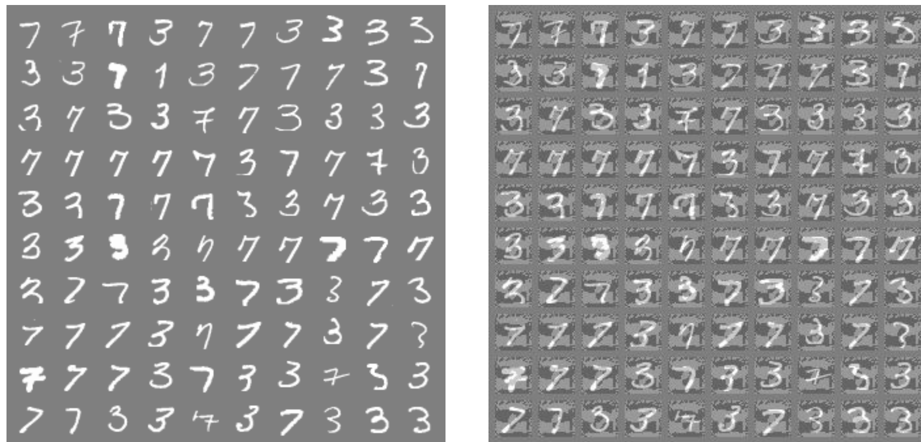

Figure 2.3: Adversarial examples generated by FGSM for a logistic regression model discriminating between pictures of 3s and 7s from MNIST. Left: the original images. Right: corresponding adversarial examples created with $\varepsilon = 0.25$. The model has 1.6% error rate on the clean inputs and 99% error rate on the adversarial inputs. Note the perturbations are much more visible than in Figure 2.2. This is because of the larger dimensionality of ImageNet pictures and consequently a much smaller ε. [Goodfellow et al., 2014b]

Fast Gradient Method

Alternatively, a *fast gradient method* (FGM) [Liu et al., 2016], similar to FGSM, can be used:

$$\eta = \varepsilon \cdot \frac{\nabla_x J_f(x, y)}{\|\nabla_x J_f(x, y)\|}.$$

Then η is the strongest perturbation subject to the given norm constraint.

2.1.3 Iterative FGSM

Kurakin et al. [2016a] introduced a simple extension of FGSM. They applied FGSM multiple times while keeping each pixel value in ε-neighborhood of the

original pixel:

$$\tilde{x}^{(0)} = x, \quad \tilde{x}^{(m+1)} = \text{Clip}_{x,\varepsilon}\left(\tilde{x}^{(m)} + \alpha \cdot \text{sgn}\left(\nabla_x J_f\left(\tilde{x}^{(m)}, y\right)\right)\right),$$

where $\text{Clip}_{x,\varepsilon} : \mathbb{R}^d \to [0,1]^d$ is a function

$$\text{Clip}_{x,\varepsilon}(\tilde{x})_i = \min\left\{1, x_i + \varepsilon, \max\{0, x_i - \varepsilon, \tilde{x}_i\}\right\}.$$

This is essentially equivalent to the *projected gradient descent* (PGD) on the negative loss function, as pointed out by Madry et al. [2017].

In the above formulation, this method again produces untargeted adversarial examples, but it is straightforward to change this by modifying the underlying FGSM (see equation 2.1).

Iterative methods create finer perturbations than FGSM; the image stays recognizable by humans with higher ε [Kurakin et al., 2016a]. See examples in Figure 2.4.

2.1.4 Papernot's method

[Papernot et al., 2015a] introduced a method in part similar to FGM. Instead of directly computing the gradient of the loss function with respect to the input, they take the gradients of the output of the model $f : \mathbb{R}^d \to \mathbb{R}^m$ with respect to the input (these form a Jacobian matrix if $m > 1$):

$$\nabla_x f(x) = \left[\frac{\partial f_j(x)}{\partial x_i}\right]_{i=1\ldots d,\, j=1\ldots m}.$$

Then they use this matrix to identify parts of the input that have a desirable effect on the model's output (*adversarial saliency map*) and perturb these parts until the output changes to the adversarial target or the maximum allowed rate of modified pixels is reached.

The algorithm is relatively complex, requiring tuning of multiple parameters, and they themselves do not adhere to it, as in their experiments on a MNIST CNN they take the gradients of the last hidden layer (as opposed to the output layer) with respect to the input. Nevertheless, the algorithm was able to find adversarial examples for 97% of inputs, modifying less than 14.5% of pixels.

2.2 Properties

Properties of adversarial examples that are being studied include their transferability, robustness to transformations, and detectability.

2.2.1 Transferability

An interesting property of adversarial examples discovered by Szegedy et al. [2013] is their *transferability* (also *generalization*). An adversarial example created for one model can sometimes serve as an adversarial example for a different model that has been trained from scratch on the same training set (*cross model generalization*) or even on a completely disjoint training set (*cross training set generalization*).

(a) Original image

(b) FGSM, $\varepsilon = 32/255$

(c) Iterative FGSM, $\varepsilon = 32/255$

(d) Iterative FGSM targeting the least likely class, $\varepsilon = 28/255$

Figure 2.4: Comparison of adversarial examples created by different FGSM variants. The maximum allowed change in individual pixel values is denoted by ε. [Kurakin et al., 2016a]

Szegedy et al. [2013] performed experiments with several models trained with different hyperparameters on either the same or a different part of the MNIST training data. They measured the *non-targeted* transfer rate – i.e. the ratio of adversarial examples generated for one model that were misclassified by another, regardless of the particular incorrect class assigned. The cross model transfer rate depended heavily on the pair of models examined (and was not symmetric); it fell between 2% an 87%. The cross training set transfer rate was between 5% and 27%.

Transferability of adversarial examples is important for most of black-box attack methods (see Section 2.4).

2.2.2 Robustness to transformations

The usually considered threat model allows the adversary to directly feed their perturbed inputs to the model being attacked. While this might be possible for models operating entirely within a computer, often the system would get its inputs from the real world through some kind of sensor. The question then arises whether adversarial examples can survive transformations introduced by the sensor, such as noise, translation, rescaling or a change of the viewing angle in the case of a camera.

Kurakin et al. [2016a] explored the effect of noise on adversarial examples crafted for Inception – an image classifying DNN [Szegedy et al., 2016] trained on ImageNet. They used FGSM, untargeted iterative FGSM and iterative FGSM targeting the least-likely class according to Inception (to avoid questionable misclassifications to very similar classes that occur in ImageNet, such as many different dog breeds). Then they printed each adversarial example, took a photo of it with a mobile phone camera and applied an affine transformation and cropping to get the image in the original shape. They measured *destruction rate* – the fraction of adversarial examples that were no longer misclassified after the printer and camera transformation.

Adversarial examples created by FGSM were generally more robust than the ones created by the iterative methods (for example 40% versus 90.5% and 94.6% destruction rate with a top-5 classification and $\varepsilon = 8/255$). Recall that iterative FGSM applies finer, harder to detect perturbations to the image; these are probably more likely to be destroyed. The experiment shows that some fraction of adversarial examples survives the photo transformation and physical adversarial examples are thus conceivable. But it did not explore the effects of viewing images from different distances or angles, which is often inevitable in the physical world (consider e.g. traffic signs). Also, when misclassification with a confidence of over 80% was required, the destruction rate was higher (70.8%, 91.2% and 97% respectively).

Robustness to geometrical transformations

Athalye and Sutskever [2017] showed it is possible to create adversarial examples robust to transformations that occur when viewing an image in the real world: scaling, rotation, translation, lightening/darkening etc. Moreover, they successfully generated adversarial textures for virtual 3D objects and even constructed two 3D-printed real-world objects with these textures that were consistently misclassified over a wide range of viewing conditions.

To this end, they introduced *Expectation over Transformation* (EOT) – an approach that maximizes the expected log probability of target response over all transformations, while trying to keep the perceived difference between the original and the perturbed image low:

$$\tilde{x} = \operatorname*{argmin}_{x'} \mathbb{E}_{t\sim T}\left[-\log \mathrm{P}(\tilde{y} \mid t(x'))\right] + \lambda \mathbb{E}_{t\sim T}\left[d(t(x), t(x')\right], \tag{2.2}$$

where T is a given distribution of transformations, x the original sample, $\tilde{x}$ its adversarially perturbed variant, $\tilde{y}$ the target (incorrect) response, d some distance function, λ is a chosen parameter and $\mathrm{P}(y \mid t(x'))$ is the probability of the model

outputting label y given randomly transformed input $t(x')$. Note the expected distance between the *transformations* of the original and transformed input is being minimized. The distance d used was the Euclidean distance in the LAB color space. The above objective function can be optimized by approximating the gradient of the expected value through sampling and running SGD.

In the case of 3D objects, the transformations in T were composed of mapping the texture to a given object and subsequent rotation, translation, perspective distortion or background addition. Instead of differentiating through the 3D renderer, the authors simulated each rendering as a sparse matrix multiplication, different at each sampling step, and differentiated through it.

The authors evaluated their method against the InceptionV3 ImageNet classifier which achieves 78% top-1 accuracy on clean images. The transformed 2D adversarial images were misclassified as the target class (randomly chosen for each image) for 96% of transformations on average. The transformed 3D models with adversarial textures were misclassified as the target class (20 were randomly chosen for each of the 10 models) for 83% of transformations on average and finally the two physical adversarial objects (a turtle and a baseball) were misclassified as the randomly chosen target class ("rifle" and "espresso") in 82% and 59% of cases, respectively (and were misclassified as other incorrect classes in additional 16% and 31% of cases, respectively). See Figure 2.5 for pictures of the turtle and its adversarial variant.

2.2.3 Detectability

Perturbed input $\tilde{x}$ is regarded as an adversarial example only if it belongs to the same class as the original image x. Thus, it must be similar enough – with similarity being assessed by human judgment, and hence difficult (or costly) to exactly quantify. But even if the perturbed input falls into the ground truth class boundary, its similarity to the original image matters: the more different it is, the easier it is to detect the perturbation.

To measure dissimilarity, *distortion* is sometimes used, defined as [Gu and Rigazio, 2014]

$$\frac{\sum_{i=1}^{d}(x_i - \tilde{x}_i)^2}{d} = \frac{\|x - \tilde{x}\|_2^2}{d}$$

or alternatively as [Szegedy et al., 2013, Liu et al., 2016]

$$\sqrt{\frac{\sum_{i=1}^{d}(x_i - \tilde{x}_i)^2}{d}} = \frac{\|x - \tilde{x}\|_2}{\sqrt{d}}.$$

2.3 Defenses

Several different approaches to defending machine learning models against adversarial examples have been proposed. Unfortunately, none of them have so far been able to completely solve the problem.

A truly successful defense would substantially decrease the misclassification rate on adversarial examples (ideally so much so that they were not misclassified more often than clean data), while not worsening the accuracy on clean data significantly.

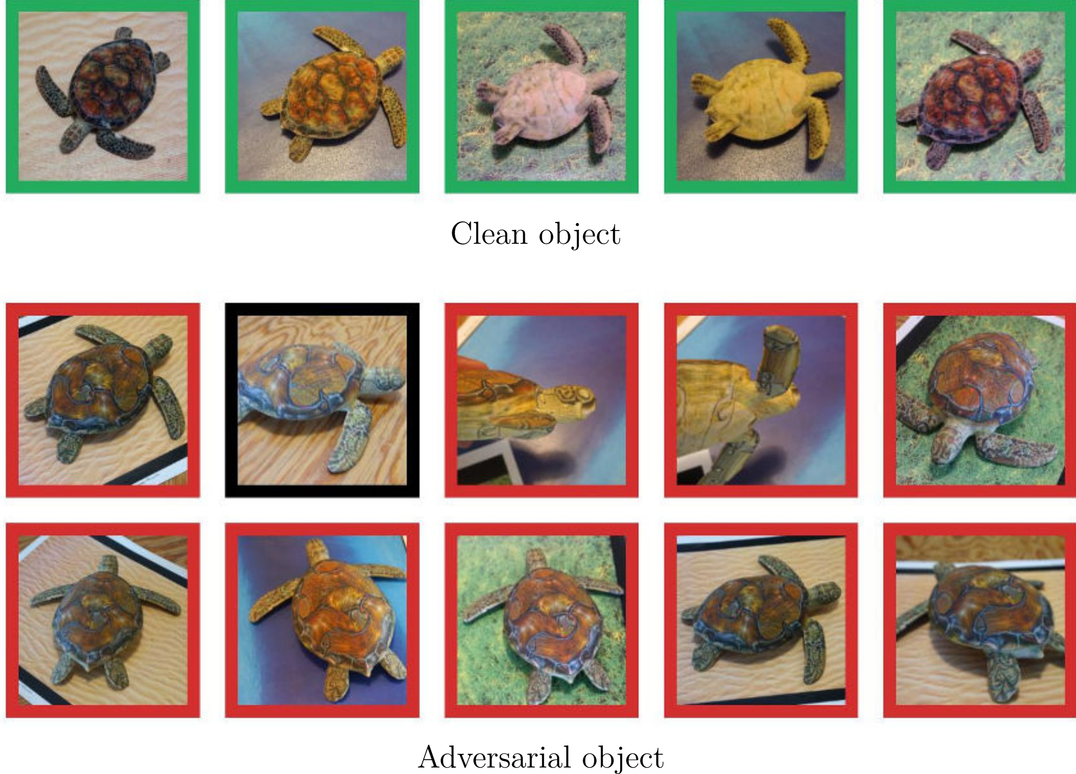

Figure 2.5: A physical adversarial example. Top row: 3D-printed model of a turtle. Bottom rows: 3D-printed model of a turtle with texture adversarially perturbed by EOT (introduced in equation 2.2). Images with green border were classified correctly, images with red border were misclassified as the target class ("rifle") and the image with black border was misclassified as a different class. [Athalye and Sutskever, 2017]

2.3.1 Adversarial training

While training a model, adversarial examples can be periodically generated and added to the training set. Szegedy et al. [2013] showed this could work as a regularizer, but did not get better results than by using dropout.

Goodfellow et al. [2014b] used FGSM to directly include adversarial examples in the cost function during training:

$$\tilde{J}_f(x, y) = \alpha J_f(x, y) + (1 - \alpha) J_f(x + \varepsilon \cdot \operatorname{sgn}(\nabla_x J_f(x, y)), y),$$

where x, y and J_f are the input, output and the cost function of the model, and $\alpha \in [0, 1]$ and $\varepsilon > 0$ are chosen parameters. This corresponds to pushing the model to minimize the error also on adversarial examples.

Using this approach along with dropout, they were able to reduce their error rate from 0.94% to 0.84% on MNIST. Early stopping on the *adversarial* validation set together with an increased number of hidden units (from 240 to 1600 per layer) further decreased the error rate to 0.78% on average. The model also acquired some resistance to adversarial examples (the error rate on these fell from 89.4% to 17.9%), but remained highly confident when fooled (average confidence 81.4%).

Instead of teaching the model that moving an example x a bit in the direction of the cost function gradient sign does not change its label, we could try to make

it resistant to all changes in x features smaller than ε by training on many points sampled uniformly from the ε max-norm box. However, Goodfellow et al. [2014b] pointed out that the expected dot product between such a perturbation and the weight vector is zero and many perturbed examples will actually have lower cost function value. The model trained in this way had a 90.4% error rate with a 97.8% confidence on FGSM adversarial examples, so it does not seem to be a viable alternative.

Kurakin et al. [2016b] studied the effects of adversarial training on the larger dataset of ImageNet. Unlike on MNIST, adversarial training caused a small drop (less than 1%) in accuracy on clean examples. This was compensated by greater increase of accuracy on adversarial examples (e.g. top-5 accuracy increased from 57.2% to 91.4% for $\varepsilon = 16/256$). However, adversarial training did not help against iterative FGSM (top-5 accuracy 23.2% with adversarial training, 25.9% without).

The authors found that larger networks benefited from adversarial training more than smaller networks. They also discovered what they called a *label leaking* effect: when trained on FGSM adversarial examples, the model then reached higher accuracy on FGSM adversarial examples than on clean images. They hypothesize this happens because the transformation that one-step FGSM applies when it has access to the true label is simple enough so that the model can learn to recognize it, and recommend to replace the true label with the label predicted by the model during adversarial training.

Madry et al. [2017] looked at adversarial robustness of neural networks from the perspective of robust optimization and ended up using iterative FGSM adversarial training.

They formulated the problem of finding the parameters that make a network most robust to the perturbations of the most apt first-order adversary as a min-max optimization problem, and argue that this formulation provides concrete security guarantee for the trained model, as opposed to previous work, in which the defenses were tested against *some* attack without ensuring its strength.

As the attack must be limited in some way, the authors consider a set of allowed perturbations $S \subseteq \mathbb{R}^d$. In line with previous work, they focus on max-norm bounded perturbations, i.e. $S = [-\varepsilon, \varepsilon]^d$.

The problem formulation stems from viewing training of machine learning models as attempting to minimize the population risk

$$\mathbb{E}_{(x,y)\sim D} J_\theta(x, y),$$

where D is the underlying data distribution, θ the model parameters and J_θ the associated loss function. Finding the parameters that result in a model most robust to attacks limited by S can then be written as a composition of an inner maximization problem and an outer minimization problem:

$$\min_\theta \mathbb{E}_{(x,y)\sim D} \left[\max_{\eta \in S} J_\theta(x + \eta, y) \right].$$

Generally, this is a difficult problem to solve, as already the inner maximization problem is highly non-concave. Still, the authors argue that the inner part can be solved by projected gradient descent (iterative FGM) in practice, much like

training of neural networks. Danskin's theorem (see Madry et al. [2017] for a formulation) then gives that descent direction for the outer minimization problem corresponds to gradients at inner maximizers, i.e. the best adversarial examples. The assumptions of the theorem technically do not hold because the function is usually not continuously differentiable (albeit only on a set of measure zero) due to ReLUs and max-pooling. Nevertheless, the application of the theorem, i.e. adversarial training with an iterative FGSM adversary, seems to work well in practice. Because PGD found very similarly valued (but distinct) local minima across many restarts, we can assume it approximated the global minimum well.

The authors reported very good accuracy of 0.89 on adversarial examples generated by iterative FGSM with $\varepsilon = 0.3$ for their best MNIST model and accuracy of 0.45 for their best CIFAR10 model. These findings contrast with those of Kurakin et al. [2016b], who could not get any benefits out of iterative adversarial training during their experiments on ImageNet. The experiments also show that having a higher capacity helps – a model more robust to adversarial examples (after adversarial training) had more parameters than a model that only achieved high accuracy on clean examples. This is in line with Kurakin et al. [2016b].

2.3.2 Denoising autoencoders and DCN

In an attempt to make a DNN trained on MNIST immune to adversarial examples, Gu and Rigazio [2014] trained an autoencoder to remove adversarial perturbations (if any) and stacked the DNN on top. They were able to reduce the misclassification rate of adversarial examples by more than 90%. However, they also noted that new adversarial examples could just as easily be generated for the network as a whole, i.e. including the autoencoder. Similar results were achieved with a mere denoising autoencoder (i.e. an autoencoder trained to remove Gaussian noise).

Gu and Rigazio [2014] then proposed *deep contractive networks* (DCN), which are still prone to adversarial examples, but should at least require larger perturbations in order to be fooled (measured by distortion, see Section 2.2.3). However, the achieved increase in average distortion is not very high (e.g. from 0.087 to 0.102 for a DNN with two hidden layers).

2.3.3 Simple image transformations

When an image classifier system gets its inputs from the real world, they are transformed in some way by the used sensor. This transformation might destroy adversarial examples (see Section 2.2.2). Similarly, we can artificially modify inputs in an attempt to increase the model's robustness to adversarial perturbations, usually trading it for accuracy on unperturbed data.

Gu and Rigazio [2014] measured the effects of adding Gaussian noise and Gaussian blurring on MNIST images. Additive Gaussian noise with a standard deviation of 0.2 increased the error rate on clean data from about 1% to about 3% and decreased it from 100% to between 93% and 63% on adversarial examples. Better results were achieved with a CNN than with a fully connected network. For a CNN, Gaussian blur with blur kernel of size 5 increased error rate on clean data from 0.9% to 1.2% and decreased it from 100% to 53.8% on adversarial examples.

Kurakin et al. [2016a] explored the effects of Gaussian blur, Gaussian noise, JPEG encoding and brightness and contrast change on adversarial examples crafted by FGSM and its iterative variant for ImageNet classifying DNN Inception. Generally, blur, noise and JPEG encoding had higher destruction rate than the change of brightness and contrast, and FGSM-created adversarial examples were more resilient than those created by iterative FGSM. Unfortunately, the size of probable corresponding decrease in accuracy on clean examples was not reported.

2.3.4 Gradient masking

A variant of *distillation* [Hinton et al., 2015] can be employed to reduce effectiveness of adversarial examples creation. Usually, distillation is used to transfer knowledge from a larger DNN to a smaller one (to save resources during inference), by training it on inputs with labels (or class probability vectors) assigned by the larger network. Instead, Papernot et al. [2015b] used what they called *defensive distillation* to improve resistance of the original network by training its substitute with the same architecture. They reported it had smaller gradients and were able to reduce the misclassification rate on adversarial examples from 95.9% to 1.3% for a DNN trained on MNIST and from 89.9% to 16.8% for a DNN trained on CIFAR10. Adversarial examples were crafted using the Papernot's method. Accuracy on clean data was lowered by less than 2%.

This seems promising, however, defensive distillation has been shown to be susceptible to substitution attacks (see Section 2.4.2) [Papernot et al., 2016].

2.4 Black-box attacks on ML models

The adversarial examples generating methods we have seen so far require access to the model parameters (and hence, its architecture). While these might be available (as state-of-the-art deep models usually take very long time to train, pre-trained models are sometimes shared and used – see for example Caffe Model Zoo[1]), often they will not be disclosed (and even an admitted pre-trained model will be probably fine-tuned to the specific task by replacing one or more of its top layers).

However, obscurity does not imply security and there are indeed methods that can perform *black-box attacks* against machine learning models, i.e successfully create adversarial examples without knowledge of their architecture and parameters.

2.4.1 Evolutionary generation of adversarial examples

Vidnerová and Neruda [2016] employed genetic algorithms to find adversarial examples for a range of MNIST image recognition machine learning models (an neural network with 2 hidden layers, a CNN, an ensemble of 10 neural networks, an RBF network, an SVM and a decision tree). They used individuals encoded as vectors of pixel values, two-point crossover, mutation by adding Gaussian noise to

[1] `https://github.com/caffe2/caffe2/wiki/Model-Zoo`

individual pixels, tournament selection and a fitness function defined as

$$F(\tilde{x}) = -\frac{\|\tilde{x} - x\|_2 + \|f(\tilde{x}) - \tilde{y}\|_2}{2},$$

where $\tilde{x}$ is the individual being evaluated (adversarial example candidate) that should resemble image x, $f(\tilde{x})$ is the model response to $\tilde{x}$ and $\tilde{y}$ is the target (i.e. incorrect) response. After 10 000 generations of 50 individuals, the neural network, CNN, ensemble, SVM with sigmoid kernel and decision tree always misclassified the resulting adversarial example, whereas SVM with polynomial kernels misclassified it only in 44% to 77% of times and RBF network and SVM with linear and RBF kernels never misclassified it.

The adversarial examples were also transferable between models (trained on the same dataset) to some extent, notably to the models resistant to adversarial examples generated specifically for them.

Depending on the exact setting, the relatively high number of inferences required to be made by the target model (one for each fitness evaluation) might render this black-box method impractical. As we will see in Section 2.4.2, even this obstacle can be overcome.

Su et al. [2017] used differential evolution to find one pixel (or few pixel) adversarial perturbations. Unlike FGSM, this method does not use the high dimensionality of image data, but on the other hand does not impose any constraint on the magnitude of the one pixel perturbation other than the image representation limits and the pixel is thus often clearly visible (see Figure 2.6).

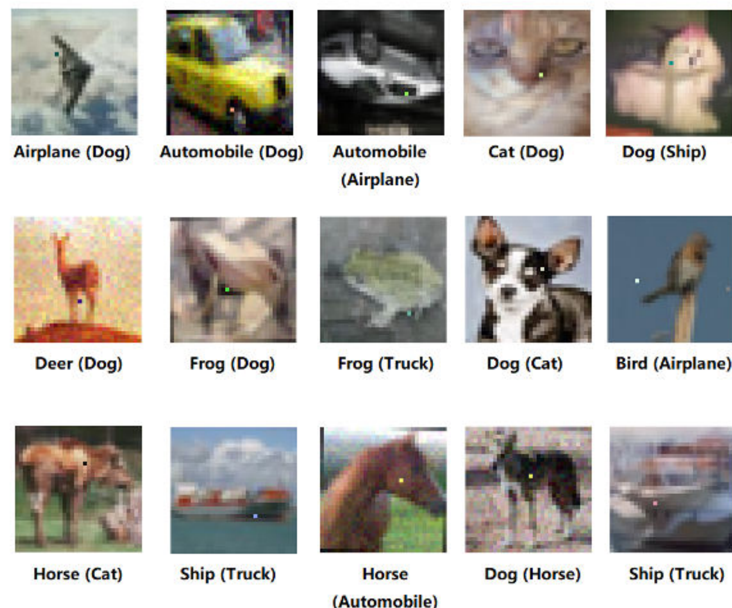

Figure 2.6: One pixel attacks against a CIFAR10 trained CNN. The target labels are in parentheses. [Su et al., 2017]

Their candidate solutions (differential evolution individuals) consisted of fixed number (1, 3 or 5) of one pixel perturbations, each containing its own coordinates and three-dimensional (RGB) perturbation value.

The fitness function they were maximizing was directly $f_t(x + \eta)$, i.e. the targeted classifier's confidence that the perturbed input $x + \eta$ belonged to the target (incorrect) class t. Because this method requires the classifier's confidences (but not its parameters), the authors call it a *semi-black-box* attack.

They tested this method on three different CNNs trained on CIFAR10. It was successful in ca. 23% when producing targeted adversarial examples and in ca. 71% producing untargeted adversarial examples. They unfortunately did not test its transferability nor robustness to transformations.

2.4.2 Substitution attacks

Because of the transferability of adversarial examples demonstrated by Szegedy et al. [2013], an adversary could train a substitution model (ideally on the same dataset, but potentially even on an independent one), create adversarial examples for it using e.g. FGSM and then apply them to the original model. Recall that Szegedy et al. [2013] reported model transfer rate between 2% an 87% and cross training set transfer rate between 5% and 27% on MNIST.

The transfer rate depended heavily on the pair of models examined; so although this method can be seen as a black-box attack, it is usually strong enough only when the target model architecture is known. The adversary also might not have access to the dataset used for training of the target model or there might not be any available dataset for a given task at all. As today's deep image recognition models are trained on datasets containing millions of images, assembling and labeling of their own dataset can be too costly for the attacker.

Papernot et al. [2016] created their substitution DNN in a different way. They trained it to mimic decisions made by the target model by using synthetic training data labeled by the target model. Their method does not need any information about the target model apart from its responses to given inputs and the number of queries for such responses is limited. The method is thus appropriate for attacking private, remotely hosted classifiers.

In the following, we will describe their algorithm in a bit more detail.

Substitute model training

First, substitute architecture is chosen and initial set of examples $S_0 \subseteq \mathbb{R}^d$ is assembled. Then m *substitute training epochs* are performed. In epoch number i for $i = 0, \ldots, m-1$, the set of examples S_i is labeled by the target model $M : \mathbb{R}^d \to \{1, \ldots, k\}$, obtaining D_i :

$$D_i \leftarrow \{(x, M(x)) : x \in S_i\}.$$

The substitute model $F_i : \mathbb{R}^d \to [0,1]^k$ is trained on D_i from scratch. Note that unlike M, F_i returns the probabilities of x belonging to a particular class. Subsequently, the set of examples is expanded by *Jacobian-based dataset augmentation*

$$S_{i+1} \leftarrow \{x + \lambda \cdot \operatorname{sgn}(J_{F_i}[M(x)]) : x \in S_i\} \cup S_i,$$

where

$$J_{F_i}[M(x)] = \left(\frac{\partial F_i(x)_{M(x)}}{\partial x_1}, \ \ldots, \ \frac{\partial F_i(x)_{M(x)}}{\partial x_d} \right)$$

is the row of the Jacobian matrix of F_i corresponding to the label $M(x)$ and $\lambda \in \mathbb{R}$ is a parameter controlling the augmentation step size.

Refinements

Papernot et al. [2016] also introduced several refinements to the original algorithm.

To reduce the number of queries made to the target model, which in the above algorithm grows exponentially with the number of epochs, for $i > \sigma$ only κ examples sampled uniformly from S_i are augmented to be included in S_{i+1}, where σ and κ are controlling parameters. Although this does reduce the substitute accuracy, the effect is relatively small (about 3–11%).

Increasing λ speeds up convergence but lowers adversarial example transferability. Somewhat surprisingly, *alternating the sign* of λ periodically helps to achieve better approximation of the target model.

Performed attacks

After crafting untargeted adversarial examples on the substitution model using FGSM, Papernot et al. [2016] deployed them against machine learning services of Metamind, Amazon and Google. These classifiers are remotely hosted and automatically trained; the user supplies only the training dataset (MNIST in this case). Their accuracy on clean images was higher than 0.92 and lower than 0.16 on adversarial examples; we can consequently consider the attack successful.

Note that the target model does not need to be differentiable; interestingly, the attack generalizes to models like decision trees and SVMs. The transferability Papernot et al. [2016] reported after six substitute training epochs was between 43% and 76%, depending on the (locally trained) target model.

2.4.3 Targeted substitution attacks

Liu et al. [2016] reported that for models trained on ImageNet, transferability of *targeted* adversarial examples is more difficult to achieve.

They measured cross-model transferability between well known deep CNNs (such as ResNet [He et al., 2016] or GoogLeNet [Szegedy et al., 2015]). While the untargeted transfer rate was between 61% and 95%, targeted transfer rate did not exceed 4%.

To increase targeted transferability on ImageNet, Liu et al. [2016] proposed to generate adversarial examples not for a single model, but for an ensemble of several CNNs. Using optimization-based approach, they were able to achieve 63% to 99% targeted transferability rate. However, adversarial examples generated by FG(S)M had no better targeted transfer rate than when created for a single model.

2.5 Related research

In this section, we will shortly describe two concepts which, while being similar to or similarly named as adversarial examples, do not quite belong among them.

2.5.1 Fooling examples

Fooling examples [Nguyen et al., 2014] (also *rubbish class examples*) are similar to adversarial examples. While adversarial examples get assigned incorrect class by the targeted machine learning model, fooling examples are assigned some class with high confidence, although none is correct – humans do not recognize the images as anything specific. The ideal answer in this case would be the same (and therefore low) confidence for each class.

Nguyen et al. [2014] generated such images by evolutionary algorithms for networks trained on MNIST and ImageNet. They used either direct encoding (genes corresponding to pixels) or indirect encoding (genes being parameters of a compositional pattern-producing network (CPPN) [Stanley, 2007]), which led to noise-like or highly regular images, respectively (see Figure 2.7).

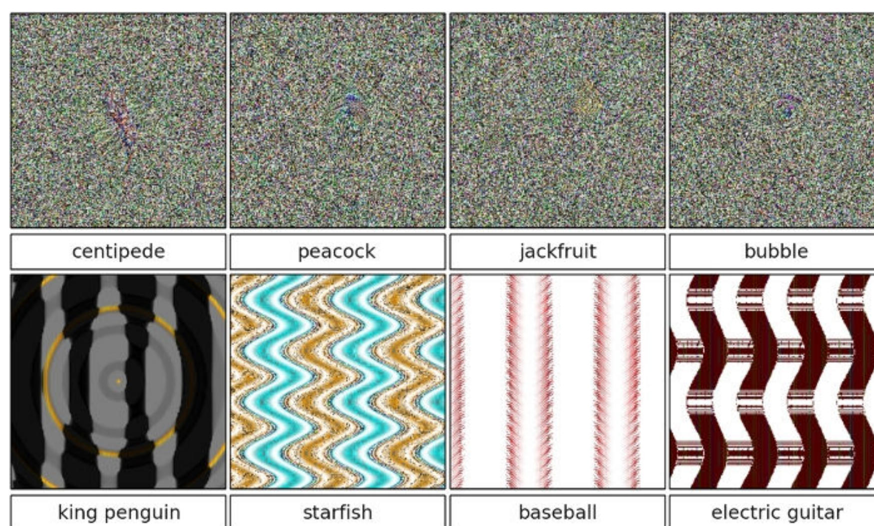

Figure 2.7: Fooling examples for an ImageNet DNN generated by evolutionary algorithms with direct (top row) or indirect (bottom row) encoding of individuals. [Nguyen et al., 2014]

For MNIST DNN, median confidence of the spurious classification was 99.99% and retraining with new "fooling image" class and labeled data did not help, as new fooling examples could be generated for the new network (and the whole procedure iterated). Generating fooling images for ImageNet network was less successful, with median confidence 22% and 88% for direct and indirect encoding, respectively. Also retraining the ImageNet DNN with new "fooling image" class did help to some extent – it reduced median confidence to 12% in the indirect encoding case. The authors hypothesize this is due to the difference between natural and CPPN images being more easily learned than the difference between MNIST digits and CPPN images.

Some fooling images were transferable to a DNN with different architecture (but trained on the same dataset).

2.5.2 Generative Adversarial Networks

Similarly named, but distinct area of research are *generative adversarial networks* (GANs), introduced by Goodfellow et al. [2014a]. As the name suggests, a GAN

consists of a *generator network* G, which is a generative model (i.e. it is trying to sample from the data probability distribution $\mathrm{P}(x)$), trained together with a *discriminator network* D, which emits a probability value $D(\theta_D, x)$, interpreted as the model's confidence that the input x is a real example (as opposed to being created by the generator). These two networks act as adversaries in a zero sum game – the generator is trying to minimize and the discriminator to maximize the value function

$$V(\theta_{\mathrm{G}}, \theta_{\mathrm{D}}) = \mathbb{E}_{x \sim p_{\mathrm{data}}(x)}[\log D(\theta_{\mathrm{D}}, x)] + \mathbb{E}_{z \sim p_{\mathrm{noise}}(z)}[\log(1 - D(\theta_D, G(\theta_{\mathrm{G}}, z)))],$$

where θ_{G} and θ_{D} are parameters of G and D, respectively. The process terminates in a saddle point.

In each step the generator network creates samples mimicking the training data. The discriminator network is then trained to recognize these as fake (in practice only a few backpropagation steps are run). Finally, the generator is updated by one backpropagation step. The training stops when the discriminator outputs 1/2 as the probability of the sample being real for every sample.

Generative adversarial networks do not have much in common with adversarial examples. The purpose of adversarial examples is to fool a classifier, whereas the purpose of GANs is to train a good generative model, discarding the only classifier involved (the discriminator network) at the end of the training. We will not deal with GANs anymore; nevertheless, it is a very active research area at the moment.

2.6 Conclusion

Adversarial examples pose a threat to machine learning models and consequently to systems where these models are used. They can greatly lower the accuracy of the targeted model while still being perceived as members of the original class by humans, and in some cases also being difficult to detect.

Several methods generating adversarial examples have been proposed and none of them is the most suitable in all circumstances. The attacker's choice will depend on the information they have about the targeted model (white box/black box), the amount of time available for adversarial examples generation (FGSM/iterative FGSM/evolutionary algorithms), whether misclassification as particular class is required, and demand for robustness to input transformations (iterative FGSM/FGSM/EOT).

Most methods have been evaluated on MNIST or ImageNet. Generally, it is possible to make the adversarial perturbation much more subtle on ImageNet, as a result of the high dimensionality where the perturbation can be hidden. Adversarial examples for MNIST models are still correctly classified by humans (and so they satisfy the definition), but their difference to the original MNIST images is usually clearly visible (although such noisy images might very well be encountered in the wild).

It seems unlikely that this problem will ever become completely solved, i.e. that the model with state-of-the-art accuracy on clean inputs will correctly classify all perturbations that do not change the true class. After all, what is known as optical illusions could be considered to be a kind of adversarial examples for humans. To decide whether the problem has been solved sufficiently well is difficult

and depends on the task at hand. Sometimes enforced misclassifications must be avoided at all cost, and in other cases, where adversaries would not have any motivation or opportunity to perturb the input, spared bits of accuracy on clean inputs might be more valuable.

Just as there are multiple kinds of attack, several defenses have been conceived. Out of these, iterative FGSM adversarial training is maybe the most promising. However, it is computationally demanding and nobody has managed to scale it to ImageNet so far. Variants of gradient masking have been shown to be bypassable by substitution attacks. Simple input transformations, both naturally occurring and artificially added, have been largely overcome by EOT.

All in all, new defenses, as well as attacks they can be tested against, are still in demand.

3. In search for new defenses and attacks

In this chapter we present several attempts at finding a successful defense against adversarial examples for image classification models. First, we compare the robustness of models with RBF units and conventional CNNs on MNIST. We do not manage to make traditional RBF models meet the expectations set up by previous work, but our network with a convolution-like RBF layer does outperform a CNN on adversarial examples. Then we try to defend a CNN MNIST model by rounding individual pixel values of input images and obtain some good results. We try applying rounding also on ImageNet, but the results are less promising.

Finally, we attempt to create adversarial perturbations without observing the particular inputs that should be perturbed. We evaluate this approach on MNIST, where it is able to decrease the accuracy of a CNN relatively strongly in an untargeted scenario, but less so in a targeted case.

In our experiments, we use TensorFlow [Abadi et al., 2015] for neural networks construction, the K-Means implementation from scikit-learn [Pedregosa et al., 2011], NumPy [van der Walt et al., 2011] and Pandas [McKinney, 2010] for data manipulation and Matplotlib [Hunter, 2007] for plotting.

3.1 RBF models on MNIST

Goodfellow et al. [2014b] suggested that deeper RBF networks could serve as a more robust alternative to CNNs when dealing with adversarial examples due to them being less linear and thus less readily generalizing in the direction of unknown data. They explored a variety of models involving quadratic units, but reported only the accuracy of a shallow RBF model on FGSM generated adversarial examples ($\varepsilon = 0.25$).

Vidnerová and Neruda [2016] found that RBF networks were resistant to adversarial examples generated by their evolutionary method. This provides some support for the above hypothesis.

To investigate the possible robustness of such networks, we evaluated several architectures incorporating RBF on images from MNIST perturbed by FGSM. We chose FGSM because it is widely used in current research, thus serving as a common benchmark, and computationally manageable. We compare these models with regular CNNs.

3.1.1 Method

We trained the following models on the MNIST training set:

CNN CNN architecture taken from the TensorFlow MNIST tutorial[1]. It has two convolutional layers (32 and 64 channels, respectively, kernel size 5×5, stride size 1) with ReLU activation and max pooling (pool size 2×2, stride size 2), a

[1] `https://www.tensorflow.org/get_started/mnist/pros`

fully connected layer (1024 neurons), a second fully connected layer (10 neurons with 0.4 dropout rate) and an output softmax layer. It was trained by Adam optimizer for 20 epochs.

CNN-adv The architecture is the same as above, but it was trained adversarially, i.e. every training batch consisted of half clean and half adversarial examples generated for the current version of the model by FGSM with $\varepsilon = 0.25$ (as recommended by Goodfellow et al. [2014b]).

RBF-500 RBF network with one hidden layer (500 neurons), one linear fully connected layer (10 neurons) and a softmax output layer. We chose the number of neurons to match the number used by Vidnerová and Neruda [2016]. The parameters of the hidden layer neurons were set by clustering the MNIST training set with K-Means (30 iterations). Then the whole network was trained by Adam optimizer to set the weights of the linear layer and fine-tune the RBF layer.

RBF-1000 The same architecture as RBF-500, but with 1000 neurons in the hidden layer.

SGD-RBF-500 The same architecture as RBF-500, but the whole model was trained using Adam optimizer for 1000 epochs (i.e. without K-Means pretraining).

SGD-2L-RBF-500 The model has two hidden RBF layers, with 500 and 125 neurons, respectively, a fully connected layer (10 neurons) and an output softmax layer. It was trained by Adam optimizer for 1000 epochs.

RBFolutional Network This model features a layer that we call *RBFolutional*. It is similar to a convolutional layer, but the operation used to combine the kernel and the underlying patch of the image is RBF instead of dot product. Recall that discrete convolution of image I by kernel $K \in \mathbb{R}^{h \times w}$ might be written as

$$(I * K)(i, j) = \sum_{i'=1}^{h} \sum_{j'=1}^{w} I(i + i', j + j') - K(i', j').$$

RBFolution of image I by kernel K is then

$$\mathrm{RBFol}(I, K)(i, j) = \exp\left(-\beta \sum_{i'=1}^{h} \sum_{j'=1}^{w} \left(I(i + i', j + j') - K(i', j')\right)^2\right),$$

where $\beta > 0$ is a parameter. Our RBFolutional network has one RBFolutional layer (32 channels, kernel size 5×5) with max pooling (pool size 2×2, stride size 2), followed by a convolutional layer (64 channels, kernel size 5×5), a dense layer (30 neurons, ReLU activation), a second dense layer (10 neurons) and an output softmax layer.

We did not manage to train this model by SGD from scratch, therefore we initialized the RBFolutional layer with KMeans (on patches extracted from the first 10 000 MNIST training images) and then trained the whole network with SGD for 10 epochs.

We generated adversarial examples for each model by perturbing the whole MNIST test set by FGSM with ε ranging from 0 to 1. After inspecting the resulting images, we discarded those generated with $\varepsilon > 0.4$, as they were already too distorted and many would not be recognized by humans. For examples of the resulting images, see Figure 3.1. We then measured the accuracy of the models on their respective adversarial examples.

Figure 3.1: Illustration of the effects of FGSM targeting RBF model on MNIST with different ε levels.

3.1.2 Results and discussion

The results are in Figure 3.2. For every model, the accuracy drops rapidly with increasing ε. Specifically for $\varepsilon = 0.25$, the accuracy of most models is lower than the reported 0.45 of Goodfellow et al. [2014b].

To our surprise, all traditional RBF models achieved worse accuracy than the CNN for every ε level. We thus found only limited support for the hypothesis of RBF models being more robust to adversarial examples than the more commonly used CNNs. Only the RBFolutional network was more robust than both the CNN and the adversarially trained CNN, with accuracy of 0.39 for $\varepsilon = 0.25$.

3.2 The effects of rounding on MNIST

Kurakin et al. [2016a] tried various approaches to defending a model against adversarial examples by distorting the input image, such as Gaussian blur, noise or change of brightness. None of them proved to be very effective. What they have in common is that they keep the space of images "mostly continuous" – not much less than the original space, which is of course discretized by the finite representation of numbers in the computer. We can then ask whether a more significant discretization of the image space could lead to higher robustness to adversarial perturbations due to removing the neighborhood of an image into which it could be moved by the perturbations. At the same time, such discretization must not greatly lower the accuracy of the defended model on clean examples.

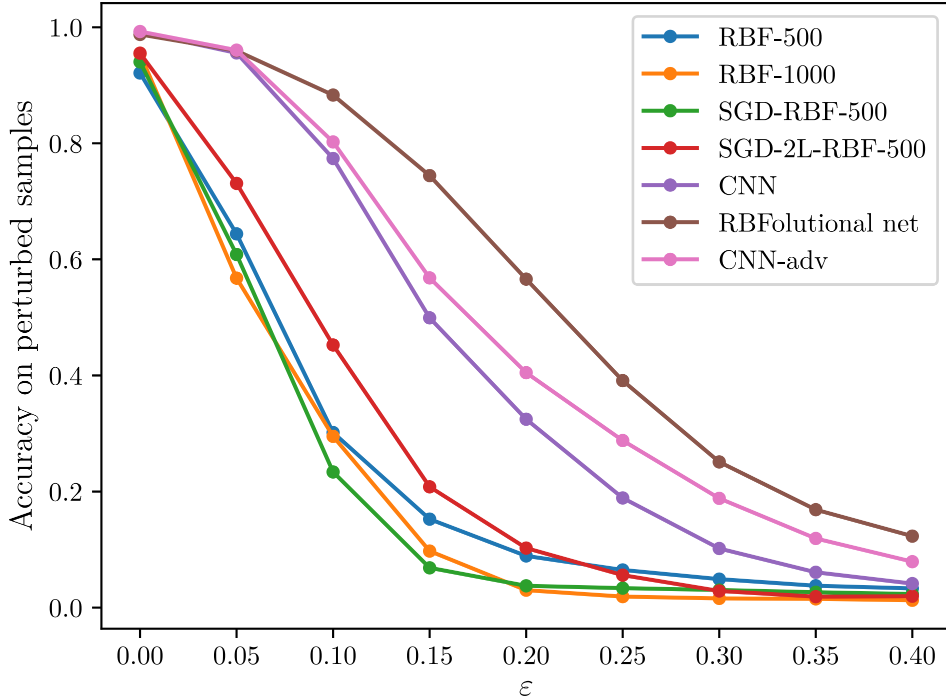

Figure 3.2: Accuracy of custom RBF and CNN models on MNIST based adversarial examples generated by FGSM with varied ε.

3.2.1 Method

We take generalized rounding as our discretization method. For each channel of each image pixel, we divide the interval $[0, 1]$ into n consecutive *rounding intervals*, each having the same length $l = 1/n$ for some $n \in \mathbb{N}, n \geq 1$. If the image is not being rounded at all we consider $l = 0$ for convenience. The new value of the channel c after rounding is then

$$\text{round}(c, l) = \begin{cases} c & \text{if } \ l = 0, \\ \lfloor (c \cdot n) + 1/2 \rfloor / n & \text{if } \ l = 1/n. \end{cases}$$

If rounding interval size is 1, $\text{round}(c, l)$ corresponds to regular rounding.

We applied this method to adversarial examples generated from the MNIST test set by FGSM and iterative FGSM for our CNN trained on MNIST training set (the architecture is the same as in Section 3.1). The parameters for iterative FGSM were chosen to match those of Kurakin et al. [2016a]: the number of iterations was $\lceil \min(255 \cdot \varepsilon + 4, 1.25 \cdot 255 \cdot \varepsilon) \rceil$ and max-norm bound on one iteration $\varepsilon_{\text{iter}} = 1/255$. We varied rounding interval size from 0 to 1 (note that only zero size or sizes of the form $1/n$ are applicable) and ε from 0 to 0.4 and measured accuracy.

3.2.2 Results and discussion

The results for FGSM are in Figure 3.3.

The accuracy as a function of rounding interval size is not monotonic. Especially for $\varepsilon = 0.1$ and 0.2, there are areas where it declines for several consecutive rounding interval sizes and then jumps higher than it was in the beginning. In some areas rounding actually reinforces the effect of FGSM. This is possible because it moves the values of individual pixels up or down much like FGSM, and their directions can agree. The reinforcement is then growing with rounding interval size, until at some point the rounding interval becomes so big that the direction of rounding is reversed. It is easily recognizable at rounding interval size 0.5: As most pixels in MNIST images are black (0) or white (1), FGSM will move them towards the middle (0.5). When $\varepsilon > 0.25$, rounding will set the perturbed pixels values to 0.5, thus increasing the effect of the perturbation and lowering the accuracy. When $\varepsilon < 0.25$, rounding sets the perturbed pixel values to 0 or 1, thus reverting the change introduced by FGSM and increasing the accuracy. See also Figure 3.4, where this effect is also visible.

The best rounding interval size is in this case 1. It is most successful at destroying adversarial examples and it still does not decrease the accuracy on clean images. This is not very surprising as MNIST images are mostly black and white. In fact, the original images from which MNIST was assembled were black and white and the gray tones emerged only as a result of size normalization [LeCun et al., 2010].

For iterative FGSM results, see Figure 3.5. The situation is simpler here, because iterative FGSM makes more subtle changes to the images than FGSM with the same ε. These are then easily reverted by rounding and at rounding interval size 1, the accuracy is over 98% for $\varepsilon < 0.4$.

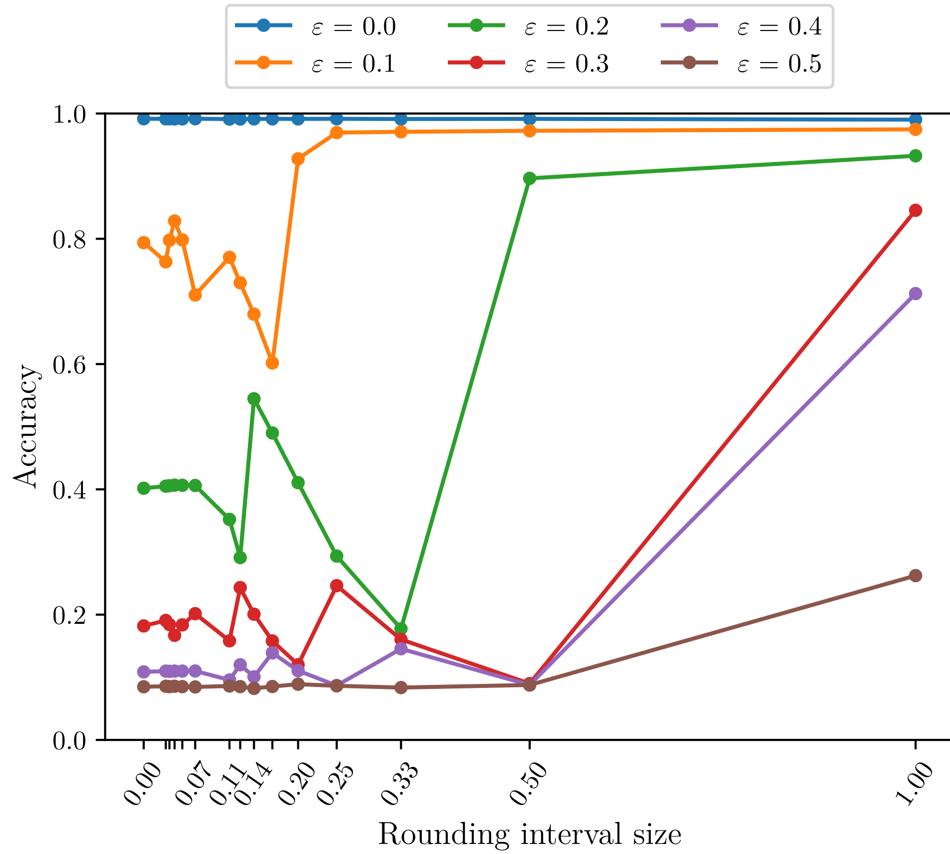

Figure 3.3: Accuracy of a CNN model on rounded adversarial examples generated by FGSM for different values of ε and rounding interval size. See Figure 3.4 for the resulting images.

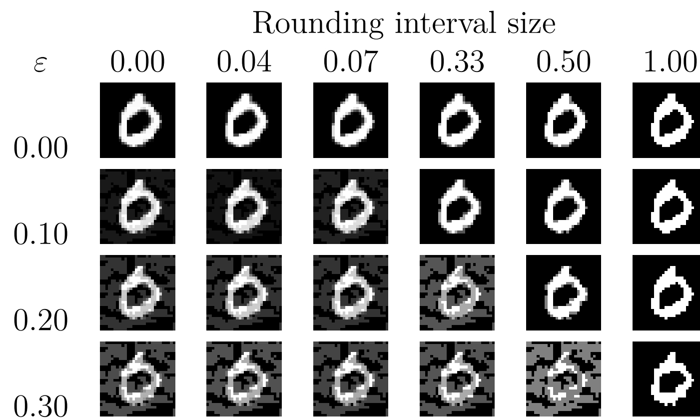

Figure 3.4: Illustration of the effects of FGSM and rounding on MNIST.

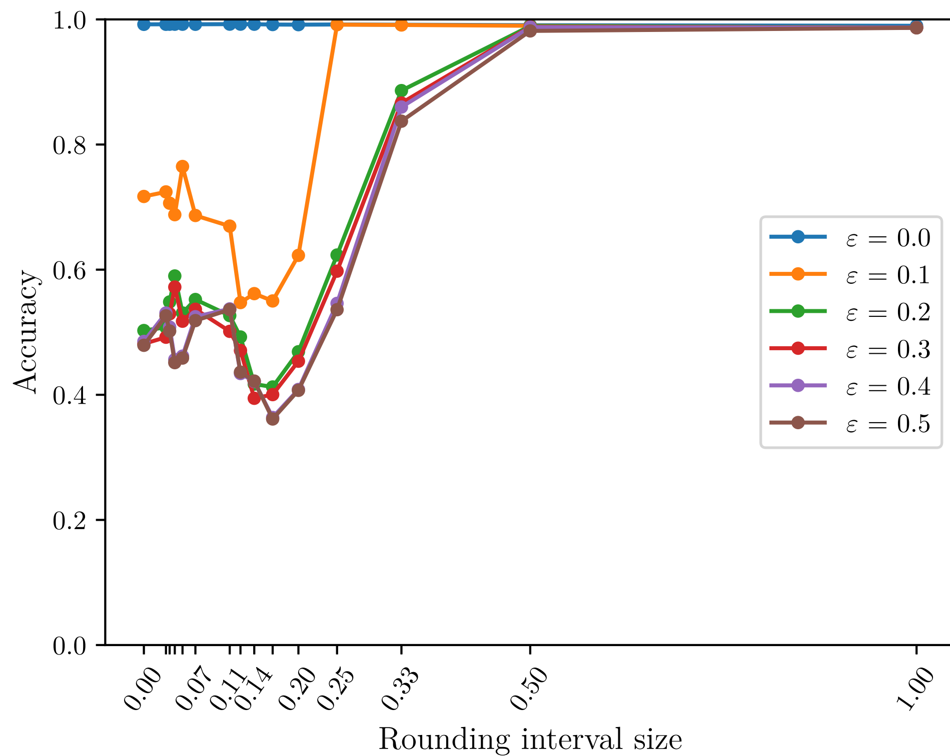

Figure 3.5: Accuracy of a CNN model on rounded adversarial examples generated by iterative FGSM for different values of ε and rounding interval size. See Figure 3.6 for the resulting images.

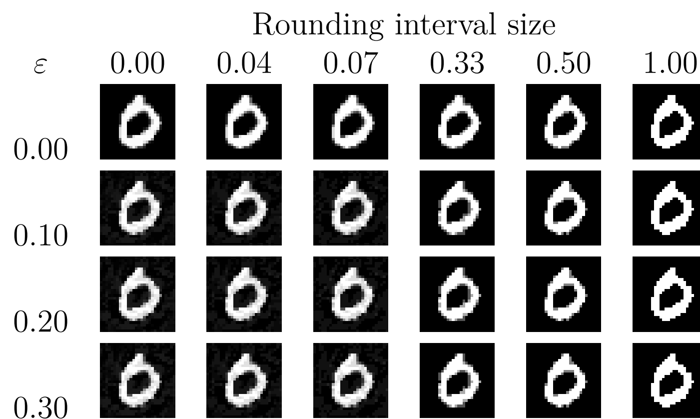

Figure 3.6: Illustration of the effects of *iterative* FGSM and rounding on MNIST.

3.3 The effects of rounding on ImageNet

As discretization of the image space proved quite successful at defending the classifier against adversarial examples on MNIST, a natural question is whether it could help also on ImageNet – a dataset bigger in virtually all measures, where adversarial examples are generally a greater problem, mostly because of the higher dimensionality and number of classes.

3.3.1 Method

Because training a CNN on ImageNet from scratch requires extensive resources, we resorted to using publicly available[2] pretrained InceptionV3 model [Szegedy et al., 2016]. This model achieves 96.54% top-5 accuracy on ILSVRC 2012 test set.

We generated adversarial examples for this model using FGSM and iterative FGSM. Again, we chose FGSM because it is widely used and efficient to compute. Iterative FGSM then serves as a representative of more expensive and subtler, but less robust methods. Its parameters were the same as in our experiment on MNIST.

Our defensive discretization was again generalized rounding of individual pixel channels. Note that the images have three color channels (red, green and blue), therefore even with rounding interval size 1, there are still 8 possible colors for each pixel.

We varied ε from 0 to 0.125 and rounding interval size from 0 to 1 and computed accuracy and average confidence on the resulting rounded adversarial examples generated from the first 10 000 (5000 for iterative FGSM) images from ILSVRC 2012 validation set.

3.3.2 Results and discussion

The results for FGSM are in Figure 3.7. Unlike on MNIST, rounding decreases accuracy on clean examples, finishing at low 69% for rounding interval size 1. At the same time it increases accuracy on adversarial examples only slightly. Maybe the most promising rounding interval size is 0.2, where the accuracy on clean images is still 90% and for a few values of $\varepsilon > 0$ it is near its maximum. For $\varepsilon = 1/255$, it brings the accuracy up from 66% to 85%. Examples of the resulting images are in Figure 3.9.

For iterative FGSM results, see Figure 3.8. In this case, rounding helped significantly more. The recommendable rounding interval size again seems to be 0.2 and even for high $\varepsilon = 32/255$, rounding with this rounding interval size brings the accuracy up from 13% to 69%. Similarly to the results of rounding on MNIST, this can be explained by the subtlety of the changes introduced by the iterative FGSM method, which are relatively easily eliminated by the greater changes made by rounding. See Figure 3.10 for examples of the perturbed and rounded images.

[2] https://www.tensorflow.org/tutorials/image_recognition

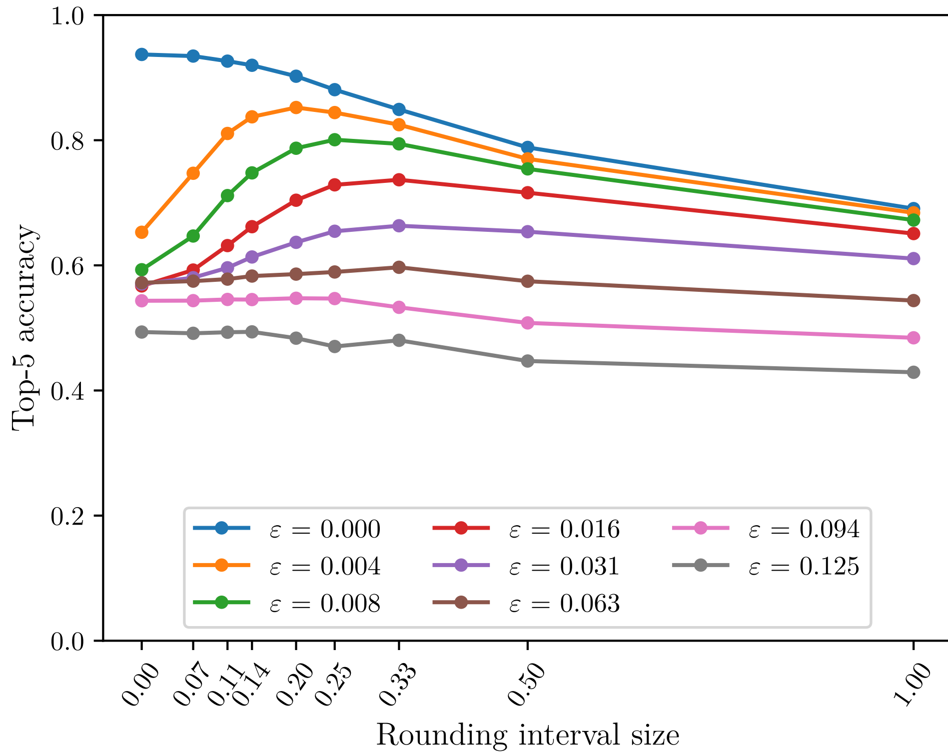

Figure 3.7: Accuracy of Inception-v3 model on rounded adversarial examples generated by FGSM for different values of ε and rounding interval size.

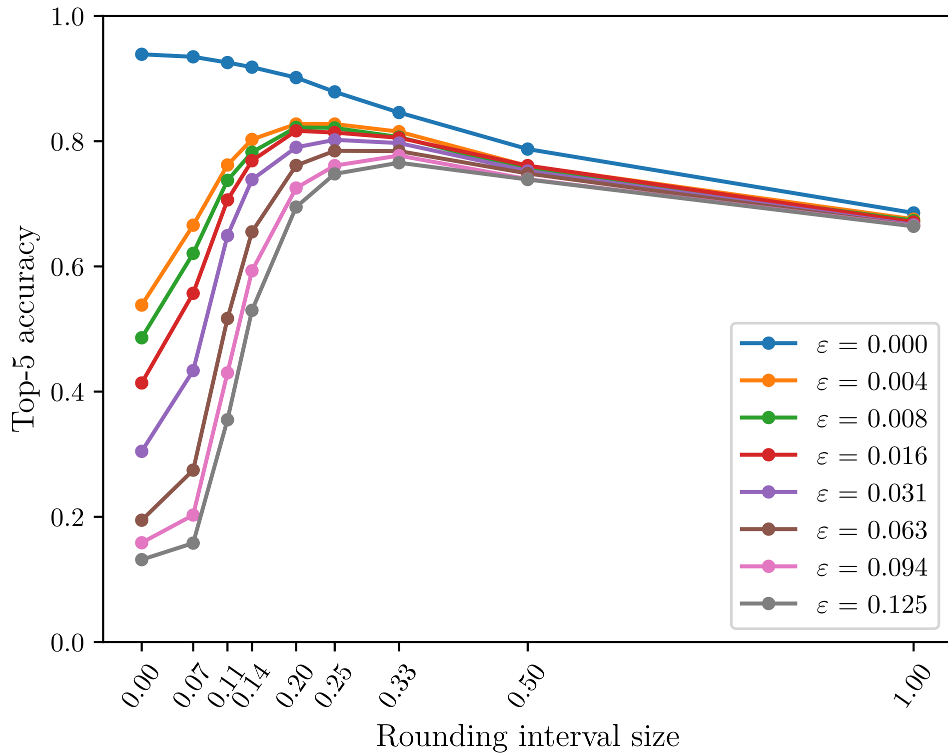

Figure 3.8: Accuracy of Inception-v3 model on rounded adversarial examples generated by *iterative* FGSM for different values of ε and rounding interval size.

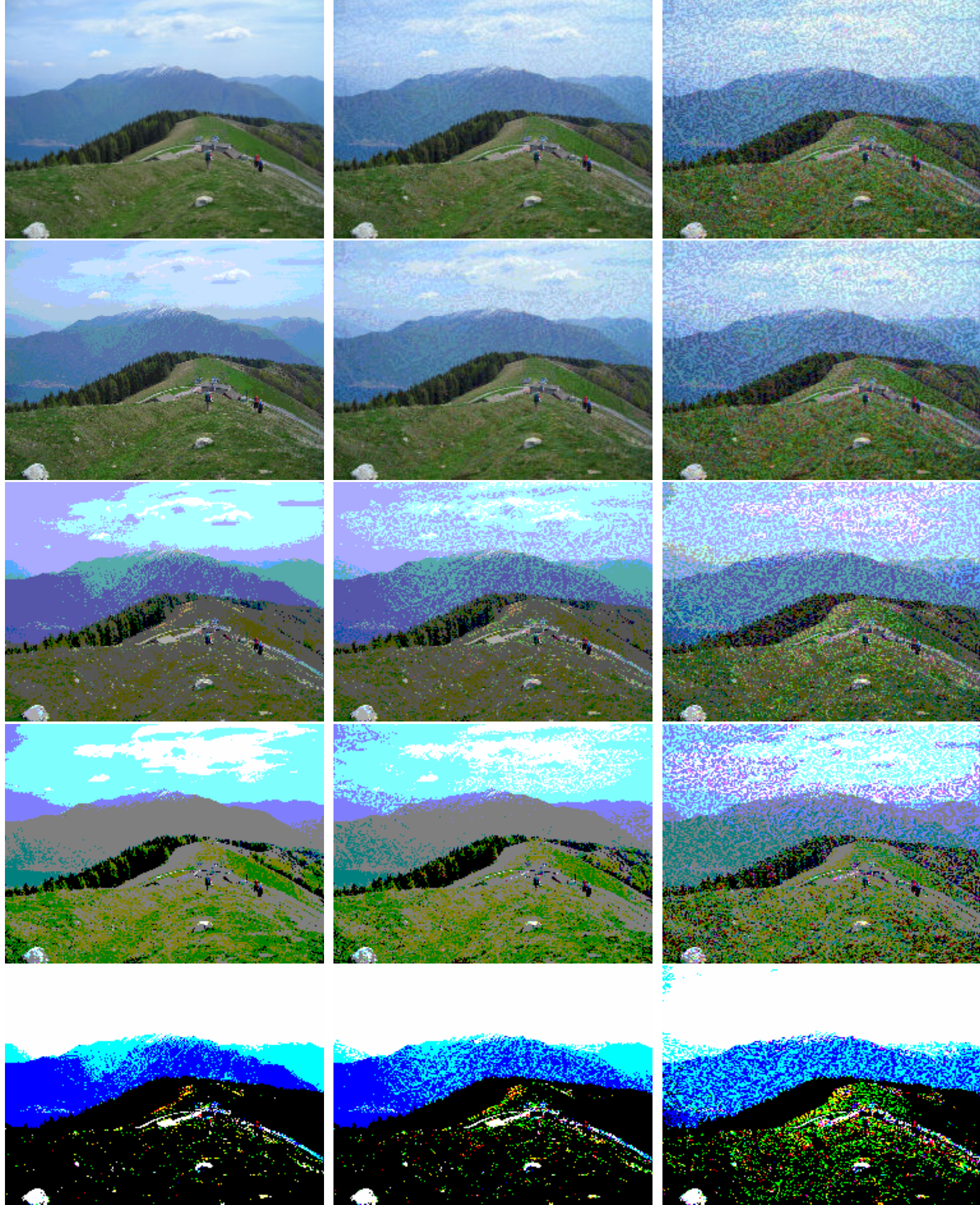

Figure 3.9: Illustration of the effects of rounding and FGSM on ImageNet sample. The original image (upper left) has been perturbed by FGSM with $\varepsilon = 8/255$ (middle column) or $\varepsilon = 24/255$ (right column) and then rounded with rounding interval size of 1/9, 1/3, 1/2 and 1 for the second to fifth row, respectively.

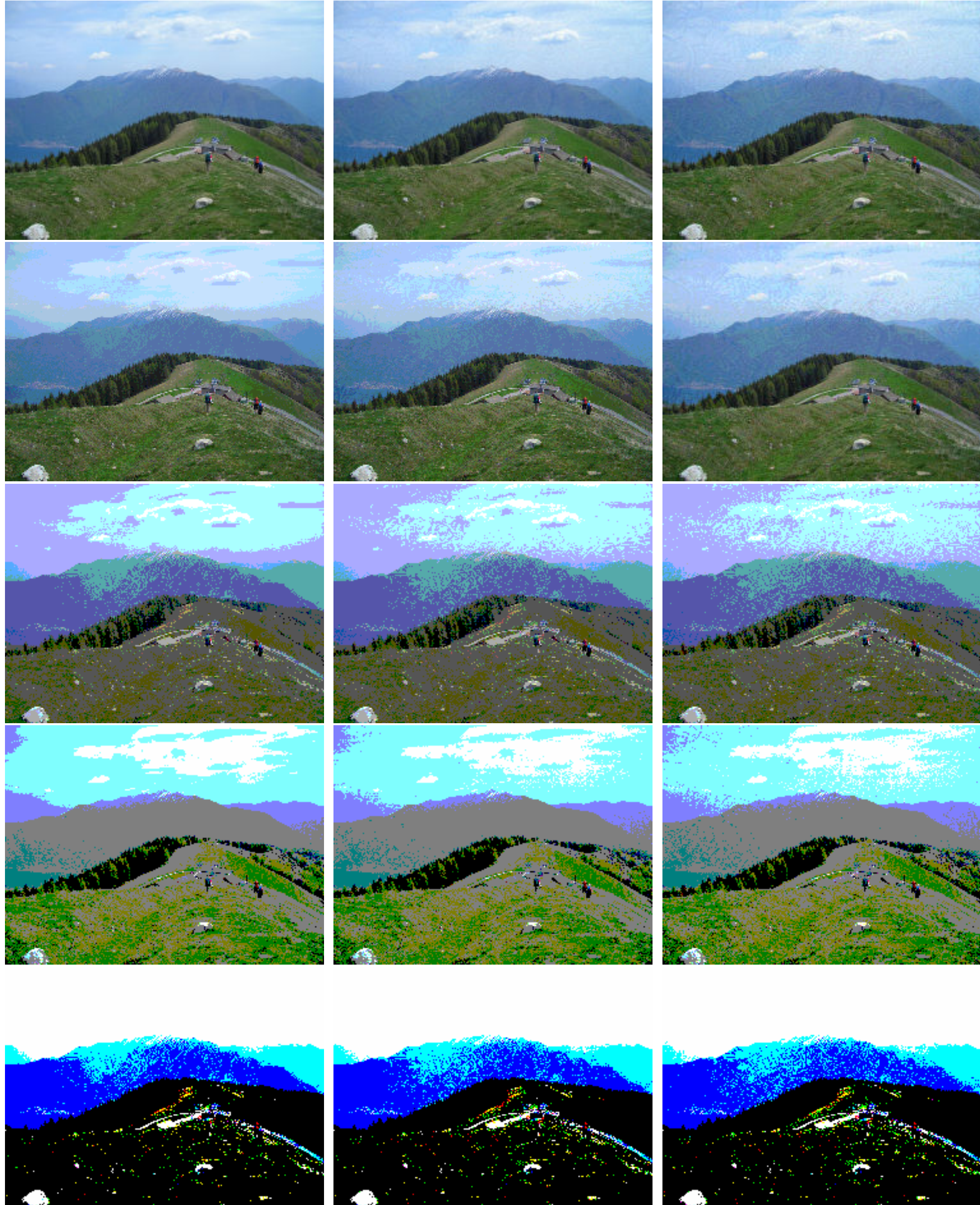

Figure 3.10: Illustration of the effects of rounding and iterative FGSM on ImageNet sample. The original image (upper left) has been perturbed by *iterative* FGSM with $\varepsilon = 8/255$ (middle column) or $\varepsilon = 24/255$ (right column) and then rounded with rounding interval size of 1/9, 1/3, 1/2 and 1 for the second to fifth row, respectively.

The effects of rounding on average confidence are in Figure 3.11. They correspond to the effects on accuracy: the confidence lowered by FGSM is not much affected by rounding; on the other hand, iterative FGSM lowers the confidence much more, but rounding quickly restores it to higher levels.

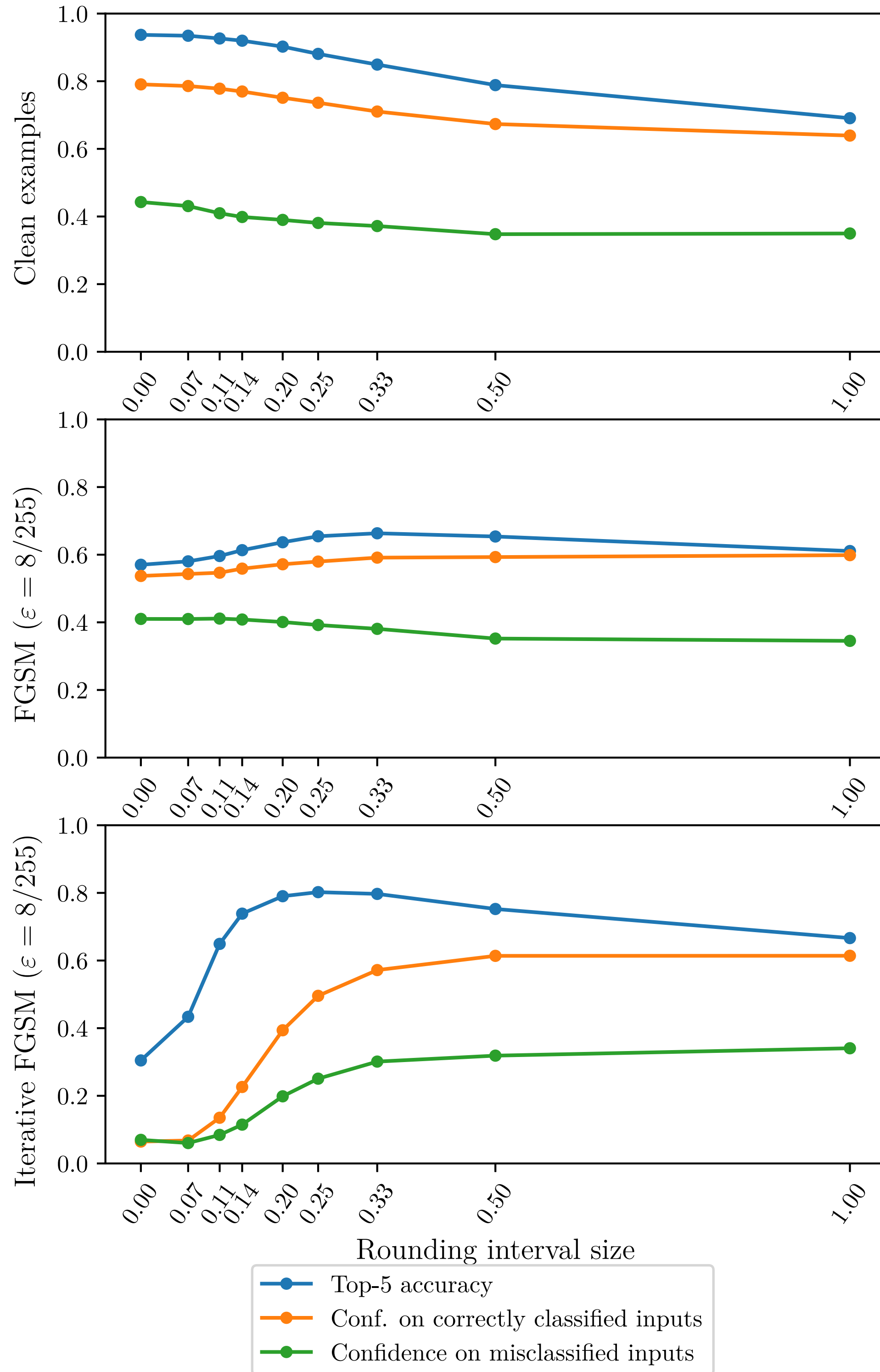

Figure 3.11: The effects of rounding on accuracy and average confidence of InceptionV3 model on clean examples (top), examples perturbed by FGSM (middle) and iterative FGSM (bottom) with $\varepsilon = 8/255$.

To better understand why rounding helps or does not help against FGSM, we tried to quantify the restrictions it poses on the images. We computed the *differing channels rate* – the rate of pixel channels that would have been rounded to a different value had they not been adversarially perturbed. For the results, see Figure 3.12 (FGSM) and Figure 3.13 (iterative FGSM). Generally, the differing channels rate is the highest when no rounding is applied, and goes quickly to zero with increasing rounding interval size. This decrease is even more pronounced for examples perturbed by iterative FGSM. After some point, almost all FGSM perturbation is removed and further decrease in accuracy is due to the rounding only.

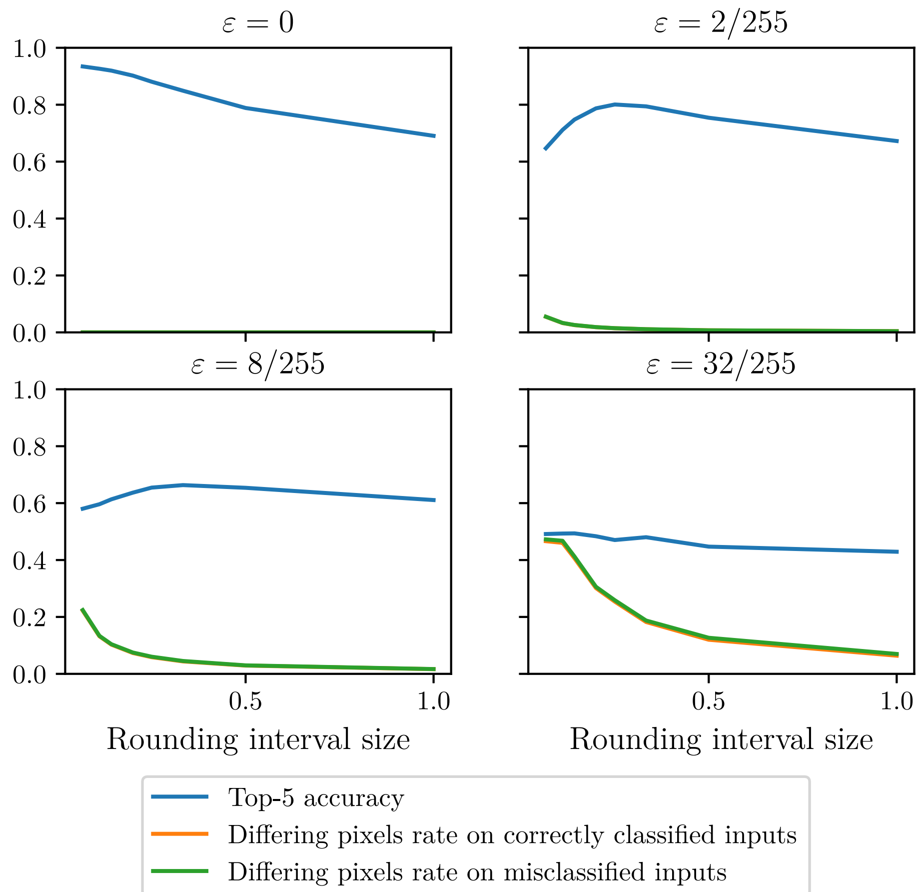

Figure 3.12: Comparison of average top-5 accuracy and the differing channels rate on adversarial examples created for InceptionV3 model by FGSM with four different values of ε. The differing channels rate goes quickly to zero with growing rounding interval size (more slowly with larger ε). At some point, almost no FGSM perturbation survives. After that, rounding is mostly responsible for any lowering of accuracy.

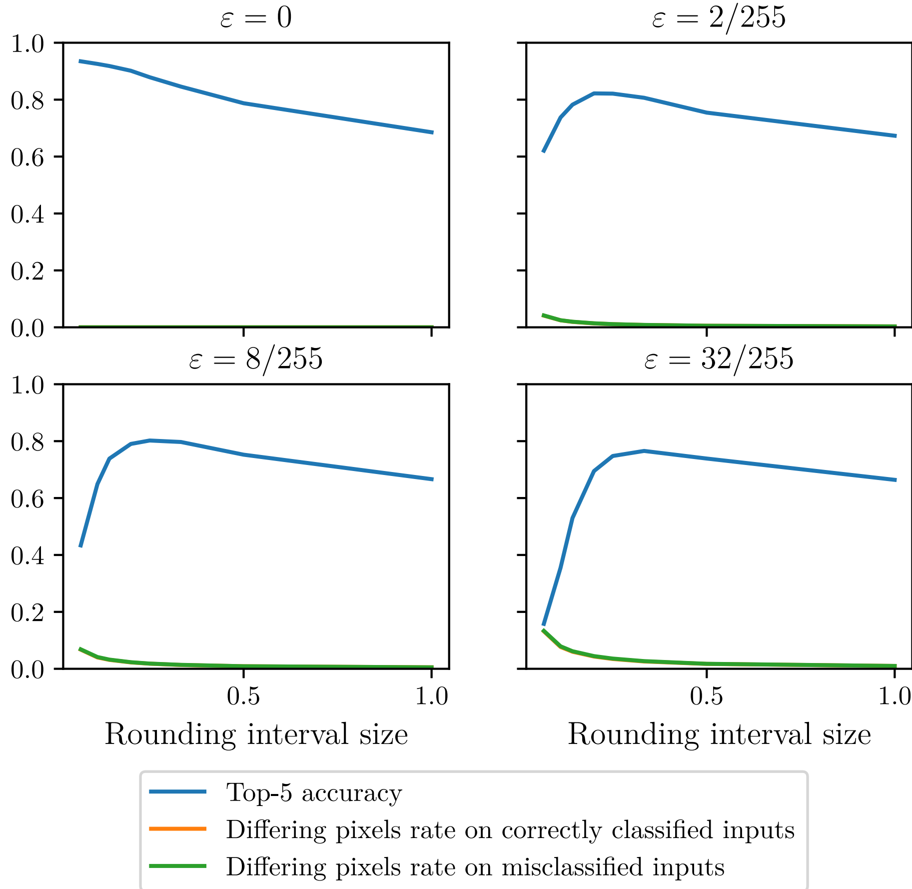

Figure 3.13: Comparison of average top-5 accuracy and the differing channels rate on adversarial examples created for InceptionV3 model by *iterative* FGSM with four different values of ε. The differing channels rate goes to zero even faster than for regular FGSM (Figure 3.12), hence rounding can increase the accuracy more before destroying the images.

As rounding with a larger interval distorts the images (see Figure 3.9) and thus decreases the accuracy of the model, we could ask whether some further modification of the images (after rounding has removed the adversarial perturbation) could make them more recognizable again. The rounded images look grainy, with big differences between neighboring pixels. Therefore we tried smoothing them with Gaussian blur (see examples in Figure 3.14). The results are in Figure 3.15 (FGSM) and 3.16 (iterative FGSM). Blurring did not help much in restoring the accuracy decreased by rounding. For rounded clean examples, it makes almost no difference. The effect is slightly better for higher ε, where it can be also caused by the blurring destroying some of the adversarial perturbation, as has been previously shown by Kurakin et al. [2016a].

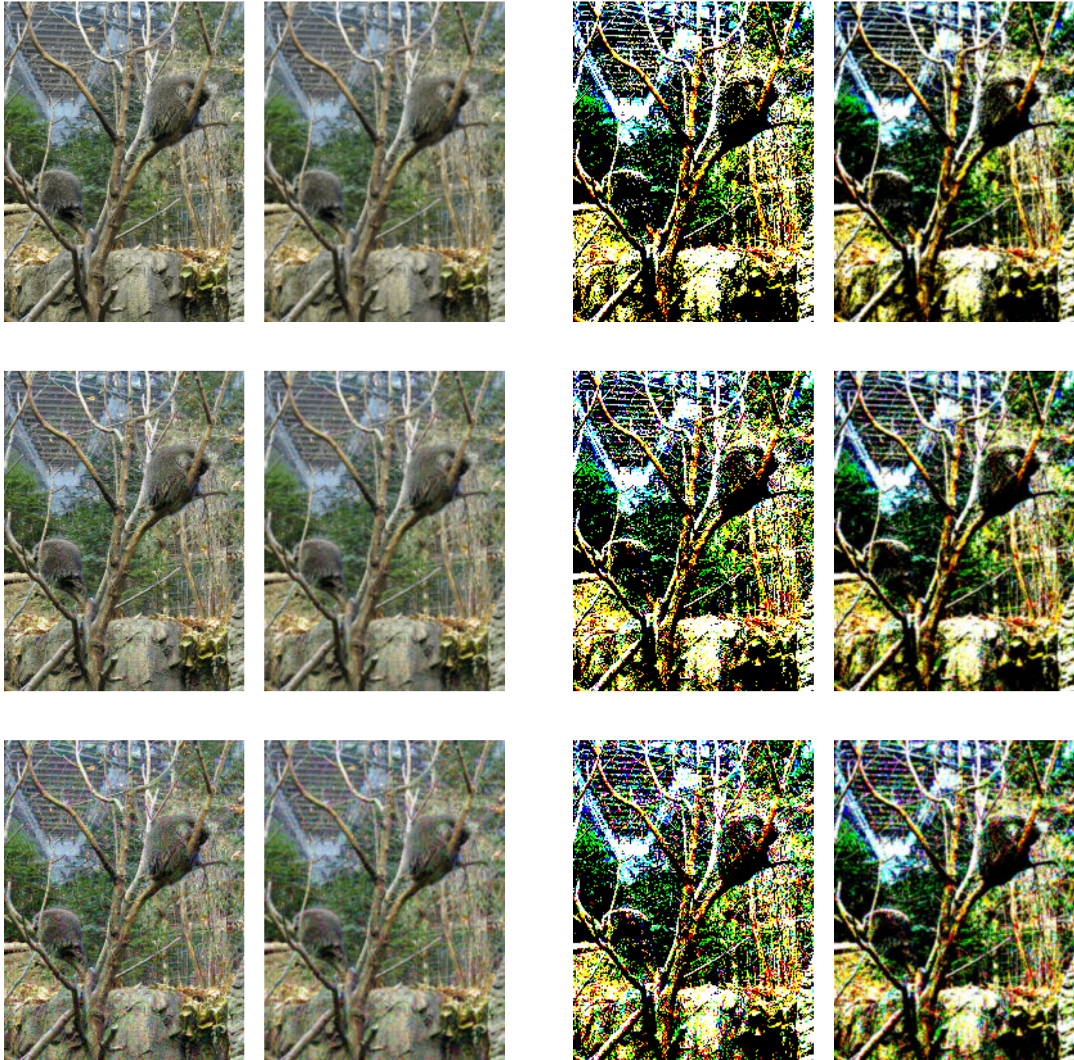

Figure 3.14: Illustration of the effects of Gaussian blur on rounded adversarial examples. The original images had been perturbed by FGSM with $\varepsilon = 8/255$ (second row) or $\varepsilon = 16/255$ (third row), rounded with rounding interval size $1/15$ (pairs in the left column) or 1 (pairs in the right column). In each pair of the images, the image on the right was then blurred.

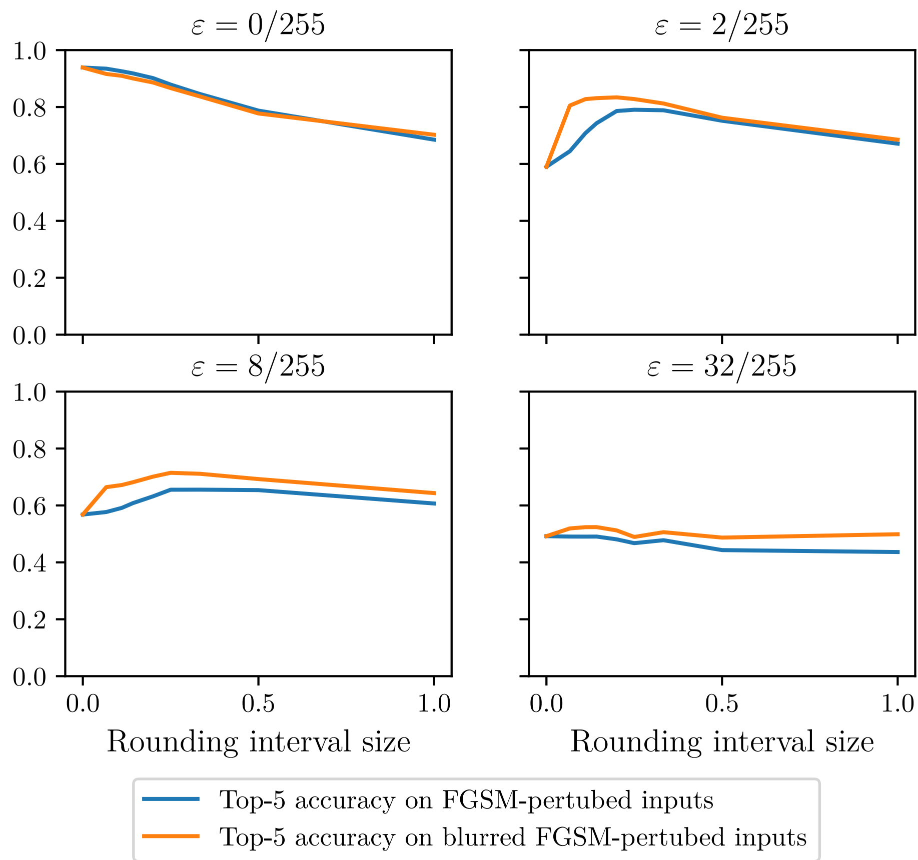

Figure 3.15: The effect of blurring on accuracy for inputs perturbed by FGSM.

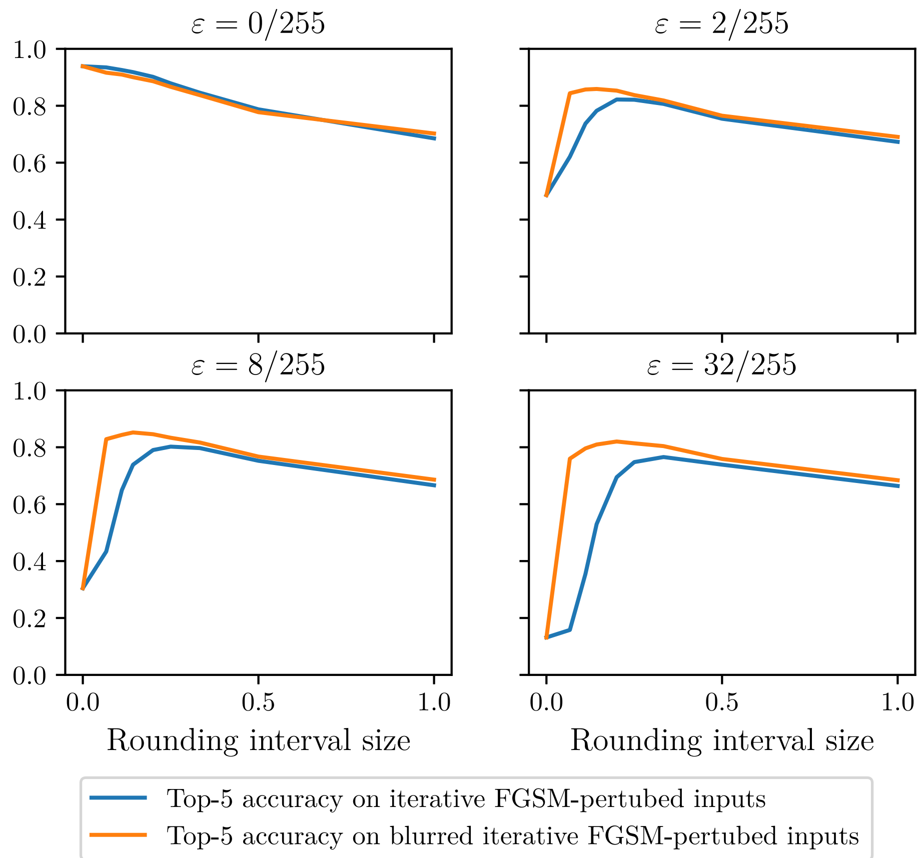

Figure 3.16: The effect of blurring on accuracy for inputs perturbed by *iterative* FGSM.

3.4 Common adversarial perturbation

All current methods of adversarial examples generation need access to the clean example in order to create the adversarial perturbation that is then added to the clean example. We might consider a situation where the attacker does not have access to the clean example when generating the perturbation. For example, they might be preparing a sticker that would be applied to a traffic sign to change the way it is perceived by an autonomous car without knowing the exact sign (or even its type) that will be available. If the attacker could do this, they would also be able to mass-produce the one perturbation without fine-tuning it to each original example they aim to perturb, so as to save time and resources.

We tried to create such adversarial perturbation on MNIST. We performed both untargeted and targeted attack against the CNN from section 3.1. We set our ε to 0.25, which is in line with Goodfellow et al. [2014b] and does not stop humans from recognizing the correct class.

In the untargeted scenario, we maximized the following error function by manipulating our common perturbation η:

$$\underset{\eta}{\operatorname{argmax}} \sum_{i} J_f(x^{(i)} + \eta, y^{(i)}),$$

where $x^{(i)}$ are images from the MNIST training set, $y^{(i)}$ their corresponding labels and $J_f(x, y)$ is the model's error function for a single input.

In the targeted case, we tried to make the model output label t on perturbed examples, whose correct label is different from t. To this end, we minimized the following error function by manipulating the common perturbation η:

$$\underset{\eta}{\operatorname{argmin}} \sum_{i:y^{(i)} \neq t} J_f(x^{(i)} + \eta, t),$$

where J_f and $x^{(i)}$ are the same as above.

To carry out the optimization, we ran the Adam optimizer for 20 epochs.

As a baseline, we used a random perturbation that modified each pixel by either adding or subtracting 0.25.

After adding a perturbation, each example was clipped so that the value of every dimension fell into the [0, 1] interval.

3.4.1 Results and discussion

We measured the accuracy of the CNN model on the MNIST test set. The model achieves accuracy of 0.993 on clean examples. When the examples were perturbed by the untargeted common perturbation, the accuracy fell to 0.650, whereas when the random untargeted common perturbation was used, the accuracy decreased only slightly to 0.982 (averaged over 10 runs). Given the restrictions, we can consider this attack to be quite successful. See Figure 3.17 for the result of applying the common perturbation to MNIST digits.

In the targeted scenario, we measured the targeted misclassification rate on examples from the MNIST test set whose original correct labels were different from t, but we replaced them with t. The average targeted misclassification rate was 0.001 on clean examples, 0.267 on examples with the targeted common

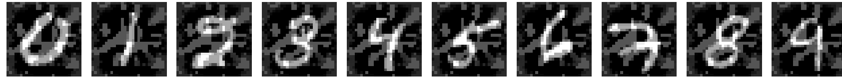

Figure 3.17: Illustration of the application of the common adversarial perturbation to MNIST digits.

target	clean acc.	common pert.	common random pert.
0	0.001	0.046	0.002
1	0.000	0.048	0.001
2	0.001	0.585	0.004
3	0.000	0.417	0.002
4	0.001	0.308	0.001
5	0.002	0.505	0.004
6	0.001	0.089	0.001
7	0.001	0.345	0.001
8	0.001	0.201	0.003
9	0.001	0.130	0.001

Table 3.1: Targeted misclassification rate.

perturbation added and 0.002 (averaged over 10 runs) on the examples with the random perturbation added. There were big differences in misclassification rate for different targets, see Table 3.1. These are probably due to some digits being similar to more digits than some others. This attack was less successful overall, but it can still cause considerable misclassification rate if the adversary selects an easy to target class.

Conclusion

In this thesis, we had three main goals, namely to review existing literature about adversarial examples, to find new kinds of defense against them and to explore new ways of adversarial example generation.

We presented a review of literature in Chapter 2. We then introduced our new defense and attack types in Chapter 3. We have focused more on the former, because at the current time, defenses are generally weaker than the attacks and are therefore in greater demand.

We compared how several models with RBF units and regular CNNs resist adversarial perturbation by FGSM on MNIST. Every traditional model with RBF units we tried achieved lower accuracy than a CNN for every level of ε, so we cannot recommend these to be employed when robustness to FGSM is needed. On the other hand, our RBFolutional network had higher accuracy than the CNN on adversarial examples and comparable accuracy on clean examples, even when the CNN had been adversarially trained. The RBFolutional network hence constitutes a promising alternative.

The second approach we tried, input space discretization by rounding of individual pixels, turned out to be quite successful at defending the CNN from both FGSM and iterative FGSM attacks on MNIST. When the rounding interval size is 1, i.e. every pixel is forced to stay either black or white, the accuracy of the model is higher than 98% for $\varepsilon < 0.4$. Therefore, we can recommend this method.

We applied rounding also as a defense against FGSM and iterative FGSM for the InceptionV3 ImageNet model. Probably the best trade-off between accuracy on clean and adversarial examples was reached for rounding interval size 0.2. This size lowered the accuracy on clean examples from 96.5% to 92% and kept it above 74% for iterative FGSM with ε as high as 32/255. But for regular FGSM, it was less successful: the accuracy stayed above 74% only for $\varepsilon \leq 2/255$. Whether rounding should be used on ImageNet depends on task specific requirements – on the importance of accuracy on clean examples and on the probability and strength of a potential adversarial attack.

Finally, we explored the possibility of generating adversarial perturbation without looking at the particular input that should be perturbed. We optimized a common perturbation for a CNN trained on MNIST. In the untargeted case, we were able to reduce the model's accuracy from 99.3% to 65%. The targeted misclassification rate depended heavily on the target, but overall was relatively low 27%.

Future research might include training and testing the RBFolutional network on ImageNet, which we did not attempt in this work due to lack of computational resources. It would also be interesting to try to combine several defenses and evaluate their collective effectiveness.

We hope that this thesis might serve as a reference about adversarial examples, and that the RBFolutional network and input space discretization could be improved upon and contribute to creating robust machine learning models.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org. [Cited on page 36]

Anish Athalye and Ilya Sutskever. Synthesizing robust adversarial examples. *CoRR*, abs/1707.07397, 2017. URL `http://arxiv.org/abs/1707.07397`. [Cited on pages 24 and 26]

Leo Breiman. *Classification and regression trees*. Wadsworth International Group, 1984. ISBN 9780412048418. [Cited on page 16]

David S Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, Royal Signals and Radar Establishment Malvern (United Kingdom), 1988. [Cited on page 15]

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. [Cited on page 17]

David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 215–242, 1958. [Cited on page 13]

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009. [Cited on pages 6, 18, and 19]

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011. [Cited on page 13]

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001. ISBN 0387848576. [Cited on pages 5 and 17]

Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982. [Cited on page 14]

Pavel Golik, Patrick Doetsch, and Hermann Ney. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *Interspeech*, volume 13, pages 1756–1760, 2013. [Cited on page 5]

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014a. [Cited on page 33]

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016. ISBN 0262035618. [Cited on pages 10, 11, 13, and 16]

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014b. [Cited on pages 18, 20, 21, 26, 27, 36, 37, 38, and 53]

Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. *CoRR*, abs/1412.5068, 2014. URL `http://arxiv.org/abs/1412.5068`. [Cited on pages 25 and 28]

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016. [Cited on pages 7, 15, 18, and 32]

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. [Cited on page 29]

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. [Cited on pages 10 and 20]

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.55. [Cited on page 36]

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`. [Cited on pages 13 and 19]

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009. [Cited on page 6]

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. [Cited on pages 7, 15, and 18]

Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016a. URL `http://arxiv.org/abs/1607.02533`. [Cited on pages 19, 21, 22, 23, 24, 28, 38, 39, and 49]

Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *CoRR*, abs/1611.01236, 2016b. URL `http://arxiv.org/abs/1611.01236`. [Cited on pages 27 and 28]

Quoc V. Le, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Marc'Aurelio Ranzato, Jeffrey Dean, and Andrew Y. Ng. Building high-level features using large scale unsupervised learning. *CoRR*, abs/1112.6209, 2011. URL `http://arxiv.org/abs/1112.6209`. [Cited on page 19]

Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990. [Cited on pages 6, 14, and 15]

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998. [Cited on page 6]

Yann LeCun, Corinna Cortes, and Christopher JC Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010. [Cited on pages 19 and 40]

Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *CoRR*, abs/1611.02770, 2016. URL `http://arxiv.org/abs/1611.02770`. [Cited on pages 19, 21, 25, and 32]

Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982. [Cited on page 17]

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017. [Cited on pages 22, 27, and 28]

Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010. [Cited on page 36]

George A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995. [Cited on page 6]

Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. *CoRR*, abs/1412.1897, 2014. URL `http://arxiv.org/abs/1412.1897`. [Cited on page 33]

Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. *CoRR*, abs/1511.07528, 2015a. URL `http://arxiv.org/abs/1511.07528`. [Cited on page 22]

Nicolas Papernot, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. *CoRR*, abs/1511.04508, 2015b. URL `http://arxiv.org/abs/1511.04508`. [Cited on page 29]

Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against deep learning

systems using adversarial examples. *CoRR*, abs/1602.02697, 2016. URL http://arxiv.org/abs/1602.02697. [Cited on pages 29, 31, and 32]

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011. [Cited on page 36]

J. Ross Quinlan. *C4. 5: programs for machine learning*. Morgan Kaufmann, 1992. ISBN 1558602380. [Cited on page 16]

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. [Cited on page 8]

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986. [Cited on page 11]

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014. URL http://arxiv.org/abs/1409.0575. [Cited on pages 6, 7, and 18]

Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1528–1540, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978392. URL http://doi.acm.org/10.1145/2976749.2978392. [Cited on page 18]

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, pages 2171–2180, 2015. [Cited on page 6]

Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014. [Cited on page 6]

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. [Cited on page 13]

Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162, 2007. [Cited on page 33]

J. Su, D. Vasconcellos Vargas, and S. Kouichi. One pixel attack for fooling deep neural networks. *ArXiv e-prints*, October 2017. [Cited on page 30]

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013. URL `http://arxiv.org/abs/1312.6199`. [Cited on pages 3, 18, 19, 20, 22, 23, 25, 26, and 31]

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015. [Cited on pages 15, 18, and 32]

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016. [Cited on pages 24 and 43]

S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2): 22–30, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37. [Cited on page 36]

Petra Vidnerová and Roman Neruda. Evolutionary generation of adversarial examples for deep and shallow machine learning models. In *Proceedings of the The 3rd Multidisciplinary International Social Networks Conference on SocialInformatics 2016, Data Science 2016*, MISNC, SI, DS 2016, pages 43:1–43:7, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4129-5. doi: 10.1145/2955129.2955178. URL `http://doi.acm.org/10.1145/2955129.2955178`. [Cited on pages 29, 36, and 37]

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, pages 1058–1066, 2013. [Cited on page 6]

Y. T. Zhou and R. Chellappa. Computation of optical flow using a neural network. In *IEEE International Conference on Neural Networks*, volume 27, pages 71–78, 1988. [Cited on page 15]

List of Figures

1.1 Examples of the MNIST digits. 6
1.2 Examples of the CIFAR10/CIFAR100 images. 7
1.3 ILSVRC 2012 examples. 8
1.4 A perceptron. 9
1.5 A multilayer perceptron. 9
1.6 Activation functions. 11

2.1 Adversarial examples generated by L-BFGS. 19
2.2 Adversarial examples generated by FGSM (ImageNet). 21
2.3 Adversarial examples generated by FGSM (MNIST). 21
2.4 Comparison of adversarial examples. 23
2.5 A physical adversarial example. 26
2.6 One pixel attacks. 30
2.7 Fooling examples. 33

3.1 Effects of FGSM targeting RBF. 38
3.2 Accuracy of RBF and CNN models on FGSM (MNIST). 39
3.3 Accuracy of a CNN on rounded adversarial examples (FGSM). . . 41
3.4 Effects of FGSM and rounding on MNIST. 41
3.5 Accuracy of a CNN on rounded adversarial examples (it. FGSM). 42
3.6 Effects of it. FGSM and rounding on MNIST. 42
3.7 Acc. of Inception-v3 on rounded adv. examples (FGSM). 44
3.8 Acc. of Inception-v3 on rounded adv. examples (it. FGSM). . . . 44
3.9 Effects of rounding and FGSM on ImageNet. 45
3.10 Effects of rounding and it. FGSM on ImageNet. 46
3.11 Effects on confidence. 47
3.12 Differing channels rate (FGSM). 48
3.13 Differing channels rate (it. FGSM). 49
3.14 Effects of blurring. 50
3.15 Accuracy on blurred inputs (FGSM). 51
3.16 Accuracy on blurred inputs (it. FGSM). 52
3.17 A common perturbation. 54

List of Abbreviations

CART	Classification And Regression Tree
CIFAR	Canadian Institute For Advanced Research
CNN	Convolutional Neural Network
CPPN	Compositional Pattern Producing Network
DCN	Deep Contractive Network
DNN	Deep Neural Network
EOT	Expectation Over Transformation
FGM	Fast Gradient Method
FGSM	Fast Gradient Sign Method
GAN	Generative Adversarial Network
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
ML	Machine Learning
MLP	MultiLayer Perceptron
MNIST	Modified National Institute of Standards and Technology
MSE	Mean Squared Error
NN	Neural Network
PGD	Projected Gradient Descent
RBF	Radial Basis Function
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine