TRABAJO FIN DE GRADO

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

Deep neural network classifiers

Study of the instability of the associated function classes

Autor
Pablo Álvarez Cabrera

Tutor
Nicolás Pérez de la Blanca Capilla

Facultad de Ciencias

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Junio de 2020

ABSTRACT

This work is focused on studying a recently discovered issue affecting deep neural networks: *adversarial examples*. Concretely, it has been found that by applying small but intentionally crafted perturbations to the inputs, it is possible to drastically change the output of the models. It is therefore a problem of stability associated with the classes of functions implemented by deep neural networks. As these models are becoming widely used to perform critical tasks due to their high efficiency, this is a serious problem in which a huge effort has been done in the last years in order to find a solution.

Throughout the work, we will focus on image classifiers based on deep neural networks. In this way, the main techniques for generating adversarial examples are going to be discussed and analyzed in detail. As some of these methods are profoundly based on geometric characteristics of high-dimensional spaces, a detailed explanation of the required geometrical and analytical concepts will be provided. Once the techniques to be used have been explained, they will be analysed experimentally and tested on different models and datasets, to obtain significant and reliable conclusions.

Lastly, the main contributions made in this field will be revisited, including some interesting proposals that could be considered in the future.

KEYWORDS: Artificial intelligence, deep learning, convolutional neural networks, image recognition, adversarial examples.

RESUMEN EXTENDIDO

Hoy en día, las redes neuronales profundas han alcanzado una gran popularidad en el campo de la Inteligencia Artificial. Se trata de modelos que son capaces de aproximar funciones realmente complejas, y que han resultado ser especialmente efectivos en áreas como la visión por computador y el procesamiento del lenguaje natural. Por poner algunos ejemplos, este tipo de modelos ha sido usado en problemas de reconocimiento facial, en juegos complejos como el *Go* e incluso en la detección de cáncer a través de imágenes de rayos X, llegando a alcanzar resultados comparables (y a veces superiores) a los de un humano. Todos estos logros han hecho que el aprendizaje profundo (*deep learning*) tenga cada vez más influencia en nuestra sociedad, pues estos modelos ya se usan en aplicaciones críticas como la conducción autónoma y la medicina.

Sin embargo, actualmente no hay una comprensión detallada del comportamiento de estos algoritmos y, por tanto, no se conocen las limitaciones subyacentes. Así, el objetivo de este trabajo es analizar uno de los mayores problemas a los que se enfrenta el aprendizaje profundo en la actualidad: los ejemplos adversarios (*adversarial examples*).

Este problema está relacionado con la estabilidad de las redes neuronales: en vista de su gran capacidad de aprendizaje, uno puede esperar que estos modelos sean robustos ante pequeñas modificaciones en los datos de entrada, pero se ha descubierto que, aplicando una pequeña modificación a una entrada, es posible cambiar por completo la salida del modelo, dando lugar a errores que ponen en riesgo su uso. Aunque el problema anterior afecta a un amplio grupo de modelos, en este trabajo nos vamos a centrar en clasificadores de imágenes basados en redes neuronales profundas, pues es el caso que más se está investigando en la actualidad. Supongamos, por ejemplo, que una red neuronal se usa para detectar una cierta enfermedad a partir de imágenes médicas. En este caso, un ejemplo adversario consistiría en una imagen que ha sido ligeramente modificada para engañar al modelo, haciendo que su predicción sea incorrecta. Para un humano, ambas imágenes serían aparentemente iguales, luego no se produciría este error. Todo esto ha llevado a un gran replanteamiento en el uso de estos algoritmos, pues queda demostrado que no podemos fiarnos por completo de sus predicciones hasta que no logremos comprender y evitar este tipo de comportamientos.

De forma más detallada, un ejemplo adversario es el dato más cercano al original (bajo una cierta medida) que consigue engañar al modelo. En el caso de imágenes, estas pueden ser vistas como puntos en un cierto espacio vectorial, considerando los valores de sus píxeles como un

vector. Así, puntos cercanos al original se corresponden con imágenes parecidas, que mantienen el contenido semántico de la imagen original.

Generar ejemplos adversarios no es nada fácil desde el punto de vista teórico, pues este proceso involucra resolver un problema de optimización complejo en espacios de dimensiones elevadas. Para este propósito, a lo largo del trabajo se describen una serie de algoritmos y técnicas que permiten obtener resultados aproximados, utilizando distintos enfoques. Así, cada método tendrá asociado un problema de optimización distinto, aunque similar al original, de forma que habrá métodos que busquen la mínima modificación posible (en sentido euclídeo) para generar un ejemplo adversario, otros que busquen modificar el menor numero de píxeles, etc. Para ello, se usarán distintos aspectos geométricos y analíticos del modelo y del espacio de entrada, por lo que la introducción de algunos conceptos y resultados matemáticos será esencial para su comprensión. De esta forma, se proporciona un resumen que contiene los principales resultados que vamos a usar de diferenciabilidad en varias variables, así como distintos resultados geométricos en espacios vectoriales reales.

Una vez se hayan detallado los métodos, se realizarán una serie de experimentos con el objetivo de extraer conclusiones sobre los mismos. Cada uno de los métodos será analizado en distintos conjuntos de datos, y usando algunos de los clasificadores más recientes. De esta forma, se pretende generar una cantidad de información suficiente como para realizar un análisis detallado en el que se consideren diversas situaciones, proporcionando así una visión más profunda del problema. En concreto, se estudiarán varios aspectos, como la efectividad de cada método, la vulnerabilidad de los modelos considerados, la magnitud de las modificaciones generadas y el tiempo de ejecución necesario, entre otros. Durante el proceso se proporcionan numerosos ejemplos de imágenes adversarias, que permitirán visualizar cómo actúa cada método, para así poder comparar los resultados obtenidos con cada uno de ellos.

Aunque el problema de los ejemplos adversarios siga estando abierto, a lo largo de estos últimos años se han dado diversas explicaciones de por qué los modelos exhiben este comportamiento, así como numerosas propuestas que intentan paliar su efecto, dando lugar a modelos más robustos. La última parte del trabajo analiza estas propuestas, proporcionando una visión general de la situación actual del problema. Todo parece indicar que, debido a la inmensa información presente en los píxeles de la imagen, en conjunto con la gran capacidad de aprendizaje de estos modelos, los algoritmos de aprendizaje actuales hacen que los modelos no aprendan en realidad las características que usan los humanos para clasificar una cierta imagen (por ejemplo, la forma, la textura...), si no otras muy distintas y posiblemente más simples. Estas *falsas estructuras* que, si bien sirven también para clasificar una imagen, no son robustas ante ciertas modificaciones de la misma, que son las

que se buscan en los ejemplos adversarios. En cuanto a las propuestas de mejora, varias de ellas se centran en realizar un preprocesado de los datos que permita reducir el efecto de cualquier modificación adversaria, y otras en controlar varios aspectos internos del modelo, como la constante de Lipschitz de la función que implementa. Sin embargo, todos estos métodos solucionan parcialmente el problema, pues nada indica que sean resistentes ante cualquier tipo de ataque. Dicho de otra forma, se pueden desarrollar ataques específicos para los que el método de defensa en cuestión no sea eficaz.

Al final del trabajo se propone una posible mejora de los modelos que podría evitar este problema en el futuro. Esta implicaría un cambio en la forma en la que los modelos aprenden, pues consistiría en adaptar la red neuronal para que extraiga información sólida en las capas intermedias, como los contornos y las texturas de la imagen, que son robustas ante las modificaciones que suponen los ejemplos adversarios. Aún así, esto requiere la existencia de alguna técnica que sea capaz de extraer dicha información de manera correcta e incluirla en el modelo.

Por último, podemos concluir diciendo que, aunque el campo del aprendizaje profundo esté actualmente en auge, no podemos olvidarnos de que aún no conocemos por completo que ocurre internamente en estos modelos, esto es, qué información concreta de los datos de entrada se usa (y cómo) para llegar a las conclusiones finales. Esto puede dar lugar a otros problemas más allá de los ejemplos adversarios. De hecho, ya se conoce el gran poder de estos modelos, pero sus limitaciones no han hecho nada más que aparecer.

PALABRAS CLAVE: Inteligencia artificial, aprendizaje profundo, redes neuronales convolucionales, reconocimiento de imágenes, ejemplos adversarios.

AGRADECIMIENTOS

A mi familia y amigos, hacia quienes sólo puedo expresar mi más sincero agradecimiento por apoyarme durante esta etapa académica.

Me gustaría agradecer también a mi tutor, Nicolás Pérez, por su apoyo durante el desarrollo de este trabajo.

CONTENTS

1 INTRODUCTION

In the last years, deep learning has triggered the current rise of artificial intelligence, becoming one of the most powerful tools in machine learning. Deep neural networks have achieved impressive results in several problems that were inconceivable using other approaches, such as image classification, speech recognition and natural language processing, reaching human-competitive results in some tasks like face recognition [49] and disease detection [36].

The reason why this high performance is achieved is that neural networks can express arbitrary computation by using several nonlinear steps. However, this makes the resulting computation to be difficult to interpret and the algorithms that are currently used to train this networks are not exempt from issues. In this work, we are going to analyze one blind spot in neural networks, which affects their stability: *adversarial examples*.

1.1 PROBLEM DESCRIPTION

In consideration of its high performance, one would expect a state-of-the-art deep neural network to be robust to small perturbations in the input, that is, the model should produce the same results when the input is slightly modified. Nevertheless, it has been revealed [47] that adding a small but intentionally crafted perturbation to an input, it is possible to change the network's output.

From a security point of view, this issue suppose a major threat, since it makes these models vulnerable to adversarial attacks, even in physical world scenarios [23]. For example, let's suppose that a deep neural network is used in a scanner that scans suitcases for weapons at the airport. A knife can be developed to avoid detection by making the system think it is an umbrella. In other example, a self-driving car can crash into another car when it ignores a stop sign, because it was designed to be recognized as a yield sign for the software of the car (see Fig. 1.1).

In parallel to avoiding this kind of problems, the study of adversarial examples will lead to go further in enhancing the stability of machine learning models, getting new insight that can be useful in the future.

Although adversarial examples has been introduced for a wide variety of machine learning models like Support Vector Machine [3], in this work we are focusing only on image classifiers based on deep neural networks,

Figure 1.1: The image on the left [43] is an ordinary image of a stop sign, while the image on the right is designed to force a particular deep neural network to classify it as a yield sign [35]. To humans, both images appear to be the same.

as an extensive research has been done in this area. Thus, the adversarial examples that we are going to consider are images (*adversarial images*) with intentionally perturbed pixels, whose aim is to deceive the classifier to assign them incorrect labels.

1.2 WORK OUTLINE

The work is organized into four main parts: 1) Preliminary Concepts, 2) Adversarial attacks, 3) Experiments, 4) Conclusions.

- We start, in Part I, by introducing some preliminary concepts that will be useful for the understanding of the work. In particular, we will describe the mathematical tools and definitions to be used throughout the work, and provide a brief introduction to deep learning, focusing on its current situation.
- In Part II, the concept of adversarial example is formally defined, and the main algorithms and techniques used to generate adversarial examples are described in detail. We conclude this part by providing an overall comparison of these approaches.
- Part III contains the experimental part of the work, in which the methods described in the previous part are tested on different models and datasets, evaluating the results obtained.
- Lastly, Part IV summarizes the main contributions that have been made in attempting to explain why this vulnerability occurs, as well as the most recent techniques used to mitigate its effect.

1.3 BASIC BIBLIOGRAPHY

Numerous sources have been consulted in this work, as the problem of adversarial examples has aroused a great deal of interest recently, resulting in a large volume of available information. Among all of them,

we indicate below those that have been essential for the development of the work.

- *Intriguing properties of neural networks* [47], by C. Szegedy et al., explains the problem and provides a first approach to generate adversarial examples, based on the L-BFGS optimization algorithm.
- *Explaining and Harnessing Adversarial Examples* [12], by Ian J. Goodfellow et al., provides a widely used technique to obtain adversarial examples, known as the Fast Gradient Sign method.
- *DeepFool: a simple and accurate method to fool deep neural networks* [31], by S.M. Moosavi-Dezfooli et al., details a very effective method for this type of attacks.
- *SparseFool: a few pixels make a big difference* [30], by A. Modas et al., provides another useful method for generating adversarial examples.
- *One pixel attack for fooling deep neural networks* [45], by Jiawei Su et al., explains the One Pixel Attack, where only one pixel of the image is modified to obtain adversarial images.
- *Universal adversarial perturbations* [32], by S.M. Moosavi-Dezfooli et al., explains an useful method to generate universal perturbations, valid for several images in a dataset.

2

OBJECTIVES

The objectives proposed at the beginning of this work were the following:

- First of all, studying in depth the field of deep learning to fully understand the basic concepts that will be essential for the development of the work.
- Secondly, to understand the problem of adversarial examples, as well as to analyse and apply state-of-the-art algorithms and techniques.
- Finally, to be able to extract relevant conclusions from the experiments performed, even making a contribution in the case of revealing some interesting progress.

The first objective is covered in chapter 4, where a complete overview of the most important concepts in deep learning is provided. In addition, a review of the state of the art in convolutional neural networks is provided.

The second objective is the main one within the work, and also the one that has required more effort. It is covered in parts II and III. Concretely, chapter 5 defines the concept of adversarial examples, chapters 6, 7, 8 and 9 describe in detail the techniques considered in this work, and chapter 10 gives an overall comparison of them. These techniques are tested in chapter 12 under different conditions, in order to have enough information to make reliable conclusions. In addition, chapter 3 provides the mathematical concepts that are needed to fully understand the ideas behind these approaches (especially the geometric ones).

Finally, chapter 13 covers the third objective, including some explications and techniques that can be useful to address this issue. It has not been easy to make a substantial contribution (apart from the existing ones), but an interesting idea that could be worked on in the future is given.

To complete this work, the knowledge acquired from some subjects (from both math and computer science) has been essential. Among them, we can highlight Computer Vision, Machine Learning, Statistics, Geometry and Analysis.

We can therefore conclude that the work largely satisfies the objectives that were initially proposed.

Part I

PRELIMINARY CONCEPTS

3

MATHEMATICAL TOOLS

This chapter covers some mathematical concepts and results that provides a theoretical basis useful for understanding some parts of the work.

3.1 ANALYSIS

We start by defining two elementary functions: the arguments of the maximum (*arg max*) and the arguments of the minimum (*arg min*).

Definition 3.1. For a real-valued function f with domain S, the *arguments of the maximum (minimum)* are the set of elements in S that achieve the global maximum (minimum) in S:

$$\arg\max_{x \in S} f(x) = \{x \in S : f(x) = \max_{y \in S} f(y)\}$$

$$\arg\min_{x \in S} f(x) = \{x \in S : f(x) = \min_{y \in S} f(y)\}$$

Remark 3.1. When the maximum (minimum) of f is undefined, the arguments of the maximum (minimum) are the empty set.

3.1.1 *Differentiation in several variables*

Here we briefly review the basic concepts of differentiation in several variables that will be useful in future chapters.

Definition 3.2. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function of n variables and $p, v \in \mathbb{R}^n$ with $\|v\| = 1$. The *derivative* of f in p in the *direction* v is defined as

$$D_v f(p) = \lim_{t \to 0} \frac{f(p + tv) - f(p)}{t}$$

supposed, of course, that the limit exists.

When the vector e_k of the canonical basis of $\mathbb{R}^n$ is considered as the direction of the derivative, then it is called the *partial derivative* of f in p with respect to the k-th variable:

$$D_{e_k} f(p) = \lim_{t \to 0} \frac{f(p + te_k) - f(p)}{t}$$

$$= \lim_{t \to 0} \frac{f(p_1, \ldots, p_k + t, \ldots, p_n) - f(p_1, \ldots, p_k, \ldots, p_n)}{t}$$

$$= \lim_{x_k \to p_k} \frac{f(p_1, \ldots, x_k, \ldots, p_n) - f(p_1, \ldots, p_k, \ldots, p_n)}{x_k - a_k}$$

The partial derivative of f with respect to x_i is also denoted as $\frac{\partial f}{\partial x_k}$.

Remark 3.2. Partial derivatives can be interpreted as computing the ordinary one-dimensional derivative treating all variables except x_i as fixed constants, hence its name.

Definition 3.3. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function of n variables. The *gradient* of f in a point $p \in \mathbb{R}^n$ is the vector of $\mathbb{R}^n$ given by

$$\nabla f(p) = \left(\frac{\partial f}{\partial x_1}(p), \frac{\partial f}{\partial x_2}(p), \ldots, \frac{\partial f}{\partial x_n}(p) \right)$$

provided that such partial derivatives exist.

Definition 3.4. A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be *differentiable in* $p \in \mathbb{R}^n$ if

$$\lim_{x \to p} \frac{f(x) - f(p) - \langle \nabla f(p), x - p \rangle}{\|x - p\|} = 0 \tag{3.1}$$

When f is differentiable in each point of $\mathbb{R}^n$, then f is said only *differentiable*.

If we define $R(x, p) = \frac{f(x) - f(p) - \langle \nabla f(p), x - p \rangle}{\|x - p\|}$, by (3.1) we get that $\lim_{x \to p} R(x, p) = 0$, so the previous expression can be equivalently written as:

$$f(x) = f(p) + \langle \nabla f(p), x - p \rangle + R(x, p) \, \|x - p\| \tag{3.2}$$

Definition 3.5. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function differentiable in $p \in \mathbb{R}^n$. The hyperplane in $\mathbb{R}^{n+1}$ given by

$$x_{n+1} = f(p) + \langle \nabla f(p), x - p \rangle$$

is called the *tangent hyperplane* of f in p, which is tangent to the graph of f at the point $(p, f(p))$.

Proposition 3.1. *If $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable in $p \in \mathbb{R}^n$ and $v \in \mathbb{R}^n$ with $\|v\| = 1$, then it holds that:*

$$D_v f(p) = \langle \nabla f(p), v \rangle$$

Proof. By taking $x = p + tv$ in (3.2), we have that

$$\begin{aligned} f(p + tv) &= f(p) + \langle \nabla f(p), tv \rangle + R(p + tv, p) \, \|tv\| \\ &= f(p) + t \langle \nabla f(p), v \rangle + R(p + tv, p)|t| \end{aligned}$$

Hence,

$$\lim_{t \to 0} \frac{f(p + tv) - f(p)}{t} = \langle \nabla f(p), v \rangle + \lim_{t \to 0} R(p + tv, p) \frac{|t|}{t} = \langle \nabla f(p), v \rangle$$

□

Corollary 3.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function differentiable in p with $\nabla f(p) \neq 0$. Then,*

1. *The direction in which the directional derivative of f in p is maximum is $u = \frac{\nabla f(p)}{\|\nabla f(p)\|}$.*
2. *The direction in which the directional derivative of f in p is minimum is $v = -\frac{\nabla f(p)}{\|\nabla f(p)\|}$*

Proof. By the above proposition, and using the Cauchy-Schwarz inequality, for any direction w it holds that

$$|D_w f(p)| = |\langle \nabla f(p), w \rangle| \leq \|\nabla f(p)\| \, \|w\| = \|\nabla f(p)\|$$

The equality holds if and only if exists $\lambda \in \mathbb{R}$ such that $w = \lambda \nabla f(p)$, from which it follows (by taking norms) that $|\lambda| = \frac{1}{\|\nabla f(p)\|}$.

Hence, the directions for which $|D_w f(p)|$ is maximum are $u = \frac{\nabla f(p)}{\|\nabla f(p)\|}$ and $v = -\frac{\nabla f(p)}{\|\nabla f(p)\|}$.

Using the first one, $|D_u f(p)| = \langle \nabla f(p), \frac{\nabla f(p)}{\|\nabla f(p)\|} \rangle = \|\nabla f(p)\|$, which is the maximum value that the directional derivative can reach.

Similarly, $|D_v f(p)| = -\|\nabla f(p)\|$, which is the minimum value for the directional derivative.

□

Remark 3.3. The above result states that the gradient vector points to the direction in which the function increases the most, and the negative gradient points to the direction in which the function decreases the most.

Theorem 3.3. *If a function $f : \mathbb{R}^n \to \mathbb{R}$ has continuous partial derivatives in an open set $O \in \mathbb{R}^n$, then it is differentiable at each point $p \in O$.*

3.2 GEOMETRY

The geometrical tools that will be used throughout the work are related to the real vector space $\mathbb{R}^n$.

3.2.1 *Norms on* $\mathbb{R}^n$

Definition 3.6. For $1 \leq p < \infty$, the *p-norm* of a vector $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is defined as

$$\|x\|_p = \left(\sum_{k=1}^{n} |x_k|^p \right)^{1/p} \tag{3.3}$$

When $p = \infty$, the ∞-norm is called the *maximum norm*, given by

$$\|x\|_\infty = \max\{|x_k| : k = 1, \ldots, n\} \tag{3.4}$$

For $p = 2$, we get the *Euclidean norm.*

Proposition 3.4. *For $1 \leq p \leq \infty$, the norms given by (3.3) and (3.4) are equivalent.*

Proof. Using the usual basis $\{e_1, \ldots, e_n\}$ of $\mathbb{R}^n$, for $1 \leq p \leq \infty$ and $x \in \mathbb{R}^n$, the triangle inequality states that

$$\|x\|_p = \left\| \sum_{k=1}^{n} x_k e_k \right\|_p \leq \sum_{k=1}^{n} |x_k| \, \|e_k\|_p = \sum_{k=1}^{n} |x_k| = \|x\|_1$$

On the other side, it is clear that $\|x\|_\infty \leq \|x\|_p$ and $\|x\|_1 \leq n \, \|x\|_\infty$. Combining the above three inequalities, we obtain

$$\|x\|_\infty \leq \|x\|_p \leq \|x\|_1 \leq n \, \|x\|_\infty$$

□

Remark 3.4. The real vector space $\mathbb{R}^n$ with the norm $\|\cdot\|_p$ defined in (3.3) and (3.4) is a Banach space, usually denoted as ℓ_p^n. For this reason, from now on we will refer to the this norm as the ℓ_p^n *norm*, or only ℓ_p norm, for $1 \leq p \leq \infty$.

3.2.2 *Distance with sign to an affine hyperplane*

Let $P = \{x \in \mathbb{R}^n : \langle w, x - p \rangle = 0\}$ be an affine hyperplane in $\mathbb{R}^n$, determined by normal the vector $w \in \mathbb{R}^n$ and the point $p \in \mathbb{R}^n$. If $b = -\langle w, p \rangle$, then we can write $P = \{x \in \mathbb{R}^n : \langle w, x \rangle + b = 0\}$.

Proposition 3.5. *The distance (with sign) between $x_0 \in \mathbb{R}^n$ and P is given by:*

$$d(x_0, P) = \frac{\langle w, x_0 \rangle + b}{\|w\|}$$

Proof. Let $v = x_0 - p$. The distance between x_0 and P is simply the length of the projection of v onto the unit normal vector $n = \frac{w}{\|w\|}$, determined by the dot product between v and n:

$$d(x_0, P) = \langle v, n \rangle = \frac{\langle w, x_0 - p \rangle}{\|w\|} = \frac{\langle w, x_0 \rangle + b}{\|w\|}$$

□

Remark 3.5. The distance with sign can be used to separate $\mathbb{R}^n - P$ into its two connected components, given by:

$$P^+ = \{x \in \mathbb{R}^n : d(x, P) > 0\}$$

$$P^- = \{x \in \mathbb{R}^n : d(x, P) < 0\}$$

4

DEEP LEARNING

The purpose of this chapter is to provide a brief overview of what deep learning means, and to describe the main techniques that are used by this kind of algorithms.

Deep learning is a class of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification [6].

Let's take a step back and introduce some terms that will be useful to understand the previous definition.

4.1 MACHINE LEARNING

Machine learning is just a subset of Artificial Intelligence, dedicated to the study of algorithms that improve automatically through experience. According to T. Mitchell [29], an algorithm is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. This gives the computers the ability to learn without explicitly being programmed to, hence the name "Machine Learning".

In general, a machine learning problem consists of selecting a *hypothesis* g from a *hypothesis set* $\mathcal{H}$, using some specific *learning algorithm* $\mathcal{A}$, to approximate an unknown *target function* $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are known as the *input* and *output space*, respectively. The learning algorithm uses a set of data samples, the *training set* $\mathcal{D} = \{(x_i, y_i) : i = 1, \ldots, n\} \subset \mathcal{X} \times \mathcal{Y}$, to determine the best global hypothesis.

The above learning paradigm is called *supervised* learning, as the training set contains explicit examples of what the correct output should be for some given inputs. It is the most common type of learning.

Typical supervised learning problems are *classification*, where the output set $\mathcal{Y}$ is discrete, representing different labels or classes, and *regression*, where $\mathcal{Y}$ is continuous.

There are other learning paradigms, such as *unsupervised* learning, where the training set contains only the inputs, i.e., $\mathcal{D} \subset \mathcal{X}$. The objectives in this case are to find structures in the data (*clustering*), to discover dependencies (*patterns*) or to select relevant features in the data (*dimensionality reduction*), among others.

In this work, we will focus on classification problems.

Deep learning models are based on *neural networks*, a learning algorithm inspired by biological neural networks that conforms an animal brain. They consist of several connected units called *neurons*, that vaguely model neurons in a biological brain. This way, each neuron can transmit a signal to other neurons, like the synapses in a real brain. Beyond this, there are no more resemblances between neural networks and biological brains.

Hence, the fundamental building block of a neural network is a neuron, which is also called a *Perceptron*. Figure 4.1 shows how it works.

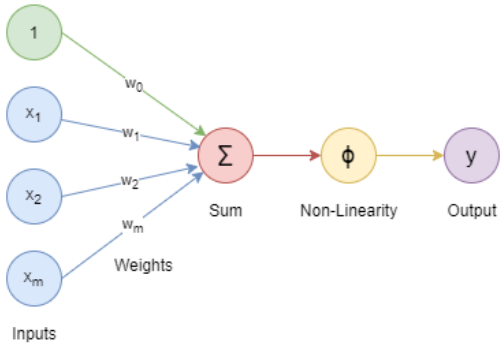

Figure 4.1: Structure of a neuron. The weight w_0 is also known as the *bias*.

Given a set of inputs $x_1, \ldots, x_m \in \mathcal{X}$, the forward propagation of information through a neuron consist in a weighted sum (a linear combination) of the inputs, followed by a non-linear activation function ϕ, so that the output is given by

$$y = \phi\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

There are many common types of non-linear activation functions, such as the sigmoid function, the hyperbolic tangent and the *Rectified Linear Unit* (ReLU). Figure 4.2 shows its properties.

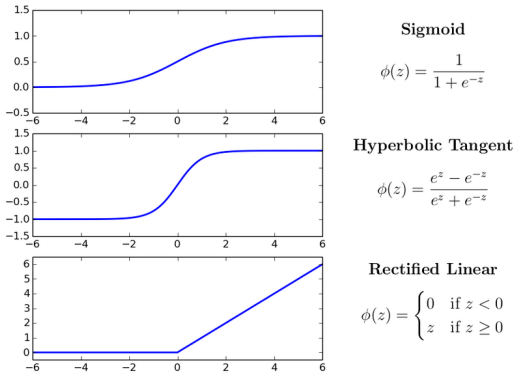

Figure 4.2: Common activation functions [18].

Combining several neurons organized in layers we obtain a neural network, as depicted in figure 4.3. The layers where all inputs are fully connected to all outputs are called *dense* layers.

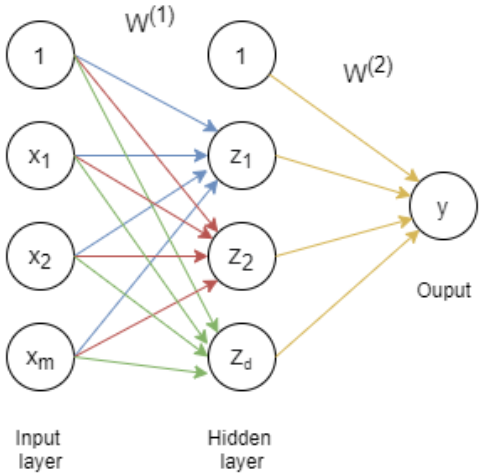

Figure 4.3: Single layer neural network. The weight matrices $W^{(k)} = w_{i,j}^{(k)}$ contains the weights associated with each arrow connecting the layers $(k-1)$ and k.

The output of the above example is given by

$$y = \phi\left(w_0^{(2)} + \sum_{j=1}^{d} z_j w_j^{(2)}\right), \text{ where } z_i = \phi\left(w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}\right)$$

The use of non-linear activation functions is crucial. Otherwise, the network will produce linear decisions, as the combination of linear functions gives a linear function. These non-linearities allow neural networks to approximate arbitrarily complex functions, and this is what makes them extremely powerful.

Depending on the activation function, a neural network can be used for different tasks. Rectified linear units generate strictly positive outputs, while sigmoid or hyperbolic tangent functions can be used for bounded outputs. These functions, combined with the *output function* in the last layer, provide a wide range of alternatives. Using the sign function in the output layer is useful for binary classification, while the identity function can be used for regression tasks. For probability estimation, it is likely to use a sigmoid function. Another useful output function is *SoftMax*, specially for multi-class classification problems. It takes a vector of real numbers and transforms it into a vector of numbers in range $(0, 1)$, which add up to 1, so it can be interpreted as the probabilities of belonging to each class of the problem. Namely,

$$Softmax(z)_i = \frac{\exp z_i}{\sum_j \exp z_j}$$

The neural network architecture is specified once the number of layers and the number of neurons in each layer is determined. Each architecture provides a different hypothesis set, containing the candidate functions where the approximated solution will be chosen from. A greater number of layers (and therefore of neurons) results in more complex models, which are called *Deep Neural Networks*.

4.2.1 *Optimizing neural networks*

The weights of the network are generally initialized with small random values, so they need to be tuned in order to achieve good results. We need a way to make the network *learn* (a way to adjust these weights using the information on the training data).

First of all, it is necessary to have a form of quantifying how good or bad the results provided by the network are. Here is where the *loss function* comes in, measuring the cost incurred from incorrect predictions of the network.

Given a dataset $\mathcal{D} = \{(x_i, y_i) : i = 1, \dots, n\}$, let f be the function implemented by a neural network with weights $W = \{W^{(0)}, W^{(1)}, \dots\}$. The loss of the model in a point $(x_i, y_i) \in \mathcal{D}$ is denoted as $\mathcal{L}(f(x_i; W), y_i)$.

The *cost function* (also known as the *objective function* or the *empirical risk*) measures the average loss over the entire dataset:

$$\mathcal{J}(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x_i; W), y_i)$$

The total loss over the dataset is known as the *empirical loss*.

In classification problems, the most common loss function is called the *softmax cross-entropy loss*. If the network uses a SoftMax function in the output layer, then the output is a vector denoting k probabilities, where k is the number of classes in the problem, and the loss is given by

$$\mathcal{L}(f(x_i; W), y_i) = -\sum_{j=1}^{k} [[y_i = j]] \ln \left(f(x_i; W)_k \right)$$

where $[[\cdot]]$ evaluates to 1 if the argument is true, and to 0 if it is false.

This function comes from information theory, and indicates the distance between what the model believes the output distribution should be and what the original distribution really is.

Now that we have a measure of the error incurred by the network, the empirical risk can be minimized by setting the weights W of the model properly:

$$W^* = \arg\min_{W} \mathcal{J}(W) \tag{4.1}$$

Solving the previous optimization problem is a really complicated task. In practice, the *Stochastic Gradient Descent* algorithm (SGD) is used, which is summarized below.

Algorithm 1: Stochastic Gradient Descent.

Input : Dataset $\mathcal{D}$, loss function $\mathcal{L}$, learning rate η
Output : Weights W

Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
while *not termination criteria* **do**
    Shuffle the samples in $\mathcal{D}$
    for *each sample* (x_i, y_i) *in* $\mathcal{D}$ **do**
        Compute gradient, $\frac{\partial \mathcal{L}(f(x_i;W),y_i)}{\partial W}$
        Update weights, $W \leftarrow W - \eta \frac{\partial \mathcal{L}(f(x_i;W),y_i)}{\partial W}$
    end
end
return W

The basic idea behind this algorithm is to use the (negative) gradient of the loss function to find the direction in the search space in which the loss function decreases the most, and moving a step in that direction. This process is repeated until it converges to a local minimum of the cost function. Figure 4.4 shows an example of applying SGD in a model with only two weights.

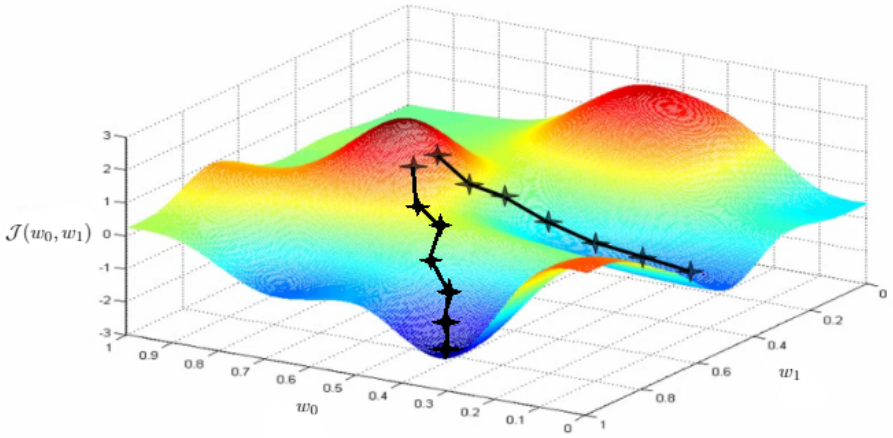

Figure 4.4: SGD on a two-dimensional weight problem. Different initial points may lead to different solutions, hence the importance of a proper initialization.

When the number of weights in the model is large, computing the gradient is not easy at all, and we will need another algorithm called *backpropagation* to make this computation efficiently. Basically, backpropagation uses elementary calculus to determine the partial derivatives of $\mathcal{J}(W)$ with respect to each weight w. The following example illustrates this process.

Let's suppose we have a very simple neural network, as shown in figure 4.5, and we want to obtain the gradient $\frac{\partial \mathcal{J}(W)}{\partial W}$.

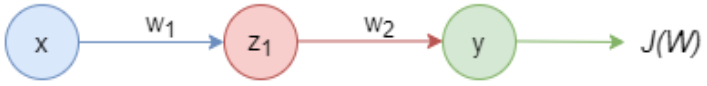

Figure 4.5: Simple neural network architecture with only two weights.

The chain rule can be used to obtain $\frac{\partial \mathcal{J}(W)}{\partial w_2}$ as

$$\frac{\partial \mathcal{J}(W)}{\partial w_2} = \frac{\partial \mathcal{J}(W)}{\partial y} \cdot \frac{\partial y}{\partial w_2}$$

By applying the chain rule in w_1, we are left with

$$\frac{\partial \mathcal{J}(W)}{\partial w_1} = \frac{\partial \mathcal{J}(W)}{\partial y} \cdot \frac{\partial y}{\partial w_1}$$

In this case, the partial derivative of y with respect to w_1 is not directly computable, so we have to apply the chain rule again to get

$$\frac{\partial \mathcal{J}(W)}{\partial w_1} = \frac{\partial \mathcal{J}(W)}{\partial y} \cdot \frac{\partial y}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

This way, the gradient of the cost function is obtained through simpler derivatives, which can be directly calculated, by using the chain rule repeatedly. This allows the error signal to propagate from the output to the input (hence the name backpropagation). A complete description of this algorithm can be found in [1].

In practice, optimization of neural networks is incredibly tough since the cost function defines really complex surfaces with many local minima, where the optimization algorithm could get stuck into.

Going back to the SGD algorithm (1), when the weights are updated in line 6, the learning rate η determines how much of a step is taken in the descent direction. Setting this parameter properly can have a huge impact on obtaining the best possible optimum in a reasonable time: while small learning rates increases the number of steps required to reach the minimum and may lead the algorithm to get stuck in local minima, using too large values may cause drastic updates which lead to divergent behaviors. For this reason, the learning rate is no longer fixed. It can be made larger or smaller depending on some aspects (how large the gradient is, how fast learning is happening, the size of particular weights...). Different approaches on adapting the learning rate lead to different optimization algorithms, with some of the most famous being Adam, Adadelta, Adagrad and RMSProp. An overview of gradient descent optimization algorithms can be found in [38].

4.2.2 *Deep Neural Networks*

A potential negative effect that we should be careful of when using a machine learning model is *overfitting*. Overfitting occurs when the model is too complex, and it starts to "memorize" certain aspects of the training data, which causes the model not to generalize well over unseen data. With this in mind, and taking into account that a neural network with a single layer is sufficient to represent any function (see the Universal Approximation Property [21]), one may be wondering why it is necessary to use deep neural networks.

There are some empirical findings that encourages us to use this type of networks. One of them is that using deeper models can reduce the number of units required to represent the desired function and thus reduce the generalization error. There are families of functions that can be approximated efficiently by a deep architecture, but require a much larger number number of units when the depth is restricted.

But the main advantege of deeper models is that they seem to result in a better generalization for a wide variety of tasks, including many of today's problems in Artificial Intelligence (e.g. in computer vision, natural language processing, etc.) [2].

There is a class of deep neural networks that is extremely effective for some specific tasks, especially those related with images: Convolutional Neural Networks (*CNN*).

4.3 CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNNs) are quite similar to the ordinary Neural Networks that have just been introduced, except that they make the explicit assumption that the inputs have local dependencies (e.g. in images and sounds), allowing us to encode certain properties into the architecture. We are going to focus on the case where the inputs are images.

The input images consist in several pixels arranged in a 3-dimensional way (wide, high and color channels). Therefore, the layers of a CNN have neurons arranged in 3 dimensions (instead of 1), and they are only connected to a small region of the input.

As can be seen in figure 4.6, layers in a CNN transform 3-dimensional input volumes to 3-dimensional output volumes. Depending on the function used to transform the inputs, there are different types of layers, which we are going to describe next.

Generally, three main types of layers are used to build CNN architectures: *Convolutional* Layers, *Pooling* Layers, and *Fully-Connected* or *Dense* Layers (exactly as seen in regular neural networks). The most

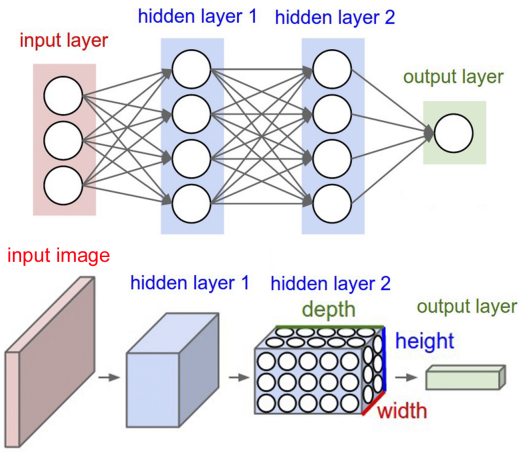

Figure 4.6: A regular neural network (top) compared with a CNN (bottom). The CNN arranges it neurons in 3 dimensions (width, height and depth).

common activation function in CNNs is ReLU, and it is also considered as a layer, which applies elementwise non-linearity to its inputs.

- **Convolutional Layer**: this type of layers are the core building block of a CNN. It uses the *convolution* operation to transform the input, hence its name. Each neuron in this layer is connected only to a local region in the input volume spatially (along width and height), but to the full depth, and represents a learnable filter. During the forward pass, each filter is convolved across the width and height of the input volume, computing dot products between the entries of the filter (the weights of the neuron) and the input at any position. This way, as we slide the filter over the width and height of the input volume, a 2-dimensional activation map that gives the responses of the filter at every spatial position is generated, as shown in figure 4.7.

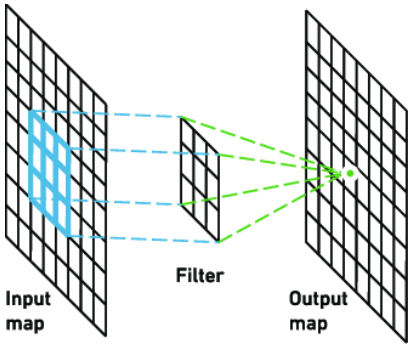

Figure 4.7: Representation of the output map generated after convolving a filter on the input map. The depth of the filter is always the same as the input volume (1 in this case).

Several filters can be used in each convolutional layer, producing different 2-dimensional activation maps, that are stacked along

the depth dimension to produce the output volume. Once the model is trained, each filter learns to detect some type of visual feature in the input image.

The local connectivity of convolutional layers exploits the local dependencies in the images.

- **Pooling layer:** the objective of a pooling layer is to reduce the spatial size of the representation to decrease the number of parameters of the network, and hence to control overfitting. It downsamples the volume spatially, independently in each depth slice of the input volume, using some function like the maximum or the average of the values.

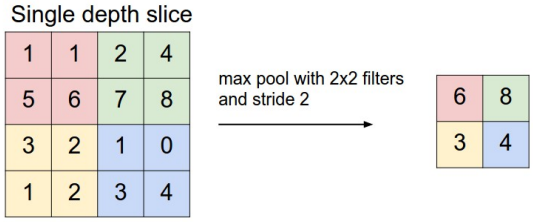

Figure 4.8: Example of max-pooling using 2x2 filters and a stride of 2. That is, each max is taken over 4 numbers, represented in the figure with the little 2x2 squares.

It is common to insert a Pooling layer in-between successive Convolutional layers. However, its use is an open discussion [42].

Some types of layers have *trainable* parameters (e.g. the weights of the neurons in Convolutional and Dense layers), which are tuned using the backpropagation algorithm, as in regular neural networks. On the other hand, ReLU and Pooling layers don't have any trainable parameter, so they apply always the same function to the input volume. Moreover, each layer may or may not have additional hyperparameters, like the filter size in Convolutional and Pooling Layers.

The basic architecture of a CNN consists in a few Convolutional-ReLU layers, followed by pooling layers. This pattern is repeated until the image has been merged spatially to a small size. Then, it is common to use some fully-connected layers. The last of them holds the output, such as the class scores if a SoftMax function is used (see Fig. 4.9).

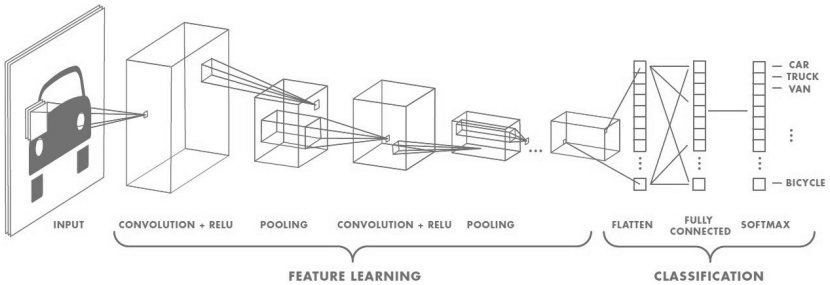

Figure 4.9: Basic CNN used to classify an image.

4.3.1 *The evolution of Convolutional Neural Networks*

To finish this section, let's review the main architectures that have emerged in recent years.

- **LeNet.** LeNet [24] was developed by Yann LeCun in 1990's, and it was the first successful application of CNNs in real problems. Concretely, LeNet was used for handwritten digit recognition. Although its architecture is quite simple (see Fig. 4.10), LeNet was able to outperform all other existing methods with ease.

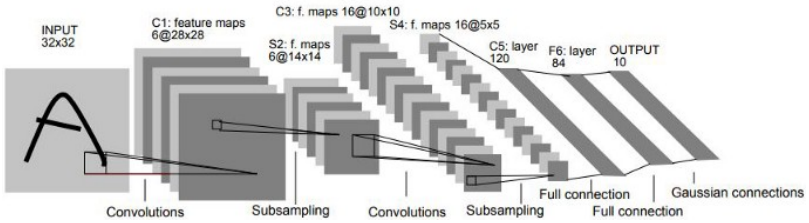

Figure 4.10: Architecture of LeNet-5. Each plane is a feature map [24].

- **AlexNet.** AlexNet, designed by A. Krizhevsky et al. in 2012 [22], was the architecture that popularized CNNs in the field of Computer Vision. It has a very similar architecture to LeNet, but with more layers (deeper architecture) and using significantly more number of filters, with different sizes (see Fig. 4.11). It also started using ReLU activations (instead of sigmoid or tahn, used in LeNet), which helped to train better models. AlexNet won the Imagenet classification challenge in 2012, significantly outperforming the second runner-up.

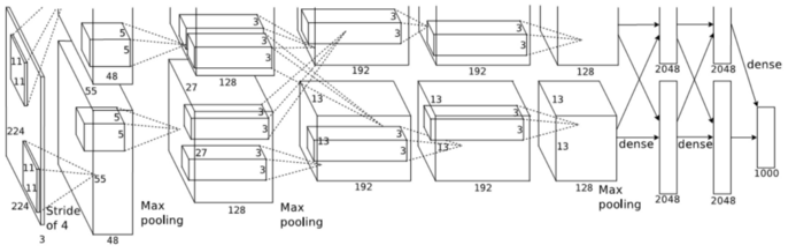

Figure 4.11: Architecture of AlexNet. It was designed to use two GPUs, so that one of them runs the layer-parts at the top while the other runs the layer-parts at the bottom, communicating at certain layers [22].

- **InceptionNet.** The Inception architectures (there are currently 4 versions), developed by Google, were a major breakthrough in the CNNs history. The first of this architectures, InceptionV1 (also known as GoogLeNet) came out in 2014 [46], and won the ILSVRC competition in the same year. Its main contribution was the development of the *Inception Module*, which allowed a

drastic reduction in the number of parameters. This module uses parallel paths with different filter sizes and operations, which are used to capture sparse patterns of correlations in the stack of feature maps. 1x1 convolutions are used for dimensionality reduction before expensive convolutions, as depicted in figure 4.12. Additionally, average pooling layers were used instead of Dense layers, further reducing the number of parameters (it has 4M, compared to the 60M of AlexNet).

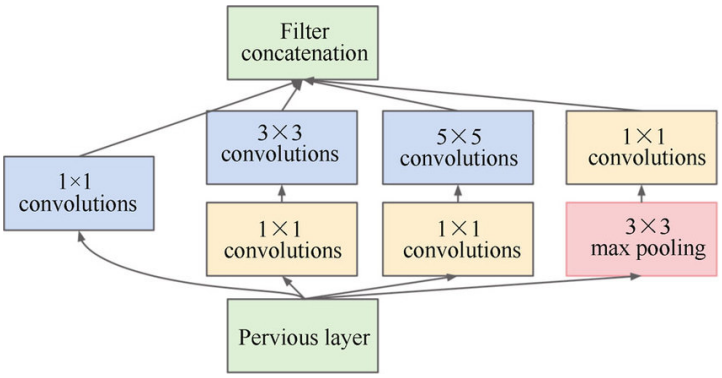

Figure 4.12: Representation of an Inception Module [46].

InceptionV2 and InceptionV3 architectures added certain improvements to the first version. They improve the training by using *batch normalization* layers to standardize the inputs, and introduced more variants of the Inception Module with aggressive factorization of filters.

To deal with the issue of training deeper models, these architectures employed auxiliary classifiers in between the model. Figure 4.13 summarizes the InceptionV3 architecture, where its depth and complexity can be appreciated.

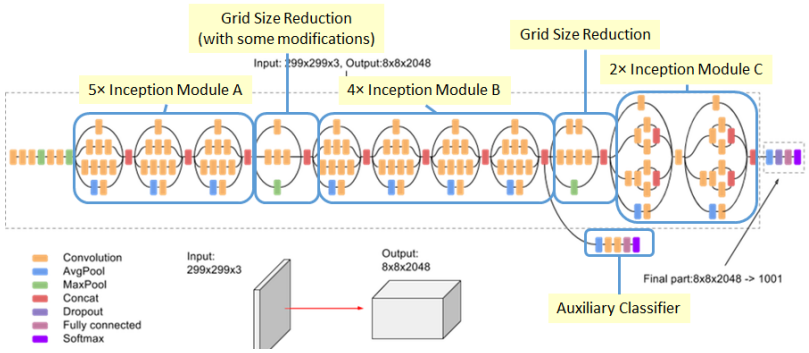

Figure 4.13: The InceptionV3 Architecture (Batch Normalization and ReLU are used after Convolutions) [37].

- **VGGNet.** VGG [41] was the runner-up in the ILSVRC 2014 competition. Its main contribution was in showing that the depth of a CNN is critical for good performance (it has 16 CONV/FC

layers). VGG introduced the idea of using small filter sizes (3x3 and 1x1), since multiple 3x3 convolutions are capable of replicatting bigger filter sizes, adding more nonlinearities between them. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot of memory and parameters (140M).

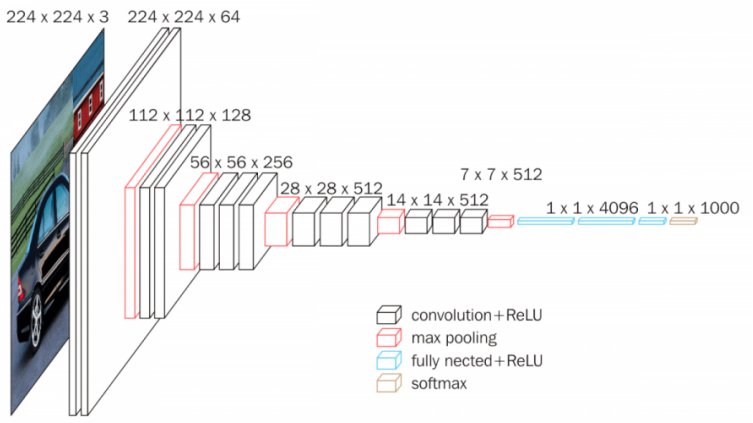

Figure 4.14: The VGGNet Architecture [41].

- **ResNet.** After the success of AlexNet in 2012, attempts to build deeper and deeper networks took off. However, the results were not as expected, since deeper neural networks are more difficult to train. The appearance of ResNet (Residual Networks) architectures in 2015 [15] brought an end to this problem, making it possible to train very deep models, with up to hundred or even thousands of layers, leading to a significant improvement in some computer vision tasks (an ensemble of deep residual networks achieved 1st place in the ILSVRC 2015 classification competition with a 3.57% error rate on ImageNet).

  The key of its success is the introduction of a new type of layer: the Residual Block, depicted in Fig. 4.15.

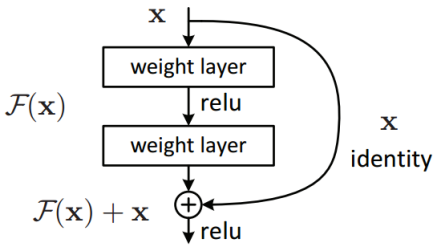

Figure 4.15: Residual learning: a building block.

The Residual Block introduces skip or shortcuts connections that make it easy for network layers to represent the identity mapping, allowing for the input x and the output $\mathcal{F}(x)$ to be combined as input to the next layer as the so-called residual mapping $\mathcal{H}(x) =$

$\mathcal{F}(x) + x$. Another benefit of these residual blocks is that they do not introduce any additional parameters to the model, so the computational time stays close.

Within the ResNet models, there are different architectures depending on the number of layers, which can be consulted in [15].

- **DenseNet.** With the idea of further exploiting the effects of shortcuts connections introduced in Residual Networks, Huang et al. proposed in 2016 [17] a novel type of architectures called Dense Convolutional Networks (DenseNet). In these architecture, all the layers all connected directly with each subsequent layer, so the input of each layer consists of the features maps of all earlier layers. To prevent the network from growing too wide and to improve the parameter efficiency, the number of input feature-maps is limited using a parameter called the *growth rate* (k). Fig. 4.16 depicts this idea.

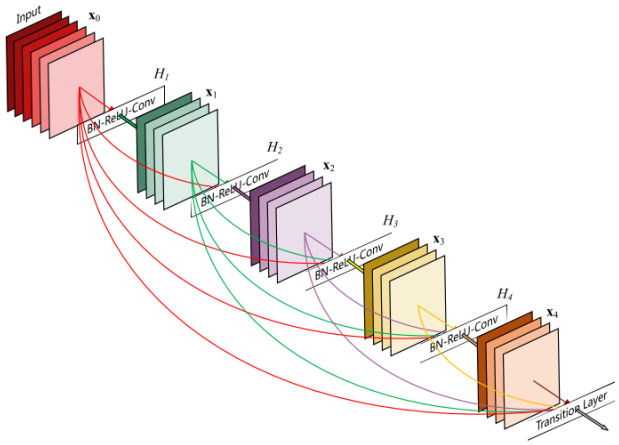

Figure 4.16: A 5-layer dense block with a grow rate of $k = 4$ Each layer takes all preceding features-maps as inputs.

Other than alleviating the vanishing-gradient problem, these architectures encourage feature reuse, making the network highly parameter-efficient. In ResNet architectures, the output of the identity mapping was added to the next block, which could impede information flow if the feature maps of two layers have very different distributions. Concatenation of feature maps can therefore preserve all of them and increase the variance of the outputs, encouraging features reuse.

The main DenseNet architectures can be found in [17].

- **MobileNet.** The MobileNet architectures were designed to run very efficiently on mobile devices.

  The first version, MobileNetV1, was introduced in 2017 [16]. The big idea behind this architecture is that convolutional layers, which are very expensive to compute, can be replaced by so-called depthwise separable convolutions. In this way, the convolution

layer is split into a first depthwise convolution layer, that filters the input, and a 1x1 convolution layer, that combines these filtered values to create new features (see Fig. 4.17). Together, they act as a traditional convolution layer, but much faster.

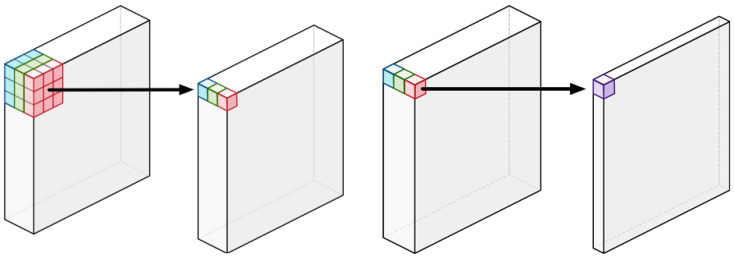

Figure 4.17: Illustration of a depthwise separable convolution block.

MobileNetV2, introduced in 2019 [39], still uses depthwise separable convolution blocks, but adds a first expansion layer, which expands the number of channels in the data before it goes into the depthwise convolution. Then, the pointwise convolution projects data into a tensor with a much lower number of dimensions, so in this version this layer is called a bottleneck layer. MobileNetV2 also adds a residual connection, inspired in ResNet architecture, and hence the full block is known as a bottleneck residual block. Figure 4.18 depicts one of this blocks.

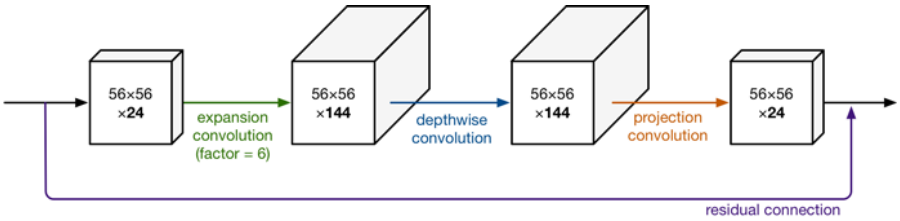

Figure 4.18: A bottleneck residual block from MobileNetV2.

The full MobileNet V2 architecture, consists basically in some of these building blocks in a row, as can be seen in [39].

Today, CNNs are tested on their image classification performance as well as transfer learning abilities across other problems, like object detection and segmentation. Some of the previous models are directly used for various deep learning tasks and so even though the image classification problem is almost solved, development in this field is going to still hold a lot of importance in the future.

Part II

ADVERSARIAL ATTACKS

5

DEFINITIONS

This chapter defines the concepts that will be covered in the following sections, related to adversarial attacks.

Let's denote by $f : \mathbb{R}^m \to [0,1]^c$ a classifier mapping image pixel value vectors to vectors denoting the probability of the image of belonging to each of the c classes considered in a certain problem. The classification is done by the following mapping, which picks the label that best fits the input (the one with the maximum probability):

$$\begin{aligned} \hat{k}_f : \mathbb{R}^m &\longrightarrow \{1, \dots, c\} \\ x &\longmapsto \hat{k}_f(x) = \arg\max_{i=1\dots c} f_i(x), \end{aligned}$$

where $f_i(x)$ is the output of $f(x)$ that corresponds to the i^{th} class. From now on, we refer to a classifier either by f or its corresponding discrete mapping $\hat{k}_f$, and we will write only $\hat{k}$ when the classifier is clear from the context.

We also denote

$$loss_f : \mathbb{R}^m \times \{1, \dots, c\} \to \mathbb{R}^+$$

the associated continuous loss function of the classifier f, which indicates the loss associated to a particular image and label.

5.1 ADVERSARIAL EXAMPLES

There exist many possible manipulations of the data to obtain adversarial examples, but we focus on the simplest one, given by additive perturbations, which consist in adding a specific vector to the original image. This approach allows to make geometric interpretations of the images generated, as we will see later.

Definition 5.1. Given an image $x \in \mathbb{R}^m$ and a classifier $f : \mathbb{R}^m \to [0,1]^c$, the ℓ_p-norm *adversarial perturbation* is defined as the minimal perturbation r that is sufficient to change the estimated label of the image x:

$$\arg\min_r \|r\|_p \quad \text{s.t.} \quad \hat{k}(x+r) \neq \hat{k}(x) \tag{5.1}$$

Hence, to get an adversarial perturbation, the optimization problem (5.1) has to be solved, but this is not an easy task. The aim of the next chapters is to present several methods that can be used to get an approach to this problem, including some interesting variations.

Definition 5.2. Given an adversarial perturbation r, the associated *adversarial example* is defined as the vector $x + r$.

Remark 5.1. The minimizer r of the problem (5.1) might not be unique, but we denote one such $x + r$ for an arbitrarily chosen minimizer.

5.2 ADVERSARIAL ROBUSTNESS

The following definitions provides a way to measure the strength of a classifier when trying to be fooled with these modified inputs.

Definition 5.3. Given an image $x \in \mathbb{R}^m$ and a classifier $f : \mathbb{R}^m \to [0,1]^c$, the *robustness* of f at the point x is defined as

$$\Delta(x; f) := \min_r \|r\|_p \quad \text{subject to} \quad \hat{k}(x + r) \neq \hat{k}(x),$$

Remark 5.2. $\Delta(x; f)$ can be interpreted as the minimum distance from x to any ℓ_p-norm adversarial example when using the classifier f.

The following definition gives a more general vision of the overall robustness of a classifier to adversarial examples.

Definition 5.4. The *robustness* of a classifier f is defined as

$$\rho_{adv}(f) := \mathbb{E}_x \frac{\Delta(x; f)}{\|x\|_p},$$

where $\mathbb{E}_x$ is the expectation over the input distribution of the data.

It can be seen as a "expected" distance, relative to the input image, to an ℓ_p-norm adversarial example when using the classifier f.

These tools will help us to compare the robustness of certain models to a specific adversarial method in the next part of the work.

6

BASIC APPROACHES

As discussed in the previous chapter, solving the optimization problem (5.1) is not easy, so we will need some efficient methods in order to obtain adversarial examples for a state-of-the-art deep neural network image classifier f. In this chapter, two basic approaches to this problem are exposed.

6.1 L-BFGS METHOD

The following algorithm is essentially based on the use of a Quasi-Newton optimization method to generate adversarial examples.

Given a classifier f, an image $x \in [0,1]^m$ (assuming that the pixels intensities are scaled in the range $[0,1]$) and a target label $l \in \{1,\dots,c\}$, the aim is to find an adversarial perturbation by solving the following box-constrained optimization problem:

$$\arg\min_r \|r\|_2 \text{ s.t. } \begin{cases} \hat{k}(x+r) = l \\ x + r \in [0,1]^m \end{cases} \tag{6.1}$$

This optimization problem is similar to (5.1), using the ℓ_2 norm, but there are two important differences:

- First, the objective here is not only to change the estimated label of an image but also to indicate which label we want the classifier to output. We will refer to this kind of attacks as *targeted attacks*.
- Secondly, the last condition ensures that the adversarial image is valid, in the sense that the image pixel values remain in the $[0,1]$ range after adding the perturbation.

In general, solving (6.1) can be really difficult. The authors in [47] suggest seeking an approximation by performing line-search to find the minimum $c > 0$ for which the minimizer r of the following problem satisfies $\hat{k}(x+r) = l$:

$$\text{Minimize } c\,\|r\|_2 + loss_f(x+r,l) \text{ s.t. } x + r \in [0,1]^m \tag{6.2}$$

Here, the parameter c is used to balance the importance given to the distance between the images ($\|r\|_2$) with respect to the distance between predictions, provided by the loss function. In this way, the term $c\,\|r\|_2$ acts like a *regularization term*, which penalizes the distance between the original and adversarial images, to get images that are as similar as possible.

To solve (6.2), the authors in [47] use a box-constrained L-BFGS[1].

This method will give us the exact solution if the loss function is convex, but this not happen in general, so we end up with an approximation. Unfortunately, this optimization method is time-consuming and therefore does not scale to high dimensional images or complex models.

Below are some adversarial images generated with this method.

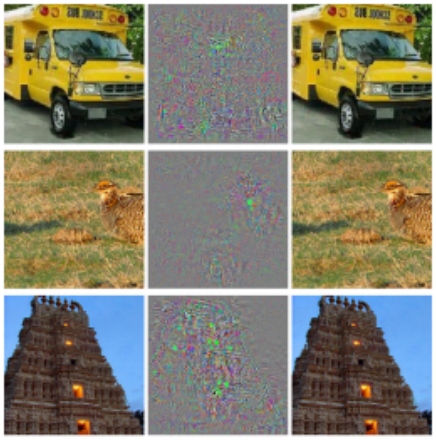

Figure 6.1: Adversarial images generated for AlexNet. In the left are the correctly predicted samples, in the center the adversarial perturbations and in the right the adversarial examples. Images in the right columns are predicted to be an *ostrich, Struthio* and *camelus* [47].

6.2 GRADIENT-BASED METHOD

Another basic, yet important method is the *Fast Gradient Sign Method* (FGSM), which is introduced in [12]. It is based on a hypothesis in which the linear behaviour in high dimensional spaces is the main reason that a broad class of classifiers are sensitive to small perturbations. Let's see this with more detail:

Given a linear model, consider the dot product between a weight vector $w \in \mathbb{R}^m$ and an adversarial example $x' = x + r$ of a given image $x \in \mathbb{R}^m$.

$$w^\top x' = w^\top x + w^\top r \tag{6.3}$$

Let's suppose that $\|r\|_\infty < \epsilon$ (i.e., $\max_{i=1\ldots n} |r_i| < \epsilon$) and ϵ is small enough to be discarded by the sensor or data storage apparatus associated with the problem (for example, $\epsilon < 1/255$ in images with 8 bits per pixel). Expression (6.3) indicates that the activation grows by $w^\top r$.

1 L-BFGS is an optimization algorithm in the family of quasi-Newton methods that finds a (local) minimum of an objective function by using objective function values and its gradient. The algorithm computes an approximation of the inverse Hessian matrix in each iteration to steer its search through variable space, using a limited amount of memory [4].

In order to maximize this increase preserving the max norm constraint on r, one can establish $r = \epsilon \cdot \text{sign}(w)$. If w has m dimensions and the average magnitude of an element of the weight vector is k, then the activation will grow by $\epsilon \cdot m \cdot k$. So, for high dimensional problems (where m is big), we can make many small changes to the input that add up to one large change to the output.

As many of today's most widely used neural networks networks are intentionally designed to behave in very linear ways, so that they are easier to optimize, the authors in [12] suggest a simple, analytical way to generate adversarial examples:

Given a classifier f, let $x \in \mathbb{R}^m$ be an input and $y = \hat{k}(x)$ the associated label. Following the previous idea, the loss function can be linearized so as to obtain a max-norm constrained (depending on $\epsilon > 0$) adversarial perturbation given by

$$r = \epsilon \cdot sign(\nabla_x \, loss_f(x, y)) \tag{6.4}$$

Remark 6.1. Here, the gradient of the loss function with respect to the input image plays the role of w in the previous discussion, and it can be efficiently computed using backpropagation. It indicates the direction in which the loss function increases its value faster. The sign of the gradient is positive if an increase in pixel intensity increases the loss and negative if a decrease in pixel intensity increases the loss.

It can be seen that FGSM does not fit the usual expression (5.1) for generating adversarial examples. In fact, it does not involve solving any optimization problem, but adding a small vector whose elements are proportional to the sign of the gradient of the cost function with respect to the input. This is actually interesting, because FGSM provides an efficient way to obtain adversarial examples.

Despite its efficiency, this method provides only a coarse approximation of the optimal perturbation vectors. In fact, it performs a unique gradient step, which often leads to sub-optimals solutions, so it is not a reliable tool in studying the worst-case robustness properties of deep classifiers.

Fig. 6.2 shows an adversarial example generated with this method.

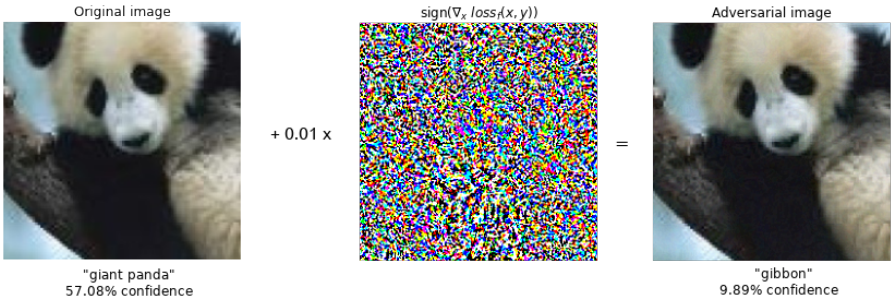

Figure 6.2: A demonstration of FGSM applied to GoogLeNet.

7

GEOMETRIC APPROACHES

The following methods seek to obtain adversarial examples by using the geometric properties of the decision boundaries in deep neural networks. The explanations here are going to be more extensive, because the geometric aspects has to be detailed to understand the procedures.

7.1 DEEPFOOL

DeepFool [31] is a simple yet accurate method to find adversarial examples. The basic idea behind it is to obtain adversarial perturbations by iteratively linearizing the classifier around the data point and finding the orthogonal projection onto a class-separating affine hyperplane.

Let's see this with more detail, using first a binary classifier to get an idea of how it works: let $f : \mathbb{R}^m \to \mathbb{R}$ be an arbitrary image classification function and let's assume that $\hat{k}(x) = \text{sign}(f(x))$. $\mathcal{F} \triangleq \{x : f(x) = 0\}$ denotes the level set at zero of f.

We begin by analyzing the case where f is an affine classifier, that is, $f(x) = w^\top x + b$, where $w \in \mathbb{R}^m$ and $b \in \mathbb{R}$. Here, the ℓ_2-norm adversarial perturbation of a point $x_0 \in \mathbb{R}^m$ corresponds to its orthogonal projection onto the hyperplane $\mathcal{F} = \{x : w^\top x + b = 0\}$ and the robustness of f at point x_0, $\Delta(x_0; f)$, is equal to the distance between x_0 and $\mathcal{F}$ (see Fig. 7.1).

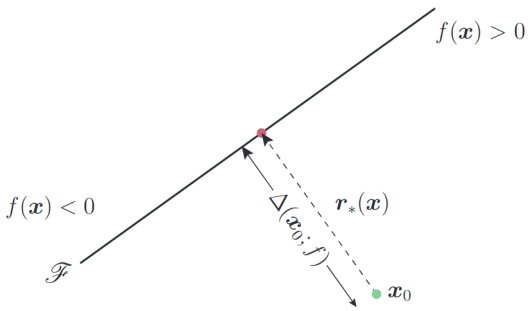

Figure 7.1: Orthogonal projection of x_0 onto the hyperplane $\mathcal{F}$ [31].

By prop. 3.5, the distance (with sign) from the point x_0 to the hyperplane $\mathcal{F}$ is given by

$$\Delta(x_0; f) = \frac{w^\top x_0 + b}{\|w\|_2} = \frac{f(x_0)}{\|w\|_2}$$

Therefore, the adversarial example is obtained by moving the data point $\Delta(x_0; f)$ units in the sense given by the unitary vector $\frac{-w}{\|w\|_2}$, so the adversarial perturbation will be

$$r_*(x_0) := \frac{\Delta(x_0; f)}{\|w\|_2}(-w) = -\frac{f(x_0)}{\|w\|_2^2} w \tag{7.1}$$

Assuming now that f is a general binary differentiable classifier, one can estimate $\Delta(x_0; f)$ by using an iterative procedure. Concretely, at each iteration i, f is linearized around the current data point x_i by the function $x \mapsto f(x_i) + \nabla f(x_i)^\top (x - x_i)$, which is tangent to the classifier function (see definition 3.5). The class-separating hyperplane is then given by $\mathcal{F} = \{x : f(x_i) + \nabla f(x_i)^\top (x - x_i) = 0\}$, so the perturbation r_i can be computed as

$$\arg\min_{r_i} \|r_i\|_2 \text{ s.t. } f(x_i) + \nabla f(x_i)^\top r_i = 0$$

using the solution given in (7.1), with $w = \nabla f(x_i)$. The datapoint is then updated as $x_{i+1} = x_i + r_i$. This process is repeated until the datapoint changes the sign of the classifier, as illustrated below. The complete algorithm is shown in Algorithm 2.

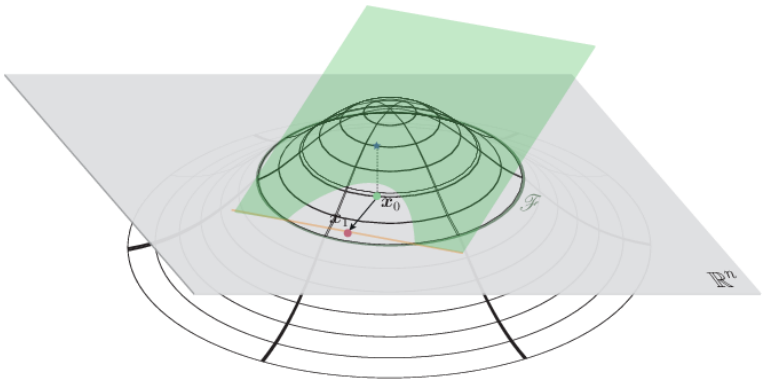

Figure 7.2: Illustration of Algorithm 2 for $x_0 \in \mathbb{R}^n$. The green plane is the graph of the function $x \mapsto f(x_i) + \nabla f(x_i)^\top (x - x_i)$ and the orange line is its zero set level (the hyperplane $\mathcal{F}$). x_1 is obtained from x_0 by projecting it on the orange hyperplane [31].

Algorithm 2: DeepFool for binary classifiers.

Input : Image x, classifier f
Output : Perturbation r

Initialize $x_0 \leftarrow x$, $i \leftarrow 0$
while $sign(f(x_i)) = sign(f(x_0))$ **do**
  $r_i \leftarrow -\frac{f(x_i)}{\|\nabla f(x_i)\|_2^2} \nabla f(x_i)$
  $x_{i+1} \leftarrow x_i + r_i$
  $i \leftarrow i + 1$
end
return $r = \sum_i r_i$

Once the binary case have been explained, we show the extended method for the multiclass case.

Let's suppose first that f is an affine classifier, given by $f(x) = W^\top x + b$. Now, W is a linear transformation between $\mathbb{R}^m$ and $\mathbb{R}^c$ and $b \in \mathbb{R}^c$. If w_k is the k^{th} column of W, then $f_k(x) = w_k^\top x + b_k$ denotes the probability of x to be labeled as k. Thus, the minimal perturbation to fool the classifier in a point x_0 can be obtained as

$$\arg\min \|r\|_2 \text{ s.t. } \exists k : w_k^\top (x_0 + r) + b_k \geq w_{\hat{k}(x_0)}^\top (x_0 + r) + b_{\hat{k}(x_0)} \quad (7.2)$$

where $\hat{k}(x_0)$ is the correct label of x_0.

Finding the robustness of f at x_0 can be interpreted geometrically as computing the distance between x_0 and the complement of the convex polytope[1] P, given by:

$$P = \bigcap_{k=1}^{c} \{x \in \mathbb{R}^m : f_{\hat{k}(x_0)}(x) \geq f_k(x)\}, \quad (7.3)$$

which defines the region of the space where f outputs the label $\hat{k}(x_0)$ (see Fig. 7.3). This distance is denoted by $\text{dist}(x_0, P^C)$. Clearly, $x_0 \in P$.

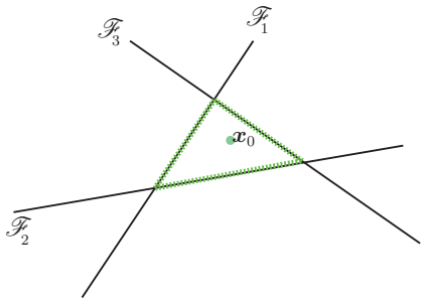

Figure 7.3: Illustration of the polytope P for x_0 belonging to class 4 and $\mathcal{F}_k = \{x : f_k(x) - f_4(x) = 0\}$. The boundary of P is shown in green dotted line [31].

In order to solve (7.2), we define $\hat{l}(x_0)$ as index of the closest hyperplane to the point x_0:

$$\hat{l}(x_0) := \arg\min_{k \neq \hat{k}(x_0)} \frac{|f_k(x_0) - f_{\hat{k}(x_0)}(x_0)|}{\|w_k - w_{\hat{k}(x_0)}\|_2}$$

Using this notation, the robustness of f at x_0 can be expressed as

$$\Delta(x_0; f) = \frac{|f_{\hat{l}(x_0)}(x_0) - f_{\hat{k}(x_0)}(x_0)|}{\|w_{\hat{l}(x_0)} - w_{\hat{k}(x_0)}\|_2}$$

Thereby, the adversarial perturbation $r_\star(x_0)$ is the vector that projects x_0 onto the hyperplane indexed by $\hat{l}(x_0)$, that is, the closest projection of x_0 on faces of P.

1 A polytope is a generalization in any number of dimensions of the three-dimensional polyhedron. It is said convex if it is also a convex set contained in $\mathbb{R}^n$.

$$r_\star(x_0) = \frac{\Delta(x_0; f)}{\|w_{\hat{l}(x_0)} - w_{\hat{k}(x_0)}\|_2}(w_{\hat{l}(x_0)} - w_{\hat{k}(x_0)}) =$$
$$= \frac{|f_{\hat{l}(x_0)}(x_0) - f_{\hat{k}(x_0)}(x_0)|}{\|w_{\hat{l}(x_0)} - w_{\hat{k}(x_0)}\|_2^2}(w_{\hat{l}(x_0)} - w_{\hat{k}(x_0)})$$

Lastly, the above procedure is extended to the general case of differentiable multiclass classifiers. The problem here is that the set P defined in (7.3), which describes the region of the space where the classifier outputs label $\hat{k}(x_0)$, is no longer a polytope. Thus, one can adopt an iterative procedure where, as in the binary case, f is linearized around the current point x_i in each iteration. Concretely, the set P at iteration i is approximated by the polytope $\tilde{P}_i$, given by:

$$\tilde{P}_i = \bigcap_{k=1}^{c} \left\{ x \in \mathbb{R}^m : f_{\hat{k}(x_0)}(x_i) + \nabla f_{\hat{k}(x_0)}(x_i)^\top x \geq f_k(x_i) + \nabla f_k(x_i)^\top x \right\}$$

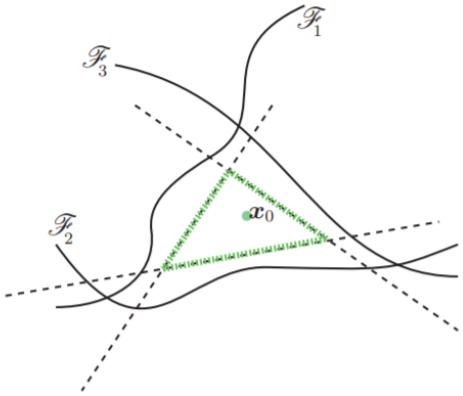

Figure 7.4: Illustration of the polytope $\tilde{P}_0$ for x_0 belonging to class 4 and $\mathcal{F}_k = \{x : f_k(x) - f_4(x) = 0\}$. The boundary of $\tilde{P}_0$ is shown in green dotted line [31].

Now, $\text{dist}(x_i, P^C)$ can be approximated by $\text{dist}(x_i, \tilde{P}_i^C)$ and update the current estimation. The full process is shown in algorithm 3.

In practice, the final adversarial perturbation is multiplied by a constant $1 + \eta$, with $\eta \ll 1$, in order to effectively reach the other side of the classification boundary.

During the whole process, we have used $p = 2$ in (5.1), but any other norm can be considered. As indicated in [31], algorithm 3 can be adapted to find ℓ_p-norm adversarial examples, by replacing the update steps in lines 7 and 8 with the following ones:

$$\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(x_0)} \frac{|f_k'|}{\|w_k'\|_q}, \qquad r_i \leftarrow \frac{|f_{\hat{l}}'|}{\|w_{\hat{l}}'\|_q^q} |w_{\hat{l}}'|^{q-1} \odot \text{sign}(w_{\hat{l}}')$$

where $q = \frac{p}{p-1}$ and $\odot$ is the pointwise product.

Algorithm 3: DeepFool.

Input : Image x, classifier f

Output : Perturbation r

Initialize $x_0 \leftarrow x$, $i \leftarrow 0$

while $\hat{k}(x_i) = \hat{k}(x_0)$ **do**

  for $k \neq \hat{k}(x_0)$ **do**

    $w'_k \leftarrow \nabla f_k(x_i) - \nabla f_{\hat{k}(x_0)}(x_i)$

    $f'_k \leftarrow f_k(x_i) - f_{\hat{k}(x_0)}(x_i)$

  end

  $\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(x_0)} \frac{|f'_k|}{\|w'_k\|_2}$

  $r_i \leftarrow \frac{|f'_{\hat{l}}|}{\|w'_{\hat{l}}\|_2^2} w'_{\hat{l}}$

  $x_{i+1} \leftarrow x_i + r_i$

  $i \leftarrow i + 1$

end

return $r = \sum_i r_i$

If $p = \infty$, the updates steps become:

$$\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(x_0)} \frac{|f'_k|}{\|w'_k\|_1}, \qquad r_i \leftarrow \frac{|f'_{\hat{l}}|}{\|w'_{\hat{l}}\|_1} \operatorname{sign}(w'_{\hat{l}})$$

This algorithm may not converge to the optimal perturbation given by (5.1), since it operates in a greedy way. However, it provides good approximations and is an efficient and accurate way to evaluate the robustness of classifiers.

Fig. 7.5 shows a couple of adversarial examples generated by DeepFool.

Figure 7.5: Adversarial images generated by DeepFool on ResNet-50.

The goal of this method is to minimize the number of perturbed pixels required to fool the network, obtaining "sparse perturbations" [30]. That correspond to minimizing $\|r\|_0$ in (5.1), but this leads to NP-hard problems.

An efficient approach to solve this problem is to use the ℓ_1 relaxation [5, 7, 33]. Briefly explained, this technique consists in approximating the original problem by solving the corresponding convex ℓ_1 problem (under linear constraints). To exploit such a relaxation, we can use DeepFool with $p = 1$ (ℓ_1-DeepFool). Here is a point to be made: although the ℓ_1-DeepFool method efficiently computes sparse perturbations, it does not explicitly respect the constraint on the validity of the perturbations, since the pixel values of the adversarial image may not be in the valid range (e.g., [0,255]). This makes the resulting adversarial images have a minor impact, since the invalid values are clipped after the computation.

Based on the previous discussion, SparseFool computes sparse adversarial perturbations using a "constrained version" of the ℓ_1-DeepFool method, that is, by solving the following optimization problem:

$$\arg\min_r \|r\|_1 \quad \text{s.t.} \quad \begin{cases} \hat{k}(x+r) \neq \hat{k}(x) \\ l \leq x + r \leq u \end{cases} \tag{7.4}$$

Here, $l, u \in \mathbb{R}^m$ denote the lower and upper bounds for the values of $x + r$, such that $l_i \leq x_i + r_i \leq u_i$, $\forall i = 1 \dots n$.

To find an efficient relaxation, the geometric characteristics of the decision boundaries are used. It has been shown [8, 9, 19] that the decision boundaries of state-of-art deep networks have a quite low mean curvature in the neighbourhood of data samples. Thus, given a datapoint x and its corresponding ℓ_2-norm adversarial perturbation v, the decision boundary at the vicinity of x can be locally well approximated by an affine hyperplane passing through the datapoint $x_B = x + v$ and a normal vector w, as depicted in Fig. 7.6.

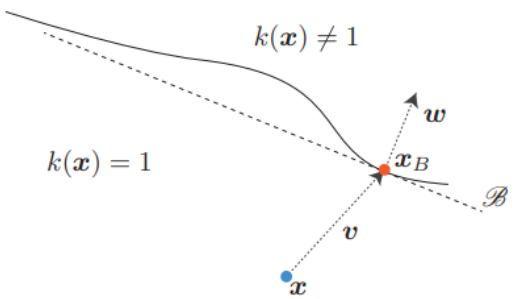

Figure 7.6: The approximated decision boundary $\mathscr{B}$ near the datapoint x. $\mathscr{B}$ can be seen as a one-vs-all linear classifier for class 1 [30].

This property can be exploited to linearize (7.4). Supposing that $\mathscr{B}$ is the approximated decision boundary (the affine hyperplane determined by x_B and w), we want to find a vector r with minimum $\|r\|_1$ such that $x + r$ belongs to $\mathscr{B}$. Since $x_B \in \mathscr{B}$, this can be expressed as $w^T(x + r) - w^T x_B = 0$, as the vectors w and $x + r - x_B$ are orthogonal. By adding the constraints on pixel values, the following optimization problem is obtained:

$$\arg\min_r \|r\|_1 \quad \text{s.t.} \quad \begin{cases} w^\top (x + r) - w^\top x_B = 0 \\ l \leq x + r \leq u \end{cases} \tag{7.5}$$

The computation of the ℓ_1 projection of x onto the approximated hyperplane does not guarantee a solution, because the resulting adversarial image may eventually not lie onto the approximate hyperplane after readjusting the invalid values to match the constraints. For this reason, the authors in [30] propose an iterative procedure where, at each iteration, the point x is projected only towards one single coordinate of the normal vector w at a time. If projecting x towards one direction does not provide a solution, it means that the perturbed image has reached its extrema value at this coordinate and it cannot contribute any further to finding a better solution, so this direction will be ignored in the next iteration.

Formally, let S be a set containing all the directions of w that cannot contribute to the perturbation. The index of the component of w which has the maximum absolute value is selected, that is, $d \leftarrow \arg\max_{j \notin S} |w_j|$. Then, the component d of the perturbation r is updated through the ℓ_1 projection of the current datapoint $x^{(i)}$ onto the estimated hyperplane as:

$$r_d \leftarrow \frac{|w^\top (x^{(i)} - x_B)|}{|w_d|} \cdot \text{sign}(w_d)$$

To ensure the validity of the values of the next iterate $x^{(i+1)}$ before proceeding to the next iteration, a projection operator Q is used to readjust the incorrect values of the update point, by projecting $x^{(i)} + r$ onto the box-constraints defined by l and u. Thereby, the new iterate is updated as $x^{(i+1)} \leftarrow Q(x^{(i)} + r)$. Note that the bounds l, u can also be used to control the perceptibility of the adversarial images.

Then, we have to check if $x^{(i+1)}$ has reached the approximated hyperplane. Otherwise, the perturbed image has reached its extrema value at the coordinate d, and we add that direction to the set S, reducing the search space.

The detailed algorithm for solving the linearized problem is shown Algorithm 4.

After that point, the only thing left to complete the solution to (7.5) is to find the boundary point x_B and the corresponding normal vector w.

Algorithm 4: LinearSolver

Input : Image x, normal w, boundary point x_B, projection operator Q

Output : Perturbed point $x^{(i)}$

Initialize: $x^{(0)} \leftarrow x$, $i \leftarrow 0$, $S = \{\}$

while $w^\top (x^{(i)} - x_B) \neq 0$ **do**

  $r \leftarrow 0$

  $d \leftarrow \arg\max_{j \notin S} |w_j|$

  $r_d \leftarrow \frac{|w^\top (x^{(i)} - x_B)|}{|w_d|} \cdot \text{sign}(w_d)$

  $x^{(i+1)} \leftarrow Q(x^{(i)} + r)$

  $S \leftarrow S \cup \{d\}$

  $i \leftarrow i + 1$

end

return $x^{(i)}$

Finding x_B is analogous to computing an ℓ_2-norm adversarial example of x. Thus, x_B can be very well approximated by $x + r_{adv}$, with r_{adv} being the corresponding ℓ_2-DeepFool adversarial perturbation. The normal vector can be estimated as:

$$w := \nabla f_{\hat{k}(x_B)}(x_B) - \nabla f_{\hat{k}(x)}(x_B)$$

Thereby, we can approximate the decision boundary by the affine hyperplane $\mathscr{B} \triangleq \{x : w^\top (x - x_B) = 0\}$, and use algorithm 4 to compute sparse adversarial perturbations.

Nevertheless, due to the fact that the decision boundaries are only locally flat [8, 9, 19], this one-step solution may not converge. To avoid that, the SparseFool method adopts an iterative procedure, where each iteration includes the linear approximation of the decision boundary, as shown below.

Algorithm 5: SparseFool

Input : Image x, projection operator Q, classifier f

Output : Perturbation r

Initialize: $x^{(0)} \leftarrow x$, $i \leftarrow 0$

while $\hat{k}(x^{(i)}) = \hat{k}(x^{(0)})$ **do**

  $r_{adv} = \text{DeepFool}(x^{(i)})$

  $x_B^{(i)} = x^{(i)} + r_{adv}$

  $w^{(i)} = \nabla f_{\hat{k}(x_B^{(i)})}(x_B^{(i)}) - \nabla f_{\hat{k}(x^{(i)})}(x_B^{(i)})$

  $x^{(i+1)} = \text{LinearSolver}(x^{(i)}, w^{(i)}, x_B^{(i)}, Q)$

  $i \leftarrow i + 1$

end

return $r = x^{(i)} - x^{(0)}$

Then, SparseFool uses the previous "Linear Solver" (Alg. 4) and the DeepFool method. Fig. 7.7 illustrates how it works.

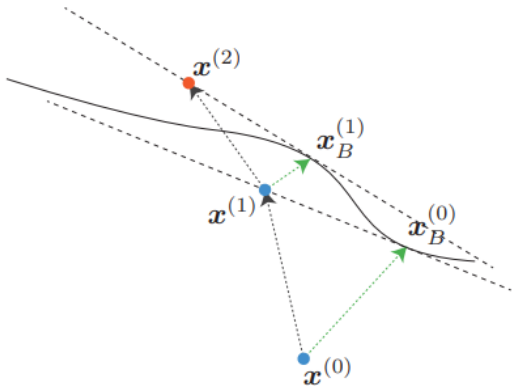

Figure 7.7: Illustration of the SparseFool algorithm. The ℓ_2-DeepFool adversarial perturbations are denoted in green. In this case, the algorithm ends after 2 iterations. The resulting adversarial perturbation is $r = x^{(2)} - x^{(0)}$ [30].

Finally, in order to obtain better results, one can go further into the other side of the boundary by using the datapoint $x^{(i)} + \lambda(x_B^{(i)} - x^{(i)})$ in algorithm 5, with $\lambda \geq 1$. The "control parameter" λ can be used to control the trade-off between the fooling rate, the sparsity and the complexity: values close to 1 lead to sparser perturbations, but also to lower fooling rate and high complexity. On the other side, higher values of λ lead to fast convergence, but less sparsity.

Below there are some examples of adversarial images generated with this method.

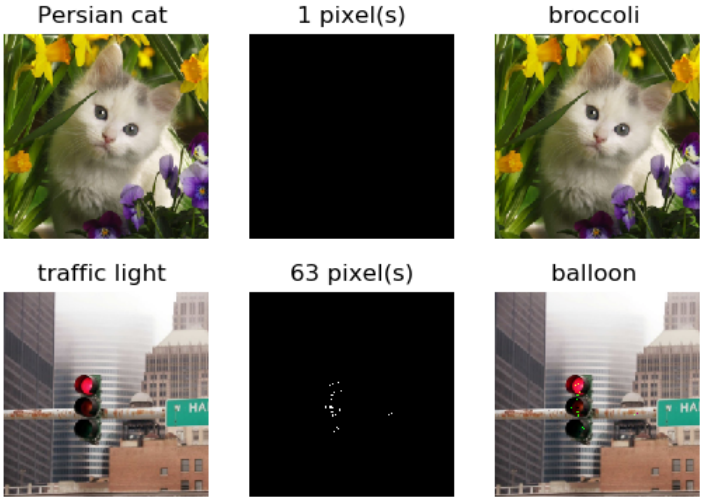

Figure 7.8: Adversarial images computed by SparseFool on a VGG-16 architecture. The perturbations are shown in the middle, with the number of pixels required to fool the network.

8

LIMITING THE SCENARIO: ONE PIXEL ATTACK

This method is focused on an extremely limited scenario, where only a few pixels can be modified.

Consequently, the problem now is different from (5.1): given a classifier f, an image $x \in \mathbb{R}^m$ and a target label $l \in \{1, \dots, c\}$, obtain an adversarial perturbation by solving

$$\arg\max_r f_l(x+r) \quad \text{s.t.} \quad \|r\|_0 \leq d, \tag{8.1}$$

where $d \in \mathbb{N}$ is a small number (generally $d \leq 3$) and f_l indicates the probability assigned by f to the class l.

In view of (8.1), the aim here is to maximize the probability of $x+r$ to be classified as l, so we are considering a targetted attack.

There is a subtle detail here: if the aim really was to find the perturbation with the minimum number of non-zero pixels (as occurs in SparseFool), the following minimization problem should have been used, instead of (8.1):

$$\arg\min_r \|r\|_0 \quad \text{s.t.} \quad \hat{k}(x+r) = l$$

The difference is that in (8.1), the maximum number of non-zero pixels is actually fixed, and we are seeing how far can we go with this constraint, which allow us to control the number of modified pixels in the adversarial image.

If $d = 1$, the modification can be seen as moving the data point along a direction parallel to the axis of one of the m dimensions. We refer to this method as the *One pixel attack*. Similarly, if $d > 1$, the d-pixel modification moves the data point within d-dimensional cubes, that is, in low-dimensional slices of input space.

In order to obtain a good approximation, the authors in [45] use Differential evolution[1] (DE) to solve (8.1). The process involves encoding the perturbation into an array (candidate solution) which is optimized (evolved) by DE.

1 DE is an evolutionary computation algorithm that optimizes a problem starting from an initial population of candidate solutions and creating iteratively new candidate solutions by combining existing ones, keeping those with the best scores on the optimization problem. In this way, the optimization problem is treated as a black box that merely provides a measure of quality given a candidate solution, so DE does not require the optimization problem to be differentiable, as is required by classic optimization methods such as gradient descent and quasi-newton methods, and can therefore also be used on optimization problems that are not even continuous, are noisy, change over time, etc. [44].

Concretely, one candidate solution contains a fixed number of perturbations (one for each pixel) and each perturbation is made up of 5 elements: x-y coordinates an RGB value of the perturbation. The initial population consists of 400 candidate solutions and at each iteration another 400 candidate solutions (children) are produced by using the usual DE formula:

$$x_i(g+1) = x_{r_1}(g) + F(x_{r_2}(g) + x_{r_3}(g)),$$

where g is the current index of generation, x_i is an element of the candidate solution, F is a scaling parameter (set to 0.5) and r_1, r_2, r_3 are different random numbers. The algorithm stops when a certain probability on the target class is reached or when the maximum number of iterations is exceeded.

In this case, no inner information of the classifier f is being used: it is sufficient to know the probability labels. This type of attacks is known as *semi-black-box* attack.

Below there are a few adversarial images generated with the One Pixel Attack in [45].

Figure 8.1: One-pixel attacks on ImageNet dataset. The modified pixels are highlighted with red circles. The original class labels are in black color while the target class labels are in blue, both with its corresponding confidence values.

9 UNIVERSAL ADVERSARIAL PERTURBATIONS

Until now, a fundamental property of adversarial perturbations has been its intrinsic dependence on datapoints: the perturbations are specifically crafted for each image independently. However, there exists universal image-agnostic perturbations that are common to most data points in the data distribution [32]. This kind of perturbations are known as "universal adversarial perturbations". Thereby, perturbing a new datapoint only involves the addition of the universal perturbation to the image, and does not require solving an optimization problem/gradient computation, as in the previous approaches.

Formally, let μ denote a distribution of images in $\mathbb{R}^m$. We want to find a small vector $v \in \mathbb{R}^m$ that fools the classifier f on almost all datapoints sampled from μ. In this way, the following two constraints have to be met by v:

i. $\|v\|_p \leq \xi, \ p \geq 1$

ii. $\mathbb{P}_{x \sim \mu}\left(\hat{k}(x+v) \neq \hat{k}(x)\right) \geq 1 - \delta$

The parameter ξ controls the magnitude of the perturbation and δ quantifies the desired fooling rate for all images sampled from the distribution μ.

Given $X = \{x_1, \ldots, x_n\}$ a set of images sampled from μ, the proposed algorithm seeks an universal perturbation v satisfying the above conditions. To do so, it proceeds iteratively over each x_i, computing at each iteration the minimal perturbation Δv_i that sends the current perturbed point, $x_i + v$ to the decision boundary of f. Then, it is aggregated to the current universal perturbation (see Fig. 9.1).

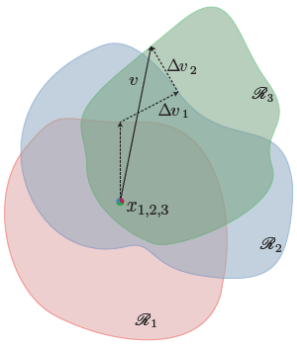

Figure 9.1: Illustration of the proposed algorithm. In this example, data points x_1, x_2 and x_3 are super-imposed, and the classification regions $\mathscr{R}$ are shown in different colors. In each step, the point x_i is send outside of the corresponding classification region $\mathscr{R}_i$ [32].

In more details, given the current universal perturbation v which does not fool the classifier on x_i, an extra perturbation Δv_i with minimal norm is computed by solving the following optimization problem:

$$\Delta v_i \leftarrow \arg\min_r \|r\|_2 \text{ s.t. } \hat{k}(x_i + v + r) \neq \hat{k}(x_i) \tag{9.1}$$

Now, to ensure that the constraint $\|v\|_p \leq \xi$ is satisfied, the updated universal perturbation is projected on the ℓ_p ball of radius ξ centered at 0, using the following operator:

$$\mathcal{P}_{p,\xi}(v) = \arg\min_{v'} \|v - v'\|_2 \quad \text{s.t. } \|v'\|_p \leq \xi$$

Hence, the update rule is given by $v \leftarrow P_{p,\xi}(v + \Delta v_i)$.

The algorithm ends when the empirical fooling rate on the perturbed data set $X_v = \{x_1 + v, \ldots, x_n + v\}$ exceeds the target threshold $1 - \delta$. That is, it stops whenever

$$\text{Err}(X_v) := \frac{1}{n} \sum_{i=1}^{n} 1_{\hat{k}(x_i+v) \neq \hat{k}(x_i)} \geq 1 - \delta$$

The complete algorithm is provided in Algorithm 6.

Algorithm 6: Computation of universal perturbations.

Input : Data points X, classifier f, desired ℓ_p norm of the perturbation ξ, desired accuracy on perturbed samples δ

Output : Universal perturbation vector v

Initialize $v \leftarrow 0$
while $Err(X_v) \leq 1 - \delta$ **do**
  for *each datapoint* $x_i \in X$ **do**
    if $\hat{k}(x_i + v) = \hat{k}(x_i)$ **then**
      Compute the *minimal* perturbation that sends $x_i + v$ to the decision boundary:

$$\Delta v_i \leftarrow \arg\min_r \|r\|_2 \text{ s.t. } \hat{k}(x_i + v + r) \neq \hat{k}(x_i)$$

      Update the perturbation:

$$v \leftarrow P_{p,\xi}(v + \Delta v_i)$$

    end
  end
end

This algorithm involves solving at most n optimization problems in each step. In order to solve (9.1), we can use one of the previous approximate methods like DeepFool.

Another aspect to consider is that the objective of Algorithm 6 is not to find the smallest universal perturbation, but rather to find one such perturbation with sufficiently small norm. Thus, we can obtain a diverse set of universal perturbations satisfying the required constraints.

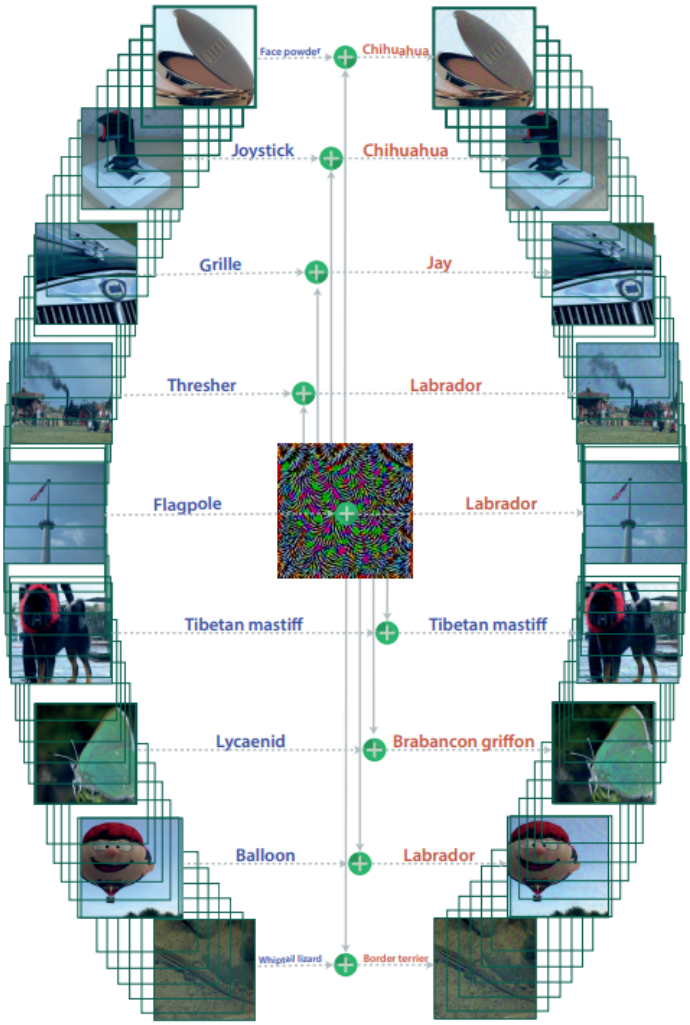

Figure 9.2: Example of universal perturbation (center). In the left are the natural images and in the right the perturbed images. The labels are shown on top of each image [32].

10 ADVERSARIAL METHODS COMPARISON

To end this chapter, we provide a table that compares the main aspects of the previously discussed methods. For each of these methods, it shows the information needed to compute the adversarial perturbations, the associated optimization problem, the parameters of the algorithm, the most remarkable pros and cons, and whether or not it is a targeted attack.

Method	**Information needed**	**Associated (opt.) problem**	**Targ.?**	**Pros**	**Cons**	**Parameters**
L-BFGS	Image x, $loss_f$, target label l	$min_{c,r}\ c\|r\|_2 + loss_f(x+r,l)$ s.t. $x+r \in [0,1]^m, \hat{k}(x+r) = l$	Yes	Constrained pixel values	Time, not scalable to high dim.	
FGSM	Image x, $loss_f$	$r = \epsilon \cdot sign(\nabla_x\ loss_f(x,y))$	No	Very quickly, constrained perturbation magnitude	Coarse approximations	ϵ: perturbation max norm
DeepFool	Image x, class. f	$\arg\min_r \|r\|_p$ s.t. (x+r) $\neq \hat{k}(x)$	No	Accuracy, efficiency	f must be differentiable, no pixel-value restrictions	η: overshoot
SparseFool	Image x, projection operator Q, class. f	$\arg\min_r \|r\|_1$ s.t. (x+r) $\neq \hat{k}(x)$, $l \leq x+r \leq u$	No	Accuracy, efficiency, constrained pixel values	f must be differentiable	l, u: lower and upper bounds for perturbation λ: control parameter
One Pixel Attack	Image x, target label l	$\arg\max_r f_l(x+r)$ s.t. $\|r\|_0 \leq d$	Yes	Simplicity, flexibility, semi-black-box attack	Low success rate	d: number of perturbed pixels
Universal	Data points X, class. f	$\arg\min_r \|r\|_2$ s.t. $\hat{k}(x_i+v+r) \neq \hat{k}(x_i)$	No	Image-agnostic No opt. required for new images	Time-consuming	ξ: desired ℓ_p norm of the perturbation, δ: desired accuracy on pert. samples

Table 10.1: Overall comparison of the different approaches.

From this perspective, the method that best fits the definition of adversarial perturbations (5.1) is DeepFool. However, the other adversarial methods provide interesting perspectives, useful for studying the behaviour of classifiers in certain scenarios.

The Quasi-Newton method was the first approach to find adversarial examples, using a conventional optimization method. Although it is really costly in terms of time, it served to demonstrate the existence of this type of perturbations, which led to the subsequent development of more precise approaches.

On the other hand, FGSM provides a very fast way to find these examples, but with reduced accuracy. Therefore, it is useful in cases where speed comes before accuracy.

DeepFool and SparseFool are designed to efficiently find adversarial images with high accuracy. DeepFool finds perturbations with minimal ℓ_p-norm, while SparseFool finds those with the minimum number of perturbed pixels. In addition, SparseFool allows to control the visibility of the perturbations, leading to different levels of sparsity.

The One Pixel Attack is really useful to examine extreme cases, where only one pixel of the image is modified. However, this makes it unsuccessful in many situations, where modifying a pixel is not enough to fool the classifier. Its main advantage is that it is a semi-black-box attack, then it can be used in situations where there is no complete information about the classifier.

Lastly, the universal adversarial method provides an universal perturbation for a specific classifier, which can be used with any input image from the input distribution. Once the adversarial perturbation is generated (which requires quite a bit of time), adversarial images can be generated without any effort.

Part III

EXPERIMENTS

11 PRELIMINARY COMMENTS

In the previous part of the work, the main methods for generating adversarial images have been exposed. Now, it is time to test them experimentally. Hence, the objective in this part will be to analyse the performance of such methods and to gain a better insight into them.

To do so, we will be using different deep neural networks architectures, trained on various datasets, so that the performance of a method can be tested in several ways. The following sections cover these aspects.

11.1 DATASETS

Two different datasets will be used during this process: CIFAR-10 and Image-Net.

11.1.1 *CIFAR-10*

The CIFAR-10 dataset, available at https://www.cs.toronto.edu/~kriz/cifar.html, will be used to test the overall performance of the adversarial methods. It contains 60.000 32x32 colour images in 10 classes, with 6.000 images per class. The dataset is separated in 50.000 training images and 10.000 test images. It will be this last set that we use to perform the tests. Fig. 11.1 shows the classes in the dataset, with 10 random images from each one.

The low resolution of the images in CIFAR-10 (32x32) makes this dataset suitable for our tests, as the images are not excessively complex, resulting in admissible execution times. It must be considered that, for each method, numerous experiments will be performed, where sometimes a huge number of adversarial images will be generated, so we cannot use high resolution images. However, this does not mean that the results obtained with this dataset are not representative. CIFAR-10 images are complex enough for this task, and they have been used in the papers where these methods were introduced, which allows us to compare the results obtained.

11.1.2 *ImageNet*

ImageNet is one of the most used databases for computer vision. It has more than 14M images, organized in more than 20.000 subcategories, which makes it very complete, but at the same time complex, since

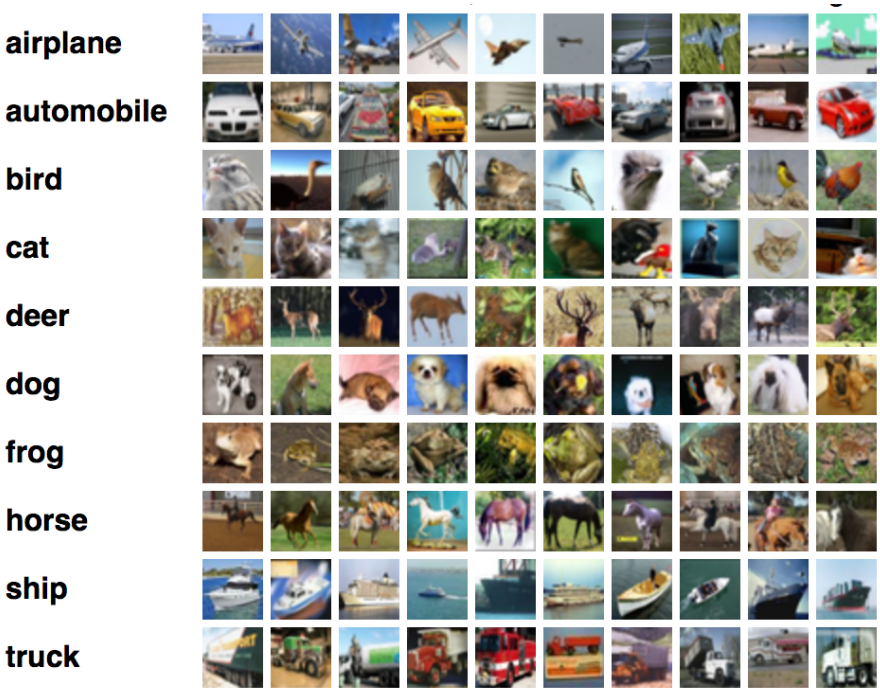

Figure 11.1: CIFAR-10 classes with 10 random images from each one.

it contains a wide variety of high resolution images (see examples in http://www.image-net.org/).

This dataset will be used only to show results in concrete natural images, considering 1.000 possible classes.

11.2 MODELS

We have considered a diverse set of state-of-the-art convolutional neural networks architectures, including ResNet-50, DenseNet-169, MobileNetV2 and InceptionV3.

The above-mentioned models will be used pretrained on the CIFAR10 dataset for performing the overall tests, using the weights available in https://github.com/huyvnphan/PyTorch_CIFAR10. For testing on natural images we will use the models pretrained on ImageNet, using the weights provided by *PyTorch* (see https://pytorch.org/docs/stable/torchvision/models.html for more information).

Those architectures are shown in detail below.

11.2.1 *ResNet-50*

		Pretrained on	
Layers	Output size	CIFAR10	ImageNet
conv1	112x112	3x3, 64, stride 1	7x7, 64, stride 2
conv2_x	56x56	3x3 max pool, stride 2	
		$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	
conv3_x	28x28	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	
conv4_x	14x14	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	
conv5_x	7x7	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	
Classification	1x1	7x7 global avg pool	
		10-d FC, softmax	1000-d FC, softmax
# trainable parameters:		23.520.842	25.557.032

Table 11.1: Resnet-50 architecture. A batch normalization layer is used after each convolution, and ReLU activations are used at the end of each group of layers.

11.2.2 *DenseNet-169*

		Pretrained on	
Layers	Output size	CIFAR10	ImageNet
Convolution	112x112	3x3, 64, stride 1	7x7, 64, stride 2
Pooling	56x56	3x3 max pool, stride 2	
Dense block (1)	56x56	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 32 \end{bmatrix} \times 6$	
Transition Layer (1)	56x56	1x1 conv, 128	
	28x28	2x2 avg pool, stride 2	
Dense block (2)	28x28	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 32 \end{bmatrix} \times 12$	
Transition Layer (2)	28x28	1x1 conv, 256	
	14x14	2x2 avg pool, stride 2	
Dense block (3)	14x14	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 32 \end{bmatrix} \times 32$	
Transition Layer (3)	14x14	1x1 conv, 640	
	7x7	2x2 avg pool, stride 2	
Dense block (4)	7x7	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 32 \end{bmatrix} \times 32$	
Classification	1x1	7x7 global avg pool	
		10-d FC, softmax	1000-d FC, softmax
# trainable parameters:		12.493.450	14.149.480

Table 11.2: DenseNet-169 architecture. Each conv. layer shown in the table corresponds the sequence BN-ReLU-Conv. The growth rate is $k = 32$.

11.2.3 *MobileNetV2*

		Pretrained on	
Layers	Output size	CIFAR10	Imagenet
Convolution	112x112	3x3, 32, stride 1	3x3, 32, stride 2
Bottleneck (1)	112x112	Inverted residual (t=1, c=16, n=1, s=1)	
Bottleneck (2)	56x56	Inverted residual (t=6, c=24, n=2, s=2)	
Bottleneck (1)	28x28	Inverted residual (t=6, c=32, n=3, s=2)	
Bottleneck (1)	14x14	Inverted residual (t=6, c=64, n=4, s=2)	
Bottleneck (1)	14x14	Inverted residual (t=6, c=96, n=3, s=1)	
Bottleneck (1)	7x7	Inverted residual (t=6, c=160, n=3, s=2)	
Bottleneck (1)	7x7	Inverted residual (c=1280, n=1, s=1)	
Classification Layer	1x1	7x7 avg pool	
		10-d fc, softmax	1000-d fc, softmax
# trainable parameters:		2.236.682	3.504.872

Table 11.3: MobileNetV2 architecture. Each bottleneck module is a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have c output channels. The first layer of each sequence has a stride s and all others use stride 1. All convolutions use 3×3 kernels. t is the expansion factor. For more information see [39].

11.2.4 *InceptionV3*

The architecture of InceptionV3 is too complex to be described in a table. For a summary, see figure 4.13. More information is available at [48].

The number of trainable parameters in CIFAR-10 version is 21.638.954, and 27.161.264 in ImageNet version.

We now introduce several metrics that will be used to measure different aspects of the adversarial methods.

- **Accuracy -** Accuracy of the model on the CIFAR10 test set ($\mathcal{D}_{test}$), defined as the number of images correctly classified by the model divided by the total number of images in $\mathcal{D}_{test}$.
- **Fooling rate -** Measures the effectiveness of a method. It is obtained by the formula $|x \in \mathcal{D}'_{test} : \hat{k}(x + r_x) \neq \hat{k}(x)|/|\mathcal{D}'_{test}|$, where $\mathcal{D}'_{test}$ is the set of images for which an adversarial image has been computed (the images on $\mathcal{D}_{test}$ that are successfully classified by the model), and r_x is the adversarial perturbation associated with the image x.
- **Average confidence -** Is the average of the confidence (highest softmax probability) assigned by the model to the adversarial images.
- **Empirical robustness in $\mathcal{D}'_{test}$ -** Average robustness of the model in each image of $\mathcal{D}'_{test}$ (see definition 5.3), calculated as

$$\hat{\Delta}(x; f) = \frac{1}{|\mathcal{D}'_{test}|} \sum_{x \in \mathcal{D}'_{test}} \Delta(x; f)$$

- **Empirical robustness -** Empirical robustness of the model f (see definition 5.4), obtained as

$$\hat{\rho}_{adv}(f) = \frac{1}{|\mathcal{D}'_{test}|} \sum_{x \in \mathcal{D}'_{test}} \frac{\Delta(x; f)}{\|x\|}$$

Unless otherwise stated, the euclidean norm will be used to calculate $\hat{\Delta}(x; f)$ and $\hat{\rho}_{adv}(f)$. The notation $\hat{\Delta}^{\infty}(x; f)$ and $\hat{\rho}^{\infty}_{adv}(f)$ will appear when using the maximum norm instead.

Finally, table 11.4 shows the accuracies obtained with the previous models in CIFAR10 test set. All of them obtain a very high score, with InceptionV3 being the best.

Model	ResNet-50	DenseNet-169	MobileNetV2	InceptionV3
Acc. (%)	92.12	92.84	92.99	95.41

Table 11.4: Accuracy of each model on CIFAR10 test set

12 EXPERIMENTAL RESULTS

In this chapter, several tests will be performed on the CIFAR-10 test set images using the adversarial methods described in the previous part. In addition, we use several high resolution images to see how these methods perform on more complex targets.

We will analyze each method separately to focus on its parameters and results on each model. The results obtained with each method are summarized in a table, which can be used to make an overall comparison.

The objective is to draw relevant conclusions about how each method works, and how they affect the models, i.e., why deep neural networks are vulnerable to this type of attack.

The L-BFGS method will not be covered here since its aim is rather to demonstrate the existence of adversarial perturbations (in fact, it was the first proposed method to find them), and it is not suitable for our experiments.

12.1 FAST GRADIENT SIGN METHOD

FGSM looks for a max-norm constrained perturbation for an image x with label y as $r = \epsilon \cdot sign(\nabla_x \, loss_f(x, y))$.

The first goal is therefore to find an appropriate ϵ for the perturbation to be both effective and barely visible in the image. We start by generating some adversarial perturbations for a randomly picked CIFAR-10 image using different values of ϵ. The results are shown in Figure 12.1.

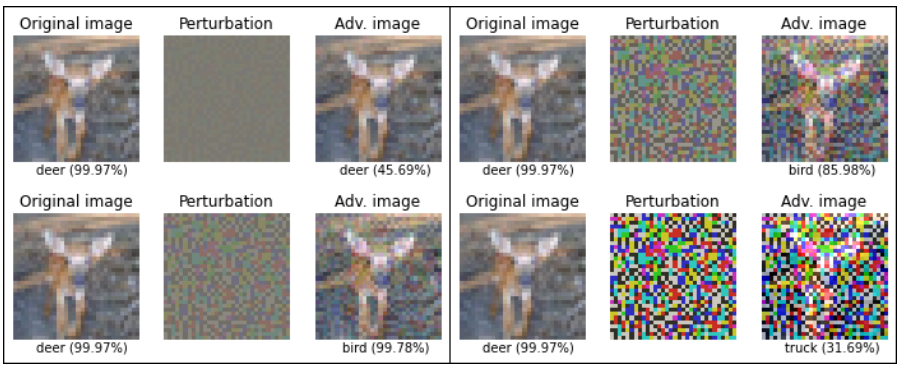

Figure 12.1: Adversarial images generated for InceptionV3 using $\epsilon = 0.01$ (top left), $\epsilon = 0.05$ (bottom left), $\epsilon = 0.1$ (top right) and $\epsilon = 0.3$ (bottom right). The first one is classified correctly as a deer, while the others are not.

Using a small ϵ may lead the generated image not to be effective, as in the top left image, but using large values instead make the resulting images be too noticeable (see adversarial images on the right).

The plot below shows the accuracy obtained by each one of the models on the CIFAR10 test set, using different values of ϵ. The accuracy decreases quickly as the value of ϵ increases. More detailed information is provided in table 12.1.

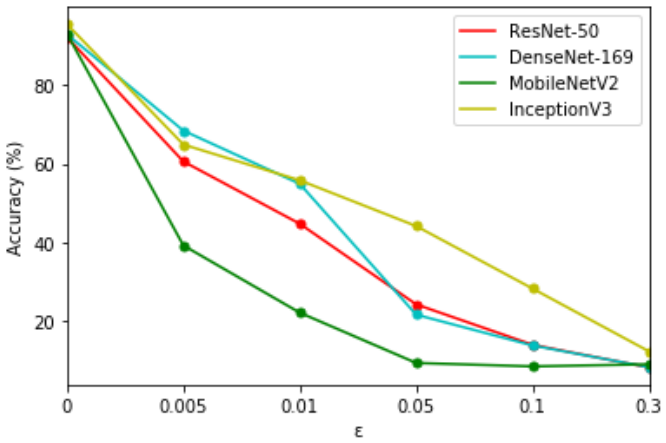

Figure 12.2: Accuracy obtained with the perturbed images generated with FGSM, using different values of ϵ.

Model	Acc. (%)	ϵ	Fool. rate (%)	Conf. (%)	Time (ms)
ResNet-50	92.12	0.01	51.45	79.42	11.6
		0.05	73.68	77.90	
		0.1	84.82	77.29	
DenseNet-169	92.84	0.01	40.91	84.09	35.9
		0.05	76.69	80.42	
		0.1	85.12	74.63	
MobileNetV2	92.99	0.01	76.19	81.49	10
		0.05	89.92	72.13	
		0.1	90.86	62.33	
InceptionV3	95.41	0.01	41.56	87.06	28.2
		0.05	53.74	83.06	
		0.1	70.38	72.49	

Table 12.1: Overall performance of FGSM on the CIFAR10 test set for different models and values of ϵ.

In view of the fooling rates, MobileNetV2 is significantly more sensitive to adversarial images generated with FGSM than the other models. Indeed, is the only model in which acceptable fooling rates are obtained

with not too high epsilon values (near 90% of adversarial images are misclassified with $\epsilon = 0.05$).

The architecture of the model also impacts on the running time, cause the gradient necessary to generate the perturbation is obtained by a backward step in the model. Complex architectures like DenseNet-169, where each of its layers is connected with several layers, results in larger times. Nevertheless, FGSM is very fast since it is a single-step method.

Another observation that can be made is that the average confidence decreases as the value of ϵ increases, specially in MobileNetV2 and InceptionV3 architectures. This may suggest that they are more sensitive to the noise introduced with this kind of perturbations.

It is important to remark that FGSM does not consider whether the adversarial examples generated are actually valid images (with values in [0,1] range), so we have clipped the incorrect values after generating each adversarial image on the previous experiments. This causes the method to be slightly less effective, as the perturbations become smaller in magnitude.

The table below shows how the clipping affects the magnitude of the perturbations and, therefore, its effectiveness. As CIFAR10 images are size $32 \times 32 \times 3$ (the last 3 represents the RGB value of each pixel), if the pixels values are not clipped, we can obtain mathematically $\|r\|_2$ for each perturbation r generated using FGSM as

$$\|r\|_2 = \sqrt{32 \times 32 \times 3 \times \epsilon^2} \tag{12.1}$$

ϵ	Fooling rate (%)		Fool. rate	$\hat{\Delta}(x; f)$		$\hat{\Delta}(x; f)$
	Clip	No clip	reduction (%)	Clip	No clip (12.1)	reduction (%)
0.005	34.27	34.32	0.15	0.2761	0.2771	0.36
0.01	51.45	51.52	0.14	0.5514	0.5542	0.51
0.05	73.68	73.74	0.08	2.7382	2.7713	1.19
0.1	84.82	85.19	0.43	5.4257	5.5426	2.11

Table 12.2: Effect of clipping in the ResNet-50 architecture.

The effect of clipping barely affects the fooling rate, because the reduction in $\hat{\Delta}(x; f)$ is small, but it increases when larger values of ϵ are used.

We can conclude by stating that FGSM is an acceptable method (more importantly, quickly) to generate adversarial examples. However, is not appropriate to estimate the robustness of a classifier since the magnitude of the perturbations depends on the parameter ϵ.

Figure 12.3 shows some adversarial examples generated for each one of the classifiers using different images from the CIFAR-10 test set. The value of ϵ has been increased starting from 0.01 until and adversarial image is successfully generated.

Confusing images lead to less visible perturbations (e.g., the image of the dog can be confused with a bird if we look at the hat). InceptionV3, which assigns very high confidences to these images, needs significantly higher values of ϵ to generate adversarial images, which indicates that it is more robust to this type of attack. It has been impossible to generate an adversarial example for the image of the frog in the InceptionV3 architecture. ResNet-50 is affected the most by FGSM, as the generated adversarial perturbations are difficult to detect.

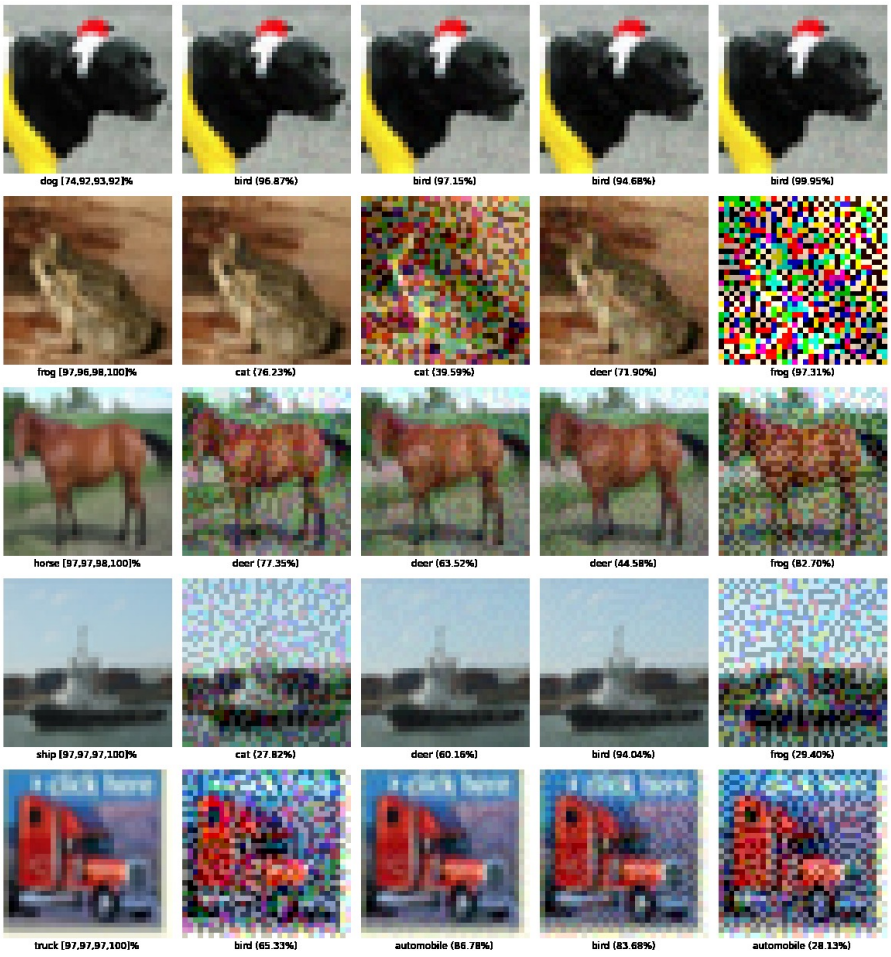

Figure 12.3: Adversarial examples generated using FGSM. Original images are on the left, with the correct label and confidences assigned by each model. The second, third, fourth and fifth columns correspond to the adversarial images generated for ResNet-50, DenseNet-169, MobileNetV2 and InceptionV3, respectively.

Figure 12.4 shows some adversarial images generated for high resolution images (obtained from https://es.freeimages.com/) using the ImageNet-pretrained versions of the models, following the same process as before.

Once again, InceptionV3 appears to be the robust model when using FGSM and ResNet-50 the most vulnerable.

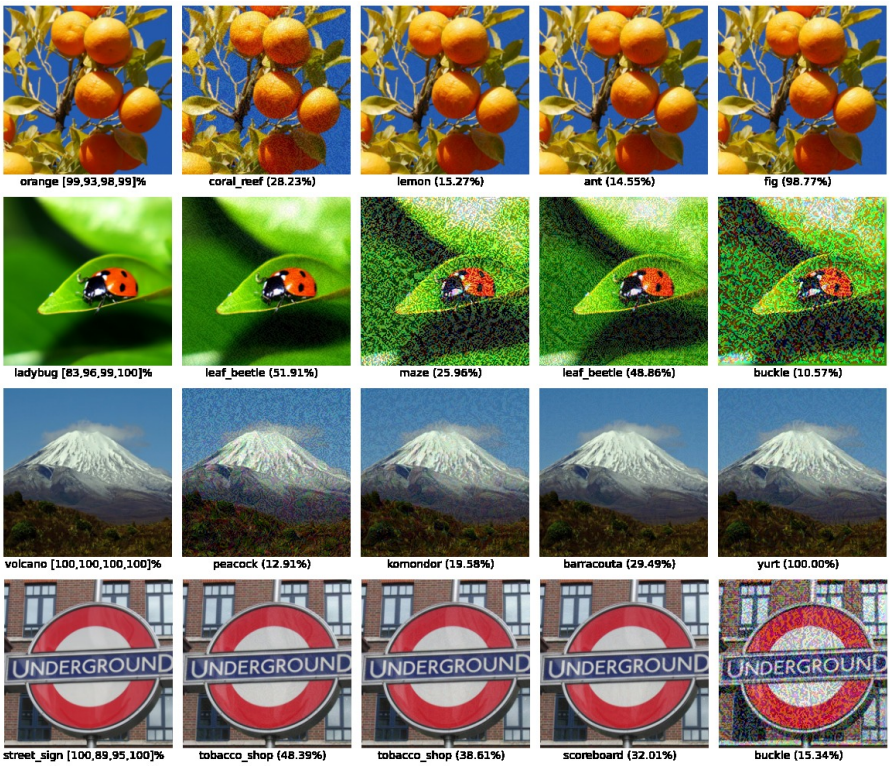

Figure 12.4: Adversarial examples generated using FGSM on high resolution images.

12.2 DEEPFOOL

This method finds adversarial examples by using the orthogonal projection of a point onto a class-separating affine hyperplane. This is an approximation, since the classifier is "linearized" around the data point to calculate such a projection. In practice, the final perturbation is multiplied by a constant $1 + \eta$, with $\eta \ll 1$, to ensure that the adversarial examples really overshoot the classification boundary. In the following experiments, unless otherwise state, we use $\eta = 0.01$.

We have performed some attacks to a few models using the DeepFool method. The results are shown in table 12.3.

Model	Fool. rate (%)	$\hat{\Delta}(x; f)$	$\hat{\rho}_{adv}(f)$	Time (s)	Avg. iters
ResNet-50	93.04	1.34	0.10	0.32	1.51
DenseNet-169	96.51	2.06	0.16	0.54	1.29
MobileNetV2	96.17	4.58	0.36	0.16	1.45
InceptionV3	91.96	0.58	0.05	0.48	2.22

Table 12.3: Overall performance of DeepFool on the CIFAR10 test set.

The first thing that jumps out is that DeepFool successfully fools the classifiers in more than 90% of the images, even excess 95% of fooling

rate in DensetNet-169 and MobileNetV2, so it is really a very effective method.

The values of $\hat{\Delta}(x; f)$ differ considerably depending on the classifier, reaching a low value (hardly perceptible perturbations) in InceptionV3 and a high value (large perturbations) in MobileNetV2. In consequence, different values of $\hat{\rho}_{adv}(f)$ are obtained. Higher values of $\hat{\rho}_{adv}(f)$ indicate that one have to move further in the input space to find adversarial examples, so the classifier f will be less sensitive to small perturbations, i.e., more robust. We can expect complex architectures like InceptionV3 or DenseNet-169 to have more complex classification boundaries, which may cause these boundaries to be closer to the data, and therefore is easier to find adversarial examples.

In the figure below there are some adversarial images generated with DeepFool for each model, using the same input image. Generally, the more robust the model, the bigger the perturbation has to be.

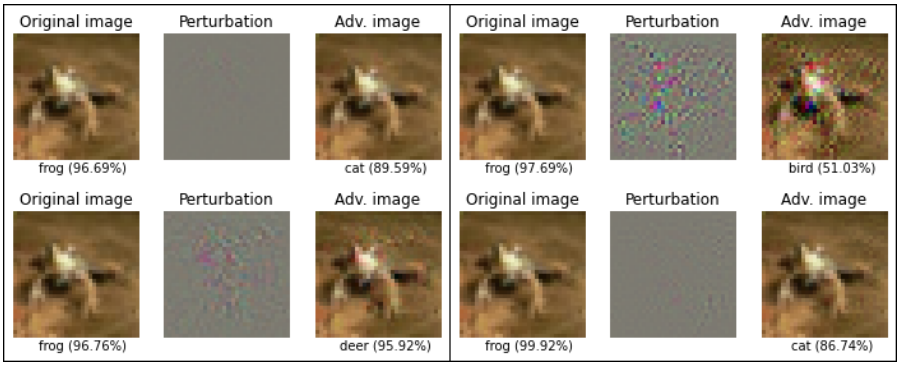

Figure 12.5: Adversarial images generated using DeepFool for ResNet-50 (top left), DenseNet-169 (bottom left), MobileNetV2 (top right) and InceptionV3 (bottom right).

The empirical robustness $\hat{\Delta}(x; f)$ varies significantly depending on the input images, even using the same model f, as each image is at a certain minimum distance from a class-separating hyperplane. Figure 12.6 shows different adversarial images generated for the same classifier. The perturbations are quite different (in magnitude) depending on the input image.

In terms of running time, DeepFool requires more time than FGSM, due to the iterative procedure and the operations needed by this method, but it is worth if we just consider the high fooling rates obtained. It can also be observed that the order of the models in terms of the required time is the same obtained when using FGSM (see table 12.1). It has also been found experimentally that DeepFool converges in a few iterations (between 1 and 3 on average in these experiments).

Note that, as in the previous method, the perturbations are obtained without considering the validity of the adversarial images, so we have

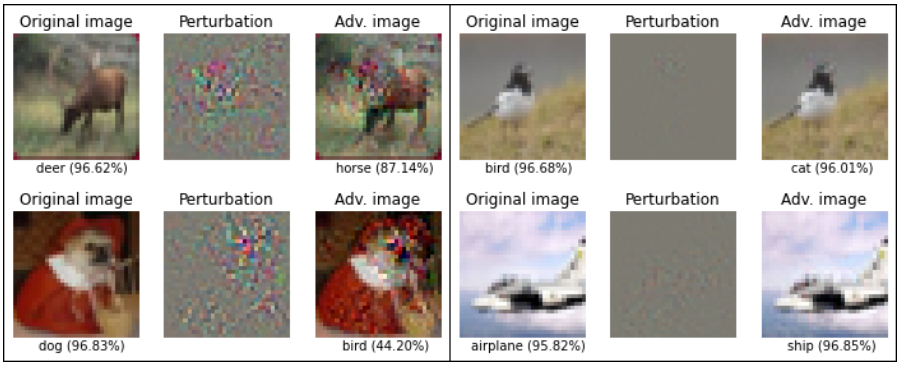

Figure 12.6: Adversarial images generated for DenseNet-169. Perturbations on the left are clearly visible, while those on the right may go unnoticed.

clipped the pixel values in the range [0,1] after generating each adversarial image. As the adversarial examples generated by DeepFool are very close to a decision boundary, it is likely that by clipping the values the image will lay on the other side of the boundary, even if the difference is small, leading to a lower fooling rate. Indeed, DeepFool has obtained a **100% fooling rate** in all the models when no clipping has been used. The table below shows how $\hat{\Delta}(x; f)$ is reduced by the clipping, and the real values of $\hat{\rho}_{adv}(f)$, obtained with the original magnitudes of the perturbations. The higher $\hat{\Delta}(x; f)$ is, the more significant is the reduction, because the wide range of perturbations makes the pixel values exceed the [0,1] range easily.

Model	$\hat{\Delta}(x; f)$		Reduction (%)	$\hat{\rho}_{adv}(f)$ (no clip)
	No clip	Clip		
ResNet-50	1.38	1.34	2.90	0.11
DenseNet-169	2.21	2.06	6.79	0.17
MobileNetV2	9.09	4.58	49.61	0.71
InceptionV3	0.59	0.58	1.69	0.05

Table 12.4: Clipping effect on $\hat{\Delta}(x; f)$.

Table 12.5 shows the results obtained when we use $p = \infty$ in DeepFool, that is, when the perturbations are measured using the maximum norm. Comparing with the results obtained when using $p = 2$ (table 12.3), the fooling rates are quite similar (maybe slightly better in this last version). We also observe similar values between $\hat{\rho}^{\infty}_{adv}(f)$ and $\hat{\rho}_{adv}(f)$ (table 12.3). The average time to generate the perturbations is higher in some models, and the average number of iterations decreases slightly.

In figure 12.7, there are some adversarial images generated with both $p = 2$ and $p = \infty$ versions of DeepFool. The use of different norms to measure the magnitude of the perturbations, makes them be quite

Model	Fool. rate (%)	$\hat{\Delta}_\infty(x;f)$	$\hat{\rho}^\infty_{adv}(f)$	Time (s)	Avg. iters
ResNet-50	95.98	0.04	0.08	0.27	1.26
DenseNet-169	97.92	0.06	0.12	0.79	1.17
MobileNetV2	95.75	0.19	0.38	0.23	1.32
InceptionV3	93.46	0.02	0.04	0.61	1.84

Table 12.5: Overall performance of DeepFool ($p = \infty$) on the CIFAR10 test set.

different. Those obtained with $p = 2$ tend to be more localized, but those obtained with $p = \infty$ are distributed uniformly throughout the image.

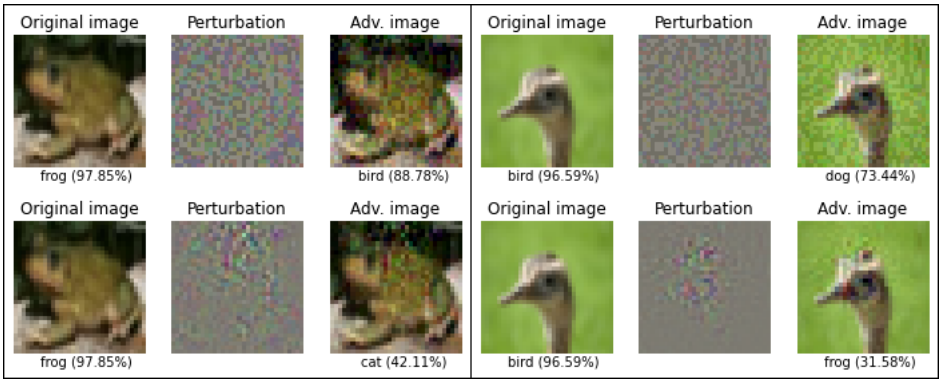

Figure 12.7: Adversarial images generated for ResNet-50 with $p = 2$ (images at the bottom) and $p = \infty$ (images at the top).

We can therefore conclude that DeepFool is a very effective method, which effectively calculates ℓ_2 and ℓ_∞-norm adversarial perturbations in a reasonable period of time.

Figure 12.8 shows some adversarial examples generated using DeepFool for each one of the classifiers (the order is always the same than in the previous section: ResNet-50, DenseNet-169, MobileNetV2 and Inception V3), using different images from CIFAR-10.

In general, DeepFool computes small perturbation, that are difficult to distinguish with the naked eye. As we have seen before, MobileNetV2 is the robust model, which means that strongest perturbations are needed to fool the classifier (see adversarial images in the second and fourth row). The fooling labels tends to be the same in all classifiers.

Finally, figure 12.9 shows adversarial images obtained on some high resolution images for the models pretrained on ImageNet. In this examples, the perturbations are all unnoticeable, which emphasizes the effectiveness of this method.

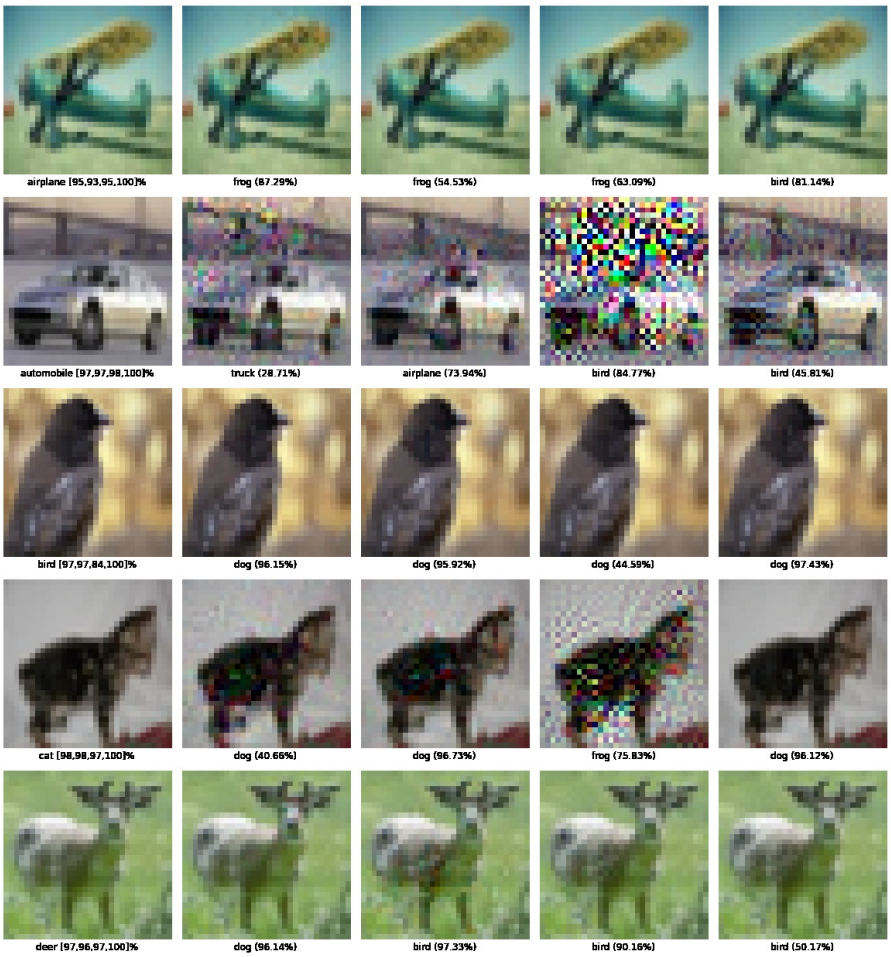

Figure 12.8: Adversarial images generated for CIFAR10 images.

Figure 12.9: Adversarial images generated for high resolution images.

In this method, the goal was to generate sparse perturbations, that is, adversarial images where only a few pixels have been modified.

If we remember, SparseFool had a control parameter λ that could be used to control the trade-off between the fooling rate, the sparsity and the complexity of the perturbations, by going further into the other side of the boundary when using DeepFool. In this way, we start by doing some experiments to select a good value for λ. The results are depicted in Fig. 12.10.

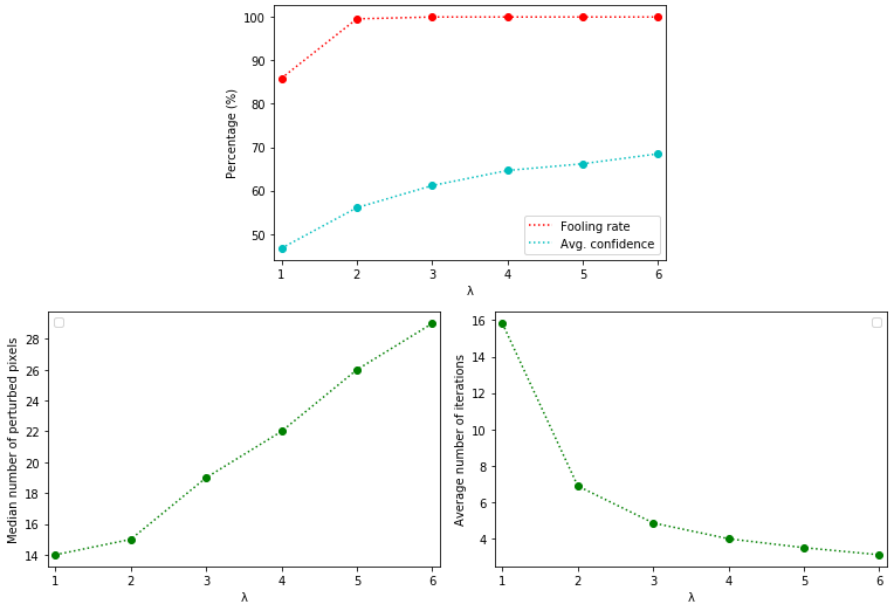

Figure 12.10: The fooling rate, the average confidence, the sparsity of the perturbations and the average iterations of SparseFool for different values of λ, on 2000 images from the CIFAR10 test set in a ResNet-50 architecture.

The fooling rate increases as we consider higher values of λ (reaching 100% fooling rate with $\lambda \geq 3$), and the average number of iterations decreases exponentially, which means a lower time needed to compute adversarial images. The average confidence also increases, presumably because the points are moving away from the boundary of decision. On the other hand, the sparsity of the perturbed images decreases, as more pixels are needed to modify the original images. In view of the results, in the following experiments we will use $\lambda = 3$.

Now we perform a complete attack to different classifiers. The results are shown in table 12.6. As this method is not intended to find adversarial perturbations with minimum $\|\cdot\|_2$ or $\|\cdot\|_\infty$ norm, but those with minimum $\|\cdot\|_0$ norm, the median number of perturbed pixels is a great estimator of the robustness of the classifiers, instead of $\hat{\rho}_{adv}(f)$

or $\hat{\rho}^{\infty}_{adv}(f)$. In this way, the column "Pert. (%)" shows the median percentage of the pixels that are perturbed per fooling sample.

Model	Fool. rate (%)	Pert. (%)	$\hat{\Delta}(x; f)$	Avg. iters	Time (sec)
ResNet-50	99.97	0.62	2.42	4.94	0.91
DenseNet-169	100	0.75	2.80	4.76	2.22
MobileNetV2	99.84	0.91	3.60	9.76	1.67
InceptionV3	100	0.55	2.15	5.58	2.20

Table 12.6: Overall performance of SparseFool on the CIFAR10 test set.

In view of the results, we can afirm that SparseFool is a reliable method to generate adversarial images, achieving fooling rates very close to 100% (even 100%) in all the models. It is also effective in the purpose of generating sparse perturbations, since barely 1% of the original image pixels have been modified to generate the adversarial image. The only handicap we can point out is its larger running time (in complex architectures like InceptionV3 or DenseNet-169, it exceeds 2 seconds per sample). This is due to the fact that SparseFool needs to run other 2 algorithms in each iteration it does, a "linear solver" and the DeepFool method. Again, the the most time-consuming models are DenseNet-169 and InceptionV3. ResNet-50 is significantly faster.

We can see how MobileNetV2 is the robust model (in terms of the median number of perturbed pixels required to fool the model), followed by DenseNet-169 and ResNet-50. InceptionV3 is in the last place. We got the same order when we used $\hat{\rho}_{adv}(f)$ in DeepFool (see tables 12.3 and 12.5). The values of $\hat{\Delta}(x; f)$ are also in line with the stated.

Fig. 12.11 shows some adversarial examples generated by SparseFool.

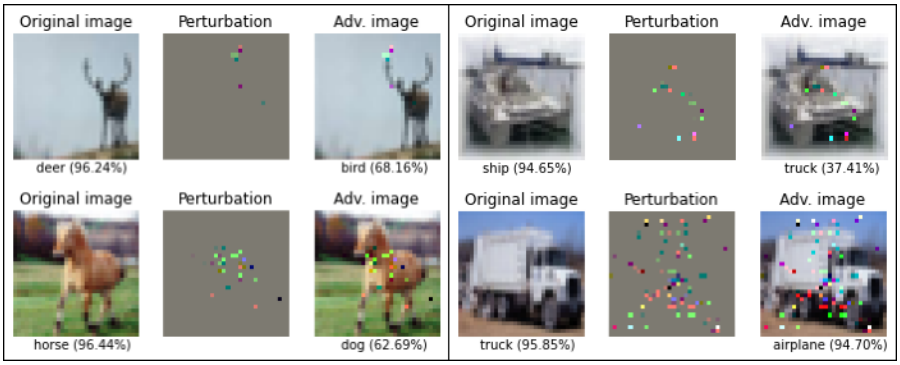

Figure 12.11: Adversarial images generated by SparseFool on a ResNet-50 architecture. We can see different levels of sparsity depending on the input image.

Now we focus on the visibility of the perturbations, by adjusting the lower and upper bounds for the generated images pixel values (the bounds l, u in (7.4)). To do so, we explicitly constrain the values of the adversarial image to lie in an interval $\pm\delta$ around the original image, such

that $0 \leq x_i - \delta \leq \hat{x}_i \leq x_i + \delta \leq 255$, where x_i and $\hat{x}_i$ denote the pixel values on the original image and the adversarial image, respectively. The higher the value of δ, the more freedom we give to the perturbations, increasing the possibility of some perturbed pixels to reach visible levels. For $\delta = 255$, the full dynamic range of the image is exploited (we used this value in the previous experiments). Fig. 12.12 shows the sparsity and the magnitude of the perturbations (in terms of its euclidean norm) obtained with different values of δ.

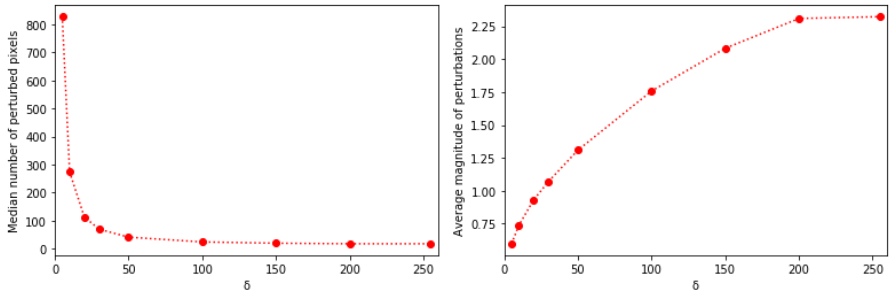

Figure 12.12: The sparsity and magnitude of the perturbations for different values of δ, generated for 200 samples from CIFAR10 test set on a ResNet-50 architecture.

The median number of perturbed pixels decreases very quickly but, after $\delta \approx 100$, these values seem to differ only slightly, which indicates that we do not need to use the whole dynamic range of the image. In the right plot it can be observed that $\hat{\Delta}(x; f)$ increases as δ does, even though far fewer pixels have been modified, which means that the pixels values are modified in a large range, as can be visualized in fig. 12.13.

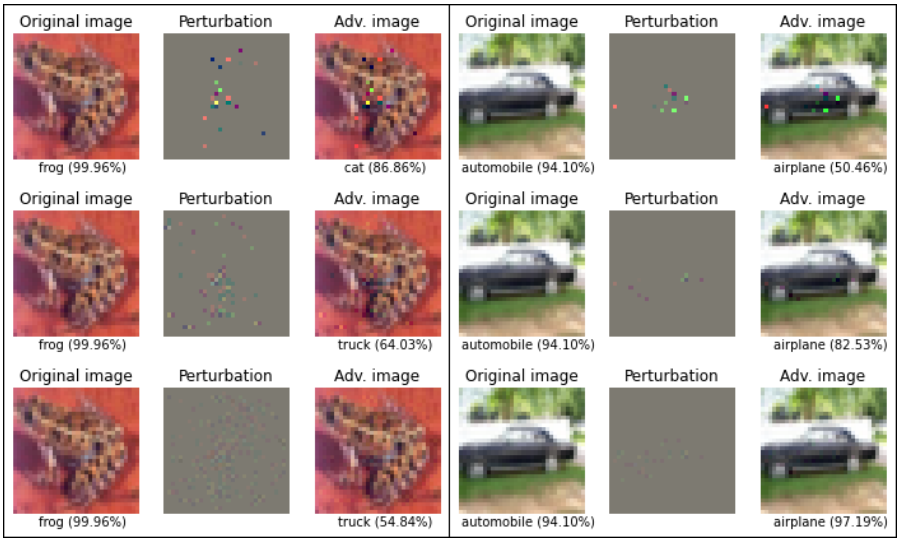

Figure 12.13: The effect of δ on the perceptibly and the sparsity of the adversarial images. Images in the first row have been generated with $\delta = 255$, images on the middle row with $\delta = 30$ and images at the bottom with $\delta = 10$. We have used the InceptionV3 model.

Figure 12.14 shows some adversarial examples generated with SparseFool in the CIFAR10 test set. The fooling labels tends to be the same and the modified pixels are in similar regions of the images. Depending on the model and the image, there are different levels of sparsity.

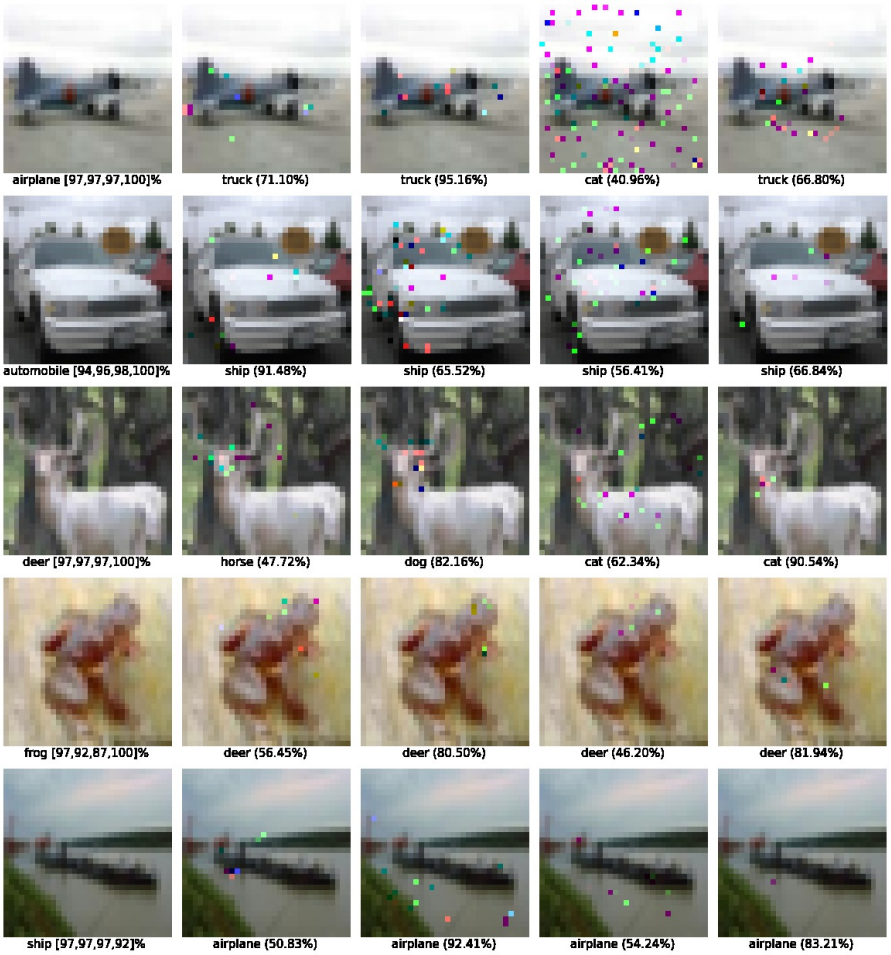

Figure 12.14: Adversarial images generated for some CIFAR10 images, using the four models.

To test the ImageNet-pretrained version of the models in high resolution images, we have used some images extracted from https://es.freeimages.com/. Below are the results.

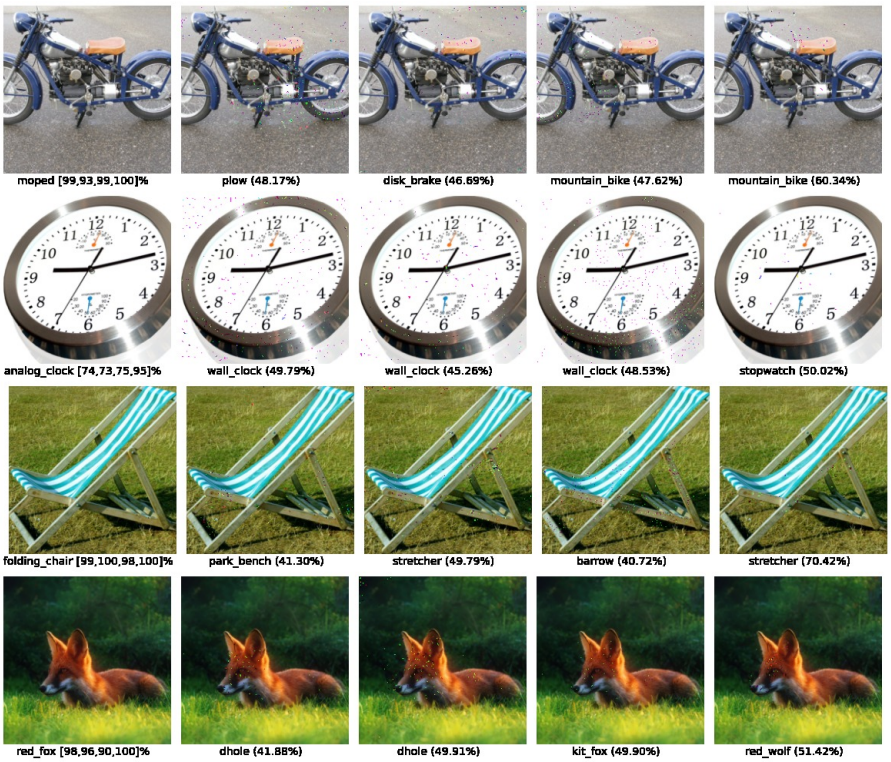

Figure 12.15: Adversarial images generated for some natural images, using the four models pretrained on ImageNet.

12.4 ONE PIXEL ATTACK

Now we pretend to fool a classifier by perturbing only one pixel of the image. Following the indications in [45], the candidate solutions are initialized by using an uniform distribution $U(1, 32)$ for x-y coordinates and a Gaussian distribution $N(\mu = 128, \sigma = 127)$ for RGB values (supposing that they are in range [0,255]), and evolved using Differential Evolution (DE). The algorithm stops when the maximum number of iterations is reached and early-stop criteria will be triggered when the probability label of target class exceeds 50% in the case of targeted attacks, and when the label of the true class is lower than 10% in the case of non-targeted attacks.

Due to the extremely high running time required by DE, we have reduced the population to 100 candidates and the number of iterations to 50 to calculate the overall performance in the 4 models we are using, and we have only performed the attack in a reduced set of 100 images of the original CIFAR10 test set. The results are shown in table 12.7.

Although we have a few data (we have used only 1% of the images in CIFAR10 test set), we could see that the fooling rate is lower than in the other methods. This is not surprising, as we are limiting the perturbation to have only one non-zero pixel. Moreover, the fooling rates may be higher than we expected, like in ResNet-50 and DenseNet-169

Model	Fool. rate (%)	Conf. (%)	$\hat{\Delta}(x; f)$	Avg. iters	Time (sec)
ResNet-50	41	76.30	1.01	39.5	60
DenseNet-169	39	75.63	1.00	39.3	156
MobileNetV2	33	74.14	0.94	42.9	40
InceptionV3	32	85.92	0.98	40.16	113

Table 12.7: Overall performance of One Pixel Attack on 100 test images.

architectures, where about 40% of the adversarial images generated have succeeded in fooling the classifier.

Note that, as we are using color images, each perturbation modifies 3 values of the original image, corresponding to the RGB value of the pixel, so $\hat{\Delta}(x; f)$ has to be lower than $\sqrt{3} \approx 1.73$.

In the matter of time, we could see extremely high values, because of the DE algorithm. This time is even greater in some models like DenseNet-169 and InceptionV3, where generate one adversarial image may take more than 2 minutes.

The average number of iterations needed by the algorithm is near to the limit of 50 iterations that we established (specially in MobileNetV2), which means that the method does not converge in most of the cases (the low fooling rates obtained by these classifiers confirm this).

Below we show 3 adversarial images generated by this method, using an initial population of 400 candidate solutions and setting the maximum number of iterations in 100.

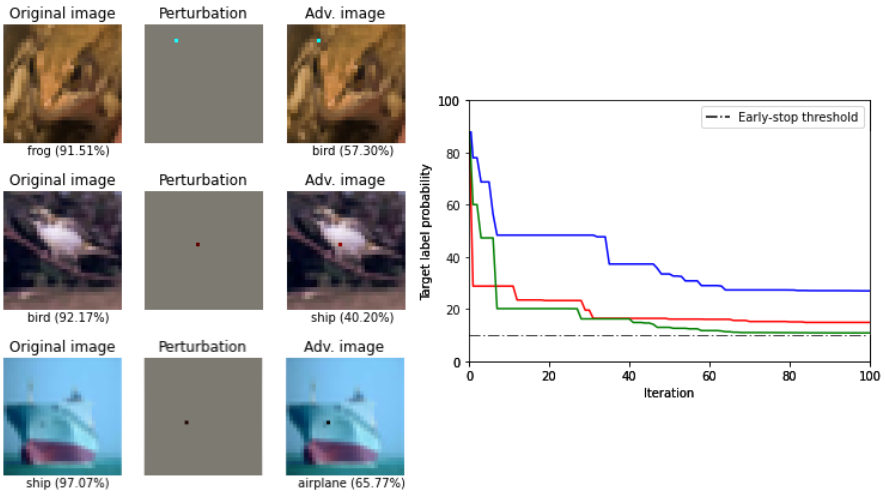

Figure 12.16: Adversarial images generated by One Pixel Attack. The right plot shows the evolution of the probability assigned by ResNet-50 to the correct label of the images (red line for the top image, green for the middle image and blue for the bottom image). None of them reach the 10% threshold but the attacks are successful.

Now we perform a targeted attack on one image, where we select previously the label we want the image to be classified as (we have used 3 different targets). The results are shown below.

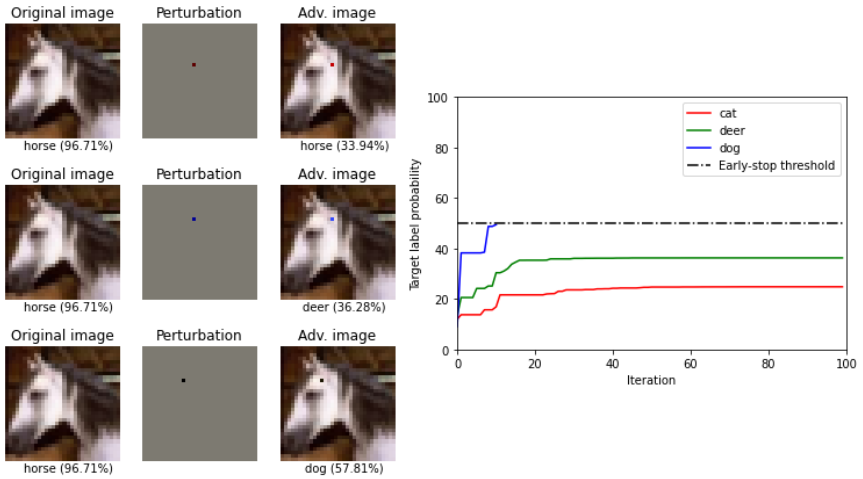

Figure 12.17: Targeted attacks in MobileNetV2. The target classes are cat (top image), deer (middle image) and dog (bottom image). The attacks succeed in the two last cases. At the right is a plot with the evolution of the probability of each target label. Only the one associated with "dog" reaches the 50% threshold.

We have observed that for most of the images, it is really difficult to successfully perform a targeted attack.

To see how the number of perturbed pixels affects the performance of this method, we have performed 3 attacks on the same image, but allowing to change a different number of pixels, as shown in figure 12.18.

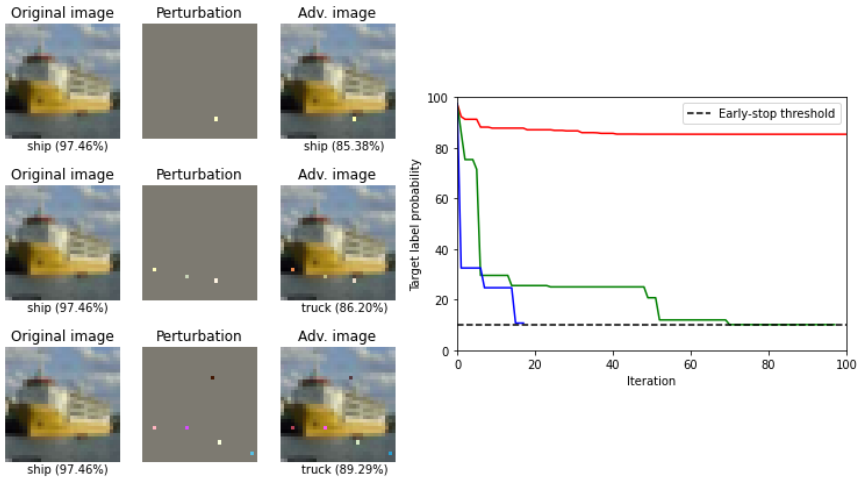

Figure 12.18: 1-pixel attack (top image, red line), 3-pixel attack (middle image, green line) and 5-pixel attack (bottom image, blue line). The performance increases with the number of pixels. These perturbed pixels seems to be quite similar in the 3 cases. The 10% threshold is reached in the 3 and 5-pixel attacks.

It has been quite difficult to successful obtain an adversarial example for a high resolution image. Below is an example generated for ResNet-50. We have had to use 10 pixels and a population of 1000 candidate solutions to generate it.

Figure 12.19: Adversarial example for ResNet-50 using the 10-Pixel Attack.

Thus, we can conclude that One Pixel Attack is an interesting method for testing some properties of adversarial attacks, but not for successfully compute adversarial perturbation, due to the large time required by the method and the poor results in most of the cases.

12.5 UNIVERSAL ADVERSARIAL PERTURBATIONS

We have left the universal method to the end, because it uses a different approach to generate adversarial images: the objective is to find a single adversarial perturbation that works for most of the images in a dataset.

In this way, the DeepFool method is used to update the universal perturbation in an iterative process that uses only a subset X of the dataset, until the empirical fooling rate exceeds the target threshold $1 - \delta$ or the maximum number of iterations is reached (one iteration consists in going through all the images in X). In our experiments, we used 1.000 images of the original CIFAR10 test set as X, $\delta = 0.2$ and a maximum of 10 iterations.

In the first experiment, we seek for an universal perturbation for each of the models we are testing, setting ξ (the maximum norm of the perturbation) to 15. The results are shown in table 12.8

Model	Fool. rate in X (%)	Fool. rate (%)	Time (s)	Iters
ResNet-50	80.20	78.54	110	4
DenseNet-19	63.20	62.69	392	10
MobileNetV2	86.70	85.48	64	1
InceptionV3	85.20	84.13	131	1

Table 12.8: Fooling ratios on the set X and the CIFAR10 test set, average time per iteration and number of iterations needed to generate an universal adversarial perturbation v in each model, with $\|v\|_2 \leq 15$.

We observe that MobileNetV2 and InceptionV3 have been more sensitive to this kind of attack, obtaining high fooling rates. For DenseNet-169, it has been impossible to surpass the fooling rate threshold in 10 iterations, obtaining a significantly lower value. We can see how the fooling rate in the entire test set (consisting in 10.000 images) is only slightly lower that the fooling rate obtained in the subset X used to compute the universal perturbation, which is 10 times smaller. This shows the effectiveness of this method in obtaining "universal" perturbations.

In the matter of running time, the model that has required the most has been DenseNet-169, 3 times more than the next, InceptionV3, which has required just over two minutes. MobileNetV2 is the faster, taking only 1 minute to generate the universal perturbation in one iteration.

Figure 12.20 shows the universal perturbations obtained for each model. We can see how some of them present some kind of structures, that are used to deceive the model.

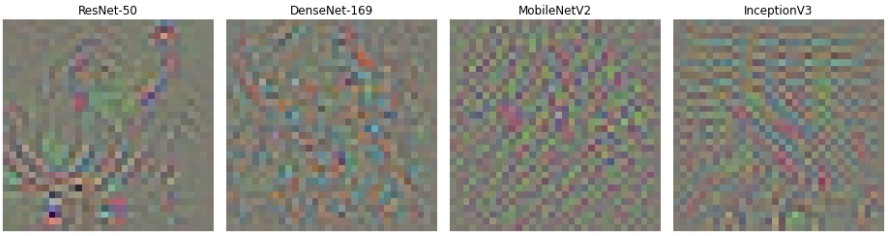

Figure 12.20: Universal perturbations obtained for each model.

Below there are a few examples of adversarial images obtained with the universal perturbation generated for InceptionV3.

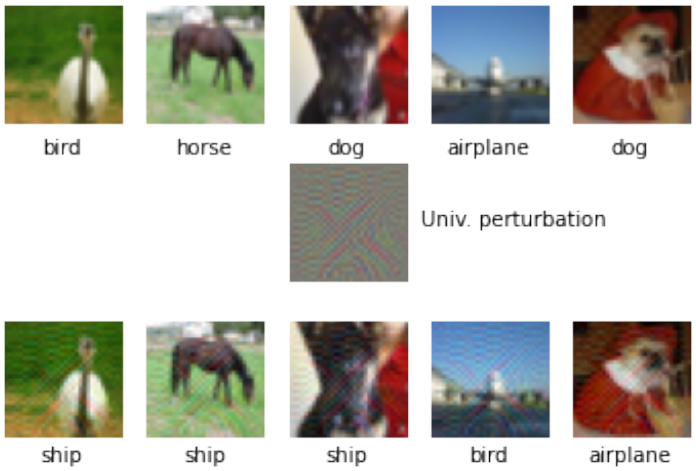

Figure 12.21: Adversarial images obtained when adding the universal perturbation to the images at the top in InceptionV3. The perturbations are clearly visible in the images.

We can use the $\|\cdot\|_\infty$ norm to constrain the values of the universal perturbations, using also the $p = \infty$ version of DeepFool. The table below shows the results obtained when this norm is used, setting $\xi = 0.5$.

Model	Fool. rate in X (%)	Fool. rate (%)	Time (s)	Iters
ResNet-50	74.60	74.69	87	10
DenseNet-169	80.20	80.54	170	2
MobileNetV2	87.20	86.12	37	1
InceptionV3	86.70	85.71	141	1

Table 12.9: Fooling ratios on the set X and the CIFAR10 test set, average time per iteration and iterations needed to generate an universal adversarial perturbation v in each model, with $\|v\|_\infty \leq 0.5$.

The results are similar, but now we get a better fooling rate in DenseNet-169, exceeding the 80% in the test set. In contrast, the fooling rate obtained in ResNet-50 has decreased a bit, where the maximum number of iterations has been reached. In this case, we obtained slightly higher values in the fooling rates associated with the complete test set than in the set X in the ResNet-50 and DenseNet-169 architectures. The running times are lower.

Below there are some adversarial images generated with the universal perturbation obtained in DenseNet-169.

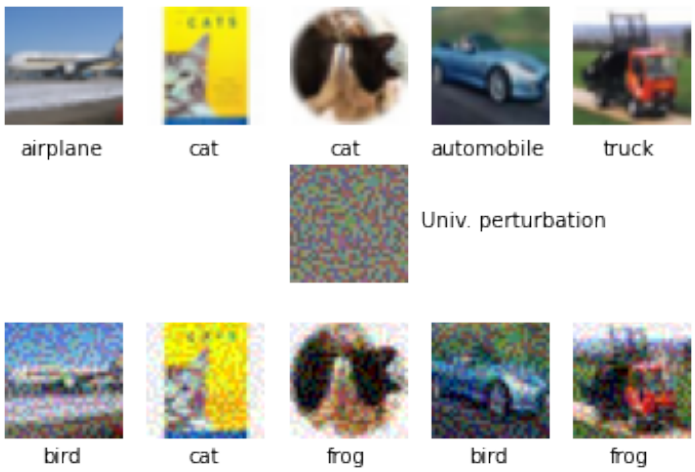

Figure 12.22: Adversarial images obtained when adding an universal perturbation to the original images at the top in DenseNet-169.

Now we focus on the perceptibility of the adversarial images. The universal perturbations obtained previously are quite visible (see Fig. 12.21 and 12.22), because we set a high value of ξ in order to obtain decent fooling rates. Let's see what happens when we reduce this value, constraining the universal perturbations to be smaller in magnitude.

The plots below show the results obtained when using different values of ξ.

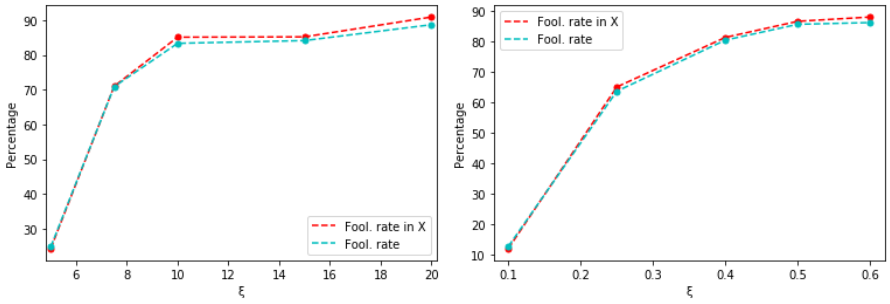

Figure 12.23: Fooling rates obtained with different values of ξ, with the universal perturbations v constrained by $\|v\|_2 \leq \xi$ (left plot) and $\|v\|_\infty \leq \xi$ (right plot).

As expected, the higher the value of ξ, the higher are the fooling rates. It can be observed that above a certain value of ξ the increase is slower, reaching levels around 90% in both cases. We can also see how the overall fooling rate stays close to the fooling rate in X.

The figures below show the universal perturbations obtained with some values of ξ, to get an idea of how visible the adversarial images will be.

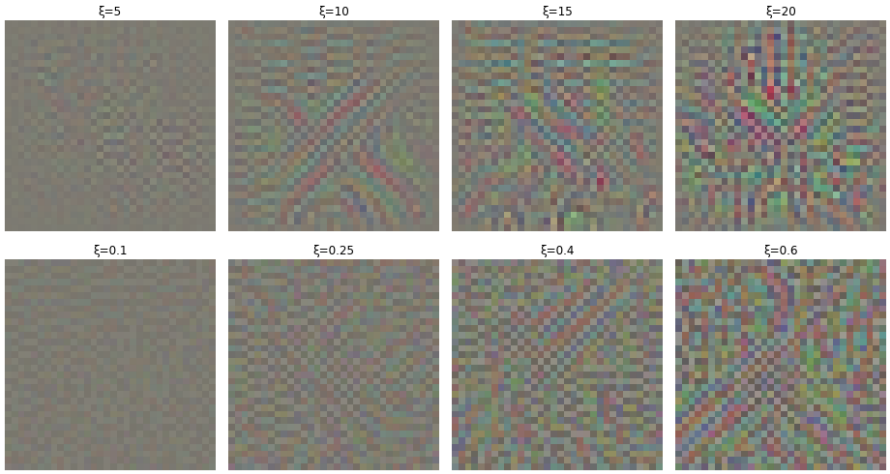

Figure 12.24: Universal adversarial perturbations generated with different values of ξ for InceptionV3, using the $\|\cdot\|_2$ norm (top images) and the $\|\cdot\|_\infty$ norm (bottom images).

In this figure we can also see how this universal perturbations are not just noise, they have some kind of structures that may be useful in fooling a specific architecture. For example, the perturbation obtained with $\xi = 10$ adds some kind of diagonal lines in the middle of the image, a structure which also appears in the perturbation obtained with $\xi = 0.6$, even when other norm is used to constraint the perturbations.

We can wonder how many images we need to include in the set X for the method to work properly. The figure below helps to answer this question: if we use a size smaller than 1000 (10% of the original), the ratios differ considerably. Generally, the lower the number of images in X, the lower the overall fooling rate, even though the fooling rate in X stays high.

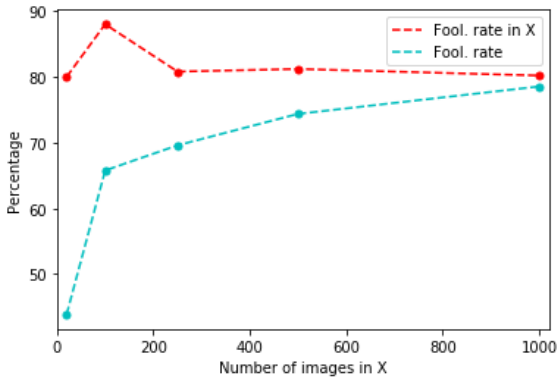

Figure 12.25: Comparison of the fooling rates obtained with sizes of the set X.

Lastly, having seen that the computed perturbations are universal across unseen data points, we now examine to which extent universal perturbations computed for a specific architecture are also valid for another one. This could be useful when we pretend to fool an unknown model. The table below summarizes the universality of such perturbations across 4 different architectures.

	ResNet-50	DenseNet-169	MobileNetV2	InceptionV3
ResNet-50	78.54	56.84	73.12	56.43
DenseNet-169	57.92	62.69	79.61	61.08
MobileNetV2	50.98	45.17	85.48	60.00
InceptionV3	48.40	45.67	79.57	84.13

Table 12.10: Generalization of the universal perturbations across different networks. The percentages indicate the fooling rates in the CIFAR10 test set. The rows indicate the architecture for which the universal perturbation is computed, and the columns the architecture for which the fooling rate is reported.

We can see that the universal perturbations obtained for a concrete architecture works well on different architectures to some extent. The model which is affected the most by perturbations obtained for other models is MobileNetV2, where the fooling rates are higher to 70%, no matter which universal perturbation we use. The universal perturbation obtained for DenseNet-169 is the one which more affects the other models, followed by the perturbation obtained for ResNet-50. The last

one in that sense is the perturbation generated for MobileNetV2. There are some pairs of models where the 50% is not reached.

We cannot show here universal perturbations obtained from ImageNet because we do not have access to the original images. To see some examples, refer to image 9.2 and [32].

12.6 OVERALL COMPARISON

To finish this part, we provide a visual comparison of some of the adversarial methods by using the same input images.

Concretely, we have used FGSM, the ℓ_2 and ℓ_∞ versions of DeepFool and SparseFool to generate adversarial examples using high resolution images from https://es.freeimages.com/. The results are shown below.

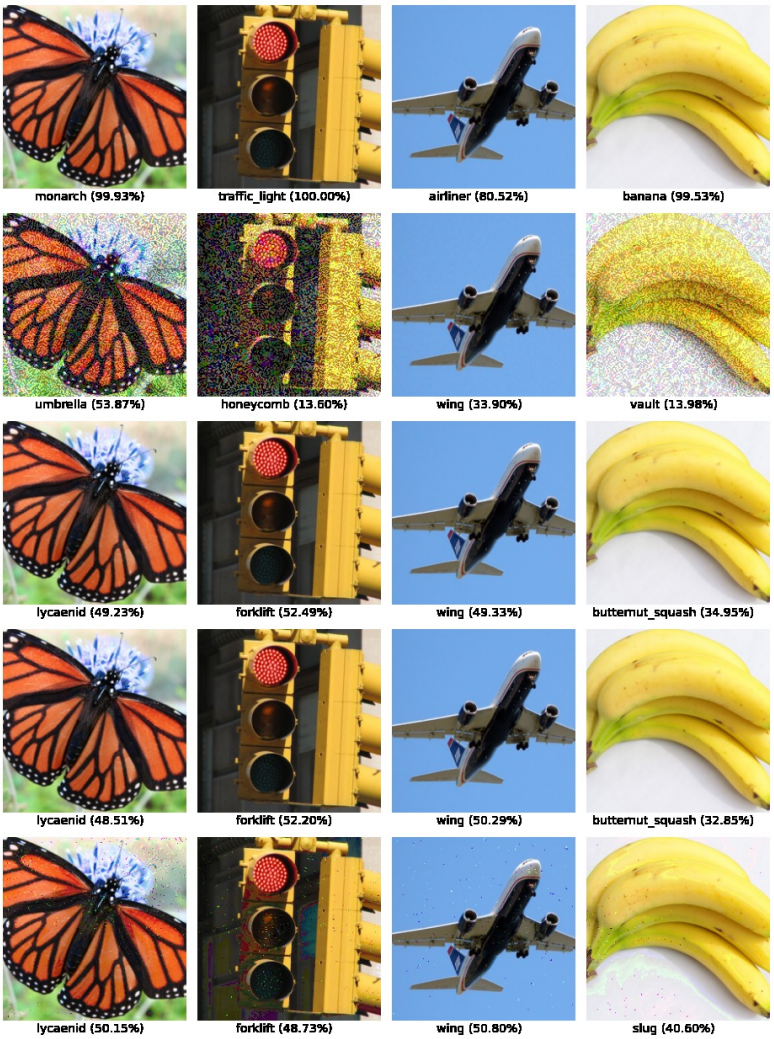

Figure 12.26: Adversarial images generated for ResNet-50 with FGSM (second row), ℓ_2-DeepFool (third), ℓ_∞-DeepFool (fourth) and SparseFool (fifth).

Part IV

CONCLUSIONS

13

A DEEPER INSIGHT INTO ADVERSARIAL EXAMPLES

Once the impact of adversarial attacks on Deep Neural Networks has been shown and analyzed, two questions remain to be answered: why are the classes of functions defined by deep neural network classifiers sensitive to these perturbations and how can we address it?

Unfortunately, this problem remains open and, although many attempts have been made to solve it, we cannot give a precise response to these questions.

This chapter provides some partial answers that may help to clarify the nature of adversarial examples. We also discuss the main methods that have been proposed up to date for designing more robust networks, ending with an interesting proposal in this aspect.

13.1 UNDERSTANDING ADVERSARIAL EXAMPLES

The following is an overview of the main explanations given for understanding the nature of adversarial examples.

13.1.1 *Spectral analysis of instability in DNNs*

Adversarial examples show that there exist small additive perturbations of the input in a DNN classifier that produce large perturbations at the output of the last layer.

If $\phi(x)$ denotes the output of a standard DNN with K layers and parameters $W = \{W_1, \dots, W_k\}$, then for any input x it holds that

$$\phi(x) = \phi_K(\phi_{K-1}(\dots \phi_1(x; W_1) \dots) W_{K-1}); W_K)$$

where ϕ_k is the operator used to map layer $k-1$ to layer k. Then, the instability of ϕ can be explained by analyzing the upper Lipschitz constant of each layer $k = 1 \dots K$, defined as the maximum $L_k > 0$ such that

$$\|\phi_k(x, W_k) - \phi_k(x + r; W_k)\| \leq L_k \|r\| \quad \forall x, r$$

In this way, if $L = \prod_{k=1}^{K} L_k$, the following inequality holds:

$$\|\phi(x) - \phi(x + r)\| \leq L \|r\| \tag{13.1}$$

Szegedy et al. obtained in [46] some upper bounds for the most common types of layers, showing that instabilities can appear as soon

as in the first convolutional layer. However, this result does not justify the existence of adversarial examples, as it provides upper bounds for the instability of a model, but points out that penalizing each upper Lipschitz bound might help to improve its robustness.

13.1.2 *False structures*

L. Thesing et al. pointed out [50] that current deep learning models do not learn the original structures that humans use when recognising images, and that is why they are vulnerable to adversarial attacks.

Definition 13.1. Consider $\mathcal{M} \subset \mathbb{R}^d$ and a string of unique predicates $L = \{\alpha_0, \dots, \alpha_N\}$ on $\mathcal{M}$ such that $\forall x \in \mathcal{M},\ \exists!\ \alpha_j \in L$ with $\alpha_j(x) = \textit{true}$. For such x define $f(x) = j$. The pair (f, L) is the *original structure* on $\mathcal{M}$. A *false structure* for (f, L) relative to $\mathcal{T} \subsetneq \mathcal{M}$ is a pair (g, L'), where $L' = \{\beta_0, \dots, \beta_N\} \neq L$ is a string of unique predicates and $g : \mathcal{M} \to \{0, \dots, N\}$ such that $g(x) = j \iff \beta_j(x) = \textit{true}$. Moreover,

$$g(x) = j \iff f(x) = j\ \forall x \in \mathcal{T}$$

as well as

$$\mathcal{C} = \{x \in \mathcal{M} \setminus \mathcal{T} : f(x) = i, g(x) = j, \text{ for some } i \neq j\} \neq \emptyset$$

Example 13.1. Consider the problem of recognising a car in an image x. Let $L = \{\alpha_0, \alpha_1\}$ be a set of predicates where $\alpha_0(x) =$ 'x demonstrates a car' and $\alpha_1(x) =$ 'x does not demonstrate a car'. Thus, the pair (f, L), with f defined as explained above, is the original structure we want the network to learn. Let's suppose that all the images in the training set that contain a car also have a horizontal line in the lower right corner and that images that are not of a car do not have this line. The network may learn the false structure given by (g, L'), where $L' = \{\beta_0, \beta_1\}$ with $\beta_0(x) =$ 'x demonstrates a horizontal line' and $\beta_1(x) =$ 'x does not demonstrate a horizontal line', and thus have high accuracy on the training set without solving the original problem.

In that way, the authors in [50] conjecture that the current learning process in deep learning classifiers forces neural networks to learn a false structure which correlates well with the original structure but is unstable, and thus the susceptibility to adversarial attacks. The false structure is simpler than the original one, and hence easier to learn.

This could not be proven mathematically, but there are some facts that support the conjecture:

- Neural networks classify unrecognizable and bizarre images as natural images with high confidence [34]. If neural networks actually capture the structures that allows for image recognition in the human brain, this behaviour is not possible.

- The One Pixel Attack shows that perturbing only one pixel of an image can change the label of classification. It is clear that this cannot happen if the model learns the original structures.
- Universal adversarial perturbations demonstrates how a single and almost invisible perturbation changes the label of a considerably set of images. Different structures in images, allowing for successful human classification, could not be susceptible to misclassification by a single perturbation.

The correctness of the proposed conjecture may have several negative consequences, some of which are that the success of deep learning is dependent of simple yet unstable structures, which we need to avoid in order to end with the adversarial examples problem, and that it becomes hard to point out what neural networks are actually learning, making it difficult to trust its predictions.

On the other hand, false structures may contain simpler information that cannot be captured by humans and can be useful to classify images in a easier way.

13.1.3 *Shortcut learning*

The authors in [11] propose that some of the current problems in deep learning (including adversarial examples) are symptoms of the same underlying problem: *shortcut learning*.

Shortcut learning reveals itself by a strong discrepancy between intended and actual learning strategy, which usually fails under slightly different circumstances, as in adversarial examples. This behaviour occurs when deep neural networks solve problems by taking shortcuts instead of learning the intended solution, leading to a worse capacity of generalization and undesired failures. In this way, false structures can be viewed as a consequence of shortcut learning.

Among all possible decision rules implemented by a machine learning model, different groups of solutions can be distinguished.

- *Training solutions*: the samples used to train and test the model are supposed to be independent and identically distributed (i.i.d.). Therefore, in order to obtain high performance on the test set, the model needs to learn a function that is approximately correct within a subset of the input domain which covers a great part of the input distribution. If the selected function successfully classify the training images, but not the i.i.d. test images, the model is said to be using *overfitting features* (see Fig. 13.1).
- *Test solutions*: decision rules that solve both the training and the test set usually obtain high scores on standard benchmarks. However, a model can learn very different functions that solve the same problem if the test are performed only on i.i.d. data, and it

is impossible to distinguish between these solutions. Thus, one can instead consider test sets that are systematically different from the i.i.d. samples, using *out-of-distribution* or o.d.d data. In this sense, if a feature helps to perform well on i.i.d test data but not in o.o.d. generalization tests, it is considered a *shortcut feature* (Fig. 13.1).

- *Intended solution*: decision rules that use the *intended features* work well on i.i.d test sets and also on o.o.d. tests, where shortcut solutions fail. Humans have a strong intuition for what the intended solution should be, but for complex problems like image classification, intended solutions are impossible to formalize.

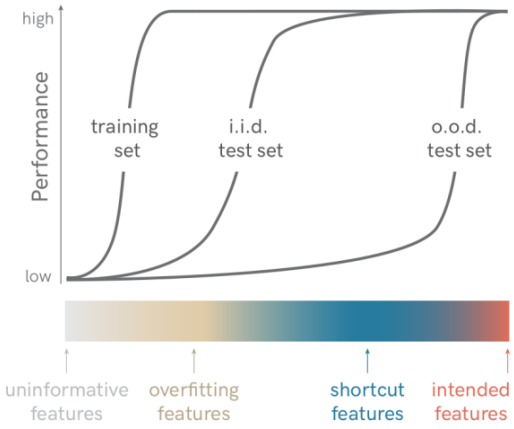

Figure 13.1: Taxonomy of decision rules [11]. Within the set of all possible rules that can be used in a specific problem, only some solve the training data and only some of them generalise to an i.i.d. test set. Among those solutions, shortcuts fail to generalise to different data (o.o.d. test sets), but the intended solution does generalise.

Adversarial attacks can change model predictions without changing semantic content, which can be an indication that something akin to shortcut learning is occurring. Therefore, adversarial attacks can be seen as testing the model on a specific worst-case o.o.d. data, where the shortcut features learned by the model fails. These shortcut features are present in the data set as the existence of different possibilities to solve the problem, since the raw data alone rarely constrains a model sufficiently.

The definition of an object relies on a potentially highly non-linear combination of information from different sources or attributes that influence a decision rule. Humans usually use interpretable object attributes like shape or texture to recognize an image, but standard DNNs do not impose any human-interpretability requirements on intermediate features extracted from an image and thus might be severely biased to consider simpler features which only generalise under the specific design of the dataset used to train the model, but easily fail otherwise. However, this generalisation failure is neither a failure to learn nor a

failure to generalise at all, but just a failure to generalise in the intended direction.

13.2 PREVENTING ADVERSARIAL ATTACKS

Extensive research has been done trying to solve the instability issue of current DNNs, but the problem is still open. Here we show some methods that have been applied for alleviating or even remove the effect of adversarial images in some cases.

13.2.1 *Natural transformations in real-world scenarios*

The first approach is based on considering the normal image acquisition process that takes place when acquiring images in a real-world scenario (e.g. a camera in an autonomous car or a scanner to recognize hand-written digits). This process involves some slight transformations, such as shifting or blurring, or even some perturbations, such as noise, that usually are enough to negate the effect of the carefully crafted perturbations used in adversarial images.

Concretely, the authors in [13] use a combination of translation, noise, blur, crop and resize transformations to fully synthetically capture this process. The results demonstrate that the image acquisition process allows the model to correctly classify a large portion of the original adversarial examples, with the most effective transformation being the cropping and subsequent resizing of input images. For the MNIST database of handwritten digits, they also used binarization (pixels are restricted to have the value 0 or 1), resulting in a very high accuracy in adversarial images.

However, only the FGSM and a variation know as the *Fast Gradient Value* method where used to generate the adversarial examples in the experiments, so we can not guarantee that this process will be valid in images generated by another adversarial method using different types of perturbations, like SparseFool. In addition, this research does not take into account the scenarios where physical acquisition of the image is not needed.

13.2.2 *Adversarial training*

Adversarial training has been used to address the instability issues in some cases, by training a model on a mixture of adversarial and clean samples (conforming and o.o.d. training set) [28]. Szegedy et al. also found [12] that training with a modified objective function based on the fast gradient sign method was an effective regularizer:

$$\mathcal{L}(x, y) = \alpha\mathcal{L}(x, y) + (1 - \alpha)\mathcal{L}(x + \epsilon\text{sign}(\nabla_x\mathcal{L}(x, y), y)$$

However, it became evident later [30, 32] that this method was not effective in many cases, where the fine-tuned networks remain largely vulnerable to adversarial examples.

13.2.3 *Defensive quantization*

Neural network quantization [14] consist in using a low-bit representation of the network. This technique is commonly used to reduce the computation and memory costs of deep neural networks, so they can be efficiently deployed on hardware platforms, while supposing only a little degradation of the original accuracy.

However, common quantization approaches results in quantized models being more vulnerable to adversarial attacks. Although it may seem counter intuitive (small perturbations should be denoised with low-bit representations), this behaviour occurs because the adversarial perturbation is amplified significantly when passing through intermediate layers of the network, which pushes some values into a different quantization bucket.

To mitigate the error amplification effect, the authors in [26] propose a *Defensive Quantization* (DQ) method that controls the Lipschitz constant of the network (13.1) during quantization, making the magnitude of the adversarial perturbation to remain non-expansive, while maintaining the efficiency.

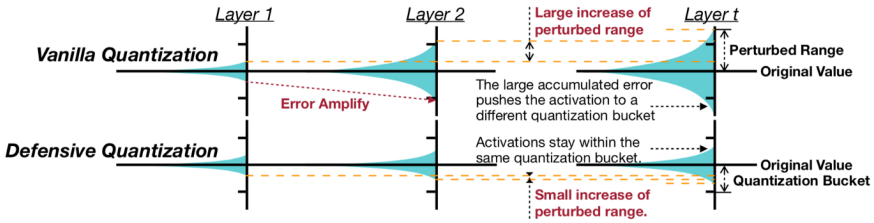

Figure 13.2: The error amplification effect that is produced in common quantization approachs vanishes in the DQ method [26].

To do so, they added a regularization term in the original objective function, which pretends to make every weight matrix be orthogonal, resulting in non-expansive networks [26]:

$$\mathcal{L}'(W) = \mathcal{L}(W) + \frac{\beta}{2} \sum_i \left\| W_i^\top W_i - I \right\|^2$$

The results obtained showed that, if perturbation at each layer is kept within a certain range, the adversarial noise will not propagate, leading also to increase the robustness of the original model.

Again, only gradient-based methods (like FGSM) where used to generate the adversarial perturbations where DQ was tested.

14

CONCLUSION AND FUTURE DIRECTIONS

In last years, the huge success of deep learning has probably overshadowed the need for a thorough understanding of the behaviour of Deep Neural Networks. Overcoming this lack of understanding is no longer a purely scientific endeavour anymore, but has become an urgent necessity due to the growing social impact of deep learning applications. If we are really going to trust such algorithms in the future to perform critical tasks, whether in autonomous vehicles or in medical check-ups, among other applications, then we really need to understand thoroughly: How does deep learning work? When does it fail, and why?

The extraordinary potential of deep neural networks is used to classify images by using the raw data contained in its pixels, which gives the model an overwhelming amount of information. As we have already discussed, this opens the door for models to learn very different functions from those initially intended, giving rise to the aforementioned problems of instability.

Therefore, solving the instability problem associated with adversarial examples will not be an easy task until we have a more detailed understanding of what goes on internally in these types of classifiers.

The strategies proposed in the previous chapter fails because they are not adaptive: one of them may block one kind of attack, but it leaves another vulnerability open to an attacker who knows about the defense being used. Designing a technique that can protect against a powerful, adaptive attacker is an important research area, in which interesting contributions have been made. For example, L. Schott et al. report [40] that they have developed a novel robust classification model, which is robust against ℓ_0, ℓ_2 and ℓ_∞-norm adversarial perturbations on the MNIST dataset.

An interesting proposal when addressing this problem would be to consider models that extract solid information from the image in the intermediate layers, such as contours and textures. Everything seems to indicate [10] that, although both humans and DNNs reach high accuracies in contour integration, DNNs perform this task in very different ways from humans. If this type of information, in which great advances have been made [20, 25, 27] were integrated into current models, we will probably solve the above-mentioned problem in the near future, since the information given by this type of features is robust against adversarial examples. Even so, this task requires an effective method to extract this type of information in the intermediate layers

of neural networks and this is a domain where there is still a lot of research to be done.

Part V

APPENDIX

15

DEVELOPED SOFTWARE

To perform the experiments in Part III, some software tools have been developed, which we are going to briefly describe now. The code is presented in four files:

- The file `adversarial_attacks.py` contains the code necessary to implement all adversarial attacks that have been used during this work, providing an useful API to test these methods in several conditions. Two additional useful functions are also provided: one can be used to test any method on a CIFAR10 image (`test_method`), and the other to perform a complete or partial attack on the CIFAR10 test set, showing the results obtained (`attack_model`). The code is commented and structured so that anyone can adapt it to their necessities
- The file `utils.py` implements all the functions required by the adversarial methods in the previous file. It also contains useful functions to work with images.
- The file `cifar10_experiments.ipynb` is a notebook document used to test the adversarial methods on the CIFAR10 test set. It offers an easy-to-follow guide, detailing the steps required to perform adversarial attacks in any model and with any method.
- The file `ImageNet_experiments.ipynb` is another notebook that is used to test the methods in high resolution images, using models pretrained on ImageNet. It can be used to obtain adversarial examples for any image extracted from the internet.

The previous files are provided with this document, but are also available in the public repository https://github.com/pabloac31/TFG.

BIBLIOGRAPHY

[1] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012. ISBN: 1600490069.

[2] Y. Bengio. "Learning Deep Architectures for AI." In: *Foundations* 2 (Jan. 2009), pp. 1–55. DOI: 10.1561/2200000006.

[3] Battista Biggio et al. "Evasion Attacks against Machine Learning at Test Time." In: *Lecture Notes in Computer Science* (2013), 387–402. ISSN: 1611-3349. DOI: 10.1007/978-3-642-40994-3_25. URL: http://dx.doi.org/10.1007/978-3-642-40994-3_25.

[4] Richard H. Byrd et al. "A limited memory algorithm for bound constrained optimization." English. In: *SIAM Journal of Scientific Computing* 16 (Sept. 1995), pp. 1190–1208. ISSN: 1064-8275. DOI: 10.1137/0916069.

[5] Emmanuel Candes and Terence Tao. *Decoding by Linear Programming*. 2005. arXiv: math/0502327 [math.MG].

[6] Li Deng and Dong Yu. *Deep Learning: Methods and Applications*. Tech. rep. MSR-TR-2014-21. Microsoft, 2014. URL: https://www.microsoft.com/en-us/research/publication/deep-learning-methods-and-applications/.

[7] D. L. Donoho. "Compressed sensing." In: *IEEE Transactions on Information Theory* 52.4 (2006), pp. 1289–1306. ISSN: 1557-9654. DOI: 10.1109/TIT.2006.871582.

[8] A. Fawzi et al. "Empirical Study of the Topology and Geometry of Deep Networks." In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 3762–3770. DOI: 10.1109/CVPR.2018.00396.

[9] Alhussein Fawzi, Seyed-Mohsen Moosavi-Dezfooli, and Pascal Frossard. "The Robustness of Deep Networks: A Geometrical Perspective." In: *IEEE Signal Processing Magazine* 34 (Nov. 2017), pp. 50–62. DOI: 10.1109/MSP.2017.2740965.

[10] Christina M. Funke et al. *The Notorious Difficulty of Comparing Human and Machine Perception*. 2020. arXiv: 2004.09406 [cs.CV].

[11] Robert Geirhos et al. *Shortcut Learning in Deep Neural Networks*. 2020. arXiv: 2004.07780 [cs.CV].

[12] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2014. arXiv: 1412.6572 [stat.ML].

[13] Abigail Graese, Andras Rozsa, and Terrance E. Boult. "Assessing Threat of Adversarial Examples on Deep Neural Networks." In: *CoRR* abs/1610.04256 (2016). arXiv: 1610.04256. URL: http://arxiv.org/abs/1610.04256.

[14] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2015. arXiv: 1510.00149 [cs.CV].

[15] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[16] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[17] Gao Huang et al. *Densely Connected Convolutional Networks*. 2016. arXiv: 1608.06993 [cs.CV].

[18] Dana Hughes and Nikolaus Correll. "Distributed Machine Learning in Materials that Couple Sensing, Actuation, Computation and Communication." In: (June 2016).

[19] Saumya Jetley, Nicholas A. Lord, and Philip H. S. Torr. *With Friends Like These, Who Needs Adversaries?* 2018. arXiv: 1807.04200 [cs.CV].

[20] André Peter Kelm, Vijesh Soorya Rao, and Udo Zölzer. "Object Contour and Edge Detection with RefineContourNet." In: *Lecture Notes in Computer Science* (2019), 246–258. ISSN: 1611-3349. DOI: 10.1007/978-3-030-29888-3_20. URL: http://dx.doi.org/10.1007/978-3-030-29888-3_20.

[21] Anastasis Kratsios. *The Universal Approximation Property: Characterizations, Existence, and a Canonical Topology for Deep-Learning*. 2019. arXiv: 1910.03344 [stat.ML].

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[23] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. "Adversarial examples in the physical world." In: *CoRR* abs/1607.02533 (2016). arXiv: 1607.02533. URL: http://arxiv.org/abs/1607.02533.

[24] Y. LeCun et al. "Gradient-Based Learning Applied to Document Recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[25] Mengtian Li et al. *Photo-Sketching: Inferring Contour Drawings from Images*. 2019. arXiv: 1901.00542 [cs.CV].

[26] Ji Lin, Chuang Gan, and Song Han. "Defensive Quantization: When Efficiency Meets Robustness." In: *CoRR* abs/1904.08444 (2019). arXiv: 1904.08444. URL: http://arxiv.org/abs/1904.08444.

[27] Li Liu et al. *From BoW to CNN: Two Decades of Texture Representation for Texture Classification*. 2018. arXiv: 1801.10324 [cs.CV].

[28] Aleksander Madry et al. "Towards Deep Learning Models Resistant to Adversarial Attacks." In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=rJzIBfZAb.

[29] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[30] Apostolos Modas, Seyed-Mohsen Moosavi-Dezfooli, and Pascal Frossard. *SparseFool: a few pixels make a big difference*. 2018. arXiv: 1811.02248 [cs.CV].

[31] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. *DeepFool: a simple and accurate method to fool deep neural networks*. 2015. arXiv: 1511.04599 [cs.LG].

[32] Seyed-Mohsen Moosavi-Dezfooli et al. *Universal adversarial perturbations*. 2016. arXiv: 1610.08401 [cs.CV].

[33] B. K. Natarajan. "Sparse Approximate Solutions to Linear Systems." In: *SIAM Journal on Computing* 24.2 (1995), pp. 227–234. DOI: 10.1137/S0097539792240406. eprint: https://doi.org/10.1137/S0097539792240406. URL: https://doi.org/10.1137/S0097539792240406.

[34] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. "Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images." In: *CoRR* abs/1412.1897 (2014). arXiv: 1412.1897. URL: http://arxiv.org/abs/1412.1897.

[35] Nicolas Papernot et al. "Practical Black-Box Attacks against Deep Learning Systems using Adversarial Examples." In: *CoRR* abs/1602.02697 (2016). arXiv: 1602.02697. URL: http://arxiv.org/abs/1602.02697.

[36] Pranav Rajpurkar et al. "CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning." In: *CoRR* abs/1711.05225 (2017). arXiv: 1711.05225. URL: http://arxiv.org/abs/1711.05225.

[37] *Review: Inception-v3 — 1st Runner Up (Image Classification) in ILSVRC 2015*. https://medium.com/@sh.tsang/review-inception-v3-1st-runner-up-image-classification-in-ilsvrc-2015-17915421f77c. Accessed: 2020-05-28.

[38] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: 1609.04747 [cs.LG].

[39] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks." In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 4510–4520.

[40] Lukas Schott et al. "Robust Perception through Analysis by Synthesis." In: *CoRR* abs/1805.09190 (2018). arXiv: 1805.09190. URL: http://arxiv.org/abs/1805.09190.

[41] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: 1409.1556 [cs.CV].

[42] Jost Tobias Springenberg et al. *Striving for Simplicity: The All Convolutional Net*. 2014. arXiv: 1412.6806 [cs.LG].

[43] Johannes Stallkamp et al. "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition." In: *Neural networks : the official journal of the International Neural Network Society* 32 (Feb. 2012), pp. 323–32. DOI: 10.1016/j.neunet.2012.02.016.

[44] Rainer Storn and Kenneth Price. "Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization Over Continuous Spaces." In: *Journal of Global Optimization* 23 (Jan. 1995).

[45] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. "One Pixel Attack for Fooling Deep Neural Networks." In: *IEEE Transactions on Evolutionary Computation* 23.5 (2019), 828–841. ISSN: 1941-0026. DOI: 10.1109/tevc.2019.2890858. URL: http://dx.doi.org/10.1109/TEVC.2019.2890858.

[46] Christian Szegedy et al. "Going Deeper with Convolutions." In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[47] Christian Szegedy et al. "Intriguing properties of neural networks." In: *International Conference on Learning Representations*. 2014. URL: http://arxiv.org/abs/1312.6199.

[48] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision." In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 2818–2826.

[49] Yaniv Taigman et al. "DeepFace: Closing the Gap to Human-Level Performance in Face Verification." In: Sept. 2014. DOI: 10.1109/CVPR.2014.220.

[50] Laura Thesing, Vegard Antun, and Anders C. Hansen. *What do AI algorithms actually learn? - On false structures in deep learning*. 2019. arXiv: 1906.01478 [stat.ML].