Treball final de grau

GRAU DE MATEMÀTIQUES

Facultat de Matemàtiques i Informtica
Universitat de Barcelona

Attribution Methods for Deep Convolutional Networks

Autor: Guillem Brasó

Director: Dr. Jordi Vitrià
Realitzat a: Departament
de Matemàtiques i Informàtica

Barcelona, 27 de juny de 2018

Abstract:
In recent years, *Deep Learning* has shown great success across several areas. However, even, though it might provide remarkable accuracy for many tasks, its application in some fields faces a fundamental problem: its predictions are not *interpretable*. Attribution Methods offer a possible solution in regards to this problem. To do so, they resource to results in Game Theory in order to *explain* individual decisions made by Deep Learning algorithms. In this work, we will be focusing, specifically, on the application of Attribution Techniques to a subset of Deep Learning algorithms: Convolutional Neural Networks.

Acknowledgements:
I would like to thank my advisor, Jordi, for all the valuable insights, discussion and general mentorship that he has provided me.

I would also like to thank my parents for their general support through this journey.

Introduction

Motivation

Deep Learning can be roughly described as a set of computational techniques that are loosely based on the *biological brain* and are used in order to *approximate* functions from observational data. In recent years Deep Learning has emerged as an overwhelmingly successful field, and it currently is one of the main foundations behind disruptive technologies such as *self-driving cars* or *virtual assistants*.

However, in spite of the fact that functions approximated by Deep Learning can achieve high accuracy, they are regarded as *black-boxes*. That is, in general, we humans do not understand *what reasons* drive the relationships between their input and output variables.

This tension between *interpretability* and *accuracy* is of increasing importance as applications of Deep Learning continue to expand accross fields such as Health Care where, arguably, the *reasoning* behind facts are just as important, if not more, than *facts themselves*.

In recent years, several approaches have been developed with the goal of tackling this problem. *Attribution Methods* are one of them. Given an input point $x \in \mathbb{R}^n$ and function $f : \mathbb{R}^n \to \mathbb{R}$, the goal of Attribution methods is to determine *how much influence* does each component of x have in the output value $f(x)$. Thus they are often described as *local explanation* methods.

Attribution methods are, a *young* field of study in which some of its most fundamental questions have not been answered yet. Namely, it is still not clear how *influence* should be defined in this context, nor how the performance of an attribution method can be measured. However, several answers for these questions have already been proposed by the *research* community, and in this work we will be addressing them.

Main Objectives

Our first goal with this project is to obtain all the preliminary knowledge that is necessary in order to understand current research in Deep Learning and Attribution Methods.

Our next goal will be to review the main lines of work that have already been proposed. We will not only attempt to study them separately but, instead, understand their connections and present them in a way where these are highlighted.

Then, our next objective will be to understand the main challenges of applying these techniques to a subset of the applications of Deep Learning: Computer Vision. This, together with the previous one, have arguably been our main goals with this project.

Finally, our last goal will be to familiarize ourselves with some of the main deep learning programming frameworks that are used nowadays. Then, use this knowledge to implement several attribution methods and apply them to real images.

Organization of this work

This work is organized in three main chapters. In the first one, we provide the basic context for all upcoming work. To do so, we start reviewing the basics of Machine Learning and we progressively restrict our area of focus until reaching *Convolutional Neural Networks*, which are the subset of Deep Learning in which we will be focusing.

The second chapter is devoted to doing an extensive review of the main two approaches that have arised in regards to applying Attribution methods to Deep Learning and, specifically, Computer Vision. This can be considered as the main chapter of our work.

Lastly, in the third chapter we explain some of the experiments that we have conducted during the development of this project, and we provide some practical ideas on the insights that we have obtained from them.

Contents

Chapter 1

Preliminaries

1.1 Machine Learning

In this section, we aim to provide a brief overview of some of the main concepts involved in Machine Learning and, more specifically, Supervised Learning. We are aware that is a broad disciple that cannot be covered in a few pages. Thus, our goal will be to simply outline some of its most basic topics in order to build a theoretical context for all upcoming work.

We will start introducing some basic definitions and notation on the basic setup of learning algorithms. Then, we will proceed to explore some simple examples and, finally, we will briefly cover the basic optimization concepts that we will be using in the rest of this work.

The Learning Problem

Machine Learning broadly consists in allowing computer to *learn patterns from data* by itself, rather than following *human-encoded rules*. However, what do we mean exactly by that?. In [7] the authors propose the following definition:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E.

Our objective with the rest of this section will be to further specify the free variables of this definitions: T, E, P, for our problem in hand.

Supervised Learning

Supervised learning is one of the three main types of learning problems in which Machine Learning is divided [1], and it will be the task T, that we will focus our work on.

Its setup is the following: consider a set $\{(X^{(1)}, Y^{(1)}), \ldots, (X^{(n)}, Y^{(n)})\}$ of independent copies of a random variable (X, Y), whose values are in $(\mathcal{X}, \mathcal{Y})$. In general, $\mathcal{X} = \mathbb{R}^m$ and, with $\mathcal{Y}$, we will make a distinction between two possibilities:

- If $\mathcal{Y} = \mathbb{R}$, then we will refer to our task as *regression*.
- If, instead, $\mathcal{Y}$ is a discrete set S, then we will refer to our task as *classification*, and we will refer to each element in S as a *class*.

In both cases, the task itself will consist in finding a mapping:

$$f : \mathcal{X} \to \mathcal{Y}$$

such that for each sample observation $(X^{(i)}, Y^{(i)})$ drawn from (X, Y), $f(X^{(i)}) = Y^{(i)}$.

However, in general, it is assumed that it is not possible to find an exact mapping f and, instead, the goal is to find find f such that $f(X) = Y + \epsilon$. The term ϵ is another random variable that accounts for several *factors not considered in the observations*, such as errors in the measurements and non-considered relevant variables.

In practice, the goal of *learning* f is to be able to *predict* the value of Y, when only X can be observed. As an example, instances of X might be pictures and Y might consist in a binary value indicating whether the corresponding image contains a cat or not. If our machine is able to build a *reliable* mapping f, it will have learned how to recognize cats.

We will generally refer to each component X_i of X as an *input feature* and, to Y, as the *target variable*. In our previous example, *pixels* are *input features*, and our target variable is a binary variable indicating whether each image is a cat or not. Typically, f is referred to as the *hypothesis*.

We will denote by $\mathcal{D} := \{(X^{(1)}, Y^{(n)}), \ldots, (X^{(n)}, Y^{(n)})\}$, the set of observed instances of (X, Y), and refer to it as *dataset*. We sill also denote by $\mathbb{X}$ the *design matrix*, consisting in a matrix in $\mathbb{R}^{n \times m}$ whose jth element in its ith row is given by $\mathbb{X}_{i,j} = X_i^{(j)}$. Observe that the superindex (j) in X runs through observations and, the subindex i, through components or *input features*.

We now turn to the problem of how to construct f with $\mathcal{D}$.

[1] The other two are Unsupervised Learning and Reinforcement Learning. Roughly speaking, these consist in, respectively, developing algorithms that are able to find structure in *raw data* and building algorithms that allow machines how to learn from *interactions* with the world.

Empirical Risk Minimization

The first step towards obtaining a mapping f consist in restricting our search of f to a set $\mathcal{H}$ of parametric functions, which we will refer to as the *hypothesis set*. An example for $\mathcal{H}$ could be $\{\theta^t x + \theta_0 : \theta \in \mathbb{R}^m, \theta_0 \in \mathbb{R}\}$, i.e., the set of linear funtions in $\mathbb{R}^m$.

Hypothesis sets are chosen by the Machine Learning practitioner, according to the task in hand. Different hypothesis sets are suited for different tasks. For instance, in the following sections we will restrict our study to those hypothesis sets suited for *computer vision*. The choice itself, cannot be considered part of *learning* process, as it is done by a human.

The learning process itself consists, instead, in determining the right parameters for the hypothesys f and the dataset $\mathcal{D}$. Given a choice of $\mathcal{H}$ parametrized by θ, let us now refer to our function as f_θ. How should θ be determined?

A natural approach is to aim for $f(X^{(i)})$ to be *close* to $Y^{(i)}$, for each $(X^{(i)}, Y^{(i)} \in \mathcal{D}$. In order to do so, we define a *loss function* $L(f_\theta, X, Y)$. Loss functions are generally only asked to be real-valued and non-negative, and are generally monotone functions of the distance*This is however, not always the case, and loss functions might not satisfy all properties required to be a distance.* between $f(x)$ and y, where (x, y) are instances of (X, Y). Loss functions are also defined depending on the task in hand. For regression, the usual choice is:

$$L(f_\theta, x, y) := (f(x) - y)^2$$

Which is known as the *ordinary least squares* or *quadratic loss* function. A natural choice for classification problems is the *misclassification error*:

$$L(f_\theta, x, y) = \mathbb{1}_{(f(x)=y)}$$

The notion of *loss function* allows us to now introduce the concept of *risk*.

Definition 1.1. *Given a random variable (X, Y), a hypothesis f, and a loss function L, the risk of f with respect to L, is defined as:*

$$R(f, L) := \mathbb{E}_{X,Y}(L(f, X, Y))$$

The ultimate goal when learning f would be to chose θ so that:

$$\theta := \underset{\theta : f_\theta \in \mathcal{H}}{\operatorname{argmin}} \; R(f_\theta, L)$$

However, in practice, computing $R(f)$ requires knowing the distribution $P(X, Y)$ and estimating this distribution might be an intractable task. Recall that X takes values in $\mathcal{X} = \mathbb{R}^n$. In general, n might be in the order of 10^6 or even greater. Think, for instance, in the number of pixels in a RGB image of resolution 1024×1024. That is, in fact, a key difference between Machine Learning and classical Statistics: the lack of assumptions, in general, about the distribution of (X, Y).

Since the real risk cannot be computed, in general, by estimating $P(X, Y)$, the general approach will be to consider, instead:

$$R_{emp}(f, L) := \frac{1}{|\mathcal{D}|} \sum_{(X^{(i)}, Y^{(i)}) \in \mathcal{D}} L(f_\theta, X^{(i)}, Y^{(i)})$$

which we refer to as the *empirical risk*. Then, the goal becomes to find θ such that:

$$\theta := \underset{\theta : f_\theta \in \mathcal{H}}{\operatorname{argmin}} \; R_{emp}(f_\theta, L)$$

The principle consisting in making this choice for the *hypothesis* is called *empirical risk minimization*. The theoretical result underlying this principle is the Law of Large Numbers which guarantees, in our case, that the empirical risk R_{emp} (i.e. sample average of L), will almost surely converge to the actual risk R as the sample size $n \to \infty$.
Returning to our initial definition, the measure P with which we will be evaluating our task T, will be *empirical risk*.

A natural question is whether *empirical risk minimization* is related to the classical paradigm in frequentist statistics: *maximum likelihood estimation*. In regression problems, where L is chosen as the ordinary least squares function we introduced above, the following proposition provides an answer.

Proposition 1.2. *Let $\{(X^{(1)}, Y^{(1)}, \epsilon^{(1)}), \ldots, (X^{(n)}, Y^{(n)}, \epsilon^{(n)})\}$ be a set of real valued i.i.d. copies of a random variable (X, Y, ϵ), $\theta \in \mathbb{R}^m$ and f_θ a hypothesis such that, for every i, $f_\theta(X^{(i)}) = Y^{(i)} + \epsilon^{(i)}$. If we assume that $\epsilon^{(i)}$ are i.i.d copies of a random variable $\epsilon \sim \mathcal{N}(0, \sigma^2)$ for some $\sigma^2 \in \mathbb{R}^+$, then, estimating θ with maximum likelihood estimation is equivalent to doing it through empirical risk minimization with the ordinary least squares loss function.*

Proof. Observe that, the assumption:

$$\epsilon \sim \mathcal{N}(0, \sigma^2)$$

implies that:

$$Y | X; \theta \sim \mathcal{N}(f_\theta(X), \sigma^2)$$

Where we denote with '$;\theta$' the parametrization by θ [2]. Now, since we are assuming that the copies of (X, Y) are independent, the likelihood function can be written as:

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p(Y^{(i)} | X^{(i)}; \theta) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(- \frac{(Y^{(i)} - f_\theta(X^{(i)}))^2}{2\sigma^2} \right) \tag{1.1}$$

Now, if we consider the log-likelihood of θ, we obtain:

[2] We do not consider it as a given variable, since θ is not a random variable but a given parameter.

$$\begin{aligned}\log(\mathcal{L}(\theta)) &= \log\left(\prod_{i=1}^{n}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(Y^{(i)}-f_\theta(X^{(i)}))^2}{2\sigma^2}\right)\right)\\ &= \sum_{i=1}^{n}\log\left(\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(Y^{(i)}-f_\theta(X^{(i)}))^2}{2\sigma^2}\right)\right)\\ &= n\,\log\frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(Y^{(i)}-f_\theta(X^{(i)}))^2\end{aligned}$$

Therefore, maximizing the log-likelihood, is equivalent to minimizing:

$$\sum_{i=1}^{n}(Y^{(i)}-f_\theta(X^{(i)}))^2$$

which is exactly the expression corresponding to the empirical risk of f_θ with the *ordinary least squares* loss function.

□

In Machine Learning, however, several others loss functions might be used and, as we said, assumptions about the distributions of the random variables involved in the learning problem might be avoided. Thus, there might not even be a defined *likelihood* function associated to the learning problem and, as a consequence, *maximum likelihood estimation* might not always be an available option.

In the classification problems we will be introducing in the upcoming sections, however, we will often be making probabilistic assumptions on the distribution $P(Y|X)$, as this will allow us to make use of differentiable loss functions, which we will be able to minimize with standard techniques. In order to clarify how these assumptions might be made in practice, we now introduce one of the most basic classification algorithms: *logistic regression*[3]. As we will see in the next section, logistic regression will constitute the basic building block of more sophisticated algorithms.

Example 1.3. *(Logistic regression)* Let $\mathcal{D}$ be a *dataset* consisting of n i.i.d copies of a random variable (X, Y). Let the set of possible values for Y, $\mathcal{Y}$, be $\{0, 1\}$ and $\mathcal{X} = \mathbb{R}^m$.

Our goal with logistic regression will be to define a function $g(x)$ that we will interpret $P(Y = 1|X = x)$. This will lead to a natural definition the mapping we are actually interested on, $f : \mathcal{X} \to \mathcal{Y}$ as:

$$f(x) = \begin{cases}1 \text{ if } g(x) > 0.5\\ 0 \text{ otherwise}\end{cases}$$

Doing so, allows us to avoid encountering the discrete optimization problem that we

[3]Ironically, the algorithm is called *logistic regression* even though it is a *classification* algorithm.

would face if dealing directly with f and the *classification error* cost function[4].

To further specify how to construct g, let us now introduce the *logistic function*, which is defined as:

$$\sigma(z) := \frac{1}{1+e^{-z}},$$

for all $z \in \mathbb{R}$. Observe that this function is defined $\sigma : \mathbb{R} \to (0,1)$, and is $\mathcal{C}^\infty(\mathbb{R})$. It is also worth observing that $\sigma(z) = 0.5$, $\sigma(z) \to 1$ when $z \to \infty$ and $\sigma(z) \to 0$ when $z \to -\infty$.

With *logistic regression* the hypothesis set for g will be $\mathbb{H} := \{\sigma(\theta^t x) : \theta \in \mathbb{R}^{m+1}\}$. As usually done for convenience, given $x \in \mathbb{R}^m$, we will be treating x as a point in $\mathbb{R}^{m+1}$ such that $x_{m+1} = 1$. This way, θ_{m+1} represents the *intercept* term in $\theta^t x$, and we do not need to write it separately.

Intuitively, observe that the mapping obtained with logistic regression can be understood as equivalent to determining an hyperplane $\theta^t x = 0$ in $\mathcal{X}$ and then, for each point $z \in \mathcal{X}$, assigning it 1 or 0 to depending on whether $\theta^t z > 0$ or otherwise. In three dimentions, we can picture this situation as separating two sets of points by a plane.

Now, how can we estimate θ? Since our assumption is that:

$$P(Y = 1|X;\theta) = g_\theta(x)$$

then,

$$P(Y = 0|X;\theta) = 1 - g_\theta(x)$$

And this allows us to write compactly the distribution of $X|Y;\theta$ as:

$$p(y|x;\theta) = (g_\theta(x))^y(1 - g_\theta(x))^{1-y}$$

Thus, since we are assuming that the observations in $\mathcal{D}$ are independent we can write the likelihood of θ as:

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{n} p(Y^{(i)}|X^{(i)}) \\ &= \prod_{i=1}^{n} (g_\theta(X^{(i)}))^{Y^{(i)}}(1 - g_\theta(X^{(i)}))^{1-Y^{(i)}} \end{aligned}$$

Now, we can simply determine θ by *maximum likelihood estimation*. However, instead of maximizing the likelihood itself, it will be more convenient to maximize the log-likelihood. This way, we obtain:

[4]Note, however, that this is not the only option available in order to avoid *classification error* loss function, use other alternative techniques without any probabilistic assumptions. One such example are *Support Vector Machines*, which we will not cover in our work.

$$\begin{aligned}\log(L(\theta)) &= \sum_{i=1}^{n} p(Y^{(i)}|X^{(i)} \\ &= \sum_{i=1}^{n} Y^{(i)}\log(g_\theta(X^{(i)})) + (1-Y^{(i)})\log(1-g_\theta(X^{(i)}))\end{aligned}$$

Once again, the maximization problem we have encountered through *maximum likelihood estimation* can be interpreted equivalently as an analogous *empirical risk minimization* problem. In the next section, with neural networks, we will address how this setting can be generalized to situations where $\mathcal{Y}$ contains more than two different classes.

Now, we turn to a problem we have already encountered twice but we still have not addressed. Once we have defined an optimization problem, how can we solve it?

Gradient Descent and its Variants

Even though there are some cases where the optimization problems resulting from empirical risk minimization[5], have a closed form solution, this does not happen in general. Thus, the general approach is to make use of iterative minimization algorithms. In Machine Learning, the most commonly used one is *Gradient-Descent*.

Algorithm 1.4. *(Gradient-Descent) Given a differentiable function $F : \mathbb{R}^n \to \mathbb{R}$, a starting point $x \in \mathbb{R}^n$ and two parameters $\epsilon, \gamma \in \mathbb{R}^+$. The minimization algorithm of F with initial point x is the following:*

Initialize:
Define: $x^{(0)} := x$
Define: $i := 0$

Do:
$x^{(i)} \leftarrow x^{(i-1)} - \gamma \nabla F(x^{(i-1)})$
$i \leftarrow i + 1$
While $|x^{(i)} - x^{(i-1)}| >= \epsilon$.

It can be proved that, with a value of γ small enough, Gradient Descent yields a sequence of points $x^{(i)}$ such that:

$$F(x^{(0)}) \geq F(x^{(1)}) \geq F(x^{(2)}) \geq F(x^{(3)}) \geq \dots$$

And by looking at the definition of the algorithm, we can see that, as long as the gradient at iteration i is not zero, then the corresponding left inequality is strict.

Thus, for convex functions, where it is known that the gradient only reaches the value zero in global extrema points, *Gradient Descent*, if applied with a parameter γ small enough, is guaranteed to converge to a global minimum.

[5] For instance, in linear regression, if the design matrix is non-singular, we can use the *normal equations* to obtain the parameters that minimize the quadratic cost.

However, in Machine Learning, many cost functions that arise are non-convex and, thus Gradient Descent could converge to local minima or saddle points.

In recent years, several techniques and *tricks* have been developed in order to avoid these problems. However, the reason why some these work is generally still not clear. Specifically, with the set of learning algorithms we will soon introduce, *Neural Networks*, there is a clear lack of theoretical understanding of why the optimization techniques employed achieve such great results. Nevertheless, issues related with the convergence of these optimization techniques are out of the scope of our work and we will not address them any further.

We will, however, briefly introduce two common variants of Gradient Descent that are generally used in practice to optimize cost functions when the amount of observations is very large.

Consider a set of n observations $\mathcal{D} = \{(X^{(1)}, Y^{(1)}), \dots, (X^{(n)}, Y^{(n)})\}$ of a random variable (X, Y). Let f_θ be a hypothesis and consider a loss function L. The empirical risk for f and L, takes the form:

$$R_{emp}(f, L) = \sum_{(X^{(j)}, Y^{(j)}) \in \mathcal{D}} L(f_\theta, X^{(j)}, Y^{(j)})$$

Thus, when trying to find the optimal value for θ, through gradient descent with a given step parameter γ, at each iteration i, we encounter:

$$\begin{aligned}
\theta^{(i)} &= \theta^{(i-1)} - \gamma \nabla_\theta R_{emp}(f_{\theta^{(i-1)}}, L) \\
&= \theta^{(i-1)} - \gamma \nabla_\theta \sum_{(X^{(j)}, Y^{(j)}) \in \mathcal{D}} L(f_{\theta^{(i-1)}}, X^{(j)}, Y^{(j)}) \\
&= \theta^{(i-1)} - \sum_{(X^{(j)}, Y^{(j)}) \in \mathcal{D}} \gamma \nabla_\theta L(f_{\theta^{(i-1)}}, X^{(j)}, Y^{(j)})
\end{aligned}$$

The third expression is the one that is actually computed. However, calculating the gradient of the loss for each individual observation for the whole dataset, at each iteration can represent a big computational burden when the the size of the dataset is big [6], and computing the derivative of the loss is *expensive* too [7]. Thus, a general strategy that is used is, at each iteration, computing the sum for only a subset of $\mathcal{D}_0 \subset \mathcal{D}$[8].

In the most *extreme* case, at each iteration, $\mathcal{D}_0$ contains a single element. This variant receives the name of *Stochastic Gradient Descent*. Generally, $\mathcal{D}_0$ has more than one element and, in this case, the variant receives the name of *Mini-Batch Gradient Descent*, in contrast to *Batch Gradient Descent*, which is the name that is generally used in the Machine Learning Community to refer to algorithm 1.4.

Optimization allows us to introduce the remaining element of our initial definition: the experience E. With these iterative algorithms computers effectively improve over time its performance on task T (i.e. supervised learning), as measured by the performance measure P (i.e. empirical risk) since it is precisely the function that is being optimized.

[6] For instance, several milions of images.

[7] For instance, θ might have hundreds of milions of components.

[8] This subset is generally taken by randomly choosing it among the whole dataset without replacement, in order to ensure that, during the process, all elements in $\mathcal{D}$ have been sampled an equal number of times.

As a practical note, reaching optimization convergence in real world-applications of machine learning, even with high-performing hardware, might take several days or even weeks of computation. This process is generally known as *training*,.

As already noted, optimization in Machine Learning models is one of the areas of this field that currently has the most open questions. However, these will not be our focus and we will end here our brief overview of Machine Learning.

1.2 Artificial Neural Networks.

In this section, we will introduce the main set of learning algorithms that we will be studying for the remaining of this work.

We will start by providing some background on the concept *Computational Graph* in order to later present *Multi-Layer Perceptrons*, which can be understood as the most basic case of *Neural Networks*. We will then introduce the most important theoretical result regarding these algorithms: The *Universal Approximation Theorem*. Finally, we will conclude this chapter introducing the *Backpropagation* algorithm, and the optimization process of *Neural Networks*.

Computational Graphs

Computational graphs can be seen as a general way to represent compositions of functions, and they provide a natural framework to represent *Neural Networks*.

There are several definitions available for Computational Graphs, and these can be formulated to allow these structures to represent arbitrary operations. However, we are not interested in general computational graphs, but only in these that we will be using with our learning algorithms. Thus, we will provide a definition tailored to our needs.

Definition 1.5. *(Computational Graph). Let $G = (V, E)$ be a non-empty directed acyclical graph (DAG). Let n be the number of leaf nodes in G* [9]*. We associate the following parameters at each node and edege of G:*

- *We associate a value $w_{ij} \in \mathbb{R}$ to each edge $(v_i, v_j) \in E$*
- *We associate a function $\sigma_k : \mathbb{R} \to \mathbb{R}$ to each non-leaf node $v_k \in V$, and a value $b_k \in \mathbb{R}$.*

Now, for every $x \in \mathbb{R}^n$, we can recursively define the following operations among nodes in G:

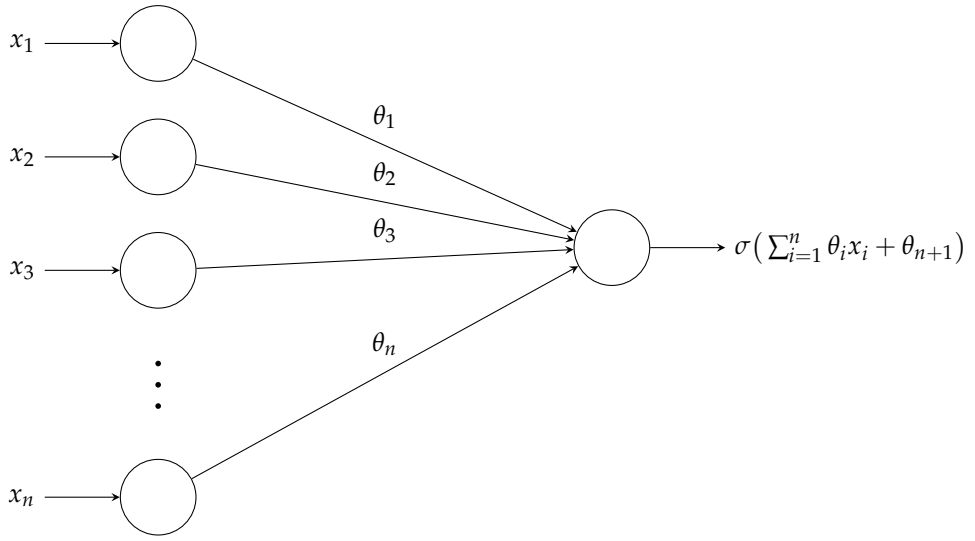
- *Let $v_1, \dots, v_n$ be all leaf nodes in V. We assign, to each v_i, $v_i \leftarrow x_i$*
- *For each non-leaf node $v_k \in V$, let v_{k_1}, v_{k_m} the set of all nodes such that $(v_{k_i}, v_k) \in E$. We assign, $v_k \leftarrow \sigma_k(\sum_{i=1}^{m} w_{k_i} v_{k_i} + b_k)$*

[9] Leave nodes are those vertices that have a single edge coming out of them. It is easy to see that non-empty directed acyclical graph must have, at least, one leaf node.

Example 1.6. *(Logistic Regression Revisited)* Recall that, for a learning problem where the vector of input features takes values in $\mathbb{R}^n$, i.e. $\mathcal{X} = \mathbb{R}$, the *logistic regression* hypothesis has the form:

$$f_\theta(x) = \sigma(\theta^t x) = \sigma\big(\sum_{i=1}^{n+1} \theta_i x_i\big)$$

for some $\theta \in \mathbb{R}^{n+1}$ [10] and σ being the *logistic function*. Thus, with the computational graph definition we have provided, f can be represented as a graph in a simple manner:



That is, we start assigning the components of x to leaf nodes, and then the only non-leaf node gets assigned the value of f itself.

Multi-Layer Perceptrons

The only classification algorithm we have introduced so far, *Logistic regression*, has a clear limitation: as already observed, it is essentially a linear function of its input features. In order for our hypothesis to be able to encode more complicated relationships, we must be able to *build* more complex functions.

With computational graphs and logistic regression in mind, we can think of an approach to building such functions. The main idea will be to, *combine* functions such as the hypothesis of logistic regression, and then *compose* these combinations.

Let's formalize this idea. Given $x \in \mathbb{R}$, and m vectors of parameters $\theta^{(1)}, \ldots, \theta^{(m)} \in \mathbb{R}^{n+1}$, we can define a mapping[11]:

[10] Recall the notation we introduced in the previous section, where the $n+1$ term in x is set to 1 and represents the intercept term in θ.

[11] as usual we for points in R^n*wedefineitsn* + 1*componenttobe*1*forconvenience.*

$$\begin{aligned} g : \mathbb{R}^n &\longrightarrow \mathbb{R}^m \\ x &\to (\sigma(\theta^{(1)T}x), \ldots, \sigma(\theta^{(m)T}x)) \end{aligned}$$

Where note that each component of g's output takes the form of a logistic regression hypothesis. Now, we can compose g with another such hypothesis h, determined by a vector of parameters $\theta' \in \mathbb{R}^{m+1}$, and obtain a mapping:

$$\begin{aligned} h \circ g : \mathbb{R}^m &\longrightarrow \mathbb{R} \\ x &\to \sigma\Big(\sum_{i=1}^{m} \theta'_i \sigma(\theta^{(i)T}x) + \theta'_{m+1}\Big) \end{aligned}$$

And observe that $(h \circ g)(x)$ no longer is an *essentially linear* function of x. In fact, we will soon see that, if we remove the outer σ from it, functions of the form of $(h \circ g)(x)$ are *universal approximators* of $n-$dimensional real-valued differentiable functions.

Before doing that, though, let us further extend its class of functions and introduce some notation. Observe that, if we represent $h \circ g$ as a computational graph, the output of g can be seen as a set of nodes that have edges both coming in and out of them. Typically the set consisting of these nodes is referred to as *hidden layer*. The set of leaf nodes which are initially assigned the input components is referred to as *input layer*, and the set of root nodes[12] is referred to as an *output layer*. Functions such as as $(h \circ g)$ are a special case of *artificial neural networks*[13].

General *neural networks* can have more than one hidden layer. That is, once we have defined a mapping $g : \mathbb{R}^n \to \mathbb{R}^m$, we can consider a different set of parameter vectors, $\theta^{(1)}, \ldots, \theta^{(l)} \in \mathbb{R}^{m+1}$, and define an analogous mapping:

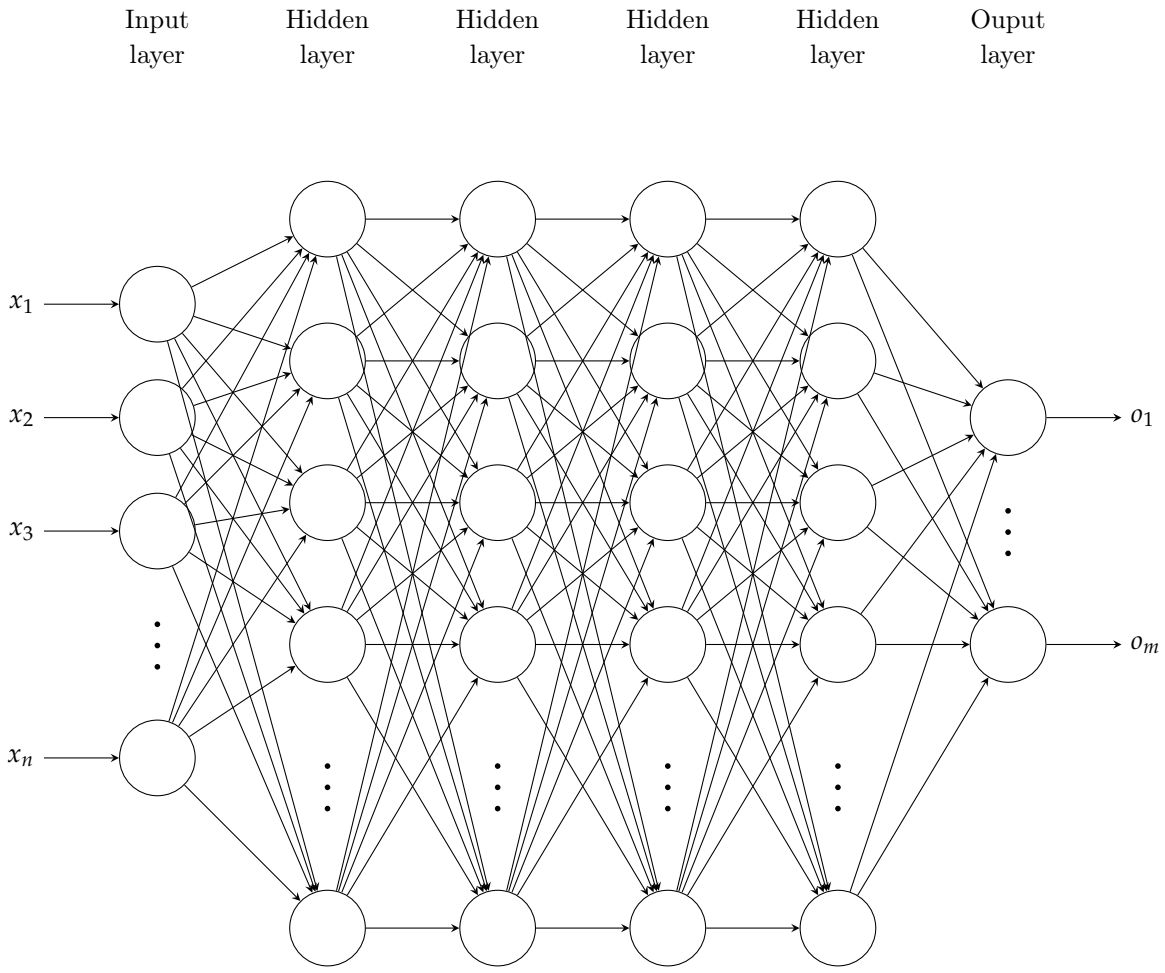
$$\begin{aligned} s : \mathbb{R}^m &\longrightarrow \mathbb{R}^l \\ x &\to (\sigma(\theta^{(1)T}x), \ldots, \sigma(\theta^{(l)T}x)) \end{aligned}$$

And then consider the composition $s \circ g$. This process could be repeated several times, and would yield a computational graph with as many *hidden layers* as functions of such kind. This family of functions is known as *Multi-Layer Perceptrons* (MLPs), and are generally considered as the most basic type of neural networks [14]. We now show the representation of the computational graph of a *MLP* with four hidden layers.

[12] those that do not have any edges coming out of them.

[13] from now on, we will be referring to them simply as *neural networks*.

[14] There is not, indeed, a stablished general definition of what an *Artificial Neural Network* is. In fact, it is often the case that the term *Neural Network* and *Multi-Layer Perceptrons* is used interchangeably, even though there are other families of functions that also are considered to fall into this category.



Observe that the function corresponding to the transformation between two consecutive layers with nodes $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ can be written as the composition of:

- A multiplication by a matrix $A \in \mathbb{R}^{m \times n}$, defined by: $A_{i,j} := w_{i,j}$, where $w_{i,j}$ denotes the parameter assigned to edge (x_i, y_j) in the computational graph.
- An element-wise evaluation of the function σ in Ax.

So far, we have been considering σ to be the *logistic* function. Historically, this has been the general practice, but there are other alternatives that currently are more popular: some examples are $\mathsf{tanh}(x), \mathsf{log}(1+\mathsf{exp}(x))$ and $\mathsf{max}(0, x)$. Generally, σ is known as the *activation function* and its relevance is, that it introduces a *non-linear* function in the computational graph. We will revisit *activation functions* in the next subsection.

It is worth observing that, in the output layer of the MLP we have represented, there are several nodes in the output layer. This setting allows to approach classification problems where there are more than two possible classes. In these cases, the vector resulting of an

output layer with k nodes composed with a mapping: $\mathbb{R}^k \to (0,1)^k$. Where each component of $\rho(z)$ is defined as:

$$\rho(z)_i = \frac{e_i^z}{\sum_{k=1}^K e_k^z}$$

This function known as *softmax* and is used in order to make the assumption that, in a problem where the possible classes are $\{1,\dots,K\}$, if z is the output of our MLP, $\rho(z)_i = P(Y = i|X)$. This allows us to define the cost function of the MLP through *Maximum Likelihood Estimation*, just as we did with *logistic regression*, and yields a natural definition of the final classifier as: $\underset{i\in\{1,\dots K\}}{\operatorname{argmax}}\{\rho(z)_i\}_{i\in\{1,\dots K\}}$.

The Universal Approximation Theorem

After this brief review of Multi-Layer Perceptrons, we now address the main result regarding their *approximation capacity*.

Theorem 1.7. *Let σ be a non-constant, bounded and monotonically increasing function and $I_n = [0,1]^n$. For every continuous function $f : I_n \to \mathbb{R}$ and $\epsilon > 0$, there exists an $N \in \mathbb{N}$, vectors $w_1,\dots,w_N,b_1,\dots,b_N \in \mathbb{R}^n$ and scalars $\alpha_1,\dots,\alpha_N \in \mathbb{R}$ such that function G, defined as:*

$$G(x) := \sum_{i=1}^N \alpha_i \sigma(w_i^t x + b_i)$$

satisfies $\|G(x) - f(x)\| < \epsilon$ for all $x \in I_n$.

A first formulation of this result, with further requirements on σ was first proved by [5] However, most recently, the result was proved without the requirement of σ being bounded [23].

The proof if this result uses concepts from *Fourier Analysis* that are out of the scope of this work. We observe, though, that it does not address two issues of much practical relevance:

1. Given a function to approximate f, can we determine *a priori*, which is the necessary value for N, i.e. the number of *hidden units*, in order to achieve an error smaller than ϵ?
2. Even if we somehow were able to determine a *right* value for N, how could we then find the parameters of such function?

Furthermore, this result raises an important question: why should we consider MLPs with more than one hidden layer if a single layer is already an *universal approximator*? We will be able to provide a partial answer to this question in the next section.

Now, since nowadays neither question (1) nor (2) have been answered yet, in practice, we will have to *heuristically* determine a value for N[15] and then resource to the optimization techniques introduced in the last section in order to *learn* the right parameters.

[15] In fact, the question of how many *units* to consider in the *architecture* of the network must be answered for every hidden layer, whose number must also be determined *heuristically*.

The Backpropagation Algorithm

In order use *Gradient Descent* and its variants on loss functions involving MLPs, we obviously need to be able to calculate the gradient of these functions with respect to its parameters.

A possibility would be to make use of a numerical approximation of the derivatives, by using the *finite differences* method. However, this solution has time complexity $o(n^2)$ on the number of parameters[16], and is clearly undesirable. Instead, we will compute the analyitical gradient. To do so, we will simply require, as tool, the chain rule of calculus for several variables. This will allow us to recursively define the gradient in a computational graph with time complexity $o(n)$.

More specifically, given a loss function L we will derive the expression of its gradient with respect to the parameters in f. In order to do that, we will first clarify some notation:

- We will denote by d the number of layers in the network, with layer 0 being the *input layer* and layer d being the *output layer*[17].
- For each layer l in the network, $1 \leq l \leq d$, W^l and b^l will denote, respectively, the associated matrix and intercept term, in the computational graph, between layers $l-1$ and l.
- For a given input vector x, we will refer with a^l to its output in the computational graph in layer l. That is, we define $a^0 = x$, and for $1 \leq l \leq d$, we define: $z^l = W^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
- Finally, we define, for each node j in layer l, $\delta_j^l = \frac{\partial L}{\partial z_j^l}$

Proposition 1.8. *(Backpropagation) Given a loss function L and a Multi-Layer Perceptron with l layers such that its parameters are specified as in the above notation, their derivatives can be computed with the following recursive formulas:*

$$\delta_j^d = \frac{\partial L}{\partial a_j^d} \frac{\partial \sigma}{\partial z}\bigg|_{z=z_j^l} \tag{1.2}$$

And for $1 \leq l \leq d-1$:

$$\delta_j^l = \sum_k \delta_k^{l+1} W_{k,j}^{l+1} \frac{\partial \sigma}{\partial z}\bigg|_{z=z_j^l} \tag{1.3}$$

$$\frac{\partial L}{\partial w_{j,k}^l} = a_k^{l-1} \delta_j^l \tag{1.4}$$

[16] In general, evaluating a neural network with n parameters requires $o(n)$ operations, and this must be done $2n$ times in order to obtain the full gradient

[17] Therefore, a network of d layers will actually have a total of $d-1$ hidden layers.

$$\frac{\partial L}{\partial b_j^l} = \delta_j^l \tag{1.5}$$

Proof. Let's start with equation 1.2. By definition of δ, and the chain rule of calculus for several variables:

$$\delta_j^d = \sum_k \frac{\partial L}{\partial a_k^d} \frac{\partial a_k^d}{\partial z_j^d} \tag{1.6}$$

But observe that, by definition of z^l and a^l, $\frac{\partial a_k^d}{\partial z_j^d} = 0$ if $j \neq k$ and, more explicitly, taking into account the fact that $a^l = \sigma(z^l)$, we obtain equation 1.2.

Now, for 1.3 observe that by definition of δ, and the chain rule of calculus for several variables:

$$\delta_j^l = \frac{\partial L}{\partial z_j^l} = \sum_k \frac{\partial L}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{1.7}$$

Now, since $z_k^{l+1} = \sum_i W_{k,i}^{l+1} \sigma(z_i^l) + b_k^{l+1}$, when we derivate with respect to z_k^l we obtain:

$$\frac{\partial z_k^{l+1}}{z_k^l} = W_{j,k}^{l+1} \frac{\partial \sigma}{\partial z}\Big|_{z=z_j^l} \tag{1.8}$$

And, thus, by substituting 1.8 in equation 1.7 we obtain 1.3.

Now, for 1.4, observe that:

$$\frac{\partial L}{\partial W_{j,k}^l} = \sum_i \frac{\partial L}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{j,k}^l} = \sum_i \delta_k^l \frac{\partial z_i^l}{\partial w_{j,k}^l} \tag{1.9}$$

And, by definition of z^l, if $i \neq j$ then $\frac{\partial z_i^l}{\partial w_{j,i}^l} = 0$, and thus the expression becomes:

$$\frac{\partial L}{\partial W_{j,k}^l} = \delta_k^l \frac{\partial z_j^l}{\partial W_{j,k}^l} \tag{1.10}$$

In the other hand, since $z_j^l = \sum_i W_{j,i}^l \sigma(z_i^{l-1}) + b_j^l$, we have $\frac{\partial z_j^l}{\partial W_{j,k}^l} = \sigma(z_k^{l-1}) = a_k^l$. Thus, if we substitute this expression in 1.10, we obtain 1.4.

Now, for 1.5, observe that it is completely analogous to 1.4, with the only difference being that we need to compute $\frac{\partial z_j^l}{\partial b_j^l}$ instead of $\frac{\partial z_j^l}{\partial W_{j,k}^l}$. Thus, it suffices to note that, by definition of z_j^l, we have $\frac{\partial z_j^l}{\partial b_j^l} = 1$, which yields the result.

□

With *Backpropagation* we have provided an efficient way to compute derivatives of MLPs parameters with respect to its weights. This allows us to use this derivatives with *Gradient-Descent*-based techniques in order to *learn* these parameters. With this algorithm, we conclude our review of *Neural Networks* and *Multi-Layer Perceptrons*.

1.3 Convolutional Neural Networks and Deep Learning

In this section, we will introduce a subset of functions within *Neural Networks* that has been particularly successful for *Computer Vision* problem: *Convolutional Neural Networks*.

We will start by providing a brief introduction to the problem of *image recognition*. Next, we will introduce the convolution operator and see how it is related to *Convolutional Neural Networks*. We will then provide some motivation on *why* this subset of neural networks is useful for image recognition and, to end up, we will provide a result regarding the role of *depth* in these networks.

Image Recognition

Image Recognition is a particular case of *Supervised Learning* where $\mathcal{X}$, the set of possible values for our random variable X, consists of digital images of a fixed size and the set of possible values for the *target variable*, $\mathcal{Y}$, consists of labels indicating *attributes* of those images, such as the presence of objects.

For humans, *Image Recognition* is a naturally eas task, as it is one of our basic perception abilities. However, the way computers *see* digital images is through the RGB format. That is, as three-dimensional matrices $M \in \{0,\dots,255\}^{n\times m\times 3}$. The first two dimentions account for spatial localization: n, the number of *rows*, corresponds to *height* in the image, m and the number of *columns*, to *width*. For each $1 \leq i \leq n$ and $1 \leq j \leq m$, $M_{i,j}$ can be understood as a *spatial location* in the image, which is known as *pixel*. The third dimension, consists of 3 different values for each *pixel* that encode its color with a value for each of the three *RGB* channels. Black-and-white images are defined analogusly as 2-dimensional matrices in $\{0,\dots,255\}^{n\times m}$, with *pixels* consisting in a single value instead of three.

The main challenge that *Image Recognition* faces is that there is not a clear way to translate *semantic concepts* for humans into values in these 3-dimensional matrices, and viceversa. For instance, once an an object is *present* in an image, humans understand that transformations such as changes in the lighting of the image and deformations, rotations or translations of the object do not alter its presence. However, from the *computer point of view*, these transformations consist in intricate modifications of the values in an array. Since there are an intractable amount of them, *rule-based* systems, thus, do not seem as an *encouraging* approach. In contrast, approaches consisting in making use of classical statistics encounter another challenge: treating such *high-dimensional* random variables.

In recent years, *Neural Networks* and, specifically, the kind that we are about to introduce, *Convolutional Neural Networks* (CNNs), have revolutionized this field, and have yielded

results that, a few years ago would not have been thought to be achievable. Namely, they have surpassed human performance in several tasks and are one of the key components in technologies such as the one behind *self-driving cars*.

CNNs are defined as *Neural Networks* such that one of its layers is a *Convolutional Layer*. We now to introduce this type of layer.

The Convolution Operator and Convolutional Layers.

A *convolution* is an operator, denoted with $*$ and defined, for two continuous functions f and g, as:

$$(f * g)(t) = \int f(a)w(t-a)da$$

Convolutions satisfy several desirable properties, such as *commutativity* and *associativity* and they are *distributive* with respect to addition.

In this work, we will be referring to their discrete version, which is analogously defined, for two discrete functions f and g, as:

$$(f * g)(t) = \sum_a f(a)w(t-a)$$

for the case in which f and g are one-dimensional functions. If they are 2-dimensional functions, convolutions are defined as:

$$(f * g)(t_1, t_2) = \sum_{a_1}\sum_{a_2} f(a_1, a_2)w(t_1 - a_1, t_2 - a_2) \tag{1.11}$$

In our problem in hand, we will be considering an operation *similar* to convolutions between three-dimensional matrices. In order to do so, given an array, we start by defining a mapping from its *indices* to its *values*. Specifically, we do that by, given an array $M \in \mathbb{R}^{n\times m\times d}$, considering a mapping I_M:

$$I_M : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^d$$

Defined as:

$$I_M(i,j) = \begin{cases} M_{i,j} & \text{if} \quad (i,j) \in \{1,\dots,n\} \times \{1,\dots,m\} \\ 0 & \text{otherwise} \end{cases}$$

Now, observe that, for each $1 \leq i \leq n$ and $1 \leq j \leq m$, $I_M(i,j)$ is a vector with d components. We will consider a variation of 1.3 that will consist in, given the mappings corresponding to two three-dimensional matrices $M_1 \in \mathbb{R}^{n_1\times m_1\times d}$ and $M_2 \in \mathbb{R}^{n_2\times m_2\times d}$, I_{M_1}, I_{M_2} modifying it in the following manner:

$$(I_{M_1} \tilde{*} I_{M_2})(t_1, t_2) = \sum_{a_1}\sum_{a_2} I_{M_1}(a_1, a_2)^T \cdot I_{M_2}(t_1 - a_1, t_2 - a_2) \tag{1.12}$$

where $\cdot$ denotes the standard dot product.

In order to obtain an array back from this operation, $(I_{M_1} \tilde{*} I_{M_2})$, we define $A^{max(n_1,n_2)\times max(m_1,m_2)}$ as:

$$A_{i,j} = (I_{M_1} \tilde{*} I_{M_2})(i,j)$$

With our case in hand, however, we can assume that one of the matrices, for instance M_2, is such that $n_1 > n_2$ and $m_1 > m_2$, that is: its *height* and *width* are both smaller than M_1's. The usual practice is to also *discard* indices $n_1 - n_2 + 1, \ldots, n_1$ and $m_1 - m_2 + 1, \ldots, m_1$ of the output matrix A. That is, we only its positions corresponding to operations between all elements in M_1, and all elements in M_2.

This setting allows us to understand this operation as *sliding* M_2 through all the spatial locations specified above in M_1 and computing a *weighted average*[18] in a neighborhood of each possible position $(i,j) \in \{1, \ldots n_1 - n_2\} \times \{1, \ldots m_1 - m_2\}$. We can visualize this situation with the following example:

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tilde{*} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{pmatrix}$$

Where the matrices are, from left to right, M_1, M_2 and A. In this example, however the matrices are two-dimensional. We could picture the general case by repeating the operation for each *slice* in the three-dimensional matrix M_1 and M_2, and then adding each resulting matrix.

These operations are often applied in *Computer Visionand outside of neural networks*, in order to extract local features of images. For instance, M_1 might be a color image of dimensions $1024 \times 1024 \times 3$ and M_2 might be a 3-dimensional array of dimensions $3 \times 3 \times 3$, *somehow* designed to highlight *edges* in M. The result of $(I_{M_1} \tilde{*} I_{M_2})$ would then be a 1021×1021 array, where positions corresponding to pixels in M_1 *containing* edges would have higher intensity values[19].

The operation we have been described is, in a nutshell, what *Convolutional Layers* consist in. Generally, we consider the values of nodes in a computational graph as an array M in $\mathbb{R}^{n\times m\times d}$, and another array $K \in \mathbb{R}^{n'\times m'\times d}$, as before, such that $n > n'$ and $m > m'$. K is referred to as a *kernel* of *filter*. After we have done that, we then apply, element-wise, an activation function σ, to the result. As noted, the result from this operation yields a two

[18] where we understand that the elements being averaged are those in M_1, and the *weight coefficients* are values in M_2

[19] In order to avoid the resulting array to have smaller *height* and *width*, there are techniques available, named padding, but we will not be fully addressing these practical considerations

dimensional matrix in $\mathbb{R}^{(n-n')\times(m-m')}$. The general practice, is to repeat the process s times, with a different *kernel* of the same size each time, and thus, obtain a total of s matrices in $\mathbb{R}^{(n-n')\times(m-m')}$, which are referred to as *feature maps*. Then, these are concatenated in order to form a single matrix $F \in \mathbb{R}^{(n-n')\times(m-m')\times s}$. F, then becomes the new *hidden* layer in the computational graph.

We now provide some general motivation on *why* these layers are generally thought to have an advantadge over *fully connected layers*.

Advantages of Convolutional Layers

In [7], its authors highlight three following reasons as the main ones explaining the success of the layer we have just introduced:

- *Sparse Connectivity*: Observe that *convolutional layers* can be seen as a special case of *fully connected layers*. Recall that, in the latter, for a given network f, each node in layer l was the result of a linear combination of all nodes in the previous layer. That is, in the computational graph of f had edges from all nodes in the previous layers to it. With *convolutional layers*, the situation is totally analogous, with the only difference being that edges between nodes in consecutive layers are further restricted: for nodes in layer l resulting from applying a kernel $K \in \mathbb{R}^{k_1\times k_2\times k_3}$ to layer $l-1$, there are exactly $k_1k_2k_3$ edges going from nodes in the previous layer into it. This reduced number of parameters allows for much fewer computations and parameters than in *fully connected layers* and thus, for a fixed computational budget greater *depth* than in an MLP. As we will soon see, this a is a key advantage.

- *Parameter Sharing*: Recall that, at each hidden layer, all units in the same *feature map* have been computed with the same *kernel*, i.e. the same parameters. The intuition behind this fact is that there is no need to use different parameters in order to *find the same feature* around an image. This fact allows to greatly reduce the amount of memory necessary to deploy the network

- *Equivariance to translation*: This property refers to the commutativity of convolutional layers with respect to translations in the input image. That is, for an three-dimensional matrix M, consider, its corresponding mapping, as defined in the previous subsection I_M, now define a translation on it as $g(I(i,j)) = I(i-a, j-b)$, for some $a, b \in \mathbb{N}$. Now, if f is a convolutional layer, then f satisfies $f \circ g = g \circ f$.

Convolutional Layers and Depth.

In *Neural Networks'* Section we saw that MLPs with a single layer can approximate any continuous function arbitrarily well in a compact subset of $\mathbb{R}^n$. However, in practice, deeper architectures are virtually always preferred. In the case *Convolutional Neural Networks*, it has been consistently shown empirically that, given a dataset, adding more *convolutional* layers seems to increase the achievable accuracy. The apparent relevant of *depth* in these

learning algorithms have earned them the general name of *Deep Learning*, which is currently used, almost interchangeably, with *Neural Networks*.

A natural question is whether there is some theoretical result justifying the success of *deep* architechtures over *shallow* ones. Poggio i his article [14] provide some results in this direction, which we will very briefly review.

In their article, they consider a class of functions that they define as *hierarchichally local compositional functions*. A $f : \mathbb{R}^n \to \mathbb{R}$ is said to be of such class if there is some natural number $d < n$ and a family of functions $S_f := \{g : g : \mathbb{R}^d \to \mathbb{R}\}$. Such that S_f can be written solely, as a composition of functions in S_f. A natural subset of these functions are *tree-like structured* functions, of which *Convolutional Neural Networks* are a particular case.

The main result provided by the authors consists in observing that, in order to achieve an approximation error of ϵ, *shallow networks* [20] need $o(\epsilon^{-n})$. Instead, *shallow networks* require only $o(n\epsilon^d)$, where d is defined as the size of its largest filter.

The question then becomes, if the task of *vision* can be understood as a function of such type. It turns out that a possible answer to this question can be obtained throguh *Neuroscience*. It turns out that the type of connections (i.e. compositional relationships) defined in *Convolutional Neural Networks* are similar to those present in *biological neural networks*, which suggests that the answer to this answer might be affirmative and thus, the explanation in [**POGGIO**] might be plausible.

Pooling Layers and Sample Architectures

There is one more type of *layer* that is generally present in *Convolutional Neural Networks*: *pooling layers*.

Pooling layers are used in order to reduce the *spatial dimensions* of *hidden layers*[21]. There are two main sub types of such layers: *average pooling* and *max pooling*. In both cases, given a 3-dimensional matrix $M \in \mathbb{R}^{n \times m \times d}$ representing a hidden layer in a neural network f. For each $1 \leq k \leq d$ a pooling layer $f_\cdot$ with *stride* s applied to M is defined as,

$$f_p(M)_{i,j,k} := g(\{M_{i',j',k} : 1 + s(i-1) \leq i' \leq si \text{ and } 1 + s(j-1) \leq j' \leq sj\})$$

where g is the $\max$ function if f_p is a *max pooling layer*, and a standard average function in case it is an *average pooling layer*.

That is, for each feature map in M, we are dividing it in *squares* of side s and computing either its maximum value, or its average.

Pooling layers are not an essential part of *Convolutional Neural Networks* and recently, [24], its authors have shown that it is possible to obtain results comparable to the *state-of-the-art* without resourcing to them. As an alternative, they propose using *strided convolutional layers*, which can be seen as analogous operations to pooling, but substituting g by applying a kernel to the corresponding, in the layer, of g's input.

[20] such as the one considered in the *Universal Approximation Theorem*

[21] i.e. the two first dimensions of their corresponding matrices

However, *standard convolutional networks* generally follow this design guideline:

1. Successive pairs/triplets of one/two *convolutional layers* followed by a pooling operation. This sequence is often referred to as the *feature extraction* part of the network.
2. Some *fully connected layers* until the output layer is reached. This sequence is often referred to as the *classification* part of the network.

Nevertheless, general *Deep Learning* is a quickly moving field, and different *exotic* and successful architectures are proposed with high frequency, and these often do not match the patterns we have just described. An example of such is [9].

We will revisit most of the concepts we have introduced in this chapter in the upcoming chapters and, now, we conclude our preliminary work and begin with our main area of focus: *Attribution Methods for Deep Convolutional Networks*.

Chapter 2

Attribution Methods for Deep Networks

In the previous chapter, we provided some basic context on the general setting of Deep Learning Algorithms. We will now start our study on the main work of this chapter. We will start by reviewing recent work on obtaining general interpretabilty for Deep Networks. After that, we will introduce some basic definitions and then proceed to study the attribution techniques in two separate groups: Gradient-Based Techniques and Perturbation Based techniques.

2.1 An overview of recent research on Deep Learning interpretability

In recent years, several techniques have been developed with the goal of making Deep Networks predictions more interpretable. These can be roughly divided into two categories: those that attempt to understand *what* concepts a network has learned, and those that attempt visualize *where* in the input the network can find them. We provide a brief overview of them below:

- In the first group, several authors attempt to understand each hidden unit's function inside a network. They do it with two main approaches:
  - *Optimisation-based techniques*: [12] take as input a trained network f and a target hidden unit j at layer l, and consider $f_{l,j}$: f's restriction at the output of this unit. Then, they set the optimization problem:

    $$\underset{x \in \mathbb{R}}{\operatorname{argmax}} f_{l,j}(x) + \lambda \|\mathcal{L}(x)\|$$

    Where the second term is a regularization penalty that constrains x to have

natural image statistics. Thus, they obtain a visual explanation of what the network is *looking for*.

– *Monitoring-based techniques*: in [4], its authors have built a dataset named BRODEN, consisting in a wide variety of images representing instances of concepts such as textures, colors, objects, etc. Given a trained network, they feed it all images in BRODEN and monitor how each hidden unit is activated through them. They observe, empirically, that responses in certain units in upper layers of the network are specific to some of human-concept labeled images. Thus, they conjecture that these units are actually encoding concepts that would be intuitive for humans. More recently, in [6], its authors have developed Net2vec, where they attempt understand how the combination of several units encodes a single *human* concept.

- The second group consists of Attributions methods, which are the main focus of this work. These techniques' goal is not to understand a single unit's function, but rather its relationship with units in lower layers in the network[1] . More specifically, given a function f and an input $x \in \mathbb{R}^n$, their goal is to construct a vector $\hat{x} \in \mathbb{R}^n$ that *quantifies* the influence of each component in x.

Lastly, there is a natural way to combine both sets of techniques. Recently, in [13] its authors have studied a wide range of possible interfaces to combine visualisations of units' activations in higher layers in the network, with localisations of these at the input pixels. With these, these interfaces attempt to tackle the whole interpretability problem at once: their goal is to shed light on *what patterns* the network is seeing and *which pixels* are responsible for it.

2.2 Basic definitions[2]

In the previous section, we got a general idea of what attribution methods do, and how they relate to other techniques within the scope of Neural Networks interpretability.

Although, some relatively successful applications of attribution techniques have already appeared, this is still a young field of research, and it is still facing many challenges regarding both its theoretical foundations and practical applications. Specifically, different authors provide different views regarding how they should be defined. Here we have tried to leverage them and provide a general framewor.*However, once we get to Perturbation-Based Methods, we will have to adapt these definitions.*

We start by providing a general definition on what an *attribution function*[3] is.

Definition 2.1. Let $f : \mathbb{R}^n \to \mathbb{R}$. An attribution of f is a function $F_f : \mathbb{R}^n \to \mathbb{R}^n$. We will refer to each component F_{fi}, as the attribution of F_f for f at x_i

[1] These could be a individual pixels which, we recall, are the network's 0th layer

[2] It turns out that the problem of attribution has been already tackled in Game Theory, in the context of *Cost Sharing* literature. Some of the definitions we will provide below and techniques that will appear later in this chapter have its origin in this field.

[3] Although some authors have defined attributions as vectors [25], we believe that defining them as functions whose output are those vectors allows us to study properties in them that are function-specific

Observation 2.2. Recall that a Neural Network f can always be written as a composition of several functions: $f_m \circ f_{m-1} \circ \cdots \circ f_0$ and that, for a given function f_i, its output component f_{ij} is referred as the jth *unit* in *layer* i. As we noted in the previous section, attributions can be considered, in particular, over a specific unit and subset of layers in a network. That is, given a network f, we could consider functions of the form $f_{lj} \circ f_{l-1} \circ \cdots \circ f_k$ to build attributions on them.

Observation 2.3. We will later see that some techniques to construct F_f, constrain its output to be in $\mathbb{R}^{+^n}$. However, other methods do not, as it is among its heuristics to distinguish input components between those having *positive* and *negative* influence on the output.

Observation 2.4. As with any functions, we can ask attributions to fulfill *continuity* and *differentiability*. However, as we will see in the following sections, these will not be satisfied by some techniques.

We now turn to the problem of characterizing zero-valued components in attributions. The next definition provides the most basic axiom in this direction:

Definition 2.5. Let $f : \mathbb{R}^n \to \mathbb{R}$ such that it does not depend on one of its input components i, and let F_f an attribution of f. F_f satisfies the *Dummy* property if, for all $x \in \mathbb{R}^n$, $F_{fi}(x) = 0$

The property above provides a sufficient condition for an attribution's component to be zero. Next, we will state a necessary one but, before that, we will introduce further heuristics.

Recall that our goal with attributions is to quantify the influence of each component of x on f's output. Thus, for each $i \in \{1, \ldots, n\}$, $F_{fi}(x) \neq 0$ should not only reflect that f depends on x ith component but rather the fact that a specific change in its particular value would have an *influence* in f's output. For functions that can be locally constant, this notion is different that mere dependency. We exemplify this fact below:

Example 2.6. Let $f(x, y) = max(0, x) + y$ and $(x_0, y_0) = (-50, 0)$. Despite the fact that f cleary depends on both of its input components, it is clear that switching x_0 for any $x' < 0$, will not affect f's output regardless of its particular value.

We would like to capture this intuitive notion in a formal property, but how can we account for it? Our main obstacle is specifying *which changes* in the input we will be considering.

One possibility is to define an additional input vector x' in the domain of f, which we will refer to as *reference*[4]. Then we can extend Definition 2.1 so that F_f now becomes a function $F_f : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, that takes both a vector x that we want to *explain* and a reference point x' in the same domain. This allows us to state the following property, called *sensitivity* in [25]:

[4]The introduction of a reference adds a free parameter to all attribution functions and, as we will see, its choice has an impact on both the theoretical properties in the function and its practical use. We will address how to approach its choice later in this chapter

Definition 2.7. Let $f : \mathbb{R}^n \to \mathbb{R}$. And F_f an attribution of f. F_f satisfies *sensitivity* if, for every pair of reference and point $x, x' \in \mathbb{R}^n$ satisfying the following:

- There is an index $j \in \{1, \ldots, n\}$ such that $x_j \neq x'_j$ and $x_i = x'_i$ for all $i \in \{1, \ldots, n\} \setminus \{j\}$
- $f(x) \neq f(x')$

Then $F_{fj}(x, x') \neq 0$.

An additional requirement that we can ask F_f to fulfill is the following[5]:

Definition 2.8. Let $f : \mathbb{R}^n \to \mathbb{R}$. And F_f an attribution of f. F_f satisfies *completeness* if, for every pair of reference and point $x, x' \in \mathbb{R}^n$, the following equality holds:

$$\sum_{i=1}^{n} F_{fi}(x, x') = f(x) - f(x')$$

Observation 2.9. In [25], the authors claims that *completeness* is a stronger property than *sensitivity*, as if an attribution F_f satisfies the former, then it must also satisfy the latter. We do not support this claim and provide a simple counter-example for it below.

Example 2.10. Let $f : \mathbb{R}^2 \to \mathbb{R}$, defined by $f(x_0, x_1) := x_0 + x_1$. For each $x, x' \in \mathbb{R}^2$ we define $F_f(x, x') := (f(x) - f(x'), 0)$. F_f trivially satisfies *completeness*. However if we consider as input point and reference $(0, 0)$ and $(0, 1)$, respectively, *sensitivity* is clearly not satisfied, as $f(0, 0) \neq f(0, 1)$, and the only non-equal component between input point and reference equals 0 in its corresponding attribution component.

Sensitivity and *Dummy* fully characterize zero-valued attributions components. Although these are important axioms, they do not address one of the most important topics in attributions: the relative magnitudes of its components. Intuitively, we would not only like attributions' components to be zero-valued in non-relevant variables, but we would rather want them to be larger in those components having the most impact on the output[6]. Namely, we will have to study how to quantify the impact of an input component at a specific point. In order to approach these concepts in a formal manner, we will require further specifications on our approaches to construct attribution functions.

Thus, we will stop our introductory theoretical study here and, from now on, additional properties of attribution functions will not be considered as general axioms any further but, instead, will be studied specificaly for every considered approach.

[5] The need for the following property will be further motivated in the next section, with example 2.13.

[6] It could be argued that this issue is *somehow* addressed by a combination of *completeness* and *sensitivity* properties. When both are satisfied we can ensure that only relevant components receive nonzero attribution, and the sum of these equals the difference $f(x) - f(x')$. However, this observation still does not consider the address the relative magnitudes of non-zero attributions.

2.3 Gradient-Based Techniques

In this section we will explore one of the main sets of attributions methods: those that use, either implicitly or explicitly, the network's gradient.

We will start by motivating the use of the gradient through the study of linear models, and this will allow us to later introduce the *Integrated Gradients* [25] method .

After that, we will study several techniques that exploit Neural Networks structure through recursive relevance propagation algorithms. We will finish this section by presenting an unified framework for these methods as variations of the *Backpropagation* algorithm.

Attributions in Linear Models

Linear Models are generally regarded as the simplest possible approach to Supervised Learning and thus, they are a good starting point to begin our practical study of attributions. In contrast to Deep Networks, they are generally regarded as models with *full algorithmic transparency*. With the following proposition, we provide a natural way to define attribution functions on them.

Proposition 2.11. Let $w, b \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$ defined by $f(x) := w^t x + b$. For every reference point $x' \in \mathbb{R}^n$ the attribution function defined by $F_f(x, x') := (w_i(x_i - x_i'), \ldots, w_n(x_n - x_n'))$ satisfies *dummy*, *sensitivity* and *completeness* . Furthermore, it is also continuous and differentiable with respect to both x and x′ in the whole of its domain.

Proof. *Sensitivity*, *dummy*, continuity and differentiability are trivially satisfied by construction. To prove that *completeness* also holds, observe that, for every $x, x' \in \mathbb{R}^n$, we have the following equalities:

$$\sum_{i=1}^{n} F_{fi}(x, x') = \sum_{i=1}^{n} w_i(x_i - x_i')) = w^t x - w^t x' = (w^t x + b) - (w^t x' + b) = f(x) - f(x')$$

□

Observation 2.12. In the attribution function above, setting the reference to be x′ = 0, provides a very intuitive use case: each component is assigned its corresponding term in the total sum: the larger the term, the larger the *contribution*.

The simplicity with which we can define an attribution function with linear models, provides motivation for the following idea: can we locally approximate arbitrary functions through linear models and then apply proposition 2.11 in order to obtain a well behaved attribution function? We explore this idea in the following subsection.

From Taylor Approximation to *Integrated Gradients*

It is well-known that differentiable functions $f : \mathbb{R}^n \to \mathbb{R}$ can be approximated in a neighborhood of a point x' by its first order Taylor Expansion:

$$f(x) \approx f(x') + (x - x')^t \nabla f(x') \tag{2.1}$$

We know that this approximation's error (concretar?) term is $o(x^2)$, and it depends on the distance between x and x'. The latter fact sheds light on an important heuristic over the choice of the reference x': it should be close to the point we want to do attributions on.

Moreover, the presence of the error in the approximation carries a disadvantage regarding fulfillment of *completeness* by the attribution function we can build from it. Indeed, if we denote with ϵ the error term of (2.1) , then applying definition 2.11 to our linear approximation yields that:

$$\sum_{i=1}^{n} \frac{\partial x_i}{\partial f(x)}(x_i - x_i') = (x - x')^t \nabla f(x') = f(x) - f(x') - \epsilon$$

Thus, the error term determines the *amount* in which *completeness* is not satisfied. We provide an example of how that can be undesirable for functions that are highly non-linear locally:

Example 2.13. Let f be a simple logistic regression function with the following form:

$$\text{SIGNES!!!! } f(x_0, x_1) = \frac{1}{1 + e^{10x_1}} =^7 \sigma(-10x_1)$$

. For every reference point (x_0', x_1'), its first order Taylor expansion is:

$$f(x_0, x_1) \approx \sigma(10x_1) + 10\sigma(10x_1')(1 - \sigma(10x_1'))(x_1 - x_1').$$

Thus, if we take, for instance, $(x_0, x_1) = (0, 1)$ and $(x_0', x_1') = (0, -3)$, from applying definition 2.11 we obtain the attributions vector $(0, 3.74{\cdot}10^{-12})$.

Observe that our function, f, only depends on x_1, and we have used a reference and input points, two points that are at opposite sides of the separating hyperplane $x_1 = 0$ and rather close. Thus, we would expect our attributions to assign a high score to x_1's component but we obtain, instead, a near-zero value, as a consequence of the error term of the local approximation. This high error is due to the fact that some functions such as the sigmoid are near-constant at most of their domain. Therefore, gradients evaluated at those points are close to zero and, as consequence, the resulting attributions from using them in linear approximations are also close to zero.

[7] Following the notation of chapter COMPLETA!!!, we refer to the logistic function with σ. Recall that its derivative takes the form $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

Observation 2.14. Despite the fact that *completeness* will generally not be satisfied by attributions functions using the above procedure, with the right choice of x', *sensitivity* will hold. There is a simple reason for that: if $f(x)$ is not locally constant with respect to any of its variables at a neighbourhood of x, then, for every x' in this neighborhood, all of its partial derivatives will be non-zero and, thus, each of the attributions components will be non zero if, and only if, $x - x'$ is non-zero at that component.

Our last example had the goal of motivating why *sensitivity* is not enough for attributions to meet our intuitive expectations, and why ensuring *completeness* fulfillment should be among our goals. The technique we are about to introduce has a simple yet effective solution to guaranteeing the satisfaction of *completeness*.

We start by introducing its linear approximation approach [8]

Definition 2.15. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function, $x' \in \mathbb{R}^n$ and, for all $x \in \mathbb{R}^n$, let $\gamma_x : [0, 1] \to \mathbb{R}^n$ be the line path defined by $\gamma_x = x' + \lambda(x - x')$. We define the *Integrated Gradients linear approximation* of f at x with reference x' as: REVISAR!!!!!!!!!!!! No té sentit

$$IG_f(x, x') := f(x') + \int_{\gamma_x} \nabla f \circ \gamma_x(u) du = f(x') + (x - x')^t \int_{\lambda=0}^{1} \nabla f(x' + \lambda(x - x')) d\lambda$$

Observation 2.16. AIXÒ A LA PROP DE MÉS ABAIX, AQUESTA INT ÉS CALCULABLE The line integral above might not have an analytical solution in general, however, it can be numerically approximated as a finite sum. In practice, if f is a deep network, evaluating it is computationally expensive and, thus, applying Integrated Gradients has a significant computational cost.

Observation 2.17. It is easy to see that when f is a linear function, the above calculation returns f itself. Indeed, let $f(x) := w^t x + b$, then $\nabla f = w$ and thus, for the integral we have $\int_{\lambda=0}^{1} \nabla f(x + \lambda(x - x'))d\lambda = w$. Therefore, the full expression becomes:

$$IG_f(x, x') = f(x') + (x - x')^t w = f(x') + f(x) - f(x') = f(x)$$

Now, the previous definition allows us to apply Definition 2.11 on it and, as the following proposition states, this provides an attribution that satisfies *completeness* regardless of the choice of x'.

Proposition 2.18. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function, $x, x' \in \mathbb{R}^n$ and let $f(\hat{x, x'})$ = be f's *Integrated Gradients linear approximation*, i.e. for every $i \in \{1, \ldots, n\}$:

$$w_i = \int_0^1 \frac{\partial f(x' + \lambda(x - x'))}{\partial x_i}$$

[8] The formulation we will provide below does not coincide with the one stated in [25] as its authors directly provide a final attributions vector. However, we believe that considering separately a linear approximation function and then applying definition 2.11 to it makes for a clearer justification.

Then the attribution $F_f(x, x') := (w_i(x_i - x'_i), \ldots, w_n(x_n - x'_n))$, which we will refer to as the *Integrated Gradients Attribution*, satisfies satisfies *dummy*, *sensitivity* and *completeness*. Furthermore, it is also continuous and differentiable with respect to both x and x′ in the whole of its domain.

Proof. Dummy is obviously satisfied by construction, and continuity and differentiability come are consequences of the fact that f is differentiable. Let $\gamma_x : [0, 1] \to \mathbb{R}^n$ be the line path defined by $\gamma_x = x' + \lambda(x - x')$ and recall that the Fundamental Theorem of Calculus for Line Integrals ensures that:

$$\int_{\gamma_x} \nabla f \circ \gamma_x(u) du = f(\gamma_x(1)) - f(\gamma_x(0)) = f(x) - f(x')$$

Thus, it follows that:

$$\sum_{i=1}^{n} w_i(x_i - x'_i) = (x - x')^t \int_{\lambda=0}^{1} \nabla f(x + \lambda(x - x'))d\lambda = \int_{\gamma_x} \nabla f \circ \gamma_x(u) du$$
$$= f(x) - f(x')$$

Lastly, *sensitivity* follows as a consequence of *completeness* and the fact that each component i in the attribution function has the factor $(x_i - x'_i)$ on it. □

We observe from Proposition 2.18's proof that its only assumption on the path, γ_x, is that it must be continuous and, therefore, we could generalize the result for any continuous path between x and x'. Thus, a natural question arises: is there any advantage in considering a straight line path over other alternatives? It turns out that the answer is affirmative, but before stating *why*, we need the following two definitions.

Definition 2.19. Let $f : \mathbb{R}^n \to \mathbb{R}$ and $x \in \mathbb{R}^n$. Let us denote by $\hat{x}_{i,j}$ a vector whose all components are as x's except for the ith and jth, that have been swapped. We say that components i and j are symmetric in f if, for all $x \in \mathbb{R}^n, f(x) = f(\hat{x}_{i,j})$.

Definition 2.20. Let $f : \mathbb{R}^n \to \mathbb{R}$ and $x, x' \in \mathbb{R}^n$ and let $F_f(x, x')$ be an attribution of f. We say that F_f is symmetry preserving if, whenever components i and j are symmetric in f, then for all $x, x' \in \mathbb{R}^n$ such that $x_i = x_j$ and $x'_i = x'_j$, $F_{fi}(x, x') = F_{fj}(x, x')$.

Now we are ready to announce the following proposition.

Proposition 2.21. From all the possible choices for γ_x in Proposition 2.18, a linear path is the only one that produces attributions that are symmetry-preserving.

Lastly, there is one last advantage with using the *Integrated Gradients Attribution*: since it is a function of f's gradient, it does not depend on the specific computational implementation of f. This might seem as an very basic condition but as we will see in the next section, it is not satisfied by some methods.

Avoiding the use of the referencepoint: *Grad*Input* and *SmoothGrads*

So far, we have described the use of a reference point as desirable, in the sense that it allows to consider the satisfaction of two properties: *sensitivity* and *completeness*. However, as observed, it also raises an important question: what makes for a good reference?

There are techniques that avoid the *reference point* and instead, compute the gradient of the network and a given point, and then multiply this result, elementwise, with the point itself. Even though there is not a clear theoretical justification for proceeding in such way, we will see in Chapter 3, that this technique obtain similar result to the ones of *Integrated Gradients*. We refere to this technique as *Gradient*Input*[**asd**].

A variation of this method, consists in given $x, \sigma \in \mathbb{R}^n$, some natural number m and a network $f : \mathbb{R}^n \to \mathbb{R}$, computing attributions by defining $F_f(x)$ as:

$$F_{f,i}(x) \sum_{i=1}^{m} \frac{\partial f(x)}{\partial x_i} + z_i$$

where $z_i \sim (N)(x_i, \sigma_i)$. Intuitively, this technique approximates the expected value of Gradient*Input in a neighborhood of x, while assuming that each component in X, its associated random variable, follows a normal random distribution. This technique is known as *SmoothGrads*[22].

Exploiting Neural Networks' Structure

In the previous subsection, we considered attributions for arbitrary functions, and we did not make any assumptions on their properties or structure other than continuity or differentiability.

However, recall that, for the task we have in hand, the class of functions used in practice is generally narrowed down to those studied in Chapter 2: Convolutional Neural Networks. Therefore, we can ask the following: can we obtain better attributions by restricting our study to Neural Networks?

This question was first tackled by (CITA: Fergus et al) with *Deconvolutional Networks*, which consist in an heuristically defined variation of the *Backpropagation* Algorithm that has empirically shown to produce sharper saliency maps than the gradient itself. More recently, more sophisticated approaches have appeared (Torralba, GRAD-CAM, Guided Backpropagation). Although some of these approaches have shown empirical success, they will not be considered in our work, since they are not only Neural Network-specific, but also require some strong assumptions on the specific architecture of the Network.

2.4 Perturbation-Based Techniques

In the last subsection, we introduced several attribution techniques that were based on the idea of obtaining a local linear approximation of a network via its gradient at a given point. In this section, we aim to change the point of view, and consider a different approach that discards information about the gradient and, instead, obtains attributions by measuring the effect of perturbating regions of the image

It could be argued that there are, at least, two main issues regarding the techniques we introduced in the last chapter:

- Partial derivatives only expose the effect of changes at an infinitesimal level, and it is not clear how meaningful these are for our purpose. In particular, none of the techniques shed any light on providing theoretical results regarding the interpretation of relative magnitudes of attributions components
- First order partial derivatives only address changes in individual components, whether these are pixels or intermediate units. They lack the capacity to capture interaction among those.

Perturbation-based methods aim to address these issues. Broadly speaking, instead of considering the impact of infinitessimal changes in individual input components, they attempt to measure the effect of abrupt changes in several pixels at a time.

The first line of work in this direction was presented by [27]. In this article, the authors consider a partition of the image's pixels by dividing it adjacent squares of equal sizes. Then, for each square on the image, they proceed to change the values of its corresponding pixels to a predetermined *uninformative* value[9] , such as 0, and evaluate this modified version on the network f. Even though they obtain some interesting results, this technique has a major limitation: it does not consider in any way how changing the values of different regions at the same time affects the network's output. The methods we are about to introduce, address this issue by drawing its techniques upon of Game Theory.

A different approach to local linearization

We start by observing that considering modifications of points in high-dimensional spaces is a challenging task:

- Let n be the number of dimensions of the space. If we consider a single alternative value for each component, we obtain, as a result, 2^n different points, which quickly turns any algorithm intractable
- Furthermore, how should we determine each individual component's alternative value?

The first issue, combined with the first limitation we observed in this subsection's introduction, motivate perturbations to be considered at several components at the same time. As

[9] We will further specify the relevance of this value later in this chapter.

for the second point, it does not have clear answer yet, and different options are used in practice, such as a 0 value or the mean over the whole image.[10]

In order to formalize these ideas and clarify what, exactly, we will be referring to when talking about *perturbations*, we introduce the following definitions.

Definition 2.22. *(Segmentation)* Let $x \in \mathbb{R}^{n \times m \times d}$, $\mathbb{M}$ a finite subset of $\mathbb{N}$ and $G := (V, E)$ a graph defined by $V := \{1, \ldots, n\} \times \{1, \ldots, m\}$ and $E := \{(v_i, v_j) : |v_i - v_j| = 1\}$. Let h_x be a function:

$$h_x : V \to \mathbb{M}$$

satisfying that, for every $p \in h_x(V)$ the set $S_p := \{v \in V : h_x(v) = p\}$ is connected. We will refere to S a *superpixel* of x. The partition $S := \{S_p : p \in h_x(V)\}$ will be referred to as a *segmentation* of x.

Observation 2.23. Broadly speaking, *segmentations* attempt to determine connected image regions in a meaningful way. Algorithms used for building image *segmentations* make for a broad area of study in Computer Vision that is beyond the scope of our work.

Definition 2.24. *(Mask)* Let $x \in \mathbb{R}^{n \times m \times d}$, $r \in \mathbb{R}$, and let h_x be a function corresponding to a segmentation S of x. Without loss of generality, we can assume that h_x takes, exactly, the values $\{1, \ldots, l\}$ for some $l \in \mathbb{N}$. Let $m_{x,h_x,r}$ be a function:

$$\begin{aligned} m_{x,h_x,r} : \{0,1\}^l &\to \mathbb{R}^{n \times m \times d} \\ x' &\to m_{x,h_x,r}(x') \end{aligned}$$

defined by:

$$m_{x,h_x}(x')_{i,j,k} \begin{cases} x_{i,j,k} & \text{if } x'_{h(i,j)} = 1 \\ r & \text{otherwise} \end{cases}$$

For each $x' \in \{0,1\}^l$, we will refer to the output image $m_{x,h_x}(x')$ as x masked by x' via S.

Observation 2.25. Intuitively, this definition states that, given a segmentation of an image that determines l different *superpixels* in it, *perturbations* in the image will consist in setting the values of a whole *superpixel* to a fixed value r. And this setting, allows us to consider perturbations as points in $\{0,1\}^l$, where the ith component of a point being 0 indicates that the corresponding *superpixel* is perturbated.

Observation 2.26. The relationship between images and its masks via a given segmentation is clearly not surjective. However, it is injective and, thus, provides a bijection:

$$m_{x,h_x,r} : \{0,1\}^l \to m_{x,h_x,r}(\{0,1\}^l)$$

[10] In our practical experiments, we have explored several alternatives, including not assigning a single value to all modified components but, instead, *shuffling* its values or assigning them random values. All options have resulted in near-equal effects and, thus, we do not believe that, in practice, the alternative value that is set on the modified component plays a relevant role.

Furthermore, if we considered both sets as metric spaces equiped with the euclidean distance, then the map $m_{x,h_x,r}$ defines an isometry between them.

The above observation provides a key insight into the new approach we are about to introduce. Broadly speaking, perturbations as we have defined them, allow us to work in a discrete metric space with a reduced number of dimensions where, the notion of *locality* has little to do with a neighborhood of the image point in $\mathbb{R}^{n\times m\times d}$, which hardly has any meaning for humans but, instead, the same image with few of its regions *deleted*[11].

In this context, our question becomes the following: given a function f and an image x, how can we build a linear approximation $\hat{f}$ in $\{0,1\}^l$ that locally approximates $f \circ m_{x,h_x,r}$ around $(1,\dots,1)$? If we are able to build a *meaningful* one, we will be then able to apply, analogously as we did with gradient-based methods, proposition 2.11 and, thus, obtain attributions over a vector of *superpixels*. However, how can we build such linear functions so that it is *somehow meaningful* for our purpose?

The LIME method

The *LIME* (Local Interpretable Model-agnostic Explanations)[16] technique provides a first attempt towards building the linear approximation we introduced above.

Its approach consists in uniformly sampling points in $\{0,1\}^l$ in order to build a design matrix X, and then fitting a linear model for these samples with a weighted version of the *least-squares function*. Specifically, if $f : \mathbb{R}^{n\times m\times d} \to \mathbb{R}$ is the function that is trying to be explained around a point $x' \in \mathbb{R}^{n\times m\times d}$ and S is a segmentation of x' given by $h_{x'}$, whose codomain is $\{1,\dots l\}$, let $\hat{f} : \{0,1\}^l \to \mathbb{R}$ be the linear approximation of $f \circ m_{x',h'_{x'},r}$ that is trying to be estimated. Then, the proposed cost function has the following form:

$$\mathcal{L}(f \circ m_{x',h_{x'},r}, \hat{f}, X, \pi_{x'}) := \sum_{x\in X} \pi_{x'}(x)(\hat{f}(x) - f(m_{x',h_{x'},r}(x)))^2 \tag{2.2}$$

Where $\pi_{x'}(x)$ is a non negative decreasing function of the euclidean distance between x and x', the point we're doing attributions on, that will be, in general, $(1,\dots,1)$. The authors suggest the use, for a given positive scalar σ, the function:

$$\pi_{x'}(x) := exp(-\frac{|x-x'|^4}{\sigma}) \tag{2.3}$$

Which is referred to as the *exponential kernel*[12]. This function is used in order to encourage the function $\hat{f}$ to have increased accuracy near closest points to x'.

Observe that, $\pi_{x'}(x) \geq 0$ and it does not depend on the parameters of $\hat{f}$. Therefore, if we write $\hat{f}(x)$ as $\sum_{i=0}^{l} w_i x_i + b$, as long as we sample, at least $l+1$ points in $\{0,1\}^l$ and we assume linear independence between columns in X, the solution to the optimization problem:

$$\underset{w_1,\dots,w_l,b\in\mathbb{R}}{\operatorname{argmin}} \ \mathcal{L}(f, \hat{f}, X, \pi_{x'}) \tag{2.4}$$

[11] The notion of actually deleting pixels in an image actually makes no sense, as pixels cannot disappear but, instead, change its value. However, by setting entire regions of an image to a fixed value, the goal is to eliminate the semantic content of that region and simulate its absence.

[12] In this case, evaluated in the euclidean distance between x and x'

can be found by simply modifying the classical *normal equations* in the following way: if H denotes a diagonal matrix where its ith diagonal element is the image of h_x at X's ith row, then the vector of parameters w of our linear model can be found by the calculation:

$$w = (X^t H X)^{-1} X^t H f(m_{x',h_{x'},r}(X) \tag{2.5}$$

where we have abused notation and denoted with $f(m_{x,h_x,r}(X))$ the vector resulting of applying row-wise, $f \circ m_{x,h_x,r}$ to matrix X.

However, this approach rises, at least, two major questions:

- Does this approximation technique satisfy any desirable properties for our goal?
- Are those dependant of the choice of $\pi_{x'}$? And if so, how can we make a *non-arbitrary* choice for it?

Shapley Values and the SHAP technique

At the beginning of this chapter, with gradient-based techniques, we mentioned that attribution techniques generally draw upon the problem of *cost-sharing*, in Game Theory. The setting we have provided with perturbation-based methods is analogous to the one of *Finite Coalitional Games*. Broadly speaking, these attempt to determine, how the cost of some task (i.e. a function f) should be split among those individuals who take part on it (i.e. non-zero input components given a point in $\{0,1\}^l$), so that shares among participants (i.e. coefficients of a linear model) satisfy some axioms.

We will soon see that, by considering a set of four simple axioms which seem quite adequate for our task in hand, there is a unique solution for the problem. In fact, we will see that they mostly consists on adaptations to the properties introduced with *Gradient-Based Techniques* to our new framework, which no longer makes use of a *reference point*. Before stating these axioms, we introduce some notation to simplify all the following statements.

Given $x \in \mathbb{R}^{n\times m\times d}$, $r \in \mathbb{R}$, a segmentation S given by a function h_x taking values in $\{1,\dots,l\}$, its corresponding mask function $m_{x,h_x,r}$, and a function $f : \mathbb{R}^{n\times m\times d} \to \mathbb{R}$:

- For every subset $I \in \mathcal{P}(\{1,\dots,l\})$, $v(I)$ will denote a point in $\{0,1\}^l$ whose only non-zero components are those indexed by values present in I. Formally:

$$v(I)_i = \begin{cases} 1 & \text{if} \quad i \in I \\ 0 & \text{otherwise} \end{cases}$$

  That is, we refer to $v(I)$ as a point in $\{0,1\}^l$ corresponding to an image whose only *non-perturbed pixels* are those such that h_x assigns them a value present in I. Note that v clearly is is a bijective function between $\mathcal{P}(\{1,\dots,l\})$ and $\{0,1\}^l$.

- For every subset $I \in \mathcal{P}(\{1,\dots,l\})$, we will write $f_x(I)$ to denote $(f \circ m_{x,h_x,r} \circ v)(I)$. That is, f applied to the perturbed image $m_{x,h_x,r}(v(I))$.

The goal of this notation is to allow us to refer to perturbed images as *sets of non-perturbed superpixels*, for which we can evaluate f on. This allows us to smoothly adapt the concepts from *Coalitional Games* that we are about to consider.

In the context of *Coalitional Games*, *Cost Sharing* is an area of study whose goal is to measure the *impact* of the presence of an agent in the outcome of task. Formally, the task f is defined as a function $\mathcal{P}(S) \to \mathbb{R}$ for some finite set S. Intuitively, S represents a set of players, and the goal is to determine how each individual player contributes to the outcome of f. In our case, our task is f, whose domain is not a power set and, instead, has a fixed number of components. With the perturbation framework we have introduced, we have simply built an artifact that allows us to refer to a given image x as a set of superpixels, whose elements' absence is defined a perturbation of them. Thus, for a given image, we can naturally refer to its perturbations as sets and, with these, consider a locally discretized version of f, f_x, that allows us to adapt the framework from *Coalitional Games*.

This framework allows us to introduce *Shapley Values*. *Shapley Values* provide a solution to the *Cost Sharing* problem by, given a real function f whose domain is $\mathcal{P}(\{1,\dots,l\})$, *Shapley Values*, defining a mapping:

$$\phi_f : \mathcal{P}(\{1,\dots,l\}) \to \mathbb{R}^l \tag{2.6}$$

that satisfies a set of axioms that we will soon state. Observe that this setting matches exactly the one we defined for attributions at the beginning of the chapter. With the only difference being that we are making assumptions on the domain of f and that is something we have already addressed. Furthermore, we will see that *Shapley Values*'s axioms allow us to obtain the same result by either directly defining ϕ_f as our attributions or as a vector of coefficients for a linear model as the one considered by the *LIME* technique.

Given a set $A \in \mathcal{P}(\{1,\dots,l\})$, we will define $\phi_f(A)$ as the vector of coefficients of the linear model in $\{0,\dots,1\}^l$. As we will see, this will be equivalent as simply defining an attribution function $F_f(x) := \phi_{f_x}(\{1,\dots,l\})$.

Now, we proceed to announce the set *Shapley Values*' axioms. Let $f : \mathbb{R}^{n\times m\times d} \to \mathbb{R}$, $x \in \mathbb{R}^{n\times m\times d}$ and $r \in \mathbb{R}$. Let S be a segmentation of x, given by a function h_x taking values in $\{1,\dots,l\}$, and let $m_{x,h_x,r}$ be its corresponding mask function. Following the notation we recently introduced, the function we will be approximating is f_x, i.e. $f \circ m_{x,h_x,r} \circ v$. We state the following axioms on ϕ_{f_x}:

Axiom 2.27. *(Efficiency)*[13]

$$f_x(\{1,\dots,l\}) - f_x(\emptyset) = \sum_{i=1}^{l} \phi_{f_x,i}(\{1,\dots,l\}) \tag{2.7}$$

This property is analogous to *Completeness*, which we considered with Gradient-Based techniques. It asks attributions to add up to the f_x's output at $\{1,\dots,l\}$, which corresponds

[13]In *Coalitional Games Theory* it is assumed that the task f satisfies $f(\emptyset) = 0$. Thus, the term $f(\emptyset)$ does not appear when announcing the *Efficiency* axiom. However the prove about the fact that *Shapley Values* satisfy this axiom, proves, indeed, that it satisfies our version of it. Thus, we can simply avoid the a assumption with the simple modification of the axiom.

to the full image, minus f_x's output at $\emptyset$, which corresponds to a fully perturbed image. It could be argued that the empty set plays here a role analogous to the reference point's from Gradient-Based techniques. In [10] they name this axiom *Local Accuracy*. The reason is that, if we interpret the components of ϕ_{f_x} as coefficients of a linear model $\hat{f}$ in $\{0,1\}^l$, taking the form $\hat{f}(z) := \sum_{i=1}^{l} \phi_{f_x,i} z_i + f_x(\emptyset)$ for $z \in \{0,1\}^l$, then this axiom requires $\hat{f}$ to be exact at $(1,\ldots,1)$ i.e. the complete image.

Axiom 2.28. *(Carrier)* If there is a set $D \subset \mathcal{P}(\{1,\ldots,l\})$ such that:

$$f_x(C) = f_x(C \cap D) \text{ for all } C \in \mathcal{P}(\{1,\ldots,l\})$$

Then, for all $k \in \{1,\ldots,l\}$ such that $k \notin D$:

$$\phi_{f_x,k}(E) = 0 \text{ for all } E \in \mathcal{P}(\{1,\ldots,l\}) \tag{2.8}$$

This axiom plays a role analogous to *Dummy*'s in Gradient-Based Methods. It is, in fact, called *Dummy* by [26] and [17]. Intuitively, it states that components in ϕ_{f_x} corresponding to elements that don not contribute to f_x's output should be zero.

Now, observe that, the two axioms together that we have announced so far, justify an observation that we anticipated earlier: with *Shapley Values*, either defining attributions as ϕ_f directly or as coefficients of a linear model in $\{0,1\}^l$ and then using proposition 2.11 yields, indeed, the same result.

Axiom 2.29. *(Symmetry)* If there are two elements $i,j \in \{1,\ldots,l\}$ such that

$$f_x(C \cup \{i\}) = f_x(C \cup \{j\}) \text{ for all } C \in \mathcal{P}(\{1,\ldots,l\})$$

Then, for all $E \in \mathcal{P}(\{1,\ldots,l\})$:

$$\phi_{f_x,i}(E) = \phi_{f_x,j}(E) \tag{2.9}$$

This axiom is completely analogous to the *Symmetry-Preserving* property we considered in the *Gradient-Based* technique *Integrated Gradients*. It basically requires elements that have an *equal role* in f_x's outcome, to have equal an equal value in its corresponding components ϕ_{f_x}.[14]

Axiom 2.30. *(Linearity)* If there exists two functions: g,h with the same domain and codomain as f such that $f = g + h$, then, for all $E \in \mathcal{P}(\{1,\ldots,l\})$:

$$\phi_{f_x}(E) = \phi_g(E) + \phi_h(E)$$

[14] That is no coincidence as, in fact, Lloyd Shapley was the original author for both techniques and defined the former as an extension of the latter. Justifying this relationship would involve exploring further concepts in *Game Theory* that are beyond the scope of this work.

We also announced a linearity-preserving property for gradient-based methods, and this axiom is self-explanatory.

Observation 2.31. The four axioms we described are the classical ones as described in the original work of Lloyd Shapley [20]. However, after this work was published, there has been some discussion relating alternative axioms. Specifically, it is proved in [8] and [10], that the following axiom implies *Linearity* and *Carrier* (proved by [8]), and *Symmetry*(proved by [10]):

Let f_1, f_2 be two functions with the same domain and codomain as f_x and $i \in \{1, \ldots, l\}$, such that for every set $S \in \mathcal{P}(\{1, \ldots, l\} \setminus i)$:

$$f_1(S \cup \{i\}) - f_1(S) \geq f_2(S \cup \{i\}) - f_2(S)$$

$$\text{Then, } \phi_{f1}(\{1, \ldots, l\}) \geq \phi_{f2}(\{1, \ldots, l\})$$

This axiom is interesting as it allows for comparison of magnitudes of *Shapley Values* components between different functions: roughly speaking if an input component of one of the function *always* contributes more to an increase of the function, that function's *Shapley Value* in that component will be larger.

We now proceed to announcing the main result of this section:

Theorem 2.32. *(Shapley Theorem) Given a function* $f : \mathcal{P}(\{1, \ldots, l\}) \to \mathbb{R}$*, there is exactly one mapping:*

$$\phi_{f_x} : \mathcal{P}(\{1, \ldots, l\}) \to \mathbb{R}^l$$

that satisfies Axioms 1, 2, 3 and 4. It is given, for each $S \in \mathcal{P}(\{1, \ldots, l\})$ *by:*

$$\phi_{f_x,i}(S) := \sum_{C \in \mathcal{P}(\{1,\ldots,l\} \setminus \{i\})} \frac{|C|!(l - |C| - 1)!}{l!} (f_x(C \cup \{i\}) - f_x(C))$$

Observation 2.33. The expression above can be considered as a sum over:

- All possible orderings of the elements in $\{1, \ldots, l\}$
- All possible subsets for each ordering

that assigns an uniform probability to each different case. [3], [26], name the quantity $(f_x(C \cup \{i\}) - f_x(C))$ the *marginalized contribution* of $\{i\}$ with C and argue that Theorem 2.32 simply calculates the expected value of the marginalized contribution over the uniform distribution defined above.

Proof. We will provide a sketch of the proof. The first part consists on checking that the definition given by Theorem 2.32 is, satisfies Axioms 1 to 4, this fact can be easily done and simple proofs can be found in [17].

We will focus on the claim about uniqueness. Given a function $f : \mathcal{P}(\{1,\dots,l\}) \to \mathbb{R}$, for $S, T \in \mathcal{P}(\{1,\dots,l\})$, define:

$$f_T(S) = \begin{cases} 0 & \text{if } S \subset T \\ 1 & \text{otherwise} \end{cases} \tag{2.10}$$

In order for an attribution map F_{f_T} to satisfy *Carrier* and *Symmetry* Axioms, it must be defined as:

$$F_{f_T}(S) = \begin{cases} 0 & \text{if } S \subset T \\ 1/|T| & \text{otherwise} \end{cases} \tag{2.11}$$

Now, we will prove that any function f' with the same domain and codomain as f can be written uniquely as a linear combination of functions as the ones we just described:

$$f' = \sum_{T \in \mathcal{P}(\{1,\dots,l\})} \lambda_T f_T$$

for some $\lambda \in \mathbb{R}^{2^n}$. Since f' has 2^n components, it suffices to see that the elements in the set $\{f_T\}_{T \in \mathcal{P}(\{1,\dots,l\}}$ are linearly independent. Assume there is some $\lambda \neq 0$ such that:

$$0 = \sum_{T \in \mathcal{P}(\{1,\dots,l\})} \lambda_T f_T$$

Let S be a set in $\mathcal{P}(\{1,\dots,l\})$ with minimum cardinality such that $\lambda_S \neq 0$. Observe that, for every set $R \in \mathcal{P}(\{1,\dots,l\})$, $R \not\subset S$, and therefore $f_R(S) = 0$ by definition 2.10. Thus, we obtain:

$$\lambda_S = \sum_{T \in \mathcal{P}(\{1,\dots,l\})} \lambda_T f_T(S)$$

Which yields a contradiction, since we assumed $0 = \sum_{T \in \mathcal{P}(\{1,\dots,l\})} \lambda_T f_T(S)$.

Now, it is clear that, if F_f satisfies *linearity* then it must be written as a unique linear combination of attributions corresponding to scaled versions of functions defined in 2.11. It is clear that, following the notation in 2.11, for each pair of sets $S, T \in \mathcal{P}(\{1,\dots,l\})$ the *efficiency* axiom enforce $F_{\lambda_T f_T} = \lambda_T F_{f_T}$. Thus, F_f admits a unique expression as a sum of such functions, which can only be the one determined by *Shapley Values*.

□

Relationship with LIME and computational considerations

At the beginning of this section we considered two questions about the *LIME* technique. Roughly speaking, we asked ourselves about: (1) which properties did the *LIME* technique satisfy and, (2), how could make a non arbitrary choice for h_x. Let us follow the notation and examples used in that chapter.

Theorem 2.32 provides a straightfoward answer for (1): uniqueness of shapley values implies that, as long as the sample size and choice for h_x does not make *LIME*'s linear coefficients coincide with *Shapley Values* [15], they will not satisfy all *Shapley Values*' axioms.

A more hopeful view can be achieved by turning to (2). In [10], the authors prove that, with the right choice of $h_{x'}$ and considering as a sample all possible points in the the given discrete input set, applying the *LIME* technique is in fact, equivalent, to calculating *Shapley Values*.

Proposition 2.34. Consider the notation introduced in *LIME's* subsection, with $\{0,1\}^l$ being the domain of the considered functions and $x' := (1,\dots,1)$ the considered point. The only choice of $\pi_{x'}$ that allows the LIME technique's coefficients to satisfy *Shapley Values'* axioms is:

$$\pi_{x'}(x) = \frac{|v(x')|-1}{\binom{|v(x')|}{|v(x)|}|v(x)|(|v(x')|-|v(x)|)}$$

for every $x \in \{0,1\}^l$, and considering the design matrix X to have as rows, all possible values in $\{0,1\}^l$.

This proposition opens the door for some practical considerations. *Shapley Values* have one major disadvantage and it is having computational complexity exponential on the cardinality of the set considered. In practice, with Convolutional Neural Networks which are also expensive to evaluate, that translates into exact computations of Shapley Values, with a segmentation of an image consisting in 10 superpixels, taking up to $10-15$ minutes on a regular laptop. Needless to say, segmentations with amounts of superpixels much higher quickly becom intractable. This is clearly undesirable and thus, several alternatives have been considered regarding how to compute *approximate Shap Values*.

Approximations are generally based on making some further assumptions about the function considered treating the computation as an expected value approximation. However, it is not clear whether these apply to Convolutional Neural Networks. Instead, the *LIME* technique avoids these hypothesis ans simply takes a *sampling-based* approach. Although Proposition 2.34 ensures that these approximations will fail to satisfy *Shapley Values*' axioms, it could be argued that it *somehow justifies* that these approximations are *headed* in the right direction. However, these are just speculative claims, and further theoretical results would be necessary in order to fully justify them.

[15] which is obviously going to be false, in general

Chapter 3

Practical Experiments

In this chapter, we will implement the main methods that we have studied theoretically. By doing so, we will try to get further insights in the behavior of these techniques.

We will start by briefly introducing the general setting of our experiments and then, we will proceed to study separately, analogously as we did in the previous chapter, *Gradient-Based* techniques and *Perturbation based* techniques.

All the code used for these experiments is available at *https://github.com/gbraso/attribution_methods*.

3.1 Frameworks Used and General Setup

Keras* and *Tensorflow

The programming language of choice for this work has been *Python*. *Python* has several libraries designed, specifically, for scientific programming and processing large *datasets*, which can significantly simplify our tasks in hand.

Specifically, this programming language allows us to work with two of the main libraries used for *Deep Learning*: *Keras* [1] and *TensorFlow* [11]. The latter is a framework that is essentially designed to do operations on *computational graphs* and and intensive computation in n-dimensional matrices [1]. In particular, it has built-in *Automatic Differentiation*[2] for these graphs. Thus, it allows us to avoid manually deriving formulas such as the ones involved in *backpropagation*. We note that, without tools such as *TensorFlow*, *modern neural architectures* would simply not be feasible to implement in practice, as they would require the practitioner to manually compute up to several thousands of millions of derivatives for each iteration step in *Gradient-Descent*.

[1] which are often called *tensors* in *computer science*, thus, the name *TensorFlow*.

[2] That is, once we define an expression, it is able to automatically calculate its symbolic derivative in an efficient manner.

Keras is a *high-level* library[3] that is built on top of *TensorFlow*, and allows for quick implementations of *Neural Networks*. For instance, it considers *layers* as *objects*, and allows the definition of *neural networks* as a sequence of such objects. However, *Keras* does not directly allow some of the operations that we need to use in order to implement *Gradient-Based* techniques. Fortunately, *Keras* and *Tensorflow* functionalities can naturally coexist and interact with each other: for instance, we are able to compute derivatives of a *network* defined with *Keras*, through *TensorFlow* in a straightforward way.

ImageNet and the VGG16 network

Some years ago, *Deep Convolutional Networks* protagonized a historical breakthrough at ILSVRC (ImageNet Large Scale Visual Recognition Competition)[18]. The dataset involved in this competition, *ImageNet*, is generally considered as the standard benchmark for *general-purpose* computer vision tasks. It consists in a total of 1.4 milions images, divided in 1.2 milions of images used for training *trainning set*, and 200000 used to evaluate their accuracy (*test set*

These images are labelled through total of 1000 classes. Some of these classes can be highly *fine-grained*. For instance, 120 of them correspond to different breeds of dogs.

In 2014, the winning team presented a network named VGG16 [21], and achieved a 70.4% accuracy on the test set. This network has a total of 21 layers, of which 13 are *convolutional*, 3 are *fully connected* and the remaining 5 are max pooling layers.

The VGG16 network has a total of 1.38 millions of *trainable parameters*. In a regular computer, training such network on such *large* dataset, would simply not be possible. Luckily for us, this network, with the parameters trained by its creators, can be downloaded by the general public. In fact, *Keras* contains a module where it is readily available, and this has allowed us easy access to it.

General Approach and Visualizations Employed

Quantitatively evaluating attributions' performance is an intricate task. Generally speaking, we do not have direct access to any *ground truth* that we can compare attributions to. More specifically, when attributions results do not match our intuitive expectations, it is not clear if the problem is related to the *network* itself, or the attribution technique. Furthermore, it is not even clear if there is any problem at all, since attributions might look *strange* to us, but accurately perceive network behavior.

Some approaches have been introduced in order to evaluate how *meaningful* attributions outputs are. In [**grad_eth**], its authors propose to, given an array of attributions, perturbate the values of pixels in the image in a sequence sorted by the values of its corresponding attribution components, and observe how the output of the network changes for this sequence. Intuitively, we would expect a rapid drop in the class- probability for which attributions were computer

[3] by *high-level*, we refer to the fact that it allows for more *abstract* operations.

However, we will not be reproducing this approach but, instead doing a qualitative analysis of the main techniques we have studied, in order to get a better understanding of how they work, in practice. Specifically, we will be mainly tackling two issues: the relevance of the layer in the network at which attributions are computed, and the role that the reference point plays in *Gradient-Based* techniques.

In order to visualize attributions, we will be plotting *heatmaps* corresponding to them. To do so, given a 3-dimensional matrix of attributions, we will add its elements through its third dimension, in order to assign a value to each element in the spatial domain. We will then will consider two alternatives:

- If the attributions have been computed at the input layer of the network, (i.e. at pixel level) we will simply plot the image corresponding to the array obtained in the anterior step, with a *colormap* applied to it.
- If the attributions have been computed at the an intermediate layer of the network, we will simply plot the image corresponding to the array obtained in the anterior step, then, *resize* it to match the image dimensions, and plot the translucid corresponding heatmap on top of the image preprocessed to be black and white .

In the first case, plotting the heatmap on top of the image yields unclear visualizations. In both cases, the *colormap* we have considered, is *red* for attribution values that are positive (i.e. with high impact), translucid for values close to zero, and blue for negative values (i.e. with high negative impact).

3.2 Gradient-Based Techniques

On the relevance of the layer at which attributions are computed

In all the articles we have reviewed, except for [19] and [13], *Gradient-Based* techniques are applied at pixel level, that is, the gradient of the network is computed with respect to the nodes of its input layer. We have empirically observed that this approach is generally able to produce *heatmaps* that distinguish *shapes* and *objects* from the image background.

However, when there are several objects present in the image, we have empirically observed that attributions are not class-specific: that is, regardless of the output node whose gradient we compute, attributions look very similar, and highlight parts of the image that seem to be non-relevant for the predicted class. We can observe this fact in figure [COMPLETA], where, regardless whether we compute the gradient corresponding to the class *french bulldog* o *egyptian cat*, the *heatmaps* we obtain highlight both objects. This fact has been previously observed in [19] and [28].

Another observation about computing attributions at pixel level, is made in [2]. In this article its authors do the following experiment: starting from the last layer in the network, they set parameters to random values, layer by layer, in a *cascade* fashion. They observe that this process minimally affects the attributions obtained, suggesting that these, in fact,

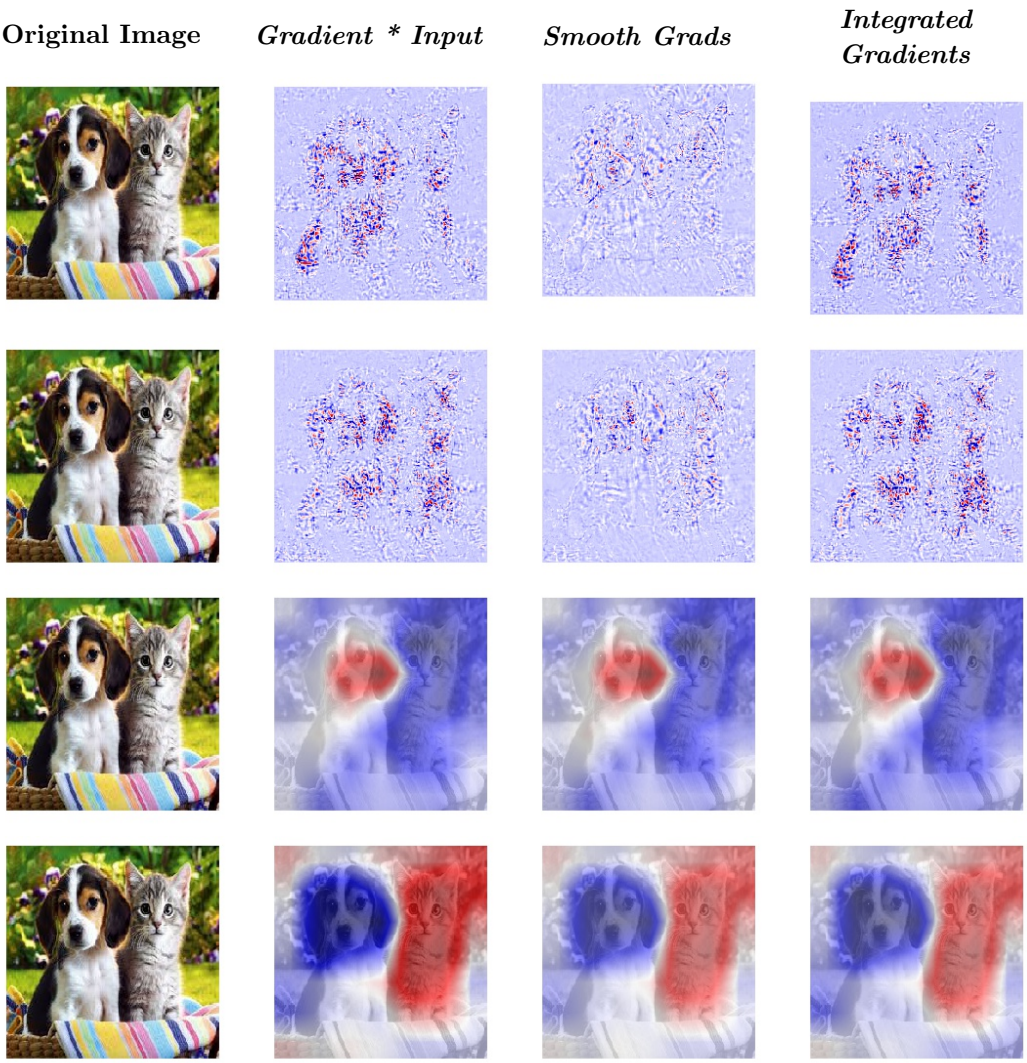


Figure 3.1: We compare how different attribution methods *explain* different classes. Rows 1 and 3 correspond to attributions for the class *beagle*, which is the one that recieves maximum probability. Rows 2 and 4, correspond to *attributions* to the class *egyptian cat*. Rows 1 and 2 correspond to attributions at pixel level, and the last two rows correspond to attributions computed at layer 19 in VGG16, the last convolutional layer. In this example, only attributions at the higher level achieve produce a meaningful heatmap.

any *information* specific to the parameter values of the network in hand. We provide an example of this fact in figure 3.3 in the appendix of this work.

Our claim[4], is that attributions at *higher* layers in the network are both more sensitive to changes in network's parameters (see Figure 3.4 in consecutive layers and class-specific.

[4]Which is solely backed by intuition and experimental results.

In order to back this claim, we have computed attributions for several images, at the last convolutional layer of the network VGG16. This operation yields a 3 dimensional matrix of size $7 \times 7 \times 512$. We then add all components in the third dimension in order to obtain a *heatmap* of size 7×7. We then obtain a *heatmap* of the same size of the image by proceeding as described in the introduction of this chapter. Ilustratory examples of these claims can be found in figure 3.1. Additional examples can be found through interactive visualizations provided in our *github* repository.

Intuitively, as already mentioned in the previous chapter, we believe that attributions at *pixel level* account for changes that have, conceptually, *little meaning*. Instead, if we take into account the results presented in [12], which suggest that higher level layers encode more *abstract features*, then it makes sense that changes in the *intensity* in which these features are *percieved by the network* make for more meaningful attributions. As an example of how this reasoning might materialize, we could consider that in order to *explain why* a network *sees* a cat, the answer should be approached regarding how *clearly it sees* ears, eyes and noses, and not by how sensitive it is to changes in values of individual pixels. Still, this is just *intuitive reasoning*.

It turns out, that this approach to building *heatmaps* from attributions in higher layers had been already proposed before we did these conclusions. It was first introduced in [19], in which the authors extend an idea first presented in [28].

3.3 Perturbation-Based techniques

Computing Shapley Values faster by getting advantage of intermediate layers

Arguably, the main drawback of computing Shapley Values is the computational burden it represents. Recall that, for a segmentation of an image in n superpixels, it requires $o(2^n)$ evaluations of the network. For *deep convolutional networks*, whose evaluation can be expensive, this translates into requiring several minutes to explain a single image on a regular laptop, which is clearly is undesirable.

However, by taking advantage of our knowledge of the network's structure, we can significantly reduce this computing time. In order to do so, we have computed attributions at the last convolutional layer of the network, which we recall, has size $7 \times 7 \times 512$. We have considered a trivial segmentation, where we consider, as superpixels, each *spatial location* in the layer, which consists in 512 values. The perturbations we have considered consist in setting all the values in a superpixel to 0. Once we have computed the Shapley values for these *superpixels*, we obtain a heatmap by proceeding as before. An example of the results of this experiment can be seen in Figure 3.2.

Observe that this procedure can be undestood as considering a *grid-like* segmentation of the input image in $7 \times 7 = 49$ superpixels consisting in equally sized squares, corresponding to the receptive fields of the units in the last layer. And then, instead of perturbating these pixels directly, perturbating the values in units corresponding to the last convolutional layer.

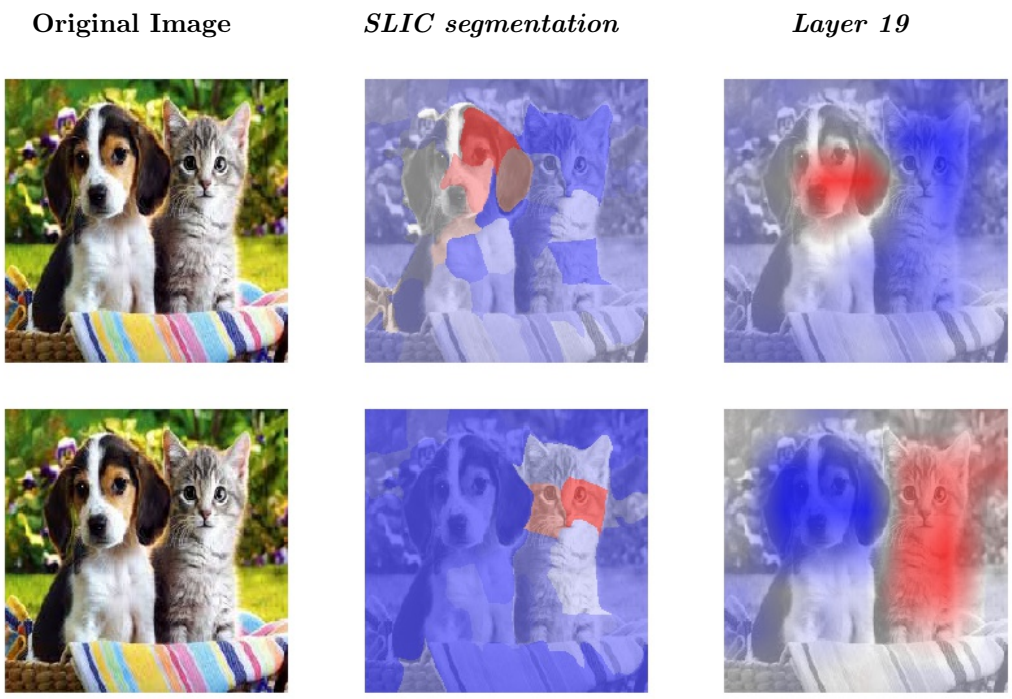


Figure 3.2: Our setting is the same as in Figure 3.1. The second column corresponds to computing Shapley Vales with a segmentation, at pixel level, obtained with a standard technique named *Slic*. The third column corresponds to computing them at the last convolutional Layer of the network. Rows 1 and 2 correspond to explanations of, classes *Beagle* and *Egyptian Cat*, respectively.

We note that computing Shapley Values at higher levels allows evaluations of the network to be significally less *computationally expensive*, by avoiding several layers of computation. In figure 3.2, we have computed attributions directly at layer 19 thus, skipping the first 18 layers of the network, which are also those that involve the most operations, despite have less parameters. In this particular case, obtaining attributions at a higher layer took 16*s*, in contrast of obtaining them through a regular segmentation, which took over 19 minutes. It could be argued, however, that there is a drawback of this *trick*: it does not allow to obtain, directly, *sharp* segmentation.

It is also worth to note that segmentations obtained throguh *Shapley Values* at higher layers of the network are generally similar tot hose obtained through gradient methods. In general however, the latter involve significantly less operations and thus, are arguably a better option in practice. Anyway, the absence of both established theoretical connections between both sets of techniques and *quantitative* performance metrics makes these comparasions rather vague.

Conclusions

One of the main objectives of this work was to provide an extensive review of attribution techniques that have been proposed in recent years. Not only that, but understanding their connections and presenting them in a unified manner, if possible.

Attempting to do so, has been a remarkably challenging task, mainly for a reason: the lack of a unified set of properties and definitions regarding what an attribution *is*, and what it *should satisfy*. We have approached this task by: (1) considering Gradient-Based and Perturbation-Based techniques separately, and (2), in each case considering the *union* of most properties suggested by different articles. In both cases, however, it has turned out that there was a single method satisfying them all: *Integrated Gradients* among Gradient-Based techniques, and *Shapley Values* among Perturbation-Based techniques.

Nevertheless, we have empirically observed that these techniques do not always seems to *perform well*. This arises a question. How can we account for *good performance*?[5] It is natural to be inclined to favor those attributions producing the most *intuitive* heatmaps, but we should bear in mind that it is not our intuitions that we are trying to reproduce, but *those of the network* in hand. Since *these* are not available to us, in order to evaluate the network, we believe it is necessary to resource to *meaningful* theoretical results. Specifically, we believe that these should be given in regards of the *quality* of the approximation that attribution methods construct.

The theoretical properties that *Integrated Gradients* and *Shapley Values* satisfy, certainly seem necessary but hardly sufficient. We believe, thus, that progress in this field should be made in the direction of *searching* for additional desirable theoretical properties that could shed some light on what an attribution should satisfy.

These observations take us to our next point: should we aim to build a general purpose *theory* for attribution methods or, instead, should we restrict it to both the problem and algorithm in hand? We believe that the answer might be the latter. First of all, it is not clear whether *influence* in Computer Vision problems, for instance, has the same meaning as in other domains. Specifically, some assumptions can be done in Computer Vision that are likely not to make sense in other domains, for instance, the possibility to group several components in the input and assume that they *play a role together*, such as we did with *perturbation based* problems. Furthermore, neither is it clear that we can apply the same

[5] Arguably, accounting for *bad performance* can be done through observations such as that Gradient-Based methods, applied at lower layers, lack sensitivity to the parameters in the network.

techniques in *high-dimensional* spaces, such as those of images, that we could apply in other domains.

In regards of assumptions concerning the algorithm in hand, we believe that they can probably lead the way to meaningful results. More specifically, in *neural networks* getting advantage of *information* provided by the activations of units in intermediate layers, might be a promising line of research. In some, sense, we have exploited such information with our practical experiments, by computing attributions at higher layers of the networks. However, we believe that much *deeper* information can be extracted from intermediate layers. Luckily, there has already been some research in this direction [15].

To conclude, we believe that our main goals were achieved, as we now feel confident about our knowledge of current approaches to attributions applied to Deep Learning and Computer Vision. Furthermore, we believe that we are now in a position to resume our research into some encouraging directions and, hopefully, getting a little closer to *opening the black box of Deep Learning.*

Bibliography

[1] François Chollet. *Keras*. https://github.com/fchollet/keras. 2015.

[2] Julius Adebayo et al. *Local Explanation Methods for Deep Neural Networks Lack Sensitivity to Parameter Values*. 2018. URL: https://openreview.net/forum?id=SJOYTK1vM.

[3] Robert Aumann. *The Shapley Value*. The Hebrew University of Jerusalem, 1991.

[4] David Bau et al. "Network Dissection: Quantifying Interpretability of Deep Visual Representations". In: (2017).

[5] G. Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In: (1989).

[6] R. Fong and A. Vedaldi. "Net2Vec: Quantifying and Explaining how Concepts are Encoded by Filters in Deep Neural Networks". In: (2018).

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[8] Young H Peyton. *Monotonic Solutions of Cooperative Games*. "pp. 65–72". International Journal of Game Theory, 1985.

[9] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[10] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: (2017). Ed. by I. Guyon et al., pp. 4765–4774. URL: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

[11] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[12] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. "Inceptionism: Going Deeper into Neural Networks". In: (May 2015).

[13] Chris Olah et al. "The Building Blocks of Interpretability". In: *Distill* (2018). DOI: 10.23915/distill.00010.

[14] Tomaso A. Poggio et al. "Why and When Can Deep - but Not Shallow - Networks Avoid the Curse of Dimensionality: a Review". In: *CoRR* abs/1611.00740 (2016). arXiv: 1611.00740. URL: http://arxiv.org/abs/1611.00740.

[15] Maithra Raghu et al. "SVCCA: Singular Vector Canonical Correlation Analysis for Deep Learning Dynamics and Interpretability". In: (2017). Ed. by I. Guyon et al., pp. 6076–6085. URL: http://papers.nips.cc/paper/7188-svcca-singular-vector-canonical-correlation-analysis-for-deep-learning-dynamics-and-interpretability.pdf.

[16] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *CoRR* abs/1602.04938 (2016). arXiv: 1602.04938. URL: http://arxiv.org/abs/1602.04938.

[17] Elvin E Roth. *The Shapley Value. Essays in honor of Lloyd S. Shapley*. Cambridge University Press, 1988.

[18] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *Int. J. Comput. Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0816-y. URL: http://dx.doi.org/10.1007/s11263-015-0816-y.

[19] Ramprasaath R. Selvaraju et al. "Grad-CAM: Visual Explanations From Deep Networks via Gradient-Based Localization". In: (Oct. 2017).

[20] L. S. Shapley. *A value for n-person Games*. "pp. 307–317". Contributions to the Theory of Games, 1953.

[21] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[22] Daniel Smilkov et al. "SmoothGrad: removing noise by adding noise". In: *CoRR* abs/1706.03825 (2017). arXiv: 1706.03825. URL: http://arxiv.org/abs/1706.03825.

[23] Sho Sonoda and Noboru Murata. "Neural Network with Unbounded Activations is Universal Approximator". In: *CoRR* abs/1505.03654 (2015). arXiv: 1505.03654. URL: http://arxiv.org/abs/1505.03654.

[24] Jost Tobias Springenberg et al. "Striving for Simplicity: The All Convolutional Net". In: *CoRR* abs/1412.6806 (2014). arXiv: 1412.6806. URL: http://arxiv.org/abs/1412.6806.

[25] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks". In: *CoRR* abs/1703.01365 (2017). arXiv: 1703.01365. URL: http://arxiv.org/abs/1703.01365.

[26] Eva Tardos. *Algorithmic Game Theory. Cost sharing - On the Shapley Value*. Cornell University, 2004. URL: https://www.cs.cornell.edu/courses/cs684/2004sp/ShapleyValue.pdf.

[27] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *CoRR* abs/1311.2901 (2013). arXiv: 1311.2901. URL: http://arxiv.org/abs/1311.2901.

[28] B. Zhou et al. "Learning Deep Features for Discriminative Localization". In: (June 2016), pp. 2921–2929. DOI: 10.1109/CVPR.2016.319.

Appendix

Additional Figures

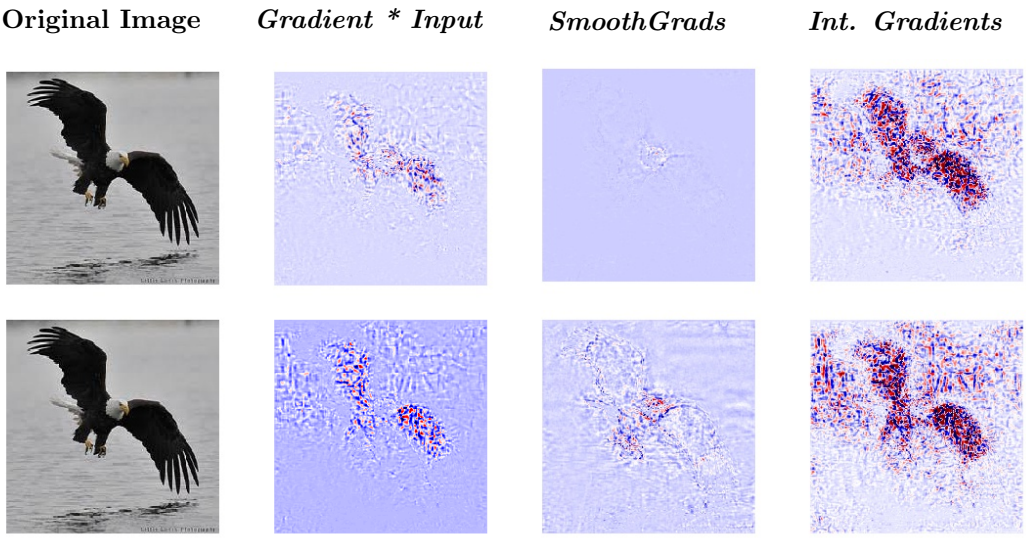


Figure 3.3: A given image in ImageNet whose class, *bold eagle* is predicted correctly by the network VGG16. Heatmaps in the top row correspond to attributions computed with the regular network. Those in the bottom row correspond to heatmaps obtained in attributions where we have set the parameters in the last four years in VGG16 to random values.

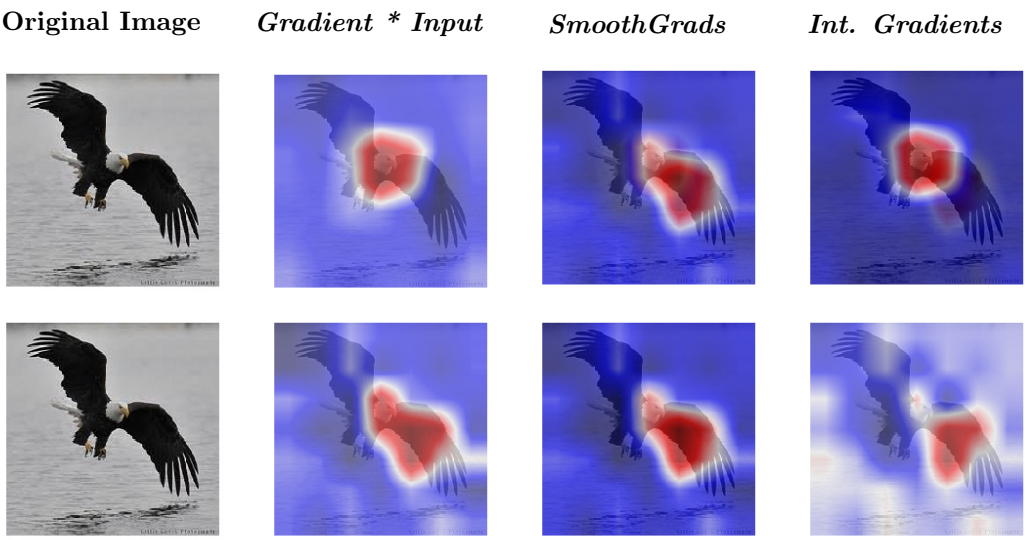


Figure 3.4: The same situation considered in figure 3.3 with attributions computed at the last convolutional layer of VGG16.