POLITECNICO DI TORINO

Department of Mathematical Sciences

Master of Science in
Mathematical Engineering

Master Thesis

Sanity Checks for Explanations of Deep Neural Networks Predictions

Supervisors:
Prof. Francesco Vaccarino
Dott. Antonio Mastropietro

Candidate:
Francesca Nuzzo

Academic year 2019-2020

A mia madre e mio padre
Ai miei nonni

Grazie
per aver sempre creduto in me
e aver reso possibile tutto questo.

Abstract

At the dawn of the fourth industrial revolution, the performance of Artificial Intelligence (AI) systems is reaching, or even exceeding, the human level on an increasing number of complex tasks. However, because of their nested non-linear structure, deep learning models often suffer from opacity and turn out to be uninterpretable black boxes. The lack of transparency represents a barrier to the adoption of these systems for those tasks where interpretability is essential, like autonomous driving, medical applications or finance. To overcome this drawback and to comply with the General Data Protection Regulation (GDPR), the development of algorithms for visualizing, explaining and interpreting deep neural networks predictions has recently attracted increasing attention. Paradigms underlying this problem fall within the so-called Explainable Artificial Intelligence (XAI) field. This class comprises a suite of methods and algorithms enabling humans to understand, trust and effectively manage the emerging generation of artificial intelligent partners. Over a relatively short period of time a plethora of explanation methods and strategies have come into existence, whose purpose is to highlight the regions of the input, typically an image, that are mostly responsible for reaching a certain prediction.

However, despite the significant performances, it remains the difficulty of assessing the scope and quality of results provided by such explanation methods. The goal of this thesis is to validate the explanations of deep neural networks predictions for the computer vision, generated by some state-of-the-art methods recently proposed. The experiments conducted in this work aim to answer to the following question: who assures us that the explanation provided by the method actually tell us reliably about what the network has learned to arrive at that decision? Along this line, a sanity check taken from the recent literature is described and experiments are performed to assess the sensitivity of explanation methods to model parameters. If one method really highlights the most important regions of the input, randomly reinitializing the parameters of the last layer of the network, then changing the output, the explanation given should change. Surprisingly, some of the methods proposed in literature are model independent and therefore fail the randomization test. By providing the same explanation even after the model parameter randomization, such methods are inadequate to faithfully explain the network prediction.

The reliability of explanation methods is a crucial aspect in tasks where visual inspection of results is not easy or the costs of incorrect attribution is high. The analysis conducted in this thesis aims to provide useful insights into developing better and more reliable visualization methods for deep neural networks, in order to gain the trust of even the most skeptical users.

When something is important enough,
you do it even if the odds are not in your favor.

Elon Musk

Contents

Chapter 1

Introduction

1.1 Artificial Intelligence

At the dawn of the fourth industrial revolution, we are witnessing a fast and widespread adoption of **Artificial Intelligence** (**AI**) by a wide range of businesses and industries, which contributes to accelerating the shift towards a more algorithmic society. Artificial Intelligence is a wide-ranging branch of computer science, concerned with building smart machines capable of performing tasks that typically require human intelligence.

AI-based technologies are increasingly being used in our daily life to make inference on classification or regression problems, such as movie recommendations of Netflix, friend suggestions on Facebook, neural machine translation of Google or speech recognition of Amazon Alexa. Artificial Intelligence is growing also into the public health sector and is going to have a major impact on every aspect of primary care. AI-enabled computer applications are helping primary care physicians to better identify patients who require extra attention and provide personalized protocols for each individual. In addition, increased AI usage in medicine not only reduces manual labor and frees up the primary care physician's time but also increases productivity, precision, and efficacy.

Not only, Artificial Intelligence offers tremendous potential also for industry. It is already making production more efficient, more flexible, and more reliable. One of the first trends to consider leveraging is predictive analytics. Through AI-based systems, this is helping companies forecast better solutions and decisions. Similarly, anomaly detection is being utilized to find opportunities and weak points in an organization to thrive and protect assets. Cybersecurity is also taking an AI approach with machine learning to get smarter at detecting threats. It is by virtue of these capabilities that AI systems are achieving unprecedented levels of performance when learning to solve increasingly complex computational tasks, making them pivotal for the future development of the human society. According to a Gartner survey, at the end of 2020, "the 85% of CIOs will try artificial intelligence programs through a combination of purchasing, development and outsourcing options". The AI market will grow to a 190 billion industry by 2025, according to research firm Markets and Markets, and even, PwC estimates that AI will contribute 15.7 trillion to the global economy by 2030. It predicts that most of this increase will be a result of stimulating consumer behavior, enhancing products and improving labor productivity. And again, Forrester predicts cognitive technologies such as robots, AI, machine learning, and automation will create 9% of new U.S. jobs by 2025, like robot monitoring professionals, data scientists, automation specialists, and content curators.

1.2 Introduction to XAI

Despite this success, in many application domains, high predictive power is not the only feature necessary to comply with user expectations. In practice, one of the main concerns in the use of AI-based systems is their *black-box* nature. Such models often lack transparency and turn out to be uninterpretable. Even if we understand the underlying mathematical theories, it is complicated, and often impossible, to get insight into the internal workings of the models and to explain their inference processes and final results in a human understandable and reconstructable way. So while AlphaGo or DeepStack can crush the best humans at Go or Poker, neither program has any internal model of its task: there is no mechanism to explain their actions and behaviour, and furthermore, there is no obvious instructional value to help humans improve. On the other hand, delegating decisions to black boxes without the possibility of an interpretation may be critical and can create discrimination and trust issues.

The nature of these decisions has spurred a drive to create algorithms, methods, and techniques to accompany outputs from AI systems with explanations. This is where the concept of **Explainable Artificial Intelligence** (**XAI**) comes into play. In order to avoid limiting the effectiveness of the current generation of AI systems, XAI proposes creating a suite of machine learning techniques that, as D. Gunning defines, produce more explainable models while maintaining a high level of learning performance, and enable humans to understand, appropriately trust, and effectively manage the emerging generation of artificially intelligent partners. For these reasons, XAI is a field widely acknowledged as a crucial feature for the practical deployment of AI models.

The future of AI is likely to depend on its ability to allow people to collaborate with machines to solve complex problems. Like any efficient collaboration, this requires good communication, trust, clarity and understanding, and XAI has precisely the aim of addressing these challenges.

1.2.1 XAI's landscape

The term "Explainable Artificial Intelligence" was first coined in 2004 by Van Lent et al., to describe the ability of their system to explain the behavior of AI-controlled entities in simulation games application. However, the eXplainable AI is not a new field: reasoning architectures existed to support complex AI systems since the 80s. But today, given the extent of the changes introduced by AI algorithms, the interpretability of AI algorithms has returned to the fore. The concept emerged with renewed vigor in late 2019 when Google, after announcing its "AI-first" strategy, announced a new XAI toolset for developers. This happened almost simultaneously with another important driving event in favor of XAI: the entry into force, in May 2018, of the *General Data Protection Regulation* (GDPR). In Article 22, the GDPR establishes that:

> *The data subject shall have the right not to be subject to a decision based solely on automated processing, including profiling, which produces legal effects concerning him or her or similarly significantly affects him or her.*

On the other side of the ocean, the US Department of Defense, through the DARPA *Defense Advanced Research Projects Agency*, responsible for the study of emerging technologies in the context of national security, started in 2017 an XAI program with an investment expected of 2 billions, which includes 11 projects and will continue running until 2021.

The problem of XAI has unleashed a fairly unprecedented tsunami of research in the last few years, as almost every major AI conference and workshop has targeted the problem, along with the emergence of conferences dedicated solely to it: FAT-ML Fairness, Accountability, and Transparency in Machine Learning, IJCAI Workshops on Explainable Artificial Intelligence, VISxAI Workshop on Visualization for AI Explainability. This year (2020) is flourishing by a wide range of dedicated workshops to the topic: CD-MAKE 2020 Workshop on Explainable Artificial Intelligence in Dublin, ETMLP 2020 International Workshop on Explainability for Trustworthy ML in Copenhagen, ACM FAT* Conference in Barcelona, for the first year in Europe.

Increasing interest in XAI has also been observed in the industrial community. Companies on the cutting edge of contributing to make AI more explainable include H2O.ai with its driverless AI product, Microsoft with its next generation of Azure, Azure ML Workbench, Kyndi with its XAI platform for government, financial services, and healthcare, FICO with its Credit Risk Models and many others.

1.3 Purpose & Outline of the Thesis

Despite the long history and a significant amount of work, making up good explanations is not trivial. The motivation for this thesis comes from the need to guarantee the reliability of explanation methods, a crucial aspect in tasks where visual inspection of results is not easy or the costs of incorrect attribution is high. The purpose of this thesis is to analyze some algorithms recently proposed in the literature in order to validate the explanations of deep neural networks predictions for computer vision.

More in detail, the main contributions of this thesis are:

- Propose an overview of the theory and the explanation methods that are suitable for deep neural networks analysis;
- Compare the results of such methods obtained thanks to the implementations provided by the Captum library for model interpretability;
- On the basis of the works of [Adebayo et al., 2018] and [Sixt et al., 2019], conduct some model parameter randomization tests for assessing the scope and quality of explanation methods;
- Analyze the class-discriminativeness of such algorithms, as a further criterion for choosing between methods.

The upcoming chapter aims to review the theoretical background of Deep Learning, starting from the definition of an Artificial Neural Network and the description of the techniques of its training. Chapter 3 describes the current state of explainable artificial intelligence, delves into the motivations which lead to the need for an explanation for black box systems and lists the main fields of application. The taxonomy presented in the chapter 4 concerns a comprehensive overview of methods proposed in the literature for explaining decision systems based on deep neural networks, some of which will be tested in the proceeding experiments. Chapter 5 describes the works of [Adebayo et al., 2018] and [Sixt et al., 2019] which inspired this thesis and provides a description of the most interesting theoretical insights. Chapter 6 looks in more depth at the recently proposed framework for interpreting predictions, SHAP, with his limitations and the proposed solutions. The thesis closes with chapter 7, in which the experiments and development possibilities are discussed.

1.4 Contribution

This thesis has been carried out as a research study conducted in Addfor SpA, a company that develops Artificial Intelligence Solutions for Engineering, based in Turin, with the supervision of Enrico Busto and Antonio Mastropietro.

The experiments have been conducted in collaboration with Rosalia Tatano.

The author thanks all the people mentioned in this section and the professor Francesco Vaccarino for their kind and supporting contribution throughout the thesis work.

Chapter 2

Background

In this chapter, important background concepts for the understanding of this thesis will be explained. Initially, broader machine learning terms will be introduced, followed by concepts and definitions specifically for the subset of machine learning challenges faced during this study, namely deep convolutional models.

2.1 Machine Learning

The emerging frontier of artificial intelligence is **Machine Learning** (**ML**): it enables machines to *learn* from past data or experiences, without being explicitly programmed, in order to make, through statistical analysis and pattern matching, classifications or predictions about the future. But what do we mean by learning? According to Tom Mitchell, professor of Computer Science and Machine Learning at Carnegie Mellon, a computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E. ML tasks are usually described in terms of how the ML system should process an example, that is a collection of *features* such as pixels in the image, which is part of a dataset. Some of the most common tasks that can be solved with ML include: classifications, regressions, transcriptions, machine translation, synthesis and sampling. In order to evaluate the abilities of a ML algorithm, we must design a quantitative measure of its performance: for tasks such as classification and transcription, we often measure the accuracy of the model, so the proportion of examples for which the model produces the correct output, or equivalently, the error rate, so the proportion of examples for which the model produces an incorrect output. By what kind of experience they are allowed to have during the learning process, ML algorithms can be broadly categorized into three classes:

- *Supervised learning*: it is the most popular paradigm for performing ML operations. It is widely used for data where there is a precise mapping between input-output data: image classification, facial recognition, email spam detection and so on. The dataset is labeled, meaning that the algorithm identifies the features explicitly and carries out predictions or classification accordingly. The term supervised learning originates from the view of the target being provided by an instructor or teacher who shows the ML system what to do. Some of the algorithms that come under supervised learning are as follows: Linear and Logistic Regression, Random Forest, Support Vector Machines, Artificial Neural Networks;

- *Unsupervised learning*: for these methods, the data is not explicitly labeled into different classes, that is, there are no labels and no instructor or teacher, so the algorithm must learn to make sense of the data without this guide. The model is able to learn from the data by finding implicit patterns: densities, structures, similar segments and other similar features. Some of the important algorithms that come under unsupervised learning are: Clustering, Principal Component Analysis and Anomaly Detection;

- *Reinforcement learning*: it covers more areas of AI which allows machines to interact with their dynamic environment in order to evaluate the ideal behavior in a specific context and reach their goals. With the help of a reward feedback, agents are able to learn the behavior and improve it in the longer run, so there is a feedback loop between the learning system and its experiences. Unlike supervised learning, there is no answer key provided to the agent when they have to perform a particular task but he learns from its own experience.

Artificial Neural Networks (**ANNs**) are the most popular supervised learning algorithms today, which constitute a very exciting and powerful subfield of ML, known as **Deep Learning** (**DL**). The invention of ANNs took place in the 1970s, when Warren McCulloch and Walter Pitts coined this term and was inspired by the goal of modeling biological neural systems. The development of DL is motivated in part by the failure of traditional algorithms, like linear regression or basic decision trees, to generalize well on AI tasks such as recognizing speech or recognizing objects, above all when working with high-dimensional data and complicated functions. Miming the human structure of the brain, these algorithms have become an indispensable tool for a wide range of applications such as image classification, speech recognition or natural language processing and they have achieved extremely high predictive accuracy, in many cases, on par with human performance. ANNs are the key technology behind driverless cars, enabling them to recognize a stop sign or to distinguish a pedestrian from a lamppost, or still, they are the key to voice control in consumer devices like phones, tablets, TVs and hands-free speakers. DL models can be used for a variety of complex tasks:

- Deep Neural Networks (DNNs) for classification tasks;

- Convolutional Neural Networks (CNNs) for computer vision;

- Recurrent Neural Networks (RNNs) for time series analysis.

ANNs are composed by a large number of highly interconnected processing elements, the **neurons**, that work in unison to solve a specific problem. The functioning of the ANNs is similar to the way neurons work in our nervous system. The basic computational unit of the brain is a neuron: each neuron receives input signals from its dendrites and processes them in the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can fire, sending a spike along its single axon and producing output signals (Fig. 2.1). The axon eventually branches out and connects via synapses to dendrites of other neurons. The synaptic strengths are learnable and control the strength of influence and its direction, excitatory or inhibitory of one neuron on another.

In order to understand the workings of an ANN, it is appropriate to understand how it is structured. ANNs are called *networks* because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph

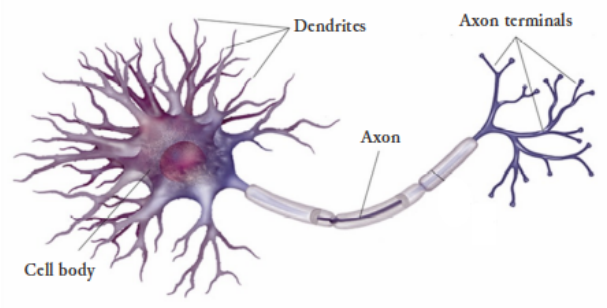


Figure 2.1: Representation of a neuron

of neurons describing how these functions are composed together. In the network there are no loops, so the outputs of some neurons can become inputs to other neurons. For this reason, such networks are called *feedforward* ANNs: information flows through the function being evaluated from the input to the output, so there are no feedback connections in which outputs of the model are fed back into itself. When feedforward ANNs are extended to include feedback connections, they are called Recurrent Neural Networks. ANNs are organized into a number of distinct layers of neurons because this structure makes it very simple and efficient to evaluate functions using matrix vector operations. In a ANN, there are three essential layers:

- *Input layer*: it is the first layer of an ANN that receives the input information in the form of various texts, numbers, audio files, image pixels and so on. The neurons within this layer are called *input neurons*;
- *Hidden layer*: in the middle of the ANN there are the hidden layers, which perform various types of mathematical computation on the input data and recognize the patterns that are part of. There can be a single hidden layer, as in the case of a perceptron or multiple, as in the DNNs, whose number determines the *depth* of an ANN.
- *Output layer*: it contains the *output neurons* and gives us the result that we obtain through rigorous computations performed by the middle layers.

2.2 From Perceptron to Deep Neural Networks

Earlier version of ANNs is represented by **Perceptron**. It is a simple algorithm, composed of one input and one output layer and at most one hidden layer in between, intended to perform binary classification, so it predicts whether input belongs to a certain category of interest or not. Obviously, the perceptron is not a complete model of human decision-making. For this reason, the last decade has witnessed the rise of more complex and deeper ANNs such as **Deep Neural Networks** (**DNNs**). DNNs receive a single input vector and transform it through an *high* number of hidden layers: hence the name *deep*. Each neuron of each hidden layer is fully connected to all neurons in the previous layer but operates completely independently and does not share any connections with the neurons of its own layer. Finally, the output layer is a fully connected layer that, in classification settings, represents the class scores (Fig. 2.2).

The goal of a DNN is to find, after a learning phase, a good approximation f of the mapping F between input-output data: given an input training set $\mathbf{X}$ of labeled examples $\mathbf{x}$,

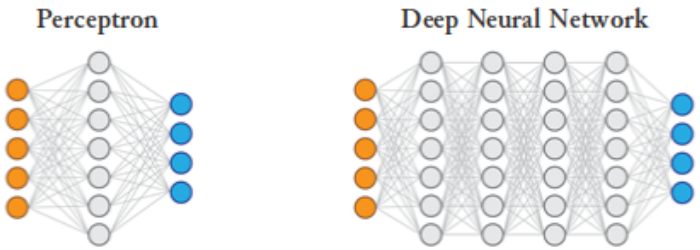


Figure 2.2: Schematic representation of a: (*Left*) Perceptron. (*Right*) Deep Neural Network. In both figures, orange balls are the input neurons, light blue balls the hidden neurons and finally, the blue balls the output neurons.

where each entry x_i is a feature, and a vector of labels $\mathbf{y}$, with y_i providing the label for the feature x_i, a DNN defines a mapping $\mathbf{y} = f(\mathbf{x}; \mathbf{p})$ and learns the value of the parameters $\mathbf{p}$ that best fit the input data, driving $f(\mathbf{x})$ to match $F(\mathbf{x})$ during the training. Each hidden layer of the network is typically vector-valued and each element of the vector may be interpreted as playing a role analogous to a neuron, in the sense that it receives input from many other units and computes its own activation value. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many units that act in parallel, each representing a vector-to-scalar function. In each layer, every neuron outputs a single real number, which is passed to every neuron in the next layer, where each neuron forms its own weighted combination of these values, adds its own bias, and applies a nonlinear function σ, called **activation function** (Fig. 2.3).

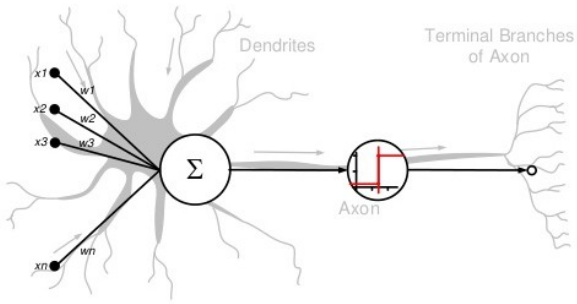


Figure 2.3: How a neuron works

The non linearity element allows for greater flexibility and creation of complex functions during the learning process, in addition to having a significant impact on the speed of learning. Currently, the most popular one for hidden layers is probably $ReLU$, defined as

$$\sigma(x) = \max\{0, x\}, \tag{2.1}$$

while, especially in the output layer, when we are dealing with a binary classification and we want the values returned from the model to be in the range from 0 to 1, the most used is *Sigmoid* (Fig. 2.4):

$$\sigma(x) = \frac{1}{1 - e^{-x}}. \tag{2.2}$$

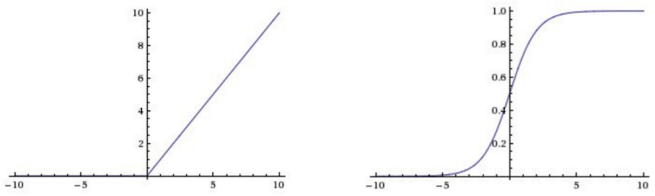


Figure 2.4: (*Left*) ReLU function. (*Right*) Sigmoid function.

Most hidden units can be described as accepting as input a vector of real numbers $\mathbf{a}$, computing an affine transformation $\mathbf{z} = \mathbf{W}\mathbf{a} + \mathbf{b}$, and then applying an element-wise nonlinear function $\sigma(\mathbf{z})$. The vector of outputs from the next layer, hence, has the form:

$$\sigma(\mathbf{W}\mathbf{a} + \mathbf{b}), \tag{2.3}$$

where $\mathbf{W}$ is a matrix of *weights*, parameters that determine how each feature affects the prediction: if a feature x_i receives a positive weight w_i, then increasing the value of that feature increases the value of our prediction, on the contrary, if a feature receives a negative weight, then increasing the value of that feature decreases the value of our prediction. The second parameter, $\mathbf{b}$, is a vector of *biases*: when initializing the parameters of the affine transformation, it can be a good practice to set all elements of $\mathbf{b}$ to a small, positive value. The number of columns in $\mathbf{W}$ matches the number of neurons that produced the vector $\mathbf{a}$ at the previous layer, while, the number of rows in $\mathbf{W}$, as well as the number of components in $\mathbf{b}$, matches the number of neurons at the current layer. To emphasize the role of the neuron i:

$$\sigma\left(\sum_j w_{ij}a_j + b_i\right), \tag{2.4}$$

where the sum runs over all entries in $\mathbf{a}$.

A key design consideration for DNNs is determining the architecture. The word architecture refers to the overall structure of the network: how many units it should have and how these units should be connected to each other. Let us suppose that the network has L layers, with layers 1 and L being the input and output layers respectively and that layer l, for $l = 1, 2, 3, ..., L$, contains n_l neurons. Overall, the n_L output neurons depend on the n_1 input neurons through a function $F : \mathbb{R}^{n_1} \longrightarrow \mathbb{R}^{n_L}$, whose analytic expression is unknown and that the algorithm will approximate. Let us use $\mathbf{W}^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ to denote the matrix of weights at layer l. More precisely, $w^{[l]}_{jk}$ is the weight that neuron j at layer l applies to the output from neuron k at layer $l-1$. Similarly, $\mathbf{b}^{[l]} \in \mathbb{R}^l$ is the vector of biases for layer l, so neuron j at layer l uses the bias $b^{[l]}_j$. Given an input $\mathbf{x} \in \mathbb{R}^{n_1}$, where n_1 is the dimension of the input data, we may then neatly summarize the action of the network, by letting $a^{[l]}_j$ denote the output, or *activation*, from neuron j at layer l. So, we have

$$\mathbf{a}^{[1]} = \mathbf{x} \qquad \mathbf{a}^{[l]} = \sigma(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}) \quad \text{for} \quad l = 2, 3, \ldots, L \tag{2.5}$$

The output from the overall network has the form:

$$f(\mathbf{x}) = \sigma(\mathbf{W}^{[L]}\sigma(\mathbf{W}^{[L-1]}\sigma(\ldots\sigma(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]})\ldots) + \mathbf{b}^{[L-1]}) + \mathbf{b}^{[L]}) \tag{2.6}$$

where $f : \mathbb{R}^{n_1} \longrightarrow \mathbb{R}^{n_L}$, defined in terms of the entries in the weight matrices and bias vectors, is the approximation, obtained by the learning process, of the real input-output map F.

The basic source of information on the progress of the learning phase is the value of the **loss function** or *cost function*, that measures how much the function f differs from F. Training a neural network corresponds to choosing the parameters, hence the weights and biases, that minimize the loss function: at each step of the training, we update our parameters following a certain direction to try to get to the lowest possible point. If now suppose we have N training data in $\mathbb{R}^{n_1}$, $\{\mathbf{x}^{\{i\}}\}_{i=1}^{N}$, for which there are given target outputs $\{\mathbf{y}(\mathbf{x}^{\{i\}})\}_{i=1}^{N}$ in $\mathbb{R}^{n_L}$, the quadratic loss function that we wish to minimize has the form:

$$\mathcal{L}(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} ||\mathbf{y}(\mathbf{x}^{\{i\}}) - \mathbf{a}^{L}(\mathbf{x}^{\{i\}})||_2^2 \tag{2.7}$$

The largest difference between the linear models and DNNs is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that DNNs are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.

A classical method in optimization to minimize this loss function is the **gradient descent method** (**GD**). At this stage it is convenient to imagine weights and biases stored as a single vector that we call $\mathbf{p} \in \mathbb{R}^s$. Let us write the loss function as $\mathcal{L}(\mathbf{p}) : \mathbb{R}^s \longrightarrow \mathbb{R}$ to emphasize its dependence on the parameters. The method proceeds iteratively, computing a sequence of vectors in $\mathbb{R}^s$ with the aim of converging to a vector that minimizes the loss function. Let us suppose that our current vector is $\mathbf{p}$: how should we choose a perturbation, $\Delta\mathbf{p}$, so that the next vector, $\mathbf{p}+\Delta\mathbf{p}$, represents an improvement? A Taylor series expansion gives:

$$\mathcal{L}(\mathbf{p} + \Delta\mathbf{p}) \approx \mathcal{L}(\mathbf{p}) + \sum_{r=1}^{s} \frac{\partial \mathcal{L}(\mathbf{p})}{\partial p_r} \Delta p_r \tag{2.8}$$

Here $\frac{\partial \mathcal{L}(\mathbf{p})}{\partial p_r}$ denotes the partial derivative of the loss function with respect to the rth parameter. For convenience, we will let $\Delta\mathcal{L}(\mathbf{p}) \in \mathbb{R}^s$ denote the vector of partial derivatives, known as the *gradient*, so:

$$\mathcal{L}(\mathbf{p} + \Delta\mathbf{p}) \approx \mathcal{L}(\mathbf{p}) + \Delta\mathcal{L}(\mathbf{p})^T \Delta\mathbf{p} \tag{2.9}$$

Our aim is to reduce the value of the loss function, so we should choose $\Delta\mathbf{p}$ to make $\Delta\mathcal{L}(\mathbf{p})^T\Delta\mathbf{p}$ as negative as possible. Hence, we should choose $\Delta\mathbf{p}$ to lie in the direction $-\Delta\mathcal{L}(\mathbf{p})$, limiting ourselves to a small step in that direction. This leads to the update:

$$\mathbf{p} \longrightarrow \mathbf{p} - \eta\Delta\mathcal{L}(\mathbf{p}) \tag{2.10}$$

Here, η is an optimization hyperparameter known as the *learning rate*, which controls the size of the step that the parameters take in the direction of the gradient.

However, when we have a large number of parameters and a large number of training points, computing the gradient vector at every iteration of the gradient descent method can be prohibitively expensive. A much cheaper alternative is to replace the mean of the individual gradients over all training points by the gradient at a single, randomly chosen, training point. This leads to the simplest form of what is called the **stochastic gradient method** (**SGD**) (Fig. 2.5).

A single step may be summarized as follows:

1. Choose an integer i uniformly at random from $\{1, 2, 3, \ldots, N\}$;
2. Update $\mathbf{p} \longrightarrow \mathbf{p} - \eta \Delta \mathcal{L}_{x^{\{i\}}}(\mathbf{p})$

In words, at each step, the stochastic gradient method uses one randomly chosen training point to represent the full training set. In particular, the index i is chosen by sampling without replacement, that is, cycling through each of the N training points in a random order. Performing N steps in this manner is referred to as completing an *epoch*.

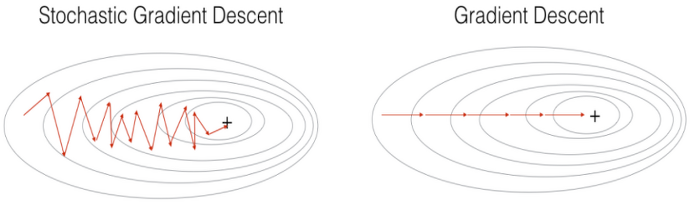


Figure 2.5: "+" denotes a local minimum of the cost. SGD leads to many oscillations to reach convergence, but each step is a lot faster with respect to GD.

If we regard the SGD as approximating the mean over all training points by a single sample, then it is natural to consider a compromise where we use a small sample average. For some $m \ll N$ we could take steps of the following form:

1. Choose m integers, $k_1, \ldots, k_m$, uniformly at random from $\{1, \ldots, N\}$;
2. Update

$$\mathbf{p} \longrightarrow \mathbf{p} - \eta \frac{1}{m} \sum_{i=1}^{m} \Delta \mathcal{L}_{x^{\{k_i\}}}(\mathbf{p}) \tag{2.11}$$

In 2.11, the set $\{x^{\{k_i\}}\}_{i=1}^{m}$ is known as a *mini-batch* and the method **mini-batch gradient descent**: it updates the parameters only after having seen a batch of all the training examples. Using mini-batches in the optimization algorithm often leads to faster optimization (Fig. 2.6).

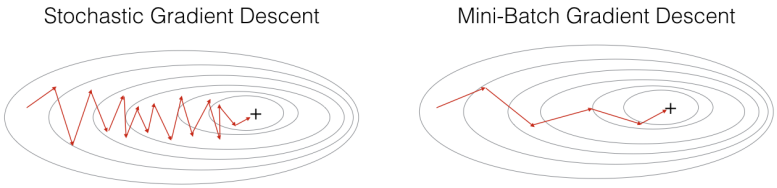


Figure 2.6: SGD vs Mini-batch SGD.

Because mini-batch GD makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch GD will oscillate toward convergence. The method of **Mini-batch GD with momentum** is designed to reduce these oscillations and accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to

move in their direction. The effect of momentum is illustrated in (Fig. 2.7). Formally, the momentum algorithm introduces a variable $\mathbf{v}$ that plays the role of velocity: it is the direction and speed at which the parameters move through parameter space. After initializing $\mathbf{v}$ at zero and the new hyperparameter β typically at 0.9, the SGD with momentum update has the form:

1. Choose m integers, $k_1, \ldots, k_m$, uniformly at random from $\{1, \ldots, N\}$;
2. Update
$$\mathbf{v} \longrightarrow \beta\mathbf{v} - \eta\frac{1}{m}\sum_{i=1}^{m}\Delta\mathcal{L}_{x\{k_i\}}(\mathbf{p}) \tag{2.12}$$
3. Update $\mathbf{p} \longrightarrow \mathbf{p} + \mathbf{v}$

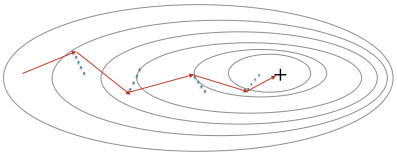


Figure 2.7: The red arrows show the direction taken by one step of mini-batch GD with momentum. The blue points show the direction of the gradient (with respect to the current mini-batch) on each step.

Nesterov momentum is a slightly different version of the momentum update that has recently been gaining popularity. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated (Fig. 2.8). With Nesterov momentum the gradient is evaluated after the current velocity is applied, thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard method of momentum. The update rules in this case are given by:

1. Choose m integers, $k_1, \ldots, k_m$, uniformly at random from $\{1, \ldots, N\}$;
2. Apply interim update $\tilde{\mathbf{p}} \longrightarrow \mathbf{p} + \beta\mathbf{v}$
3. Update at interim point
$$\mathbf{v} \longrightarrow \beta\mathbf{v} - \eta\frac{1}{m}\sum_{i=1}^{m}\Delta\mathcal{L}_{x\{k_i\}}(\tilde{\mathbf{p}}) \tag{2.13}$$
4. Update $\mathbf{p} \longrightarrow \mathbf{p} + \mathbf{v}$

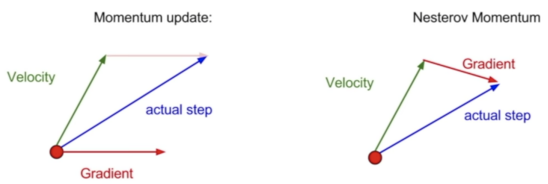


Figure 2.8: Standard momentum vs Nesterov momentum

Finally, we have to calculate these gradients. We recall that the output from the DNN $\mathbf{a}^{[L]}$ can be evaluated from a *forward pass* through the network, computing $\mathbf{a}^{[1]}, \mathbf{a}^{[2]}, \ldots, \mathbf{a}^{[L]}$ in order. Having done this, we can immediately calculate $\delta^{[L]}$, which measures the sensitivity of the loss function to the *weighted input* $\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$ for $l = 2, 3, \ldots, L$, as follows:

$$\delta^{[L]} = \sigma'(\mathbf{z}^{[L]}) \circ (\mathbf{a}^{[L]} - \mathbf{y}) \tag{2.14}$$

where

$$\sigma'(z_j^{[L]}) = \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \qquad \text{with } a_j^{[L]} = \sigma(z_j^{[L]}) \quad \text{for } 1 \leq j \leq n_L \tag{2.15}$$

Then, $\delta^{[L-1]}, \delta^{[L-2]}, \ldots, \delta^{[2]}$ may be computed in a *backward pass*:

$$\delta^{[l]} = \sigma'(\mathbf{z}^{[l]}) \circ (\mathbf{W}^{[l+1]})^T \delta^{[l+1]} \qquad \text{for} \quad 2 \leq l \leq L-1 \tag{2.16}$$

with a component-wise product $(x \circ y)_i = x_i y_i$, for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. Now we are able to compute gradients as:

$$\frac{\partial \mathcal{L}}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \text{for} \quad 2 \leq l \leq L, \quad 1 \leq j \leq n_l \tag{2.17}$$

Computing gradients in this way is known as **backpropagation** algorithm, that represents how to efficiently compute the gradient of the loss with respect to the model parameters.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs or ConvNets) represent a special class of artificial neural networks, which have become a standard tool in computer vision applications. They are very similar to ordinary neural networks from the previous chapter: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. However, ConvNet architectures make the explicit assumption that the inputs are *images*, which make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. In particular, unlike a regular neural network, the layers of a CNN have neurons arranged in 3 dimensions: *width*, *height*, *depth*, where depth, here, refers to the third dimension of an activation volume, not to the depth of a full ANN.

The name of these algorithms indicates that the network employs a mathematical operation called *convolution*. In its most general form, convolution is an operation on two functions of a real-valued argument: given, for example, the functions $x(t)$ and $w(t)$, the convolution operation, typically denoted with an asterisk $s(t) = (x \star w)(t)$, in its discrete form consists on:

$$s[t] = (x \star w)(t) = \sum_{a=-\infty}^{+\infty} x[a]w[t-a] \tag{2.18}$$

In convolutional network terminology, the first argument, the function x, is often referred to as the input and the second argument, the function w, as the *kernel* or *filter*. The output is referred to as the *activation map*. In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of learnable parameters. We will refer to these multidimensional arrays as *tensors*.

Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements. Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$s[i,j] = (I \star K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n] \tag{2.19}$$

that, being commutative, is equivalent to write

$$s[i,j] = (I \star K)[i,j] = \sum_m \sum_n I[i-m, j-n]K[m,n] \tag{2.20}$$

that is more straightforward to implement in a machine learning library.

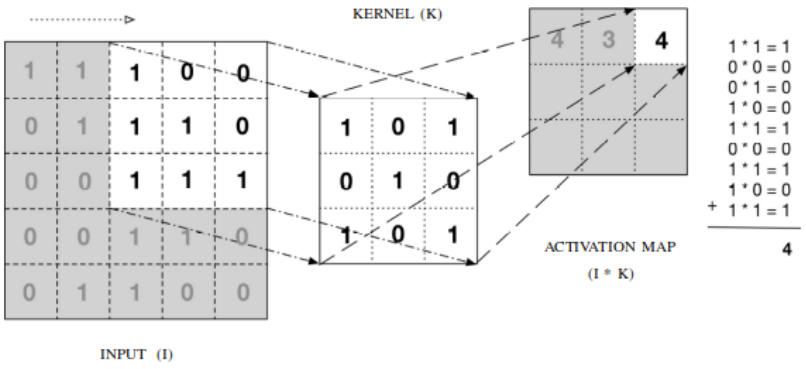


Figure 2.9: Convolution operation

To build a CNN architecture, the main types of layers are Convolutional layers and Pooling layers.

- The **Convolutional layer** is the core building block of a ConvNet. His parameters consist of a set of learnable **filters**. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size $5x5x3$ (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels RGB). During the forward pass, we slide, or more precisely, *convolve*, each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional **activation map** or *feature map* that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer. Now, we will have an entire set of filters in each convolutional layer, and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume. Furthermore, convolutional networks typically have sparse interactions. When dealing with high-dimensional inputs such

as images, it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron, that equivalently is the filter size. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency, computing fewer operations.

- It is common to periodically insert a **Pooling layer** in-between successive convolutional layers in a CNN architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. The pooling layer operates independently on every depth slice of the input and resizes it spatially, using the *max* or *average* operation. The most common form is a pooling layer with filters of size 2×2 that downsamples every depth slice in the input by 2 along both width and height, while keeping the depth dimension unchanged. Pooling helps to make the representation become invariant to small translations of the input. Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.

A ConvNet architecture ends with a **Fully-Connected layer** that drives the final classification decision: first, the output of convolution/pooling is flattened into a single vector of values, each representing a probability that a certain feature belongs to a label. For example, if the image is of a cat, features representing things like whiskers or fur should have high probabilities for the label "cat". These values, then, are multiplied by weights and they pass through an activation function (typically ReLU), just like in a classic artificial neural network. Finally, they pass forward to the output layer, in which every neuron represents a classification label.

The most common form of a ConvNet architecture stacks a few *Conv-ReLU* layers, follows them with *Pool* layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transition to fully-connected layers, which holds the class scores(Fig. 2.10).

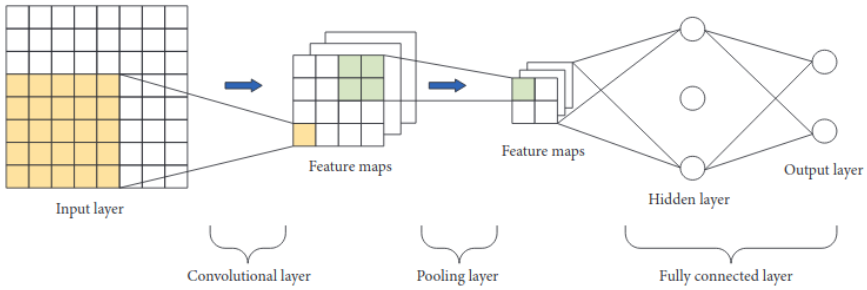


Figure 2.10: Generic CNN architecture

Chapter 3

Explainable Artificial Intelligence

3.1 Why are DNNs black-box?

Despite their success in a broad range of tasks, the severe drawback of deep learning architectures is their multilayer nonlinear structure and millions of parameters to optimize. Training data is fed to the bottom layer, the input layer, and it passes through the succeeding layers, getting multiplied and added together in complex ways, until it finally arrives, radically transformed, at the output layer. During training, the weights and thresholds are continually adjusted until training data with the same labels consistently yield similar outputs. These reasons make them be considered as **black-box models**, so they suffer from opacity and lack transparency, as compared to generalized linear models. The black-box nature of DNNs allows powerful predictions, but it cannot be directly explained, so it is hard to grasp *what* makes them arrive at a particular classification or recognition decision given a new unseen data sample.

3.2 Explainable Artificial Intelligence

Since the beginning of artificial intelligence, researchers, engineers and practitioners have felt the urge to understand these complex and non-linear models. While in the early days of AI, researchers aimed to find connections between models such as perceptrons and human neurodynamics, later works focused more on understanding the learned representations and the system's behavior. What if the computer says "No"? The absurdity of inexplicable black box decision making is lampooned in the famous "Computer says No" sketch. It is funny for a number of reasons, not least that a computer should hold such sway over such an important decision and not in any way be held to account. There is no way of knowing if it is an error or a reasonable decision.

The problem of explaining the behaviour of deep neural networks has recently gained a lot of attention. Models that are able to summarize the reasons for neural network behavior, gain the trust of users, or produce insights about the causes of their decisions. Paradigms underlying this problem fall within the so-called **eXplainable Artificial Intelligence (XAI)** field, which is widely acknowledged as a crucial feature for the practical deployment of AI models. According to *DARPA*, XAI proposes creating a suite of machine learning techniques that produce more explainable models while maintaining a high level of learning performance. This will enable humans to understand, appropriately trust, and effectively manage the emerging generation of artificially intelligent partners. XAI has become a

hotspot in machine learning research community and is thus expected by most of the owners, operators and users to answer some questions like: Why did the AI system make a specific prediction or decision? Why did not the AI system do something else? When did the AI system succeed and when did it fail?

3.2.1 The need of explanation

In general, humans are reticent to adopt techniques that are not directly interpretable, tractable and trustworthy. To win the users trust, the link between the features and the predictions should be understandable. Based on the explored literature, the need for explaining AI systems may stem from at least four reasons:

- **Explain to justify**. The past several years have seen multiple controversies over AI enabled systems yielding biased or unintentional discriminatory results. The problem is that the neural network learns only from the training data, which should characterize the task. The enormous amount of data may contain human biases and prejudices, thus, decision models learned on them may inherit such biases, possibly leading to unfair and wrong decisions. That implies an increasing need for explanations to ensure that AI based decisions were not made erroneously. Furthermore, henceforth AI needs to provide justifications in order to be in compliance with legislation: in force from 25 May 2018, the European Parliament adopted the General Data Protection Regulation (**GDPR**), which for the first time introduce, to some extent, a right of explanation for all individuals to obtain "meaningful explanations of the logic involved" when automated decision making takes place.

- **Explain to control**. Explainability can also help prevent things from going wrong. Indeed, understanding more about system behavior provides greater visibility over unknown vulnerabilities and flaws, and helps to rapidly identify and correct errors. Currently, models are evaluated using accuracy metrics on an available validation dataset. However, real-world data is often significantly different, and further, the evaluation metric may not be indicative of the product's goal. Inspecting individual predictions and their explanations is a worthwhile solution, in addition to such metrics. Furthermore, metrics like accuracy or the mean average precision are depending on the quality of manually hand annotated data. But suitable training data is tedious to create and annotate, hence it is not always perfect. These metrics are often the only values that evaluate the learning algorithm itself and they offer an incomplete description. Techniques for interpreting and understanding what the model has learned have therefore become a key ingredient of a robust validation procedure.

- **Explain to improve**. Another reason for building explainable models is the need to continuously improve them. A model that can be explained and understood is one that can be more easily improved. Because users know why the system produced specific outputs, they will also know how to make it smarter.

- **Explain to discover**. Asking for explanations is a helpful tool to learn new facts, to gather information and thus to gain knowledge. Only explainable systems can be useful for that. For example, given that AlphaGo Zero can excel at the game of Go much better than human players, it would be desirable that the machine can explain its learned strategy to us. So it will come as no surprise if, in future, XAI models taught us about new and hidden laws in biology, chemistry and physics.

For commercial benefits, for ethics concerns or for regulatory consideration, we conclude that explainability is a powerful tool for justifying AI based decisions. It can help to verify predictions, for improving models, and for gaining new insights into the problem at hand.

3.3 Where is XAI crucial?

While traditional machine learning models are being employed in an increasing number of fields, the black-box nature of DNNs is still a barrier to the adoption of these systems for those tasks where interpretability is essential, like autonomous driving, medical applications and finance. In these fields, failure is not an option: the importance of explainability does not just depend on the degree of functional opacity caused by the complexity of your machine learning models, but also the impact of the decisions they make. Even a momentarily dysfunctional computer vision algorithm in autonomous vehicles easily leads to fatality. In the medical field, clearly human lives are on the line. Detection of a disease at its early phase is often critical to the recovery of patients or to prevent the disease from advancing to more severe stages. More in detail:

- **Medicine** The amount of medical data generated is growing exponentially, from medical images to connected devices. The use of diagnostic imaging has increased significantly and continues to grow. This has an impact on the workload of healthcare practitioners. Additionally, new kinds of healthcare data are large and complex, rendering traditional diagnostic methods obsolete. Unfortunately, the growth of radiologists is only half of the growth in medical images. Deep Learning will be critical in ensuring that healthcare practitioners can focus on cases that truly matter, letting machines diagnose or treat the easier ones. AI will also contribute in combining various data sources from a patient's history (medical records, radiology imaging, pathology imaging, genome sequences, fitness band data, heart monitor data, etc.) to generate a personalized diagnosis or a personalized treatment plan. Most of the deep learning models are classification models which predict a probability of abnormality from a scan. However, just the probability score of the abnormality does not amount much to a radiologist if it is not accompanied by a visual interpretation of the model's decision. Interpretability of deep learning models is very much an active area of research and it becomes an even more crucial part of solutions in medical imaging: for medical professionals, new technologies can change the way they work, enable more accurate and faster diagnoses and improve care. For patients, healthcare innovations lessen suffering and save lives.

- **Finance** Financial services are demanding more advanced AI solutions. They want to apply AI to numerous situations that require high credibility, such as sales predictions, risk management for investment, and detection of money laundering and illegal transactions. Companies need to know they can trust AI results. They need transparency and credibility, and they need to know how to effectively use AI results to transform decision making. With traditional AI, customers are left asking why the results are what they are. For example, when using AI to predict the risk of an investment, a firm wants to know what the criteria behind the predictions are, and if the risk is high, what could be altered to reduce the risk. Explainable AI allows the user to understand the criteria behind a prediction, which factors increase risk, and the actions that can be taken. The impact of this increased transparency is significant:

there are savings costs that come from better understanding potential vulnerabilities. Firms can reduce their time costs by being able to understand where to prioritise and focus on, and firms can implement more effective strategy and allocation of resources based on the ability explainable AI offers to modify behaviour and outcomes.

- **Transportation** The emerging arena of self-driving vehicles is one where AI will certainly play a role and where explainable AI will be paramount. Deep neural networks are an increasingly important technique for autonomous driving, especially as a visual perception component: autonomous vehicles have to make split-second decisions based on how they classify the objects in the scene in front of them. Self-driving cars began to migrate from laboratory development and testing conditions to driving on public roads. Deployment in a real environment offers a decrease in road accidents and traffic congestions, as well as an improvement of our mobility in overcrowded cities for a sustainable future. On the other hand, a wrong decision by a self-driving AI could lead to the car crashing into a guard rail, colliding with another vehicle or running down pedestrians and cyclists. This is not a possibility, but this is already happening, recently, a self-driving Uber killed a woman in Arizona. It was the first known fatality involving a fully autonomous vehicle. Only an explainable system can clarify the ambiguous circumstances of such a situation and eventually prevent it from happening. Hence, the explainability and inspectability of the algorithms controlling the vehicle are necessary: such insightful explanations are relevant not only for legal issues and insurance matters, but also for engineers and developers in order to achieve provable functional quality guarantees.

Healthcare, finance and transportation are only a part of the many fields where XAI is a fundamental tool for their development. The need for an explanation to accompany the decisions taken by AI-based systems is also strong in the legal and military fields.

Chapter 4

Explanation methods

Before presenting the methods taxonomy and the classification of the problems addressed in the literature with respect to black box predictors with the corresponding solutions, it is convenient to establish a common point of understanding on what the terms as interpretation and explanation stand for in the context of AI. By defining these words as [Montavon et al., 2018] suggest:

- An **interpretation** is the mapping of an abstract concept, e.g. a predicted class, into a domain that the human can make sense of. Examples of domains that are interpretable are images (arrays of pixels), or texts (sequences of words): a human can look at them and read them respectively.
- An **explanation** is the collection of features of the interpretable domain that have contributed for a given example to produce a decision, e.g. classification or regression. An explanation can be, for example, a heatmap highlighting which pixels of the input image most strongly support the classification decision. The explanation can be coarse-grained to highlight which regions of the image support the decision.

4.1 Taxonomy

Over a relatively short period of time a plethora of explanation methods and strategies have come into existence, driven by the need of expert users to analyze and debug their deep neural networks, in the quest to make them interpretable. In this axis, we propose an overview of existing methods. We note that, since explainability in artificial intelligence is still an emerging field, the classes of methods belonging to the proposed taxonomy are neither mutually exclusive nor exhaustive. As such it is possible that in the future the taxonomy needs to be modified.

Based on the conducted survey of the literature, we arrive to classify the methods according to three criteria:

1. The complexity of interpretability;
2. The scope of interpretability;
3. The level of dependency from the used ML model.

4.1.1 Intrinsic vs Post-hoc interpretability

The complexity of a machine learning model is directly related to its interpretability. Generally, the more complex the model, the more difficult it is to interpret and explain. Interpretable machine learning techniques can generally be grouped into two categories: *intrinsic* and *post-hoc interpretability*.

Intrinsic interpretability

The most straightforward way to get to interpretable AI models would be to design an algorithm that is inherently and intrinsically interpretable. Intrinsic interpretability is achieved by constructing self-explanatory models which incorporate interpretability directly to their structures, by generating explanations at training time. This category, which includes decision tree, rule-based model or linear model, aims to improve the interpretability of internal representations with methods that are part of the machine learning architecture, so that we arrive at explanations by reading off parameter estimates or a few decision rules. A common challenge, which hinders the usability of this class of methods, is the tradeoff between interpretability and accuracy. In a sense, intrinsic interpretable models come at a cost of accuracy.

Post-hoc interpretability

An alternative approach to interpretability in machine learning, when machine learning models do not meet any of the criteria imposed to declare them transparent, is to construct a separate method to be applied to the existing model to explain its decisions. This is the purpose of post-hoc interpretability techniques, which aim at communicating understandable information about how an already developed model produces its predictions for any given input. Since most useful machine learning models have a level of complexity beyond the threshold of intrinsic interpretability, post-hoc techniques are the most used despite, while keeping the underlying model accuracy intact, they are limited in their approximate nature.

4.1.2 Global vs Local interpretability

Interpretability implies understanding an automated model and this supports two variations according to the scope of interpretability: understanding the entire model behavior (model-level) or understanding a single prediction (instance-level). Accordingly, we distinguish between two types of interpretability: *global* and *local interpretability*.

Global interpretability

Global interpretability facilitates the understanding of the whole logic of a model and follows the entire reasoning leading to all the different possible outcomes. Users can understand how the model works globally by inspecting his structures and his parameters. However, these models are usually specifically structured, thus limited in predictability to preserve interpretability. Arguably, global model interpretability is hard to achieve in practice, especially for models that exceed a handful of parameters.

Local interpretability

Explaining the reasons for a specific decision or single prediction means that interpretability is occurring locally. The scope of local interpretability is to generate an individual explanation, generally, to justify why the model made a specific decision for an instance. Several explored papers propose local explanation methods. Analogically to humans, who focus effort on only part of the model in order to comprehend the whole of it, local interpretability can be more readily applicable.

Figure 4.1: (*Left*) Model-level interpretability. (*Right*) Instance-level interpretability.

4.1.3 Model-specific vs Model-agnostic interpretability

In this section we categorize and review different algorithmic approaches for post-hoc interpretability, discriminating among those that are designed for their application to machine learning models of any kind and those that are designed only for a single type or class of algorithm and thus, can not be directly extrapolated to any other: *model-specific* and *model-agnostic interpretability*

Model-specific interpretability

Model-specific interpretability methods are limited to specific model classes, like deep neural networks. DNN-specific methods treat the networks as *white-boxes* and explicitly utilize the interior structure to derive explanations. The drawback of this practice is that when we require a particular type of interpretation, we are limited in terms of choice to models that provide it, potentially at the expense of using a more predictive and representative model.

Model-agnostic interpretability

Model-agnostic interpretability methods are not tied to a particular type of machine learning model. They allow explaining predictions of arbitrary machine learning models independent of the implementation. They provide a way to explain predictions by treating the models as *black-boxes*, where explanations could be generated even without access to the internal model parameters. They bring some risks at the same time, since we cannot guarantee that the explanation faithfully reflects the decision making process of a model.

Deep learning is the model family where most research has been concentrated in recent times and they have become central for most of the recent literature on XAI. While the division between model-agnostic and model-specific is the most common distinction made, the community has not only relied on this criteria to classify XAI methods. For instance, some model-agnostic methods such as SHAP, are widely used to explain deep learning models. That is why several XAI methods can be easily categorized in different taxonomy branches depending on the angle the method is looked at.

4.2 Techniques

Before proceeding with the detailed review of the methodologies in interpretable models, we provide an overview of existing state-of-the-art methods in machine learning and categorize them into two main categories: *rule-extraction* and *attribution methods*.

4.2.1 Rule-extraction methods

Works supporting the rule-extraction methods propose approaches that provide a symbolic and comprehensible description of the knowledge, learned by the network during its training. They extract human interpretable rules that approximate the decision-making process in a deep neural network, by utilizing the input and output of it. These classifiers use a collection of *if...then...* rules to infer the class labels. In a sense, rule-based classifiers are the text representation of the decision trees. So, it is arguably the most interpretable category of methods in our taxonomy, considering that the resulting set of rules can unambiguously be interpreted by a human being as a kind of formal language. Therefore, we can say that it has a high degree of clarity. In terms of explanatory power, rule-extraction methods can validate whether the network is working as expected in terms of overall logic flow, and explain which aspects of the input data had an effect that lead to the specific output.

4.2.2 Attribution methods

In the realm of local interpretability, attribution methods have received particular attention in the last years. The starting point of these algorithms is to view the data point $\mathbf{x}$ as a collection of features $(x_i)_{i=1}^m$. Given a specific output neuron, which can be described as a function $f(\mathbf{x})$ of the input, the goal of an attribution method is to produce explanations by assigning to each features x_i a scalar attribution value R_i, sometimes also called "**relevance**" or "**contribution**", determining how relevant the feature x_i is for explaining $f(\mathbf{x})$. For a classification task, the output neuron of interest is usually the target neuron associated with the correct class for a given sample. When the attributions of all input features are arranged together to have the same shape of the input sample, we talk about **attribution maps**, which are usually displayed as *heatmaps*, where green color indicates features that contribute positively to the activation of the target output, and red color indicates features that have a suppressing effect on it, as in Fig. 4.2.

Figure 4.2: Example of heatmap.

In practice, these techniques often do seem to highlight regions that can be meaningful to humans. Some attribution methods derive these importance scores using the local sensitivity of the model to the variation of the input's elements. Another group of attribution methods adopts a more global approach, by defining importance relative to a **baseline** (reference) input, i.e describes the marginal effect of a feature on the output with respect to a baseline, as opposed to local attribution methods, which describe how the output of the network changes for infinitesimally small perturbations around the original input.

4.3 Attribution methods

The problem of finding attributions for deep neural networks has been tackled in several previous works. Various new attribution methods have been published in the last few years to assign an importance value to individual features which is meant to reflect their influence on the final classification. Some of them use perturbation techniques, while others backpropagation rules with gradients. Similarly, these approaches can be partitioned into two broad categories: *gradient-based* methods limited to neural networks and in particular to convolutional neural networks, and more general, *perturbation-based* methods which can be used with arbitrary prediction models.

Unless otherwise specified, all explanations that we will show from here on concern images belonging to the Tiny Imagenet dataset. The heatmaps are obtained through the CNN used in our experiments, thanks to implementation provided by the Captum library.

4.3.1 Gradient-based methods

A starting point for identifying features that strongly influence the final decision is to compute the **gradient** of the class score function with respect to the input, holding the weights found during the training stage fixed. By leveraging the back propagation rules to track information from the network's output back to its input, or an intermediate layer, gradient-based methods elaborate on this basic idea: computing the gradient determines which input elements (e.g., which pixels in case of an input image) need to be changed the least to affect the prediction the most. In fact, the most relevant input features are those to which the output is most sensitive and one can expect that such pixels correspond to the object location in the image. Since they are based on simple modifications of the backpropagation algorithm, they only require a single forward and backward pass through the model: gradient-based methods are efficient in terms of implementation and generally faster then perturbation-based methods. On the other hand, they are limited in their *heuristic* nature and may generate explanations of unsatisfactory quality.

Local importance

The most basic attribution technique for interpreting deep neural networks is **Saliency Map** of [Simonyan et al., 2014]. The purpose of this approach is to query an already-trained classification CNN about the spatial support of a particular class in a given image, i.e. "figure out *where* a given object is in the photo without any explicit location labels". Saliency method computes the attributions by taking the partial derivative of the score function with respect to the input image, using a single backpropagation pass through a classification CNN. The saliency map, sometimes referred to as "backpropagation map", is a visual representation of the absolute value of the partial derivative evaluated locally.

More formally, in the case of deep CNN, the score of the c-th class $f_c(\mathbf{x})$ is a highly non-linear function of a generic input image $\mathbf{x}$. However, we can approximate $f_c(\mathbf{x})$ with a linear function in the neighbourhood of the image of interest $\tilde{\mathbf{x}}$ by computing the first-order Taylor expansion:

$$f_c(\tilde{\mathbf{x}}) \approx w^T \mathbf{x} + b \tag{4.1}$$

where w is the derivative of $f_c(\mathbf{x})$ with respect to the image $\mathbf{x}$ at the point (image) $\tilde{\mathbf{x}}$:

$$w = \frac{\partial f_c}{\partial \mathbf{x}}\bigg|_{\tilde{\mathbf{x}}} \tag{4.2}$$

The partial derivative in 4.2, computed by backpropagation, has the same shape of the image considered, so the saliency map is obtained taking the absolute value of it. It is important to note, however, that saliency map does not produce an explanation of the function value $f(\tilde{\mathbf{x}})$ itself, but rather a *variation* of it. Intuitively, when applying this technique, e.g. to a neural network detecting cars in images, we answer the question "what makes this image more/less a car?", rather than the more basic question "what makes this image a car?".

Despite the simplicity of this approach, the absolute value prevents the detection of positive and negative evidence that might be present in the input. Furthermore, the resulting visualizations are typically visually **noisy**, thus making the interpretation difficult, as we can see in the Fig. 4.3.

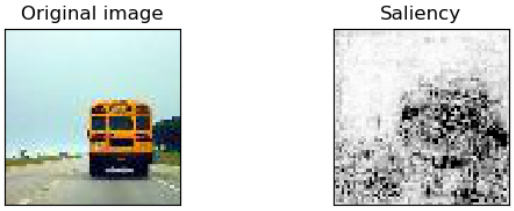


Figure 4.3: A noisy saliency map

Seeking better explanations of network decisions, several works proposed modifications to the basic technique of Saliency Map. The central idea of [Zeiler and Fergus, 2014] is to visualize layer activations of a convolutional neural network, by running them through a **DeconvNet**. It is a network that reverses the convolution and pooling operations of the CNN, until it reaches the input space, with the goal to approximately reconstruct the input of each layer from its output. Since max-pooling layers are in general not invertible, in order to perform such reconstruction, DeconvNet method requires first to perform a forward pass of the convolutional neural network to compute *switches*, the positions of maxima within each pooling region. The positions saved in the switch variables are then used in the DeconvNet to obtain a discriminative reconstruction and, as we will see in the next chapters, they strongly determine the property of this explanation method. The key difference from vanilla backpropagation of Saliency Map method is for the linear rectifier ReLU. As a recap, here we show the equations of a forward ReLU and a backward ReLU, obtained simply by taking the derivative:

$$ReLU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{altrimenti,} \end{cases} \qquad \frac{d}{dx} ReLU(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{altrimenti} \end{cases} \tag{4.3}$$

When backpropagating importance using gradients, the gradient coming into a ReLU during the backward pass is zeroed out if the input to the ReLU during the forward pass is negative. In formulas, let us consider input feature x_i and denote its activation in layer l as $a_i^{(l)}$ and gradient propagated up to $a_i^{(l)}$ as $R_i^{(l+1)}$. Let $\mathbb{I}(\cdot)$ be the indicator function. By the chain rule, backward pass through the ReLU non linearity for vanilla gradient is achieved by

$$R_i^{[l]} = \mathbb{I}(a_i^{[l]} > 0) \cdot R_i^{[l+1]} \tag{4.4}$$

In the Fig. 4.4, the first row shows the feature map $a_i^{(l)}$ produced by the l-th layer and the result of backward ReLU operation on it. In the second row, the latter is multiplied by the gradient map $R_i^{[l+1]}$, resulting in the calculation of backpropagation.

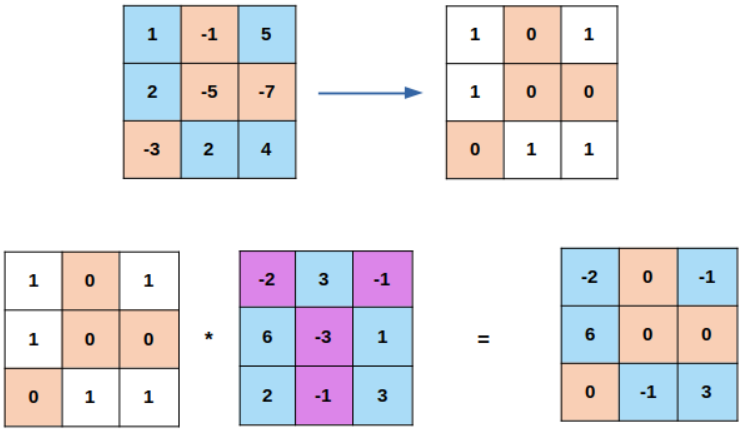


Figure 4.4: Backward ReLU used by Saliency Map.

By contrast, when backpropagating an importance signal in deconvolutional networks, the importance signal coming into a ReLU during the backward pass is zeroed out if and only if it is negative, with no regard to sign of the input to the ReLU during the forward pass. In formulas:

$$R_i^{[l]} = \mathbb{I}(R_i^{[l+1]} > 0) \cdot R_i^{[l+1]} \tag{4.5}$$

In the Fig. 4.5, the first row shows the result of the backward ReLU on gradient map $R_i^{[l+1]}$. The second row concerns the product between the latter and the gradient map itself. As we can see, there is no reference to the feature map calculated in the forward pass.

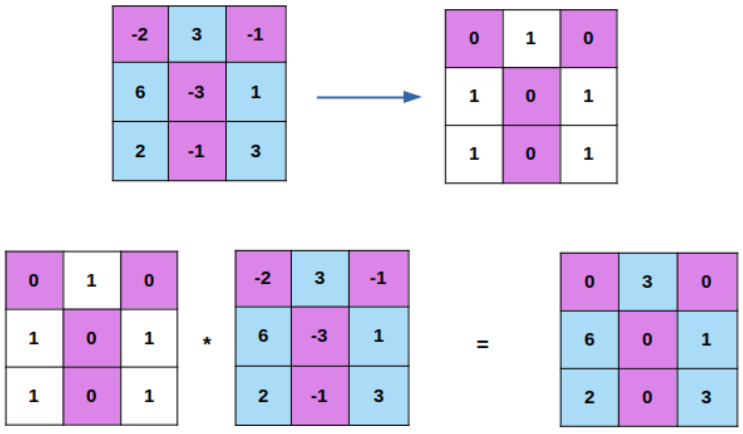


Figure 4.5: Backward ReLU used by DeconvNet.

In the following year, [Springenberg et al., 2015] propose to combine these two methods: rather than masking out values corresponding to negative entries of the top gradient (DeconvNet) or bottom data (Saliency), they mask out the values for which at least one of these values is negative. More formally, backward pass through the ReLU non linearity is achieved by

$$R_i^{[l]} = \mathbb{I}(a_i^{[l]} > 0) \cdot \mathbb{I}(R_i^{[l+1]} > 0) \cdot R_i^{[l+1]} \tag{4.6}$$

In the Fig. 4.6, it is shown the product between, in order, the result of the backward ReLU on the feature map computed by forward pass (Fig.4.4), the result of the backward ReLU on the importance signals map computed by backward pass (Fig.4.5) and finally the importance map itself.

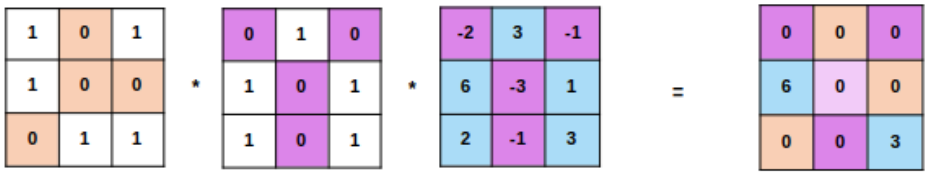


Figure 4.6: Backward ReLU used by Guided Backpropagation.

Since it adds an additional guidance signal from the higher layers to usual backpropagation, such method is called **Guided Backpropagation**. It prevents backward flow of negative gradients, corresponding to the neurons which decrease the activation of the higher layer unit we aim to visualize. Furthermore, unlike the DeconvNet, Guided Backpropagation works remarkably well without switches.

Many authors tried to identify the cause that makes Saliency Maps visually noisy:

- [Smilkov et al., 2017] suggest the hypothesis that noisy saliency maps are faithful descriptions of what the network is doing. That is, pixels scattered seemingly at random are crucial to how the network makes a decision. In short, this hypothesis claims that noise is actually informative;
- [Shrikumar et al., 2017] state that saliency maps are noisy due to the piecewise linearity of the score function. Specifically, since typical DNNs use ReLU activation functions and max pooling, the derivative of the score function with respect to the input will not be continuously differentiable. Under this hypothesis, noise is caused by meaningless local variations in the gradient.
- [Sundararajan et al., 2017] suggest that important features may have small gradients due to saturation. In other words, the score function can flatten in the proximity of the input and have a small derivative. This hypothesis explains why informative features may not be highlighted in saliency maps even though they contributed significantly to the decision of the DNN.
- [Kim et al., 2019] believe that saliency maps are noisy because deep neural networks do not filter out irrelevant features during forward propagation.

Gradient⊙Input of [Shrikumar et al., 2017] was at first proposed as a technique to improve the sharpness of the saliency maps and address the problem of gradient saturation. The attribution is computed taking the partial derivatives of the output with respect to the input, as in 4.2, and multiplying them with the input itself:

$$R_i(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial x_i} \cdot x_i \tag{4.7}$$

The multiplication in 4.7 tends to produce visually simpler and sharper images, although it can be unclear how much of this can be attributed to sharpness in the original image itself.

Another strategy for enhancing saliency maps and alleviating noise has been proposed by [Smilkov et al., 2017]: their method, called **Smooth Grad**, attempts to smooth discontinuous gradient through a Gaussian noise with standard deviation σ. However, since calculating the local average in a high dimensional space is intractable, the authors proposed a stochastic approximation which takes random samples of dimension n in a neighborhood of the input $\mathbf{x}$ and then averages their gradients $S(\mathbf{x})$:

$$\hat{S}(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} S(\mathbf{x} + \mathcal{N}(0, \sigma^2)) \tag{4.8}$$

In our experiments, we will apply this technique thanks to **Noise Tunnel**, a method of Captum library that can be used on top of any of the attribution methods. It computes attribution multiple times, adding Gaussian noise to the input each time, and combines the calculated attributions based on the chosen type. In the Fig.4.7, we show the effects of noise level, represented by the value of the standard deviation, on explanations generated by the Saliency Map with Smooth Grad technique applied through the Noise Tunnel. In the following Figure (4.8), we show instead the effect of the sample size n.

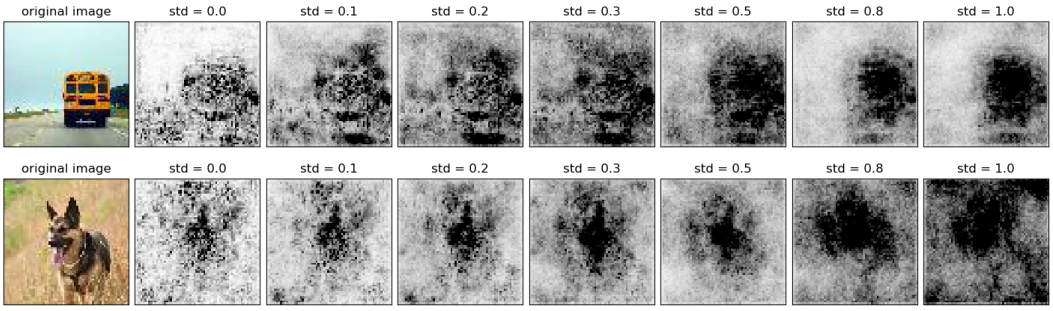


Figure 4.7: Effect of noise level on visualizations. The second column corresponds to the vanilla gradient of Saliency Map ($std = 0$). We observe that the ideal noise level depends on the input image.

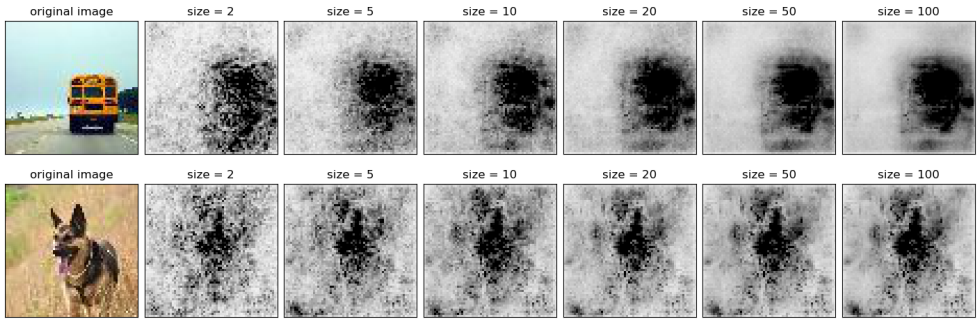


Figure 4.8: Effect of sample size on visualizations, keeping the noise level fixed. For the first image we used a $std = 0.8$, while for the second, a $std = 0.2$. As expected, the estimated gradient becomes smoother as the sample size increases.

A technique that significantly improves the quality of saliency maps and generalizes DeconvNet and Guided Backpropagation is **Rectified Gradient**, or RectGrad in short, proposed by [Kim et al., 2019]. It is a method that alleviates the noisy visualizations problem through layer-wise thresholding during backpropagation: the gradient propagates only through units whose importance scores exceed some threshold. By construction, RectGrad drops irrelevant features while retaining relevant features in a layer-wise fashion. Importance score for a unit is calculated by multiplying its activation with gradient propagated up to the unit.

By the chain rule, backward pass through the ReLU nonlinearity for vanilla gradient is achieved by Eq. 4.4. [Kim et al., 2019] modified this rule such that

$$R_i^{[l]} = \mathbb{I}(a_i^{[l]} \cdot R_i^{[l+1]} > \tau) \cdot R_i^{[l+1]} \tag{4.9}$$

for some threshold τ. Finally, importance scores for input features are calculated by multiplying gradient propagated up to input layer $l = 0$ with input features and thresholding at zero:

$$x_i \cdot R_i^{[1]} \cdot \mathbb{I}(x_i \cdot R_i^{[1]} > 0) \tag{4.10}$$

Instead of setting τ to a constant value, they use the qth percentile of importance scores at each layer. This prevents the gradient from entirely dying out during the backward pass.

Despite producing fine-grained visualizations, all methods presented so far are not class-discriminative. Visualizations with respect to different classes are nearly identical. We will elaborate on this aspect in chapter 7. To overcome this limitation, work by [Zhou et al., 2016] showed that the convolutional units of various layers of CNNs with a global average pooling layer actually behave as unsupervised object detectors. Despite having this remarkable ability to localize objects in the convolutional layers, this ability is lost when fully-connected layers are used for classification. On this study, [Zhou et al., 2016] proposed a new technique called Class Activation Maps **CAM**: visualizing the weighted combination of the resulting feature maps at the pre-softmax layer, they were able to obtain heatmaps that explain which parts of an input image were looked at by the CNN for assigning a label. CAM estimates these weights by training a linear classifier for each class using the activation maps of the last convolutional layer generated for a given image. However, this limits its explainability process to a particular kind of CNN architectures performing global average pooling over convolutional maps immediately prior to prediction and requires retraining of multiple linear classifiers, one for each class, after training of the initial model.

To address these issues, [Selvaraju et al., 2019] subsequently came up with an efficient generalization of CAM, known as Gradient-weighted Class Activation Mapping **Grad-CAM**. It uses the gradient information flowing into the last convolutional layer of the CNN to assign importance values to each neuron for a particular decision of interest. Grad-CAM can thus work with any deep CNN where the final score is a differentiable function of the activation maps, without any retraining or architectural modification. More formally, let us denote a class-discriminative localization map GradCAM $L^c_{\text{GradCAM}} \in \mathbb{R}^{w \times h}$ of width w and height h for any class c. We first compute the gradient of the score for class c, y^c (before the softmax), with respect to feature map activations A^k of a convolutional layer, i.e $\frac{\partial y^c}{\partial A^k}$. These gradients flowing back are global-average-pooled over the width and height

dimensions, indexed by i and j respectively, to obtain the neuron importance weights ω_k^c:

$$\omega_k^c = \underbrace{\frac{1}{Z}\sum_i\sum_j}_{\text{global avg pool}} \overbrace{\frac{\partial y^c}{\partial A_{ij}^k}}^{\text{via backprop}} \tag{4.11}$$

where Z is the number of pixels in the activation map. During computation of ω_k^c while backpropagating gradients with respect to activations, the exact computation amounts of successive matrix products of the weight matrices and the gradient with respect to activation functions till the final convolution layer that the gradients are being propagated to. Hence, this weight ω_k^c represents a partial linearization of the deep network downstream from A, and captures the importance of feature map k for a target class c. Finally, we perform a weighted combination of forward activation maps, and follow it by a ReLU to obtain:

$$L^c_{\text{GradCAM}} = ReLU\left(\sum_k \omega_k^c A^k\right) \tag{4.12}$$

However, while GradCAM is class-discriminative and localizes relevant image regions, it lacks the ability to highlight fine-grained details like pixel-space gradient visualization methods such as Guided Backpropagation or DeconvNet. In order to combine the best of both worlds, [Selvaraju et al., 2019] thought of fusing existing gradient visualizations to create **Guided GradCAM** method that generates both high-resolution and class-discriminative heatmaps. To obtain these fine-grained pixel-scale representations (Fig. 4.9), the GradCAM saliency maps are upsampled and fused via point-wise multiplication with the visualizations generated by Guided Backpropagation.

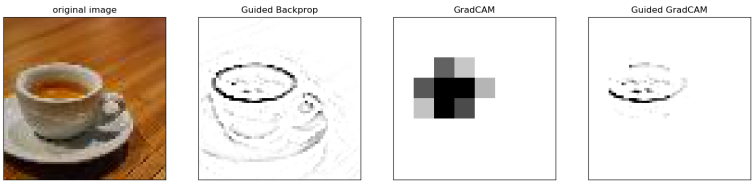


Figure 4.9: Comparison between heatmaps generated Guided BP and CAM-based methods.

However, the performance of this method drops when localizing multiple occurrences of the same class. In addition, for single object images, Guided GradCAM heatmaps often do not capture the entire object in completeness. To address these limitations, [Chattopadhyay et al., 2018] proposed a generalized method called **GradCAM++** that provides better visual explanations of CNN model predictions. This method uses a weighted combination of the positive partial derivatives of the last convolutional layer feature maps with respect to a specific class score as weights to generate a visual explanation for the corresponding class label. Formally:

$$\omega_k^c = \sum_i\sum_j \alpha_{ij}^{kc} \cdot ReLU\left(\frac{\partial y^c}{\partial A_{ij}^k}\right) \tag{4.13}$$

where the α_{ij}^{kc}'s are weighting coefficients for the pixel-wise gradients for class c and convo-

lutional feature map A^k:

$$\alpha_{ij}^{kc} = \left(\sum_{lm} \frac{\partial y^c}{\partial A_{lm}^k} \right)^{-1} \quad \text{if } \frac{\partial y^c}{\partial A_{ij}^k} = 1 \tag{4.14}$$

$\alpha_{ij}^{kc} = 0$ otherwise. In this way, the presence of objects in all feature maps are highlighted with equal importance.

Similar to GradCAM, to generate the final saliency maps, [Chattopadhyay et al., 2018] carried out pointwise multiplication of the upsampled saliency map with the pixel-space visualization generated by Guided Backpropagation: the representations thus generated are hence called **Guided GradCAM++**.

All these approaches arbitrarily invoke different back propagation or activation rules, which results in aesthetically pleasing, heuristic explanations of image saliency. Their solution is not black box agnostic, but it requires specific architectural modifications or access to intermediate layers. While the gradient provides information about which features can be locally perturbed the least in order for the output to change the most, applied to a highly non-linear function only provides local information and it does not help to compute the marginal contribution of a feature.

Reference-based importance

To overcome the limitation of the gradient-based methods, other techniques have been proposed, such as **Layer-wise Relevance Propagation** (LRP) of [Bach et al., 2015], **DeepLIFT** (in its two variants, *Rescale* and *RevealCancel*) of [Shrikumar et al., 2017], and **Integrated Gradients** of [Sundararajan et al., 2017]. They are characterized by different propagation rules for non-linear operations in the network than the vanilla gradient, making explicit use of feedforward graph structure of a DNN.

Let us start to analyze in detail the method proposed by [Bach et al., 2015]: **LRP**. It explains the model's decision by decomposing the prediction $f(\mathbf{x})$ as a sum of relevance scores. The algorithm starts at the output of the network and moves in the graph in reverse direction, progressively redistributing the prediction score (or total relevance), until the input is reached. Such backward pass is a conservative relevance distribution procedure, where neurons that contribute the most to the higher-layer receive most relevance from it. Formally, let us consider a deep neural network where j and k are indices for neurons at two successive layers. Let $(R_k)_k$ be the relevance scores associated with neurons in the higher layer. We define $R_{j \leftarrow k}$ as the share of relevance that flows from neuron k to neuron j. This share is determined based on the contribution of neuron j to R_k, subject to the following local relevance conservation constraint:

$$\sum_j R_{j \leftarrow k} = R_k \tag{4.15}$$

On the other hand, the relevance of a neuron in the lower layer is then defined as the total relevance it receives from the higher layer:

$$R_j = \sum_k R_{j \leftarrow k} \tag{4.16}$$

Combining the equations 4.15 and 4.16, a **relevance conservation property** between all consecutive layers is ensured. This, in turn, also leads to a global conservation property from the neural network output to the input relevance scores:

$$\sum_{i=1}^{m} R_i = \ldots = \sum_j R_j = \sum_k R_k = \ldots = f(\mathbf{x}) \tag{4.17}$$

The LRP procedure is shown graphically in the figure below.

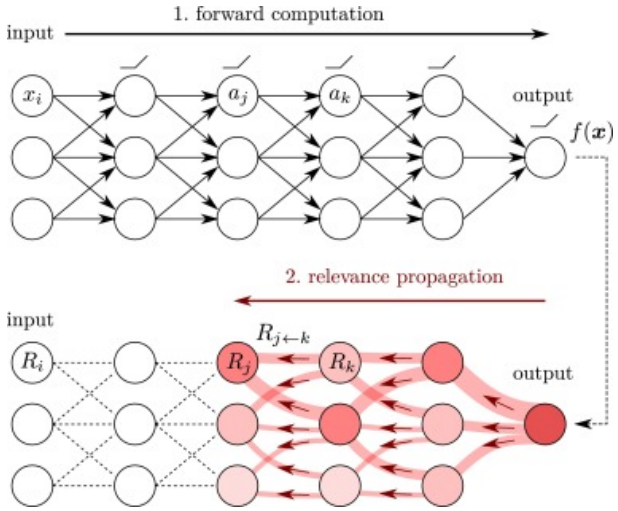


Figure 4.10: Diagram of the LRP procedure

[Bach et al., 2015] introduced two relevance propagation rules. Let the neurons of the DNN be described by the equation

$$a_k = \sigma\left(\sum_j a_j w_{jk} + b_k\right) \tag{4.18}$$

with a_k the neuron activation, $(a_j)_j$ the activations from the previous layer, w_{jk}, b_k the weight and bias parameters of the neuron and σ the activation function. The first rule, called ϵ-LRP, consists of adding a small positive term ϵ in the denominator:

$$R_j = \sum_k \frac{a_j w_{jk}^+}{\epsilon + \sum_j a_j w_{jk}^+} R_k \tag{4.19}$$

The role of ϵ is to absorb some relevance when the contributions to the activation of neuron k are weak or contradictory. As ϵ becomes larger, only the most salient explanation factors survive the absorption. It prevents the numerical instability for the case in which the denominator becomes zero, so it typically leads to explanations that are less noisy.

The second rule that fulfills local conservation properties is the $\alpha\beta$-rule, given by:

$$R_j = \sum_k \left(\alpha \frac{a_j w_{jk}^+}{\sum_j a_j w_{jk}^+} - \beta \frac{a_j w_{jk}^-}{\sum_j a_j w_{jk}^-}\right) R_k \tag{4.20}$$

where $()^+$ and $()^-$ denote the positive and negative influences respectively. In this way, LRP performs and then merges separate decompositions for the activatory (w_{jk}^+) and inhibitory

(w_{jk}^{-}) parts of the forward pass. Here, the non-negative α parameter permits a weighting of relevance distribution towards activations and inhibitions. The β parameter is given implicitly s.t. $\alpha - \beta = 1$ in order to uphold conservativity of relevance between layers. Different combinations of parameters α, β allow us to modulate the qualitative behavior of the resulting explanation, as we will see in Chapter 7.

However, propagating the positive and negative relevance simultaneously without considering the amount and direction of the contribution, may lead to defective interpretation. [Nam et al., 2020] found some shortcomings of both rules described so far. In their experiments, they propagated the relevance recursively using the $\alpha\beta$-rule with $\alpha = 1$ and $\beta = 0$ from the output layer to the input, obtaining the positive propagation image. After that, they did the same process, but only for the negative value ($\alpha = 0$ and $\beta = 1$), obtaining the negative propagation image. It turned out that the same locations of the object received both high positive relevance and high negative relevance. When these are combined, using the $\alpha\beta$-rule, many of the positive relevance values are canceled out by equally large negative values, except for specific locations in which one contribution dominates. Consequently, these locations can be either positive or negative and appear in close proximity to each other.

Motivated by the above issues, [Nam et al., 2020] propose a new perspective for interpreting the relevance of each neuron, accounting for each neuron's influence among connected neighbors and allocating it with relative importance. Their method, called **Relative Attributing Propagation** (**RAP**), redistributes the relevance by changing the priority and rearranging it into positive and negative while preserving the conservation. This way, the relevance is assigned to each neuron directionally in line with the degree of importance to the output, which is highly focused on the object, rather than on the sign of the neuron's contribution, which leads to a similar distribution of the positive and negative attributions. In order to separate the relatively (un)important neurons according to their influence across the layers, RAP method has three main steps, illustrated in the following figure (Fig.4.11).

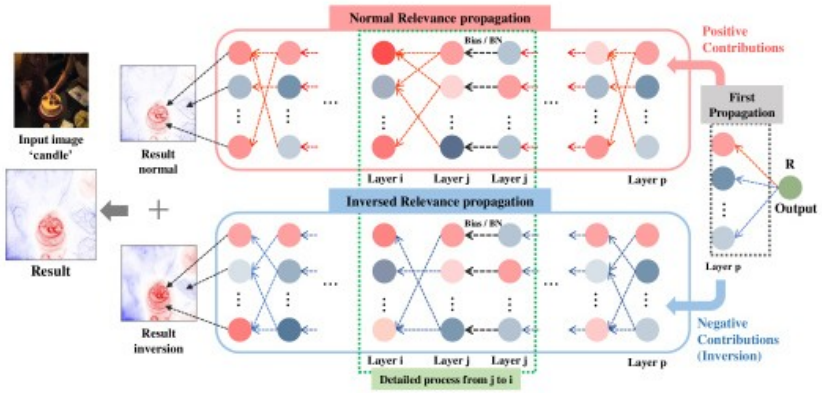


Figure 4.11: Overall structure of RAP algorithm. After the first propagation of relevance, RAP is separated into two flows. Red and blue colors in weights and neurons denote positive and negative values, respectively. After both propagations are finished, the final result is produced by adding the result of each case.

In this way, RAP distinguishes between the relevant and unimportant parts of the input image without overlapping each other.

Another important method for decomposing the output prediction of a neural network

on a specific input is **DeepLIFT** of [Shrikumar et al., 2017]. It explains the difference in output from some "reference" output in terms of the difference of the input from some "reference" input. The "reference" input represents some default or "neutral" input that is chosen according to what is appropriate for the problem at hand. Formally, let t represent some target output neuron of interest and let $x_1, x_2, \ldots, x_n$ represent some neurons in some intermediate layer. Let t_0 represent the reference activation of t. We define the quantity Δt be the *difference-from-reference*, that is $\Delta t = t - t^0$. DeepLIFT assigns contribution scores $C_{\Delta x_i \Delta t}$ to Δx_i s.t. it is satisfied the *summation-to-delta property*:

$$\sum_{i=1}^{n} C_{\Delta x_i \Delta t} = \Delta t \tag{4.21}$$

$C_{\Delta x_i \Delta t}$ can be regarded as the amount of difference-from-reference in t that is attributed to or "blamed" on the difference-from-reference of x_i. Furthermore, $C_{\Delta x_i \Delta t}$ can be non-zero even when $\frac{\partial t}{\partial x_i}$ is zero. This allows DeepLIFT to address a fundamental limitation of gradients, which cause noisy attribution maps. For a given input neuron x with difference-from-reference Δx, and target neuron t with difference-from-reference Δt that we wish to compute the contribution to, we define the **multiplier** $m_{\Delta x \Delta t}$ as:

$$m_{\Delta x \Delta t} = \frac{C_{\Delta x \Delta t}}{\Delta x} \tag{4.22}$$

In other words, the multiplier $m_{\Delta x \Delta t}$ is the contribution of Δx to Δt divided by Δx.

However, in some situations, it is essential to treat positive and negative contributions differently. To do this, let us consider a neuron y with inputs $x_1, x_2, \ldots$ such that $y = f(x_1, x_2, \ldots)$. For every neuron y, we introduce Δy^+ and Δy^- to represent the positive and negative components of Δy, such that:

$$\Delta y = \Delta y^+ + \Delta y^- \qquad C_{\Delta y \Delta x} = C_{\Delta y^+ \Delta x} + C_{\Delta y^- \Delta x} \tag{4.23}$$

Now, we present two rules for assigning contribution scores for each neuron:

- The **Rescale rule**: for this rule, which applies to nonlinear transformations that take a single input, such as the ReLU, tanh or sigmoid operations, we set Δy^+ and Δy^- proportional to Δx^+ and Δx^- as follows:

$$\Delta y^+ = \frac{\Delta y}{\Delta x} \Delta x^+ = C_{\Delta x^+ \Delta y^+} \tag{4.24}$$

$$\Delta y^- = \frac{\Delta y}{\Delta x} \Delta x^- = C_{\Delta x^- \Delta y^-} \tag{4.25}$$

Based on this, we get:

$$m_{\Delta x^+ \Delta y^+} = m_{\Delta x^- \Delta y^-} = m_{\Delta x \Delta y} = \frac{\Delta y}{\Delta x} \tag{4.26}$$

While the Rescale rule improves upon simply using gradients, there are still some situations where it can provide misleading results: let us consider the $\min(i_1, i_2)$ operation, with reference values of $i_1 = 0$ and $i_2 = 0$. Using the Rescale rule, all importance would be assigned either to i_1 or to i_2. This can obscure the fact that both inputs are relevant for the *min* operation.

- The **RevealCancel rule**: this rule represents one way to address this problem. It treats the positive and negative contributions separately. Instead of assuming that Δy^+ and Δy^- are proportional to Δx^+ and Δx^- and that $m_{\Delta x^+ \Delta y^+} = m_{\Delta x^- \Delta y^-} = m_{\Delta x \Delta y}$, as is done for the Rescale rule, we define them as follows:

$$\Delta y^+ = \frac{1}{2}(f(x^0 + \Delta x^+) - f(x^0)) + \frac{1}{2}(f(x^0 + \Delta x^- + \Delta x^+) - f(x^0 + \Delta x^-)) \quad (4.27)$$

$$\Delta y^- = \frac{1}{2}(f(x^0 + \Delta x^-) - f(x^0)) + \frac{1}{2}(f(x^0 + \Delta x^+ + \Delta x^-) - f(x^0 + \Delta x^+)) \quad (4.28)$$

$$m_{\Delta x^+ \Delta y^+} = \frac{C_{\Delta x^+ \Delta y^+}}{\Delta x^+} = \frac{\Delta y^+}{\Delta x^+} \quad (4.29)$$

$$m_{\Delta x^- \Delta y^-} = \frac{C_{\Delta x^- \Delta y^-}}{\Delta x^-} = \frac{\Delta y^-}{\Delta x^-} \quad (4.30)$$

  where x^0 is the reference activation of the input. In other words, we set Δy^+ to the average impact of Δx^+ after no terms have been added and after Δx^- has been added, and we set Δy^- to the average impact of Δx^- after no terms have been added and after Δx^+ has been added. RevealCancel would assign a contribution of $0.5 \min(i_1, i_2)$ to both inputs.

When formulating the DeepLIFT rules, we talked about a "reference activation". The need of a *baseline* arises from the common way for humans to perform attribution, that is, based on counterfactual intuition. When we assign blame to a certain cause, we implicitly consider the absence of the cause as a baseline for comparing outcomes. However, this requires the ability to test a process with and without a specific feature, which is problematic with current neural network architectures that do not allow to explicitly remove a feature without retraining. Defining a baseline x^0, for example a constant black image or an image of random noise, represents the usual approach to simulate the absence of a feature. When a baseline value has to be defined, zero is the canonical choice, that will represent absence of information, but sometimes a constant color baseline is blind to the baseline color: it could not highlight black pixels as important even if black pixels make up the object of interest. As we can see in the Fig. 4.12, the black baseline is not always the better choice.

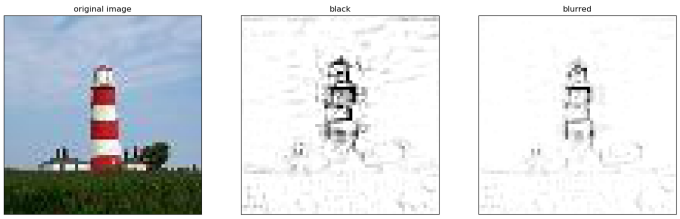


Figure 4.12: Example of explanation for which a blurred version of image as baseline is better than black baseline.

In many domains, it is not clear how to choose a baseline that correctly represents a lack of information, hence choosing the right baseline remains a challenge.

Besides DeepLIFT, other methods require choosing a baseline, like **Integrated Gradients**, the method proposed by [Sundararajan et al., 2017]. Unlike previously proposed methods, Integrated Gradients do not need any instrumentation of the network and can be computed easily using a few calls to the gradient operation.

Formally, suppose we have a function $F : \mathbb{R}^n \longrightarrow [0,1]$ that represents a deep neural network. Specifically, let $\mathbf{x} \in \mathbb{R}^n$ be the input and $\mathbf{x}' \in \mathbb{R}^n$ be the baseline input. We consider the straight line path in $\mathbb{R}^n$ from the baseline $\mathbf{x}'$ to the input $\mathbf{x}$ and compute the gradients at all points along the path. Integrated gradients are obtained by accumulating these gradients. The method of [Sundararajan et al., 2017] defines the importance value for the ith feature value as follows:

$$IG_i(\mathbf{x}) = (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F(\mathbf{x}' + \alpha \times (\mathbf{x} - \mathbf{x}')}{\partial x_i} d\alpha \tag{4.31}$$

where $\frac{\partial F(\mathbf{x})}{\partial x_i}$ is the gradient of $F(\mathbf{x})$ along the i-th dimension. By integrating over a path, integrated gradients avoid problems with local gradients being saturated. However, this integral can be efficiently approximated via a summation. We simply sum the gradients at points occurring at sufficiently small intervals along the straight line path from the baseline x' to the input $\mathbf{x}$:

$$IG_i^{\text{approx}}(\mathbf{x}) = (x_i - x_i') \times \sum_{k=1}^{m} \frac{\partial F(\mathbf{x}' + \frac{k}{m} \times (\mathbf{x} - \mathbf{x}')}{\partial x_i} \times \frac{1}{m} \tag{4.32}$$

where m is the number of steps in the Riemman approximation of the integral.

As mentioned before, choosing the appropriate baseline is not easy: a good solution is to average over multiple different baselines. Although doing so may not be particularly natural for constant color images (which colors do you choose to average over and why?), it is a very natural notion for baselines drawn from distributions. Simply draw more samples from the same distribution and average the importance scores from each sample. When we average over multiple baselines from the same *distribution* D, we are attempting to use the distribution itself as our baseline. The **Expected Gradients** method of [Erion et al., 2019] is based on this idea. It is an extension of Integrated Gradients with fewer hyperparameter choices. Indeed, it is possible to avoid specifying $\mathbf{x}'$, defining the expected gradients value for feature i as:

$$EG_i(\mathbf{x}) = \int_{\mathbf{x}'} \left((x_i - x_i') \int_{\alpha=0}^{1} \frac{\partial F(\mathbf{x}' + \alpha \times (\mathbf{x} - \mathbf{x}')}{\partial x_i} d\alpha \right) p_D(\mathbf{x}') d\mathbf{x}' \tag{4.33}$$

where D is underlying data distribution. However, directly integrating over the training distribution is intractable, so we instead reformulate the integrals as expectation:

$$EG_i(\mathbf{x}) = \mathbb{E}_{\mathbf{x}' \sim D, \alpha \sim U(0,1)} \left[(x_i - x_i') \frac{\partial F(\mathbf{x}' + \alpha \times (\mathbf{x} - \mathbf{x}')}{\partial x_i} \right] \tag{4.34}$$

This expectation-based formulation lends itself to a natural sampling based approximation method: draw samples of $\mathbf{x}'$ from the training dataset and α from $U(0,1)$, compute the value inside the expectation for each sample, and average over samples. In this way, Expected Gradients removes the choice of a single reference value that many existing feature attribution methods require, instead using the training distribution as a reference.

Expected gradients and Integrated Gradients belong to a family of methods known as **path attribution methods**, because they integrate gradients over one or more paths between two valid inputs. The most important aspect of these methods is that are the only attribution methods that always satisfy the axioms of **Sensitivity** (or *Dummy*), **Implementation Invariance**, **Linearity** and **Completeness**. An axiom is a self-evident

property of the attribution method that should be satisfied for any explanation generated by the method itself. By leveraging these properties, attribution methods with stronger theoretical guarantees can be designed.

The first axiom states that if the function implemented by the deep network does not depend (mathematically) on some variable, then the attribution to that variable is always zero. Saliency Map violates Sensitivity, because the prediction function may flatten at the input and thus have zero gradient despite the function value at the input being different from that at the baseline (saturation problem). For the same reason, also DeconvNet and Guided Backpropagation violate this property. Unlike these, methods like DeepLIFT and LRP tackle the sensitivity issue by employing a baseline, and in some sense try to compute "discrete gradients" instead of instantaneous gradients at the input: the idea is that a large, discrete step will avoid flat regions, avoiding a break-age of sensitivity. Before seeing the Implementation Invariance axiom, it is useful to define that two networks are *functionally equivalent* if their outputs are equal for all inputs, despite having very different implementations. It is clear that attribution methods should satisfy Implementation Invariance, i.e., the attributions are always identical for two functionally equivalent networks. However, DeepLIFT and LRP break this property: the chain rule does not hold for discrete gradients in general. For the Linearity axiom: suppose that we linearly composed two deep networks modeled by the functions f_1 and f_2 to form a third network that models the function $a \times f_1 + b \times f_2$, i.e. a linear combination of the two networks. Then we would like the attributions for $a \times f_1 + b \times f_2$ to be the weighted sum of the attributions for f_1 and f_2 with weights a and b respectively. Finally, for the Completeness axiom, the attributions add up to the difference between the output of F at the input $\mathbf{x}$ and the baseline $\mathbf{x}'$, as formalized here:

$$\sum_i R_i(\mathbf{x}) = F(\mathbf{x}) - F(\mathbf{x}') \tag{4.35}$$

We will take up these concepts later, placing them in the context of game theory.

4.3.2 Perturbation-based methods

The philosophy of perturbation-based interpretability is perturbing the original input and observing how prediction score changes when the feature is altered: perturbation-based methods directly compute the attribution of an input feature (or set of features) by removing, masking or altering them, and running a forward pass on the new input, measuring the difference with the original output, as a consequence of this operation. This line of works tries to answer the question: which parts of the input, if they were not seen by the model, would most change its prediction? The perturbation is performed across features sequentially to determine their contributions, and can be implemented in two ways: *omission* and *occlusion*. For omission, a feature is directly removed from the input, but this is impractical in practice since few models allow setting features as unknown. As for occlusion, the feature is replaced with a reference value, such as zero for word embeddings or specific gray value for image pixels. Unlike the gradient-based methods, perturbation-based methods allow a direct estimation of the marginal effect of a feature, so they tend to be very slow as several evaluations of the network are necessary.

Occlusion-based

The simplest perturbation method of this category is **Occlusion**, the method proposed by [Zeiler and Fergus, 2014]. This technique has been applied to Convolutional Neural Networks in the domain of image classification to visualize the probability of the correct class as a function of the position of a grey patch occluding part of the image. Using a patch of pixels, as it captures the notion of multiple pixels as a feature, yields better results than single pixel occlusion: observing output change by removing one single element does not take the interdependence between elements into account. Notice that the procedure of replacing features with a zero value implicitly defines a baseline that can be used to indicate features that are toggled off.

A representative and pioneer framework that generates local explanations of black-box models performing superpixel occlusion is **LIME** (Local Interpretable Model-agnostic Explanations) created by [Ribeiro et al., 2016]. This technique generates the explanations by approximating the underlying complex model by an interpretable surrogate one, such as a linear model, learned on perturbations of the original instance. The key intuition behind LIME is that it is much easier to approximate a black-box model by a simple model *locally*, so in the neighborhood of the prediction we want to explain, as opposed to trying to approximate a model globally (Fig. 4.13). This is done by weighting the perturbed images by their similarity to the explained instance.

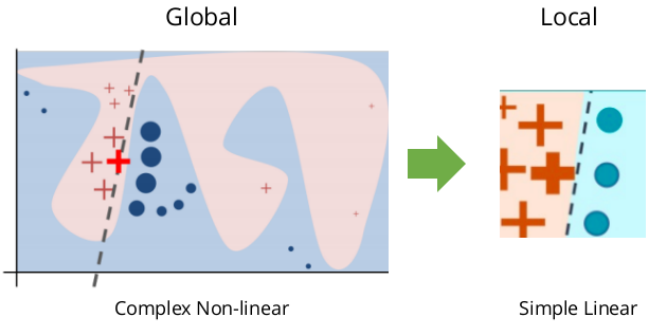


Figure 4.13: LIME localises a problem and explains the model at that locality, rather than generating an explanation for the whole model.

Formally, we denote $\mathbf{x} \in \mathbb{R}^d$ be the original representation of an instance being explained, and we use $\mathbf{x}' \in \{0,1\}^d$ to denote a binary vector for its **interpretable representation**. For image classification, an interpretable representation may be a binary vector indicating the "presence" or "absence" of a contiguous patch of similar pixels (a super-pixel). We define an explanation as a model $g \in G$, where G is a class of potentially interpretable models, such as linear models, whose domain of is $\{0,1\}^d$, i.e. g acts over absence/presence of the interpretable components. As not every $g \in G$ may be simple enough to be interpretable, we let $\Omega(g)$ be a measure of complexity (as opposed to interpretability) of the explanation $g \in G$. For example, while for linear models, $\Omega(g)$ may be the number of non-zero weights. Let the model being explained be denoted $f : \mathbb{R}^d \longrightarrow \mathbb{R}$. We further use $\pi_{\mathbf{x}}(\mathbf{z})$ as a proximity measure between an instance $\mathbf{z}$ to $\mathbf{x}$, so as to define locality around x. Finally, let $\mathcal{L}(f, g, \pi_{\mathbf{x}})$ be a measure of how unfaithful g is in approximating f in the locality defined by $\pi_{\mathbf{x}}$. In

order to ensure both interpretability and local fidelity, we must minimize $\mathcal{L}(f, g, \pi_{\mathbf{x}})$ while having $\Omega(g)$ be low enough to be interpretable by humans. The explanation produced by *LIME* is obtained by the following:

$$\xi(\mathbf{x}) = argmin \ \mathcal{L}(f, g, \pi_{\mathbf{x}}) + \Omega(g) \tag{4.36}$$

This formulation can be used with different explanation families G, fidelity functions $\mathcal{L}$, and complexity measures Ω.

By presenting explanation as an optimisation problem to find a trade-off between local fidelity of explanation and its interpretability, however, it offers no guarantees that the explanations are faithful and stable. Using neighbourhood around explanation instances, it may fall into a curse of dimensionality trap. This problem is fueled by the limited theoretical understanding of some of these methods and lack of reliable quantitative metrics to evaluate explanations in the absence of a groundtruth.

Shapley values-based

To overcome this issue, an idea worth pursuing seems to be an axiomatic approach based on the **Shapley values**. They are classic game theory solutions for the distribution of credits to players participating in a cooperative game. Such an approach addresses another limitation of LIME: the ordering of variables has an impact on the contributions calculated, especially for models that are non-additive. Shapley-based approaches ameliorate this issue by averaging the value of a variable's contribution over all, or a large number, of possible orderings. Shapley values can be considered a particular example of perturbation-based methods, known to be the unique method that satisfies certain properties, where no hyperparameters, except the baseline, are required. A popular model agnostic explanation method based on these values is **SHAP** (SHapley Additive exPlanations), proposed by [Lundberg and Lee, 2017] as an unified framework of different commonly used techniques for model explanations, which go by the name of **additive feature attribution methods**, like DeepLIFT, LRP and LIME in addition to the classic Shapley values' estimation techniques. It is about Shapley values of a conditional expectation function of the original model as the set function: SHAP values attribute to each feature the change in the expected model prediction when conditioning on that feature. It has been observed that attributions based on Shapley values better agree with the human intuition empirically. Unfortunately, computing Shapley values exactly requires us to evaluate all 2^N possible feature subsets. Different ways of approximate them were proposed: a model-agnostic approach, known as **KernelSHAP** which takes advantage of LIME, a faster model-specific technique called **DeepSHAP** which comes from the connection between Shapley values and DeepLIFT and last but not least **DASP**, a perturbation-based method that can reliably approximate Shapley values in DNNs with a polynomial number of perturbation steps.

Despite it can be argued to be the "gold standard" for model interpretability, because it provides both local and global measures of feature attribution and through the KernelSHAP algorithm is model-agnostic, many later works highlighted his weaknesses, first of all unconditional expectations used as approximation for the conditional ones, justified by the simplifying assumption of *feature independence*.

Chapter 5

Validation of Explanation Methods

In the previous chapter, we gave an overview of various explanation methods that fit well to deep neural networks for computer vision. As we have seen, these are reliable, efficient algorithms, many of which are inexpensive from the computational point of view, well able to provide explanations that highlight the most important regions of the input image for the prediction made by the network.

However, explanation methods suffer from a limitation that unites them all: the difficulty of being validated. Despite the significant results, it remains the difficulty of assessing the scope and quality of such explanations. The problem arises why for attribution, no ground truth exists. If an attribution heatmap highlights subjectively irrelevant areas, this might correctly reflect the network's unexpected way of processing the data, or the heatmap might be inaccurate.

Then, who assures us that the explanation provided by the attribution method really highlights the most important region for the network? Do the pretty visualizations actually tell us reliably about what the network is doing internally? Who assures us that that visualization is not biased? Among an abundance of competing methods, what is the method that gives us the correct explanation?

5.1 Related works

[Adebayo et al., 2018] first tried to answer these questions by proposing a sanity check of some explanation methods to perform before deploying them in practice. Their work arises from the need for an explanation method to be sensitive to the parameters of the model. The parameter settings of a model encode in fact what the model has learned from the data during training and determine test set performance. So, if the method really highlights the most important region of the input, randomly re-initializing the parameters of the last fully connected layer, then changing the network, the explanation given should change. A network with the last layer fully randomized should give very different importance to the input than a network that has the original trained configuration. However, for some methods their experiments show the opposite: the heatmaps provided by Guided Backpropagation and Guided GradCAM methods stay identical also after the randomization of several layers of the network. Consequently, [Adebayo et al., 2018] consider such methods inadequate for computer vision tasks that are sensitive to model, such as, explaining the relationship between inputs and outputs that the model learned and debugging the model: a method ignoring the last layer cannot explain the network's prediction faithfully.

After the publication of the work of [Adebayo et al., 2018], another group of researchers, [Sixt et al., 2019], recently repeated the sanity check on several other explanation methods. In addition to [Adebayo et al., 2018], which only reported Guided Backpropagation and Guided GradCam to fail the test, they found many other methods fail too: LRP, Deep Taylor Decomposition (DTD), PatternAttribution, Excitation Backpropagation, DeconvNet and RectGrad. Of the methods tested, instead, Saliency, GradCAM, LRP $\alpha_5\beta_4$, DeepLIFT and Integrated Gradients pass the sanity check.

5.2 Methodology

To assess the sensitivity to model parameters of explanation methods, [Adebayo et al., 2018] conducted two kinds of model parameter randomization test:

- **Cascading randomization**
  In the cascading randomization, they randomly re-initialize all weights of the model in a cascading fashion, destroying completely the learned weights. In detail, they randomize the weights of the network, starting from the top layer (last fully-connected), successively, all the way to the bottom layer (first convolutional).

- **Independent randomization**
  As a different form of the model parameter randomization test, they conduct an independent layer-by-layer randomization with the goal of isolating the dependence of the explanations by layer. In the independent randomization, they randomly re-initialize the weights of a single layer at a time, while keeping all others fixed.

In both cases, they compare the resulting explanation from a network with random weights to the one obtained with the trained original weights. After a comparison with the edge detector, that does not depend on model or training data but yet produced results that bear visual similarity with saliency maps, they show that visual inspection is a poor guide in judging whether an explanation is sensitive to the underlying model. Given the need for a quantitative metric of comparison, they rely on the Structural Similarity Index Measure (**SSIM**) for both randomization tests. SSIM is a method for measuring the similarity between two images, a reference image and a processed image, from the same image capture. In this case, they compare the original heatmap (reference image) with the one generated after weights randomization (processed image). More formally, given two images x and y of common size $N \times N$, the SSIM index is computed as follows:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{5.1}$$

where μ_x and μ_y are the average of x and y respectively, σ_x^2 and σ_y^2 ate the variance of x and y respectively, σ_{xy} is the covariance of x and y, $c_1 = (k_1 L)^2$ and $c_2 = (k_2 L)^2$ are two variables to stabilize the division with weak denominator, where L is the dynamic range of the pixel-values (typically this is $2^{\#bitsperpixel} - 1$) and $k_1 = 0.01$, $k_2 = 0.03$ by default.

The method fails the test if the explanation provided with the last fully connected layer randomized shows a SSIM equal to 1 with the original explanation.

5.3 Theoretical analysis

Why do some methods fail the sanity check and therefore are not reliable in interpreting how deep neural networks make classification decisions? Many researchers have tried to justify this behavior. In this section, we provide an overview of the most interesting theoretical insights provided recently.

5.3.1 Bottleneck information

[Mahendran and Vedaldi, 2016] showed that some backpropagation-based saliency methods lack neuron discriminativity. They focused in particular on the DeconvNet method and found that the output of reversed architectures is mainly determined by the bottleneck information rather than by which neurons are selected for explanation. By bottleneck information they mean limited information computed by the forward pass: the setting of the pooling switches in the max pooling layers and the setting of the mask (binary tensor with a 1 for every positive element of input and 0 otherwise) in the ReLU units (Fig. 5.1).

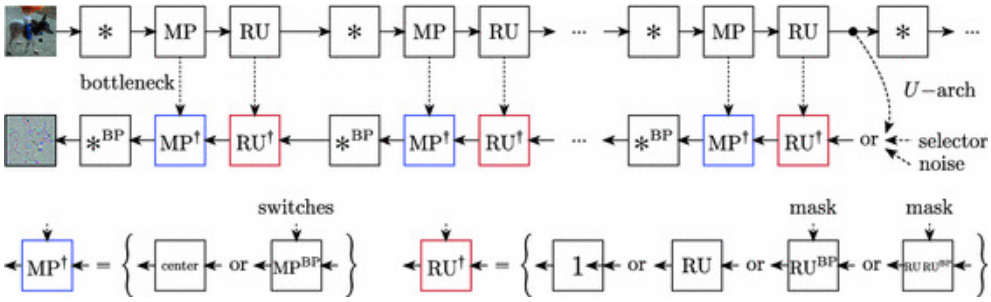


Figure 5.1: The top row shows a typical CNN obtained by repeating short chains of convolution ($\star$), max pooling (MP) and ReLU (RU) operators. The middle row shows a generic deconvolutional architecture, in which information flows backward to the image using the convolution transpose ($\star^{BP}$), the backward ReLU (RU^T) and backward max pooling (MP^T)

These informations contain much less information (bottleneck info) than the input data, but they play a crucial role in determining the explanation. More formally, each layer ϕ_i of a CNN is associated with a corresponding layer ϕ_i^T that reverses input $\mathbf{x}$ and output $\mathbf{y}$. The reverse layer is influenced by auxiliary (bottleneck) information $\mathbf{r}$ computed by the forward layer. Thus a layer ϕ_i and its reverse ϕ_i^T are maps:

$$\text{forward} \quad \phi_i : \mathbf{x} \longrightarrow (\mathbf{y}, \mathbf{r}) \qquad \text{reversed} \quad \phi_i^T : (\hat{\mathbf{y}}, \mathbf{r}) \longrightarrow \hat{\mathbf{x}} \tag{5.2}$$

The $\hat{}$ symbol emphasizes that, in the backward direction, the tensors $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ have the same shape as $\mathbf{x}$ and $\mathbf{y}$ in the forward pass, but different values. In their experiments, denoted with $\phi^T(e_i, \mathbf{r})$ the reverse network, function of the neuron indicator e_i as well as the bottleneck information $\mathbf{r}$ extracted from the forward pass through the network ϕ, the response changed very little and insignificantly with different choices of e, by keeping $\mathbf{r}$ fixed. If the output of $\phi^T(e_i, \mathbf{r})$ had been a direct characterization of the i-th neuron, they would have expected the generated image meaningfully would have changed as the input e_i to the deconvolutional network changed. Therefore, [Mahendran and Vedaldi, 2016] conclude that DeconvNet does not depend on the learned network weights or data and this explains why in the work of [Sixt et al., 2019] this method fails the sanity check. Please refer to their paper for further insights.

5.3.2 Image recovery

Along similar lines, the analysis made by [Nie et al., 2018] shows that the backward ReLU introduced by Guided Backpropagation and DeconvNet, together with the local connections in convolutional neural networks, are the two main causes of much cleaner visualizations than Saliency. However, the theoretical explanation provide by [Nie et al., 2018], reveals that Guided Backpropagation and DeconvNet essentially do a (partial) image recovery instead of highlighting class-relevant pixels or visualizing the learned weights, which means in principle they are unrelated to the decision-making of neural networks. At the first glance, in fact, the Guided Backpropagation visualizations are very similar to the results of an edge detector. In other words, Guided Backpropagation pays much attention to the edge information like a Gabor filter. However, Guided Backpropagation has the additional ability to filter out some background image patches. Anyway, after introducing the backward ReLU, both DeconvNet and Guided Backpropagation modify the true gradient in a way that they create much cleaner results but their functionality as an indicator of important pixels to a specific class has disappeared. For the same reason, their visualizations are less class-sensitive than saliency map, as we will see in Chapter 7.

5.3.3 The convergence problem

[Sixt et al., 2019], in their similar work to assess the faithfulness of new and existing modified back propagation methods, have provided theoretical insights for this surprising behavior and also analyzed why DeepLift does not suffer from this limitation. To explain their theoretical analysis, we need to take a step back. They consider a feed-forward neural network with a ReLU activation function $[x]^+ = max(0, x)$ and n layers, each with weight matrices W_l (to simplify notation, they absorb the bias terms into the weight matrix). The output of the l-th layer is denoted by h_l and is given by

$$h_l = [W_l h_{l-1}]^+, \tag{5.3}$$

while they refer to the input with $h_0 = \mathbf{x}$ and to the output with $h_n = f(\mathbf{x})$. The gradient of the k-th output of the neural network w.r.t. the input $\mathbf{x}$ is a product of matrices given by:

$$\frac{\partial f_k(\mathbf{x})}{\partial \mathbf{x}} = W_1^T M_1 \frac{\partial f_k(\mathbf{x})}{\partial \mathbf{h}_1} = \prod_l^n (W_l^T M_l)\mathbf{v}_k \tag{5.4}$$

where $M_l = diag(1_{h_l>0})$ denotes the gradient mask of the ReLU operation, the vector $\mathbf{v}_k$ is a one-hot vector to select the k-th output and the last equality follows from recursive expansion. As we can see, the derivative of output with respect to the input image is just the weight vector of the model. Therefore, Saliency method visualizes the learned weights and for this reason the explanation provided with the last fully-connected layer destroyed is very different from the original one.

The modified back propagation methods studied by [Sixt et al., 2019] modify this definition of gradient and estimate relevant areas by backpropagating a custom relevance score instead of the gradient. Methods like Deep Taylor Decomposition, LRP $\alpha_1\beta_0$ and Excitation Backpropagation, all failing the sanity check, use the z^+-rule, that yields a multiplication chain of non-negative matrices. Each matrix corresponds to a layer and the attribution map is a function of this matrix chain. The z^+-rule back propagates only positive relevance values, which are supposed to correspond to the positive evidence for the prediction.

Let w_{ij} be an entry in the weight matrix W_l, the relevance at layer l for an input $\mathbf{x}$ is given by:

$$r_l^{z^+}(\mathbf{x}) = Z_l^+ \cdot r_{l+1}^{z^+}(\mathbf{x}) \qquad \text{where} \qquad Z_l^{+T} = \left(\frac{[w_{ij}\mathbf{h}_{l_{[j]}}]^+}{\sum_k [w_{ik}\mathbf{h}_{l_{[k]}}]^+} \right)_{[ij]} \tag{5.5}$$

Each entry in the derivation matrix Z_l^+ is obtained by measuring the positive contribution of the input neuron i to the output neuron j and normalizing by the total contributions to neuron j. The relevance from the previous layer $r_{l+1}^{z^+}$ is then distributed according to Z_l^+. In contrast to the vanilla backpropagation, algorithms using the z^+-rule do not apply a mask for the ReLU activation. The relevance of multiple layers is computed by applying the z^+-rule to each of them. Similar to the gradient, we obtain a product of non-negative matrices: $C_k = \prod_l^k Z_l^+$. The following theorem states a result that justifies the behaviour of these back propagation methods:

Theorem 1. *Let $A_1, A_2, \ldots$ be a sequence of non-negative matrices for which $lim_{n\longrightarrow\infty} A_n$ exists. Excluding the cases where one column of $lim_{n\longrightarrow\infty} A_n$ is the zero vector or two columns are orthogonal to each other, then the product of all terms of the sequence converges to a rank-1 matrix $\bar{C}$:*

$$\bar{C} := \prod_{i=1}^{\infty} A_i \tag{5.6}$$

which can be written as an outer product $\bar{C} = \bar{\mathbf{c}}\gamma^T$, $\bar{C} \in \mathbb{R}^{n\times m}$, $\bar{\mathbf{c}} \in \mathbb{R}^n$, $\gamma \in \mathbb{R}^m$.

The geometric intuition of the proof is depicted in Fig 5.2. The column vectors of the first matrix are all non-negative and therefore in the positive quadrant. For the matrix multiplication A_iA_j, observe that A_ia_k is a non-negative linear combination of the column vectors of A_i, where a_k is the k-th column vector $A_{j_{[:k]}}$. The result will remain in the convex cone of the column vectors of A_i. The conditions stated in the theorem ensure that the cone shrinks with every iteration and it converges towards a single vector. Analogously, if an attribution method converges, the contributions of the layers shrink by depth.

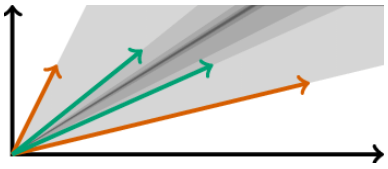


Figure 5.2: The positive column vectors a_1, a_2 of matrix A_1(orange) form a cone. The resulting columns of A_1A_2 (green) are contained in the cone as they are positive linear combinations of a_1, a_2. At each iteration, the cone shrinks.

The column vectors of a rank-1 matrix are linearly dependent $C = \mathbf{c}\gamma^T$, so a rank-1 matrix C always gives the same direction of $\mathbf{c}$: $CZ_{k+1}^+ = \mathbf{c}\gamma^T Z_{k+1}^+ = \mathbf{c}\lambda^T$ and for any vector $\mathbf{v}$: $CZ_{k+1}^+\mathbf{v} = \mathbf{c}\lambda^T\mathbf{v} = t\mathbf{c}$, where $t \in \mathbb{R}$. In this way, the influence of later matrices decreases and, since Z^+ matrices of fully-connected layers fulfill the conditions of the theorem, this explains why the explanations produced by the network with the last randomized layer remain identical to the original ones.

5.3.4 Why does DeepLift pass the test?

DeepLift is the only tested modified back propagation method which does not converge to a rank-1 matrix. Additionally to the vanilla gradient, DeepLIFT separates positive and negative contributions. The rule for linear layers is most interesting because it is the reason why DeepLIFT does not converge:

$$r_l^{DL+}(x, x_0) = M_{>0}^T \odot \left({W_l^+}^T r_{l+1}^{DL+}(x, x_0) + {W_l^-}^T r_{l+1}^{DL-}(x, x_0) \right) \tag{5.7}$$

where x_0 is a so-called reference point and the mask $M_{>0}$ selects the weight rows corresponding to positive deltas ($0 < \Delta \mathbf{h}_l = h_l - h_l^0$). For negative relevance r_l^{DL-}, the rule is defined analogously. An interesting property of this rule is that negative and positive relevance can influence each other. If the intermixing is removed by only considering W^+ for the positive rule and W^- for the negative rule, the two matrix chains become decoupled and converge to a rank-1 matrix. For the positive chain, this is clear. For the negative chain, observe that the multiplication of two non-positive matrices gives a non-negative matrix.

5.3.5 Other methods

The paper of [Sixt et al., 2019] focuses on modified back propagation methods, as other attribution methods do not suffer from the converges problem. Methods like Input $\odot$ Gradient, Smooth Grad and Integrated Gradients are heavily dependent on the weights because they rely on the gradient directly and therefore they do not converge to a rank-1 matrix. Differently, also methods like LIME or SHAP pass the test because they consider the model as a black-box, so they generate explanations without access to the internal model parameters.

Only when the back propagation algorithm is modified, the convergence problem can occur.

In Chapter 7, we will repeat this experiment on an even larger set of methods, including several versions of LRP $\alpha\beta$ and some belonging to the SHAP framework.

Chapter 6

The SHAP framework

The combination of a solid theoretical justification and a fast practical algorithm makes the SHAP framework a powerful tool for confidently interpreting machine learning models predictions. In this chapter, we will delve into this class of methods describing the link between Game Theory and Machine Learning and the properties that characterize it.

6.1 Shapley values: from the Game Theory...

How Shapley values, an idea with roots in game theory, are being applied to innovative ways to explain machine learning predictions? We will start by understanding the problem that economist Lloyd Shapley wanted to solve when he introduced his work [Shapley, 1953]: a group of differently skilled participants are all cooperating with each other for a collective reward. How should the reward be fairly divided amongst the group?

Let us define a toy scenario: four friends **A**shley, **B**en, **C**laire and **D**on are leaving the bar one night after work. They realized that they all live along the same route so they can split an Uber ride home, but the distance between stops is not evenly split and the total cost is some complicated function of distance duration surge pricing and tolls. What is the *fairest* way to split the contributions from each rider? In this case, we should consider what the fare would be for each possible subset of riders: Ashley riding alone, Ashley and Mark, Mark and Thomas but no Ashley and so on, and look at the marginal contribution to the cost when each rider joins the each possible subset. According to Shapley the fairest contribution for each rider is their average marginal contribution over all these possible subsets.

Expressing the concept in mathematical formulas: let us consider a set of N players P and a set function $f : 2^N \longrightarrow \mathbb{R}$, that maps each subset $S \subseteq P$ of players to real numbers, modeling the outcome $f(S)$ of a game when players in S participate in it. The Shapley value is one way to quantify the total contribution of each player to the result $f(P)$ of the game when all players participate. For a given player i, it can be computed as follows:

$$\phi_i = \sum_{S \subseteq P \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} [f(S \cup \{i\}) - f(S)] \tag{6.1}$$

The Shapley value for player i defined above can be interpreted as:

- the average: $\frac{|S|!(N-|S|-1)!}{N!}$
- marginal contribution of player i: $[f(S \cup \{i\}) - f(S)]$

- to all possible coalitions S that can be formed without it: $\sum_{S \subseteq P \setminus \{i\}}$

To better understand, the first thing we will do is rewrite Eq. 6.1 somewhat:

$$\phi_i = \frac{1}{N} \sum_{S \subseteq P \setminus \{i\}} \binom{N-1}{|S|}^{-1} [f(S \cup \{i\}) - f(S)] \tag{6.2}$$

Let us start calculating the Shapley value for Don. If we relate this back to the parameters of the Shapley value formula we have that $P = \{A, B, C, D\}$ and $i = D$, so D is our player i and the entire group P consists of all four friends A, B, C and D. With that laid out, let us start by looking a bit closer at this part of the Shapley Value formula: $S \subseteq P \setminus \{i\}$ says that we need to take our group of people and exclude the person that we are focusing on now. Then, we need to consider all of the possible subsets that can be formed. So if we exclude D from the group, we are left with $\{A, B, C\}$. From this remaining group we can form the following 8 subsets, i.e. these are the sets that S can take on:

$$\begin{bmatrix} \emptyset & \{A\} & \{B\} & \{C\} \\ \{AB\} & \{AC\} & \{BC\} & \{ABC\} \end{bmatrix} \tag{6.3}$$

Now, let us turn our focus to this part: $[f(S \cup \{i\}) - f(S)]$ is the marginal value of adding player i to the game. So for any given subset S we are going to compare its value to the value that is has when we also include the player i in it:

$$\begin{bmatrix} \Delta f_{\emptyset,D} & \Delta f_{A,D} & \Delta f_{B,D} & \Delta f_{C,D} \\ \Delta f_{AB,D} & \Delta f_{AC,D} & \Delta f_{BC,D} & \Delta f_{ABC,D} \end{bmatrix} \tag{6.4}$$

We can view each of these as a different scenario that we need to observe in order to fairly assess how much D contributes to the overall cost. Alright, we have now figured out that we need to compute 8 different marginal values. The summation in the Shapley value equation (Eq. 6.1) is telling us that we need to add all them together. However, we also need to scale each marginal value before we do that, which we are told by this part of the Eq. 6.1: $\binom{N-1}{|S|}^{-1}$. It calculates how many permutations of each subset size we can have when we are constructing it out of all remaining team members excluding player i. Or in other words: if we have $N-1$ players, how many groups of size $|S|$ can we form with them? So, for our scenario $N - 1 = 3$. We then use this number to divide the marginal contribution of player i to all groups of size $|S|$. This means that we should apply the following scaling factor to each of our 8 marginal values:

$$\begin{bmatrix} 1\Delta f_{\emptyset,D} & 1\Delta f_{ABC,D} & \frac{1}{3}\Delta f_{A,D} & \frac{1}{3}\Delta f_{B,D} \\ \frac{1}{3}\Delta f_{C,D} & \frac{1}{3}\Delta f_{AB,D} & \frac{1}{3}\Delta f_{AC,D} & \frac{1}{3}\Delta f_{BC,D} \end{bmatrix} \tag{6.5}$$

By adding this scaling factor, we are averaging out the effect that the rest of the team members have for each subset size. This means that we are able to capture the average marginal contribution of D when added to a team of size 0, 1, 2 and 3 regardless of the composition of these teams. Now, we only have one final part of the Shapley Value equation to break down which also at this point should be fairly straightforward to understand: $\frac{1}{N}$. We have one final scaling factor that we need to apply to all of our marginal values before being able to sum them. We have to divide them with the number of players participating in the game, i.e. the number of team members that we have in total. The final piece of the puzzle is to average out the effect of the group size as well, i.e. how much does D contribute

regardless of the size of the team. For our scenario, we do this by dividing with 4. We have now arrived at the point where we can finally compute the Shapley value for D. We have observed how much he marginally contributes to all different constellations of the team that can be formed. We have also averaged out the effects of both team member composition as well as team size which finally allows us to compute:

$$\phi_D = \frac{1}{4} \sum \begin{bmatrix} 1\Delta f_{\emptyset,D} & 1\Delta f_{ABC,D} & \frac{1}{3}\Delta f_{A,D} & \frac{1}{3}\Delta f_{B,D} \\ \frac{1}{3}\Delta f_{C,D} & \frac{1}{3}\Delta f_{AB,D} & \frac{1}{3}\Delta f_{AC,D} & \frac{1}{3}\Delta f_{BC,D} \end{bmatrix} \tag{6.6}$$

There we have it, the Shapley value for D. After we have done this for the rest of the team will know each person's contribution to the X total cost of the car, allowing us to fairly divide the money amongst all team members.

$$X = f(\{A, B, C, D\}) = \phi_A + \phi_b + \phi_C + \phi_D \tag{6.7}$$

6.1.1 Properties of Shapley values

The properties of Shapley values are given as four axioms:

1. **Efficiency**: This assumption forces the model to correctly capture the original predicted value:
$$f(x) = \sum_{i=0}^{M} \phi_i \tag{6.8}$$

2. **Symmetry**: This states that if two features contribute equally to the model then their effects must be the same. Formally, let $\mathbb{I}_S \in \{0,1\}^M$ be an indicator vector equal to 1 for indexes $i \in S$, and 0 elsewhere, and let $f_x(S) = f(h_x^{-1}(\mathbb{I}_S))$. If for all subsets S that do not contain i or j
$$f_x(S \cup \{i\}) = f_x(S \cup \{j\}) \implies \phi_i(f, x) = \phi_j(f, x) \tag{6.9}$$

3. **Null player**: A feature ignored by the model must have an effect of 0: if for all subsets S that do not contain i
$$f_x(S \cup \{i\}) = f_x(S) \implies \phi_i(f, x) = 0 \tag{6.10}$$

4. **Linearity**: The effect a feature has on the sum of two functions is the effect it has on one function plus the effect it has on the other. For any two models f and f'
$$\phi_i(f + f', x) = \phi_i(f, x) + \phi_i(f', x). \tag{6.11}$$

However, [Young, 1985] showed that *linearity* and *null player* can be eliminated using a **Monotonicity** axiom: for any two model functions f and f' if for all subsets S of the simplified input features Z that do not contain i

$$f_x(S \cup \{i\}) - f_x(S) \geq f'_x(S \cup \{i\}) - f'_x(S) \implies \phi_i(f, x) \geq \phi_i(f', x) \tag{6.12}$$

Furthermore, the symmetry is also implied by the monotonicity.

6.2 ...to Machine Learning

What does this have to do with machine learning? We can think of the riders as the features in our model and the total cost of the car as the model output. Hence, a prediction can be explained by assuming that each feature value of the instance is a player in a game where the prediction is the payout. The definition of ϕ_i can be adapted for a neural network in this way: f is the function to analyze, so the map from the input layer to a specific output neuron in a DNN, S is a given subset of input features from the entire input dataset P. The Shapley value is the average marginal contribution of a feature with respect to all subsets of other features, so in other words, the value of the feature i contributed ϕ_i to the prediction of a particular instance compared to the average prediction for the dataset:

$$\phi_i = \sum_{S \subseteq \{x_1,\ldots,x_p\} \setminus \{x_i\}} \frac{|S|!(N - |S| - 1)!}{N!} [f(S \cup \{x_i\}) - f(S)] \tag{6.13}$$

That is the real strength and appeal behind Shapley values. However, it does come at a cost. For a set of N players participating in a game we will have 2^N subsets that we will need to analyse in order to compute the Shapley values. SHAP method makes this computation more practically feasible when applying it to machine learning.

6.3 From Shapley values to SHAP

SHAP's novel components include:

- the perspective of viewing any explanation of a model's prediction as a model itself, which we term the *explanation model*. This lets us define the class of **additive feature attribution methods**, which unifies six current methods;
- SHAP values as a unified measure of feature importance.

6.3.1 Additive feature attribution methods class

The work of [Lundberg and Lee, 2017] starts from the idea that the best explanation of a simple model is the model itself because it perfectly represents itself and is easy to understand. However, for complex models, such as deep networks, we cannot use the original model as its own best explanation because it is not easy to understand. Instead, we must use a simpler **explanation model**, which we define as any interpretable approximation of the original model.

Let f be the original prediction model to be explained and g the explanation model. Here, we focus on local methods designed to explain a prediction $f(x)$ based on a single input x. Explanation models often use *simplified inputs* x' that map to the original inputs through a mapping function $x = h_x(x')$. Local methods try to ensure $g(z') \approx f(h_x(z'))$ whenever $z' \approx x'$.

Additive feature attribution methods are characterized by an explanation model that is a linear function of binary variables:

$$g(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i \tag{6.14}$$

where $z' \in \{0,1\}^M$, M is the number of simplified input features and $\phi_i \in \mathbb{R}$.

Methods with such explanation models attribute an effect ϕ_i to each feature and, summing the effects of all feature attributions, approximate the output $f(x)$ of the original model. To this class of methods belong **LIME**, **DeepLIFT**, **LRP**, Shapley regression values, Shapley sampling values and Quantitative Input Influence.

A surprising attribute of the class of additive feature attribution methods is the presence of a single unique solution in this class with three desirable properties:

1. **Local accuracy** (also named **Completeness**): The explanation model $g(x')$ matches the original model $f(x)$ when $x = h_x(x')$:

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^{M} \phi_i x'_i \tag{6.15}$$

   where $\phi_0 = f(h_x(0))$ represents the model output with all simplified inputs toggled off (i.e. missing).

2. **Missingness** (also named **Dummy**): If the simplified inputs represent feature presence, then missingness requires features missing in the original input to have no impact:

$$x'_i = 0 \implies \phi_i = 0 \tag{6.16}$$

   hence, missingness constrains features where $x'_i = 0$ to have no attributed impact.

3. **Consistency** (also named **Monotonicity**): It states that if a model changes so that some simplified input's contribution increases or stays the same regardless of the other inputs, that input's attribution should not decrease. Formally, let $f_x(z') = f(h_x(z'))$ and $z' \setminus i$ denote setting $z'_i = 0$. For any two models f and f', if

$$f'_x(z') - f'_x(z' \setminus i) \geq f_x(z') - f_x(z' \setminus i) \quad \forall z \in \{0,1\}, \tag{6.17}$$

   then

$$\phi_i(f', x) \geq \phi_i(f, x)$$

   .

From Consistency the Shapley properties Linearity, Dummy and Symmetry follow, as described in the Appendix of [Lundberg and Lee, 2017].

In their work, [Lundberg and Lee, 2017] state that there is only one possible explanation model g s.t $g(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i$ and that satisfies these 3 properties:

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus \{i\})] \tag{6.18}$$

where $|z'|$ is the number of non-zero entries in z' , and $z' \subseteq x'$ represents all z' vectors where the non-zero entries are a subset of the non-zero entries in x'. Under the three axioms, for a given simplified input mapping h_x, there is only one possible additive feature attribution method: **SHAP**. This result implies that methods not based on Shapley values violate local accuracy and/or consistency.

6.3.2 SHAP values

Before giving a formal definition of SHAP values, let us consider a more realistic example: let us imagine we are some data scientists and our goal is to build a model for a bank that, sucking in data from applicants, it is trained to predict if a customer could have any repayment problem. For a customer named John. that he sends in some information to this model, it returns a chance that he will have repayment problems equal to 22%: the model thinks that John is a too high risk customer and so declines his loan. As data scientists, the first question that should come to our mind when we are building kind of a high-impact model like this, is how do we debug this thing? How do we understand? How is it behaving and how do we trust it?

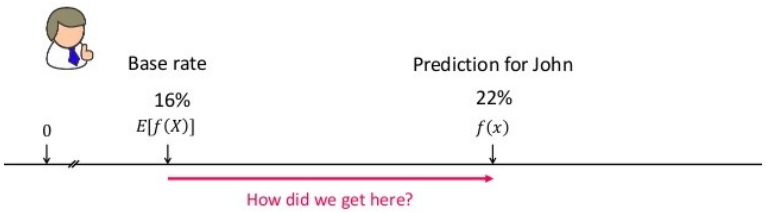


Figure 6.1: Output of the model

If we want to explain why the model took this decision for John, we need to explain what is so unique about him versus someone else. In this case the model has a base rate of 16%, so there are the 16% repayment problems in this dataset, but for John he had a prediction of 22%. So, we are trying to explain how it got from when we knew nothing about John to when we know everything about him. This is going to explain what is so special about John or Sue or whatever other customer arranged in explaining. One way to go about this, is to realize that the base rate of the model is really just the **expected value** of model output, then we can start from there and we fill out John's application, one form at a time. So what we will do is to compute the effects of certain interventions on the classifier prediction that alter the model in which we work: from the original one, that represents our dataset, to the intervened one, that contains the desired intervention. The first variable that comes to the model is whether John's income was verified or not. What we do is literally fill out that form for John, hence we set the application to say his income is not verified and then we observe the change that was made in the model: this intervention increases the risk from 16% up to 18.2%. Now, if we continue doing this and we introduce the debt-to-income ratio, we see it is equal to 30 that is rather high and this pushes the risk all the way up to 22%. Then, we see that he was delinquent on payment 10 months ago and this is not good news for John but, on the other hand, he has not applied for any recent account opening, so that enables drops this risk because apparently he is not applying through credit recently. However, when we consider that he has 46 years of credit history, that apparently is bad for him, the risk jumps up to the final outcome (Fig. 6.2).

We just showed a way to just literally fill out the form one at a time and measure the delta as we put each value into his application. However the model potentially could be very complex, and when it is non-linear or the input features are not independent, the order in which features are added to the expectation really matters. In fact we could test that by swapping the order in which we introduce these features, for example, we could introduce credit history first and then we could introduce his account openings, just filling out his application in a different order, we get totally different attributions given to these features:

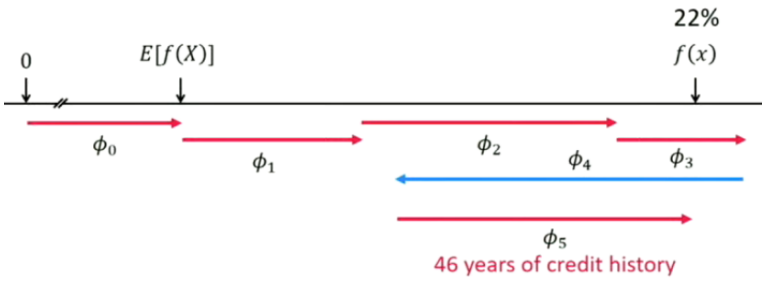


Figure 6.2: SHAP values explain how to get from the base value $\mathbb{E}[f(X)]$ that would be predicted if we did not know any features to the current output $f(x)$ (ϕ_0). This diagram shows a single ordering: ϕ_1 indicates the not verified income, ϕ_2 is the debt-to-income ratio, ϕ_3 considers that John was delinquent on payment 10 months ago, ϕ_4 considers no recent account openings and finally ϕ_5 represents the 46 years of credit history.

indeed 46 years of credit structure now helps them a little bit and then no recent count openings is just meaningless given the fact that we knew his long credit history (Fig. 6.3).

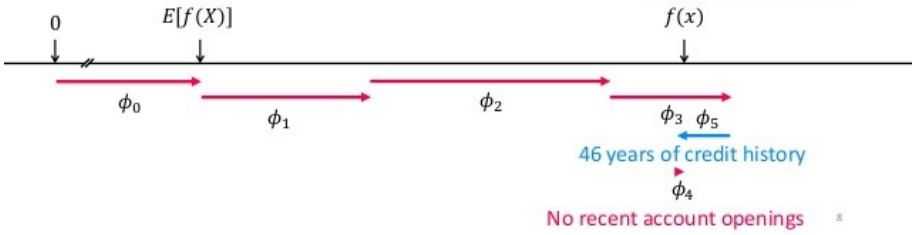


Figure 6.3: A different order in which the features are introduced.

SHAP values arise from **averaging ϕ_i over all N! possible orderings** (Eq. 6.13). Hence, SHAP values are the Shapley values of a conditional expectation function of the original model. They are the solution of Eq. 6.18, where $f_x(z') = f(h_x(z')) = \mathbb{E}[f(z)|z_S]$, and S is the set of non-zero indexes in z'. They use conditional expectations to define simplified inputs: implicit in this definition of SHAP values is a simplified input mapping, $h_x(z') = z_S$, where z_S has missing values for features not in the set S. Since most models cannot handle arbitrary patterns of missing input values, we approximate $f(z_S)$ with $\mathbb{E}[f(z)|z_S]$. SHAP values attribute to each feature the change in the expected model prediction when conditioning on that feature. They explain how to get from the base value $\mathbb{E}[f(z)]$ that would be predicted if we did not know any features to the current output $f(x)$. This definition of SHAP values is designed to closely align with the Shapley regression, Shapley sampling, and quantitative input influence feature attributions, while also allowing for connections with LIME, DeepLIFT, and LRP.

6.4 Approximation of SHAP values

As already mentioned, unfortunately, the exact evaluation of Shapley values is prohibitively expensive, NP-hard and only feasible for less than 20-25 players (i.e. input features for our case) because it is exponential in the number of input features. However, by combining insights from current additive feature attribution methods, we can approximate them. We

describe two approximation methods, one model-agnostic that is known as *Kernel SHAP* and another model-specific, named *Deep SHAP*. When using these methods, feature independence and model linearity are two assumptions simplifying the computation of the expected values:

$$f(h_x(z')) = \mathbb{E}[f(z)|z_S] = \mathbb{E}_{z_{\bar{S}}|z_S}[f(z)] \underbrace{\approx \mathbb{E}_{z_{\bar{S}}}[f(z)]}_{\text{feature independence}} \underbrace{\approx f([z_S, \mathbb{E}[z_{\bar{S}}])}_{\text{model linearity}} \tag{6.19}$$

6.4.1 KernelSHAP

KernelSHAP comes from the combination of *LIME* and *Shapley values*. At first glance, the regression formulation of LIME seems very different from the classical Shapley value formulation. However, since linear LIME is an additive feature attribution method, we know the Shapley values are the only possible solution that satisfies local accuracy, missingness and consistency. The solution of LIME equation

$$\xi = \underset{g \in G}{argmin} \ \mathcal{L}(f, g, \pi_{x'}) + \Omega(g) \tag{6.20}$$

recovers the Shapley values when:

- $$\Omega(g) = 0 \tag{6.21}$$
- $$\pi_{x'}(z') = \frac{M - 1}{(M \text{ choose } |z'|)|z'|(M - |z'|} \tag{6.22}$$
- $$\mathcal{L}(f, g, \pi_{x'}) = \sum_{z' \in Z} [f(h_x(z')) - g(z')]^2 \pi_{x'}(z') \tag{6.23}$$

Since $g(z')$ is assumed to follow a linear form, and $\mathcal{L}$ is a squared loss, the LIME equation can still be solved using linear regression. As a consequence, the Shapley values from game theory can be computed using *weighted linear regression*. Since LIME uses a simplified input mapping that is equivalent to the approximation of the SHAP mapping given in 6.19, this enables model-agnostic estimation of SHAP values. It has been show that estimating all SHAP values using regression provides better sample efficiency than the direct use of classical Shapley equations.

6.4.2 DeepSHAP

While KernelSHAP can be used on any model, including deep models, it is natural to ask whether there is a way to leverage extra knowledge about the compositional nature of deep networks to improve computational performance. The answer is provided by **DeepSHAP**, which comes from the connection between *DeepLIFT* and *Shapley values*. It is sufficient to interpret the reference value r in DeepLIFT equation

$$\sum_{i=1}^{n} C_{\Delta x_i \Delta o} = \Delta o \tag{6.24}$$

where $o = f(x)$ is the model output, $\Delta o = f(x) - f(r)$, $\Delta x_i = x_i - r_i$ as representing $\mathbb{E}[x]$, then DeepLIFT approximates SHAP values assuming that the input features are independent of one another and the deep model is linear. Since DeepLIFT is an additive feature attribution method that satisfies local accuracy and missingness, we know that Shapley values represent the only attribution values that satisfy consistency. This motivates our adapting DeepLIFT to become a compositional approximation of SHAP values, leading to Deep SHAP. It combines SHAP values computed for smaller components of the network into SHAP values for the whole network. It does so by recursively passing DeepLIFT's multipliers, now defined in terms of SHAP values, backwards through the network.

6.4.3 DASP

Both KernelSHAP and DeepSHAP are based on the strong assumptions of model linearity and feature independence and this is a big flaw. To overcome this limitation, [Ancona et al., 2019] proposed **DASP**, a perturbation-based method that can reliably approximate Shapley values in DNNs with a polynomial number of perturbation steps. This approximation is based on the following intuition. According to the definition of Shapley values, the Shapley value of an input feature is given by its marginal contribution to all possible 2^{N-1} coalitions that can be made out of the remaining features. Since we are interested in an average value, we can compute the expected contribution to a random coalition instead of enumerating each of them. In particular, we consider the distribution of coalitions of size k, for each $0 \leq k \leq N-1$, that do not include the feature x_i, and compute the expected contribution of that feature with respect to these distributions. The average of all these marginal contributions gives the feature's approximate Shapley value:

$$\mathbb{E}[\phi_i^c] = \frac{1}{N} \sum_{k=0}^{N-1} \mathbb{E}_k[\phi_{i,k}^c] \tag{6.25}$$

where the expectations $\mathbb{E}_k$ are over the distribution of size k, and $\phi_{i,k}$ denotes the contribution of feature x_i to any random coalition of size k. More explicitly, we can write the expected contribution of a feature x_i for a given coalition size as the expected target output difference with and without it, i.e.

$$\mathbb{E}_k[\phi_{i,k}] = \mathop{\mathbb{E}}_{S \subseteq P \setminus \{i\}} f(x_{S \cup \{i\}}) - \mathop{\mathbb{E}}_{S \subseteq P \setminus \{i\}} f(x_S) \qquad |S| = k \tag{6.26}$$

So the main problem is approximating these expected values for all coalitions of size $0 \geq k \geq N-1$. However, these expected values can be computed and propagated along with variances from layer to layer in a DNN. Such a propagation can be achieved by transforming the architecture of a given DNN to replace the point activations at all layers by probability distributions. Please refer to [Ancona et al., 2019] for a description of the architecture Lightweight Probabilistic Deep Networks (LPN) employed to propagate the expected values.

As we have extensively discussed in this chapter, Shapley values are a powerful solution for interpreting machine learning models predictions, because they have a solid theoretical background. However, at the moment, they do not fit so well with computer vision networks, which are very heavy since they include many parameters. For this reason, our experiments will focus on the gradient-based methods, described in Chapter 4 and on the methods that approximate Shapley values, like DeepSHAP.

Chapter 7

Experiments

In this chapter, we will repeat the methodology proposed by [Adebayo et al., 2018], described in Chapter 5, in order to experience model dependency of several explanation methods and validate their results.

For a clearer treatment, we divided such methods into 4 groups. The first group consists of *Saliency*, *Input* $\odot$ *Gradient*, *DeconvNet*, *Guided Backpropagation*, *GradCAM* and *Guided GradCAM* methods. The second of *DeepLift*, *DeepShap* and *DeepLift with SmoothGrad* methods. The third group includes *Integrated Gradients*, *Integrated Gradients with SmoothGrad* and *Expected Gradients with SHAP*, and finally, in the last group are present several variants of the *LRP-*$\alpha\beta$ rule and the recently proposed *RAP* method. To the best of our knowledge, we are the first to validate the heatmaps generated by the latter. All explanation methods are studied thanks to the implementation provided by the **Captum** library for model interpretability built on PyTorch, except for the methods belonging to the last group, for which we used the implementation introduced by [Nam et al., 2020], available in the GitHub repository, linked here.

For the purpose of this thesis, we have implemented and trained from scratch a variation of *VGG-16* convolutional neural network, whose details are provided below. The CNN is trained and evaluated on the *Tiny Imagenet* dataset, for which it reaches 52% top-1 accuracy, result that get close to state-of-the-art CNN performance from scratch on this dataset. The code concerning the network and his training is implemented in Python using the Torch library. All code is available in the GitHub repository, linked here.

7.1 Experimental setup

7.1.1 Dataset

The Tiny ImageNet dataset is a subset of the ImageNet dataset that is used as a benchmark for the annual image recognition competition ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). The dataset, prepared by Stanford University for their $CS231n$ course, contains 120.000 labeled images belonging to 200 object categories. Each of the 200 categories consists of 500 training images, 50 validation images and 50 test images, all down-sampled to a fixed resolution of $64x64$. Training and validation sets are provided with RGB images and class labels, while the test set is released without labels. For this reason, we decided to split the original training set in two subsets: 80% of the initial one constitutes the new training set and the remaining 20% forms the validation set. In this way, we used the original labeled validation set as a test set for the testing of our network.

The images were all pre-processed by subtracting the mean image, computed on the training set, from each image of the entire dataset and dividing by the standard deviation.

We have chosen this dataset because, despite the low resolution of the images makes it more challenging to extract information from there, his smaller size allowed us to see meaningful results after only a few hours of training.

7.1.2 Network Architecture

Motivated by the success of [Krizhevsky et al., 2012], [Szegedy et al., 2015] and several other groups in applying convolutional neural networks to image classification, and in particular, to the ILSVRC benchmark, we have decided to work with a deep convolutional neural network. The most famous model submitted to ILSVRC is the VGG-16 architecture introduced by Simonyan at al: consisting of 16 weight layers, it is designed to predict class labels for 224x224 RGB images from the ImageNet dataset (Fig 7.1).

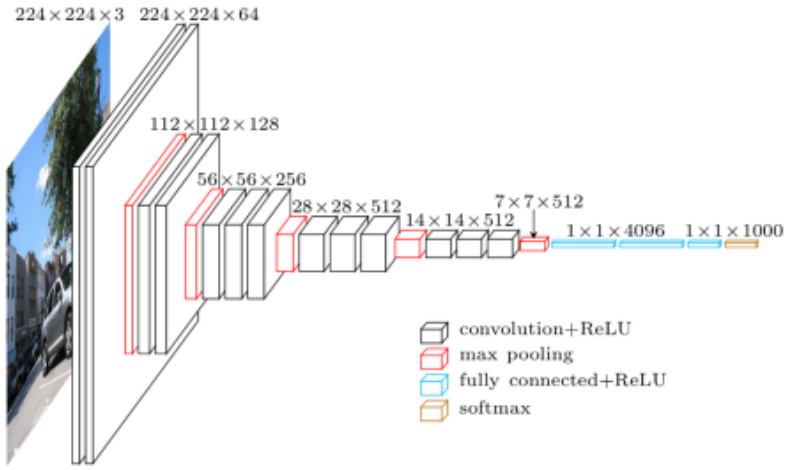


Figure 7.1: Architecture of VGG-16

However, since Tiny ImageNet has much lower resolution than the ImageNet data, some changes had to be made to the original configuration, as reported in the Table 7.1.

In our network, the 64x64 RGB image passes through a first stack of convolutional layers, where, similar to the original network, the 64 filters have a very small receptive field, 3x3, a convolution stride fixed to 1 pixel and the spatial padding such that the spatial resolution is preserved after convolution. To follow, a spatial pooling is carried out by a max-pooling layer, performed over a 2x2 pixel window with stride 2. The scheme (conv-conv-maxpool) is repeated a second time with 128 convolutional filters. The third and fourth stack consist of 3 convolutional layers with 256 and 512 filters, respectively, both followed by a max pooling layer. Unlike the original network that continues with another stack of 3 convolutional layers and another max pooling layer, our architecture goes on with 3 fully-connected layers: the first two have 2048 channels each, the third contains 200 channels as many as there are classes of Tiny ImageNet. The network ends without a softmax layer, differently from what proposed by Simonyan et al., for the reasons we will explain in the next lines.

All hidden layers are equipped with the activation function ReLU for both configurations, outlined in Table 7.1, one per column.

VGG-16 Architecture	
Original	Modified
input size 224x224	input size 64x64
conv3-64	conv3-64
conv3-64	conv3-64
maxpool	
conv3-128	conv3-128
conv3-128	conv3-128
maxpool	
conv3-256	conv3-256
conv3-256	conv3-256
conv3-256	conv3-256
maxpool	
conv3-512	conv3-512
conv3-512	conv3-512
conv3-512	conv3-512
maxpool	
conv3-512	
conv3-512	
conv3-512	
maxpool	
FC-4096	FC-2048
FC-4096	FC-2048
FC-1000	FC-200
softmax	

Table 7.1: VGG-16 configurations. The convolutional layer parameters are denoted as "conv(receptive field size)-(number of channels)". The ReLU activation layer is not shown for brevity.

Non-Linearity: *torch.nn.functional.relu* vs *torch.nn.ReLU*

In the first network we implemented for our experiments, we have defined the ReLU activation function with a functional approach. However, in Captum, some explanation methods which require placing hooks during back-propagation, including DeepLift, DeepLiftShap, Guided Backpropagation and Deconvolution, did not work appropriately with functional non-linearities. They must use the corresponding module activation which should be initialized in the module constructor.

In the following figures (Fig.7.2 and Fig.7.3) we can see how visualizations change after this correction. But not only, the use of the ReLU activation function with a functional approach led to sanity check results inconsistent with the literature and the experiments of [Adebayo et al., 2018] and [Sixt et al., 2019]. Further details and insights will be provided in the next sections.

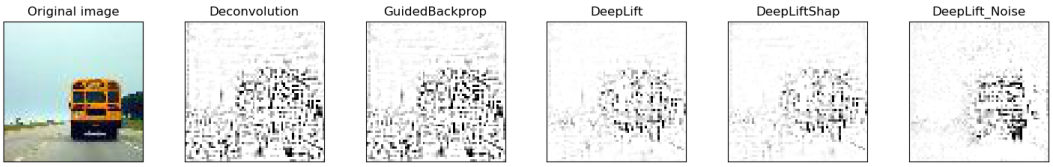


Figure 7.2: Network with torch.nn.functional.relu

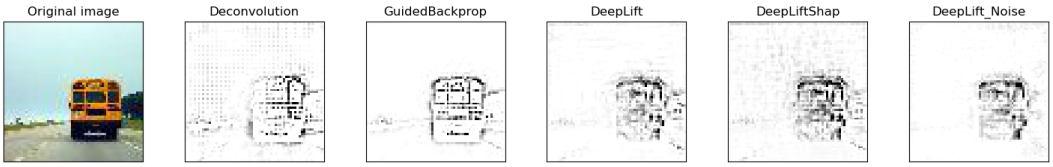


Figure 7.3: Network with torch.nn.ReLU

7.1.3 Training

In this section we report the choices made to train our network from scratch, which led to achieving 52% top-1 accuracy on the test set.

All convolutional and fully-connected layers were randomly initialized using a normal distribution for weights, known as He initialization, and setting all biases to 0. In addition to careful initialization choices, the related works suggest that carefully tuned first-order momentum methods are important to training deep networks. Consequently, we experimented with SGD with momentum to train the network, setting the parameter μ to 0.9 and choosing 64 as batch size. The learning rate was initialized to 0.01 and reduced by a factor of 0.2 every 2 epochs until convergence. Finally, for our model, we used accuracy as a metric and categorical cross-entropy as a loss function. The Fig. 7.4 illustrates the significant accuracy improvement after the first learning rate decay.

Most image classification datasets contain significantly more images per class than the Tiny Imagenet. For this reason, it is insufficient to train a deep neural network without considerable overfitting. We have considered the following strategies to overcome this obstacle.

L2-Regularization

The first main approach to overcome overfitting is the classical weight decay, which adds a term λ to the cost function to penalize the parameters in each dimension, preventing the network from exactly modeling the training data and therefore help generalize to new examples. Larger values of λ correspond to stronger regularization, which makes overfitting more difficult, but limits model capacity. We found the value of $\lambda = 0.005$ as optimal.

Batch Normalization

One of the most powerful ways to increase regularization, reduce overfitting, is Batch Normalization. It normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. We applied a layer of Batch Normalization over a 4D input after every convolutional layer and before non linearity. For fully-connected layers, we used a Batch Normalization over a 2D input.

Dropout

Another widely used technique increasing regularization, preventing overfitting is Dropout. Dropout sets the output of each neuron to zero with a certain probability p. The majority of the parameters in the convolutional network are in the fully-connected layers, so dropout is applied to the fully-connected layers with default value of $p = 0.5$. However, by adding a dropout with a lower p, equal to 0.25, also to the convolutional layers, we have achieved an improvement in the performance of our network.

Early Stopping

Early stopping is a form of regularization used to avoid overfitting on the training dataset. It keeps track of the validation loss: if the loss stops decreasing for several epochs in a row, the training stops. The corresponding weights are saved into a file named *bestmodel.pt* and will be use in the experiments that will be described in the following section. The number of epochs to wait before early stop is identified by the parameter *patience* that we setted to 10 The Fig. 7.5 shows that from the 25th epoch, the validation loss has no longer decreased. The early stopping finished the training after exactly 10 epochs, at the 35th.

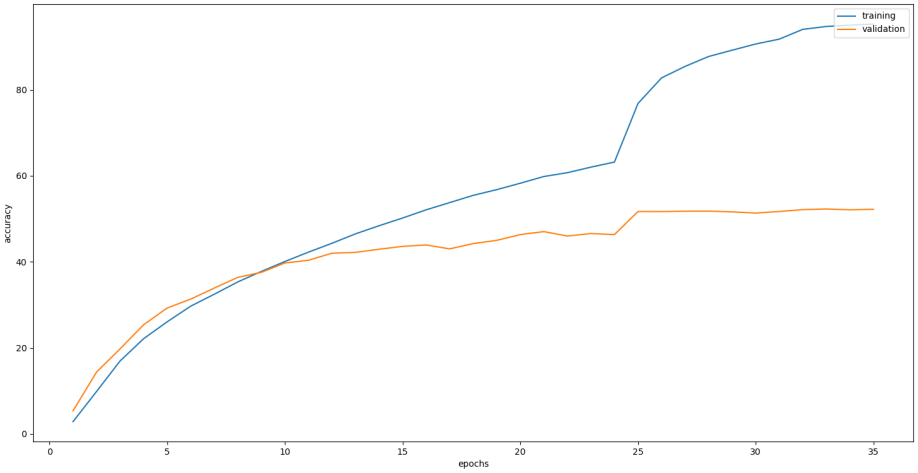


Figure 7.4: Accuracy plot

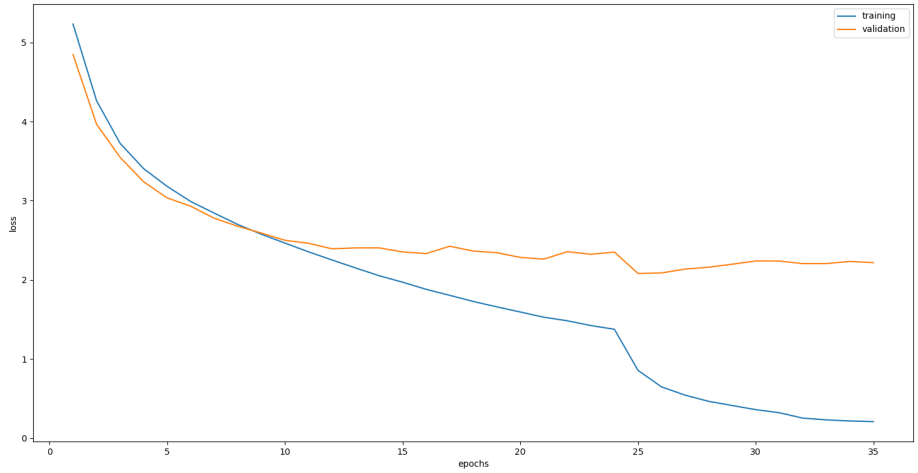


Figure 7.5: Loss plot

7.2 Sanity check

As anticipated in the introductory chapter, the goal of the thesis is to assess the scope and quality of explanation methods. Following the works of [Adebayo et al., 2018] and [Sixt et al., 2019] described in Chapter 5, we sanity check of several explanation methods, some of them tested for the first time, in order to answer to the following question: if the parameters of the model are randomized and therefore the network output changes, do the saliency maps change too?

We report results of some model parameter randomization tests on a deep convolutional neural network, inspired by the VGG-16 architecture, and trained on Tiny Imagenet dataset, as described in the previous section. All results were computed on 5 images from our test set, one of which containing two classes, in order to evaluate the class sensitivity of methods considered. We setted 5 different seeds in order to conduct an analysis more robust and then we took the mean. The experiments were run on a single machine with an Nvidia GPU and took about a day to complete.

7.2.1 Tools

Given the attention deep learning applications are currently receiving, several software tools have been developed to facilitate model interpretation. [Alber et al., 2019] presented the *INNvestigate* library, an interpretability tool based on Keras which provides a common interface and out-of-the-box implementation for many analysis methods. All the works published so far concerning the analysis of image classifications use the interface of iNNvestigate. However, a prominent very recent library is **Captum**, an extension of the PyTorch deep learning package, announced at PyTorch Developer Conference in 2019 by Facebook and released this year (2020). Captum provides support for most of the feature attribution techniques described in this work and, to date, it is the only library that contains implementations of methods approximating the Shapley values, like DeepLiftShap and GradientShap. For this reason, in our experiments, we chosen to use the interface provided by Captum library for all explanation methods, except for all variants of $LRP\text{-}\alpha\beta$ and RAP methods for which we used the code released by [Nam et al., 2020], linked at the beginning of this chapter.

In order to confirm our results, we tested also the algorithms implemented by another PyTorch interpretability library, **TorchRay**, released recently by [Fong et al., 2019], to provide researchers and developers with an easy way to understand which features are contributing to a model's output.

As we will see in the next chapters, the results obtained with both libraries are consistent with the literature.

7.2.2 Experimental results

As anticipated in the introduction of this chapter, we will study the quality and faithfully of attributions resulting from several explanation methods, which we divided into 4 different groups. By way of example, we visualize attribution maps for a German Shepherd image from Tiny Imagenet dataset on our VGG16-like network. In section Additional Figures, we will provide explanations for other images, in order to prove the robustness of our analysis. For visualization, we normalized the heatmaps to be in $[0, 1]$ if the method produces only positive relevance and in $[-1, 1]$ if it estimates also negative ones. Both values are scaled equally by the absolute maximum.

This section is organized as follows. For each group:

- We visualize the original heatmaps in order to understand which of the features, according to the method, were most important for the prediction reached by the net.
- We provide the heatmaps generated after a model parameter cascading randomization and the corresponding mean SSIM plot.
- We provide the heatmaps generated after a model parameter independent randomization and the corresponding mean SSIM plot

For all methods and tests, we provide explanations in two ways: first we display all relevance values, both positive and negative, and then we highlight only positive attributions.

Group 1

The following plot shows the explanations generated by methods belong to the first group: **Saliency**, **Input** $\odot$ **Gradients**, **DeconvNet**, **Guided Backpropagation**, **GradCAM** and **Guided GradCAM**.

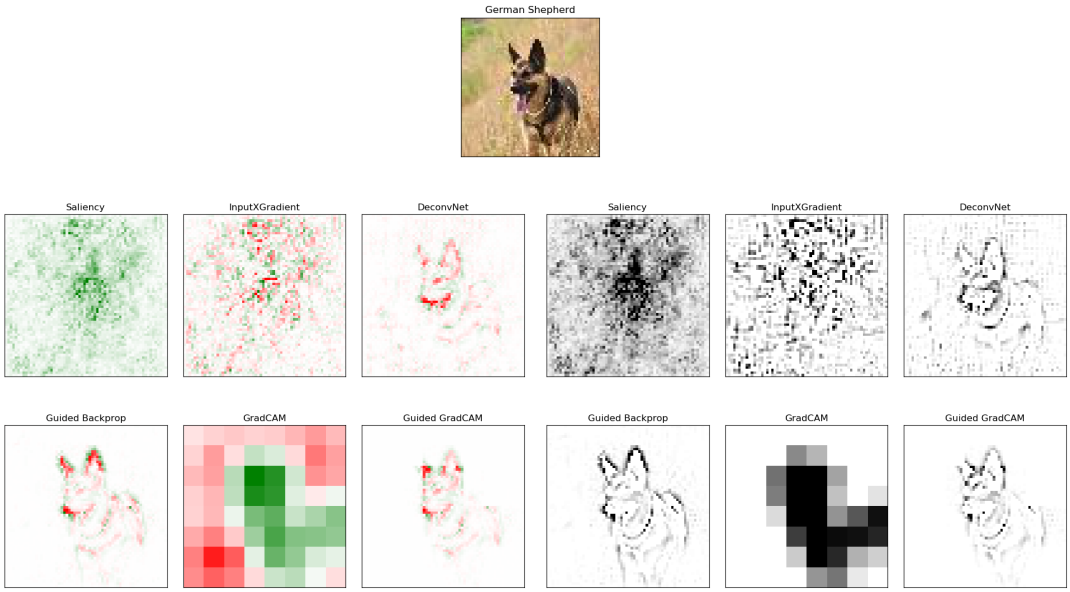


Figure 7.6: All relevances

Figure 7.7: Positive relevances

From Fig. 7.6 and 7.7, we can make a first qualitative comparison between explanation methods belonging to the fist group. Consistently with what is discussed in Chapter 4, the heatmap provided by Saliency is very noisy. In Input$\odot$Gradient attribution, we see a reduction of visual diffusion, but it is still hardly interpretable. Better visualizations are obtained with the DeconvNet and Guided Backpropagation methods, in which are recognizable the contours of the object. Regarding the DeconvNet method, Captum library implements the hybrid model of [Mahendran and Vedaldi, 2016] that produces convincingly sharper images that network saliency while being much more selective to foreground objects than DeconvNet of [Zeiler and Fergus, 2014]. GradCam is applied to the last convolutional layer, similarly to Guided GradCam that computes element-wise product of guided backpropagation attributions with upsampled non-negative GradCam attributions.

Now we report the sanity check of these explanation methods carried on our network for the German Shepherd input image. Randomizing the model parameters, first in a cascading fashion, then in independent manner, as we described in Chapter 5, we are going to compare the output of an explanation method on a trained network, with the output of the same method on a randomly initialized untrained network of the same architecture. If the explanation method depends on the learned parameters of the model, we should expect its output to differ substantially between the two cases.

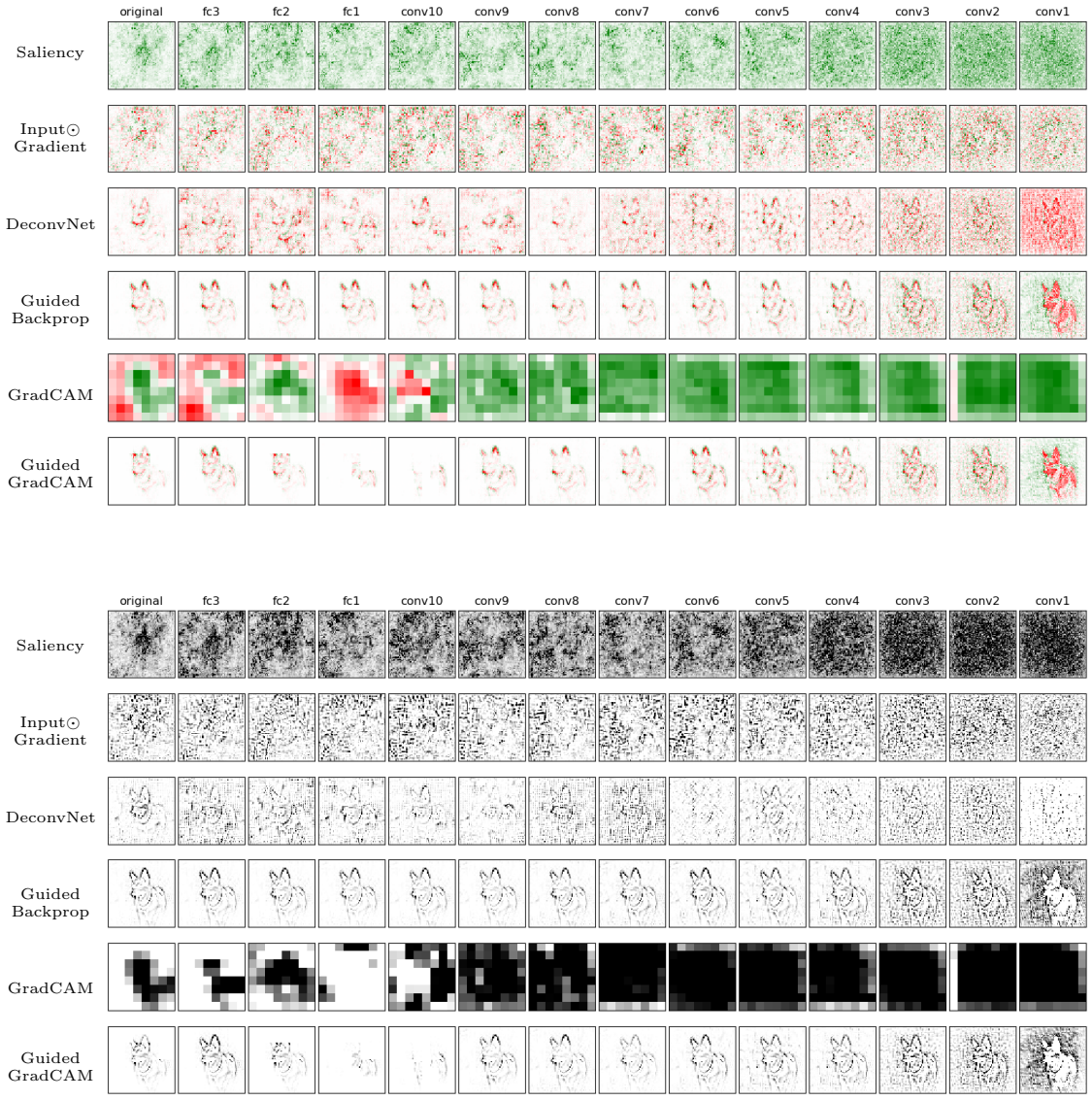


Figure 7.8: Cascading randomization for German Shepherd class. Figure shows the original explanations (first column). Progression from left to right indicates complete randomization of network weights. The last column corresponds to a network with completely reinitialized weights. (Top) Green denotes positive and red negative relevance. (Bottom) Only positive relevance are displayed.

On visual inspection of Fig 7.8, we find that Guided Backpropagation is invariant to the several layers reparameterization, at least until the fourth convolutional layer. Let us try to read the graph in reverse, from right to left: we load the trained weights up to a given layer and leave later layers randomly initialized. Matching the results of [Nie et al., 2018], from *conv1* to *conv5* Guided Backpropagation keeps filtering out more image patches as the number of trained convolutional layers increases. However, from *conv6* to *fc3* (i.e. the almost fully trained network), Guided Backpropagation behaves almost the same, no matter weights in the higher convolutional layers or dense layers are random or trained. From this we can deduce two results: (1) it is the trained weights in the lower convolutional layers rather than those in the dense layers that account for filtering out image patches, (2) the earlier convolutional layers has more important impact in the Guided Backpropagation visualization than the later convolutional layers.

Differently, Guided GradCAM, that comes from the combination of Guided Backpropagation and GradCAM, shows a slight sensitivity to parameter randomization of the last fully-connected layer, probably due to the influence of the latter. GradCAM, in fact, together with remaining methods, show a rather evident model dependency. In this regard, we confirm the bug founded by [Sixt et al., 2019] in the implementation proposed by [Adebayo et al., 2018], resulting in saliency maps of Guided Backpropagation and Guided GradCAM to remain identical until the input layer.

However, in order to have a more concrete numerical result than a visual perception, we computed the structural similarity index (SSIM). This metric allowed us to assess similarity between two explanations. In particular, we scaled all saliency maps to be in $[0, 1]$ and we compared the explanation obtained with the trained network to the one obtained from a network with random weights, one layer at a time, for all layers. Being a randomization of the parameters, we set a seed to allow replicable analysis. However, since some methods are sensible to randomized weights, we decided to consider 5 different seeds and then take the mean. In the following plot, the line represents the mean SSIM deriving from cascading randomization for the German Shepherd image on our network, while the shaded area shows the interval of standard deviation. The method fails if already with the last randomized layer $fc3$, the SSIM remains equal to 1.

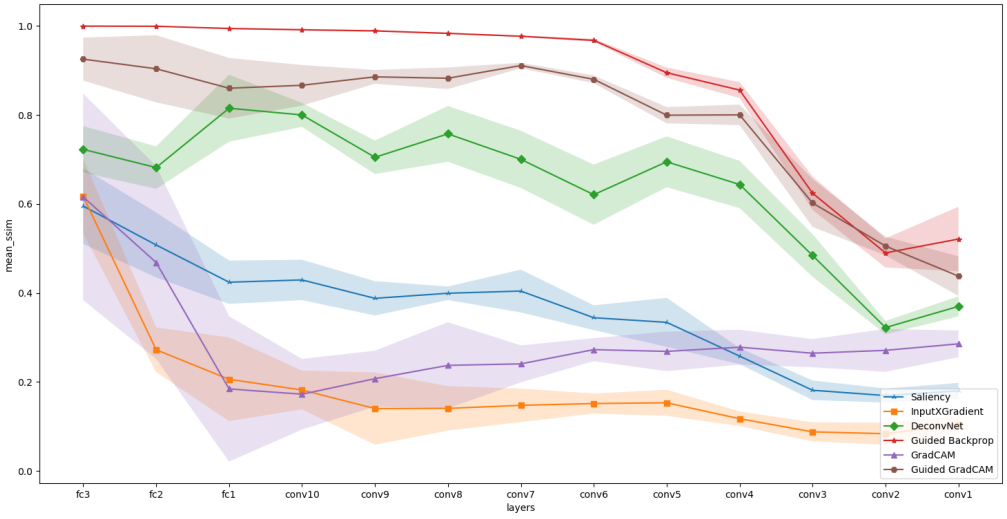


Figure 7.9: Mean SSIM plot for cascading randomization test

The mean SSIM plot in Fig. 7.9 confirms what we previously noted: Guided Backpropagation, produces identical saliency maps to the original one, after the parameter randomization of several layers (SSIM equal to 1), regardless of the seed considered (standard deviation equal to 0). Guided GradCAM, in contrast to the results obtained by Adebayo et al., shows a SSIM index close to 0.9, already after the randomization of the last fully-connected layer. For Saliency, Input $\odot$ Gradient and GradCAM methods, the values of SSIM are consistent with the works cited above: all three methods abundantly pass the test. As we can see, the formulation of DeconvNet proposed by [Mahendran and Vedaldi, 2016] is sensitive to parameter randomization, differently from DeconvNet of [Zeiler and Fergus, 2014] studied in the work of [Sixt et al., 2019] that, instead, shows a SSIM equal to 1 for several layers.

In order to find a match with the results so far obtained with the Captum library, let us see now what happens with the implementation of Saliency, DeconvNet and Guided Backpropagation methods provided by the **TorchRay** library. We show here the resulting heatmaps from a cascading randomization starting from the last fully connected layer until the first convolutional layer, as described before. For a sensible comparison, we experimented such methods on the same input image.

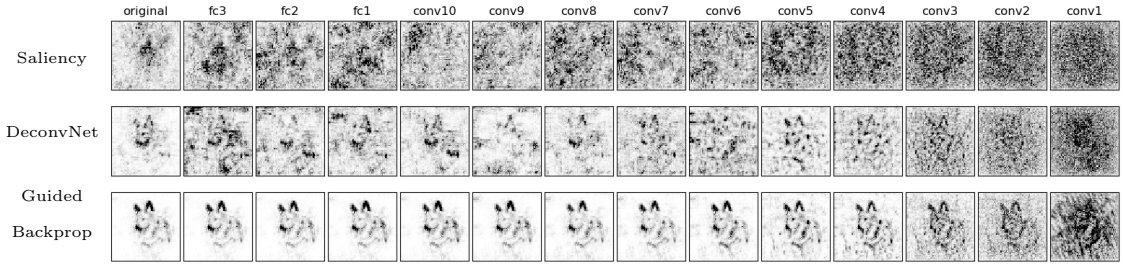


Figure 7.10: Cascading randomization with TorchRay.

The results so far obtained with TorchRay confirm the ones obtained with Captum: Guided Backprop shows no change regardless of model randomization for multiple layers, Saliency and DeconvNet pass the test. Further confirmation is given by the next SSIM plot:

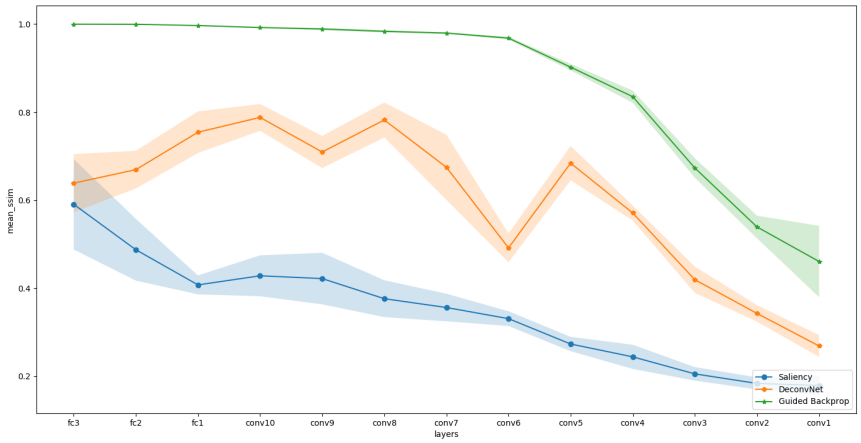


Figure 7.11: Mean SSIM plot for cascading randomization test with TorchRay

Now we go back to Captum and we show the results of a different form of the model

parameter randomization test. In addition to a cascading randomization, we conducted an independent layer-by-layer randomization with the goal of isolating the dependence of the explanations by layer. Consequently, we can assess the dependence of heatmap on lower versus higher layer weights.

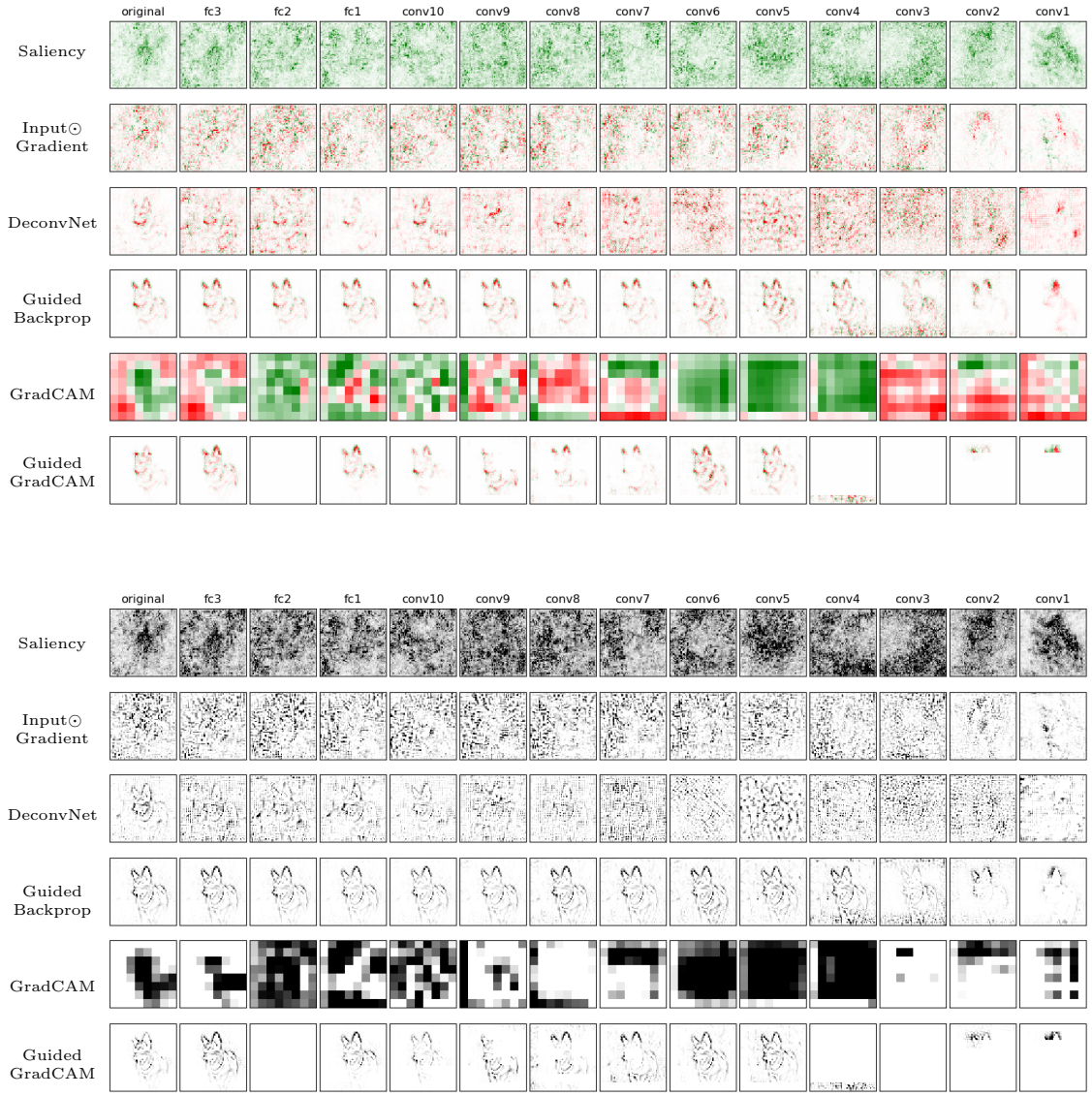


Figure 7.12: Independent randomization for German Shepherd class. Figure shows the original explanations (first column). Progression from left to right indicates a layer randomization of network weights at the time. (Top) Green denotes positive and red negative relevance. (Bottom) Only positive attribute values are displayed.

As previous analysis, we computed the mean SSIM index over 5 different seeds.

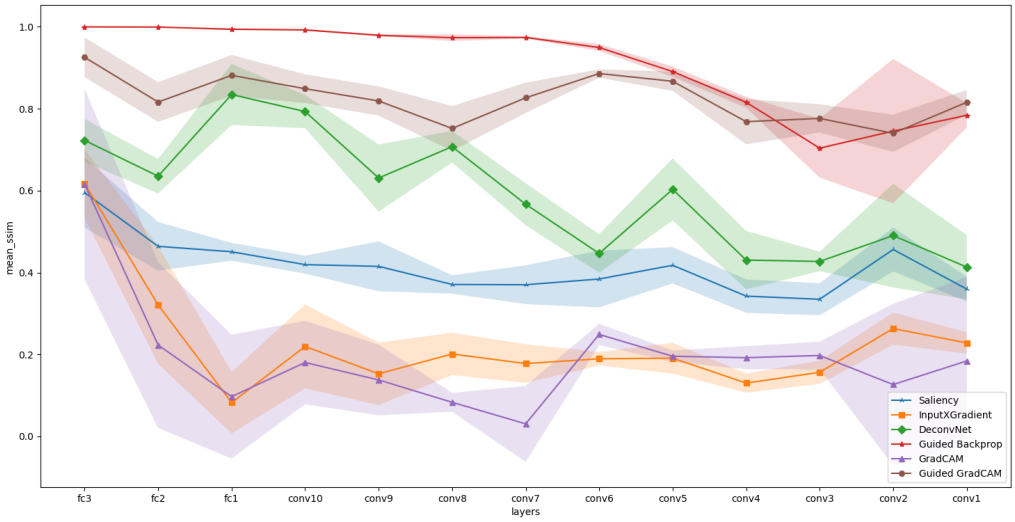


Figure 7.13: Mean SSIM plot for independent randomization.

As we can see, none of the tested methods maintain a similarity of 1 when the first convolutional layer of the network *conv1* is destroyed, keeping the others unchanged (trained). We note that the Guided Backpropagation visualization is difficult to interpret for randomly re-initialized *conv1* and *conv2*, it is more clean but with much background information for randomized *conv3* and *conv4* and it is clean without background information for *conv6* destroyed. It further confirms that the earlier convolutional layers have a greater impact in the Guided Backpropagation visualization than the later convolutional layers.

However, we observe a correspondence between the results from the cascading and independent layer randomization experiment.

Group 2

The following plot shows the explanations generated by methods belong to the second group: **DeepLIFT**, **DeepSHAP** and **DeepLIFT with SmoothGrad**.

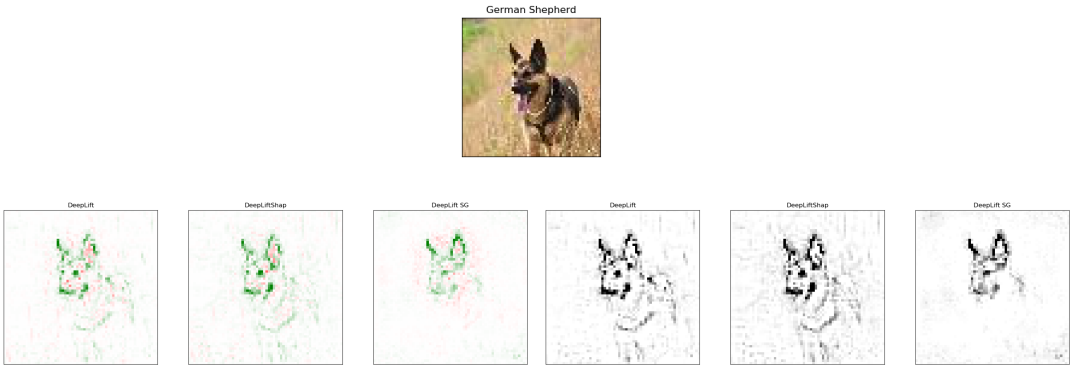


Figure 7.14: All relevances

Figure 7.15: Positive relevances

About DeepLift, the Captum library supports only the Rescale rule. In order to implement this method, we chose as baseline a blurred version of the original image with a gaussian filter of size 15×15, because with this, the method provides a more satisfactory explanation than the black baseline, as we can see in the following figure:

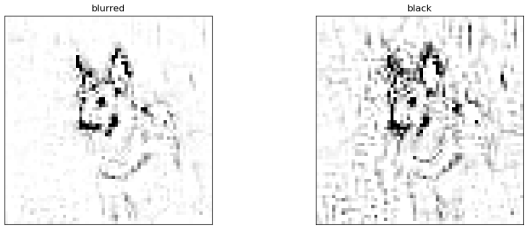


Figure 7.16: (Left) DeepLift explanation with a blurred baseline. (Right) DeepLift explanation with a black baseline.

As mentioned earlier, Captum is the only library that supports the implementation of methods which approximate the SHAP values. An example is DeepLift Shap, which in the original paper the authors call *DeepSHAP*. It takes a distribution of baselines that we got from an entire test set's batch of 64 images, computes the DeepLift attribution for each input-baseline pair and averages the resulting attributions per input example. A prominent method existing between implementations of Captum is Noise Tunnel, that can be used on top of any of the attribution methods. It computes attribution multiple times, adding Gaussian noise to the input each time, and combines the calculated attributions based on the chosen type. The supported types for Noise Tunnel are SmoothGrad, SmoothGrad Squared and VarGrad.

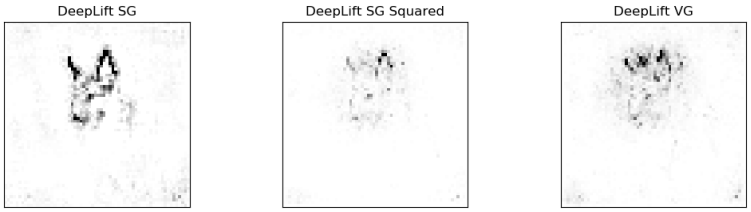


Figure 7.17: From left: DeepLift with SmoothGrad, DeepLift with SmoothGrad Squared, DeepLift with VarGrad.

Intrigued by this, we decided to apply Noise Tunnel to DeepLift, choosing to add Gaussian noise with SmoothGrad and setting for this input image the standard deviation to 0.7. As we can see, the resulting visualization is very satisfactory.

Now we report the sanity check of these explanation methods carried on our network for the German Shepherd input image. First, we show heatmaps from cascading randomization:

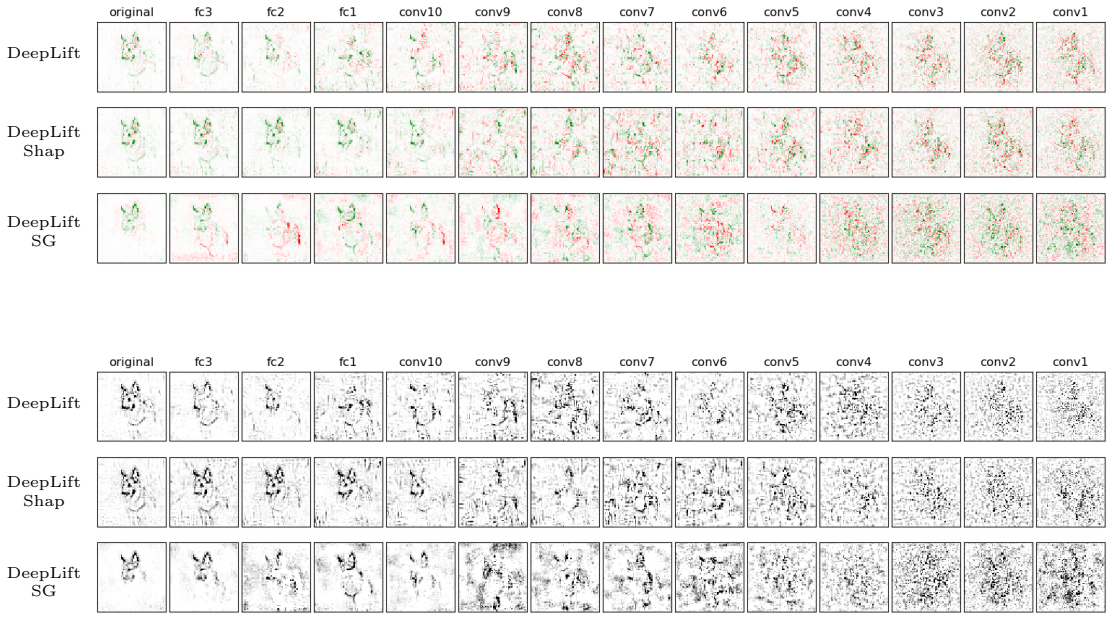


Figure 7.18: Cascading randomization for German Shepherd class. Figure shows the original explanations (first column). Progression from left to right indicates complete randomization of network weights. The last column corresponds to a network with completely reinitialized weights. (Top) Green denotes positive and red negative relevance. (Bottom) Only positive attribute values are displayed.

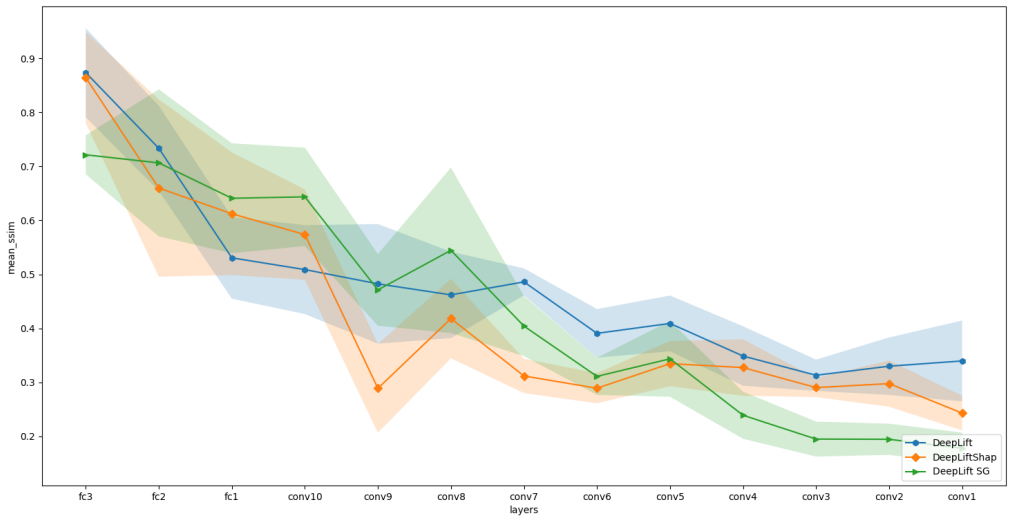


Figure 7.19: Mean SSIM plot for cascading randomization test

To follow, the results of independent randomization test:

Figure 7.20: Independent randomization for German Shepherd class. Figure shows the original explanations (first column). Progression from left to right indicates a layer randomization of network weights at the time. (Top) Green denotes positive and red negative relevance. (Bottom) Only positive attribute values are displayed.

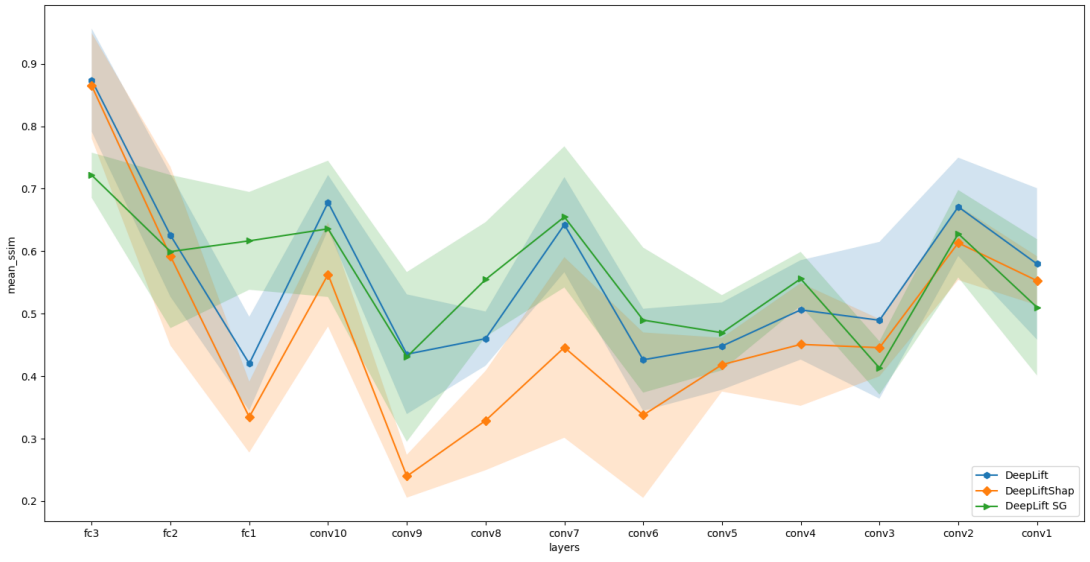


Figure 7.21: Mean SSIM plot for independent randomization.

In Chapter 4, we saw that the choice of reference image is not determined a priori by the method, but it is a hyperparameter of the attribution task. As such, it plays a fundamental role in generating the explanation and can be decisive for the properties of the method.

In this regard, we wanted to validate the explanations generated by DeepLift, using two different baselines: a black image and a blurred version of the input image. We carried the cascading randomization with both baselines, separately, keeping all other parameters fixed, and in the following figures, we show the results.

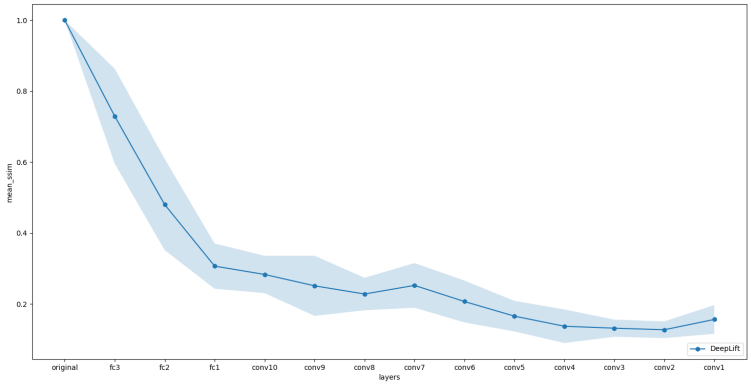


Figure 7.22: Mean SSIM plot for DeepLift method with a black baseline

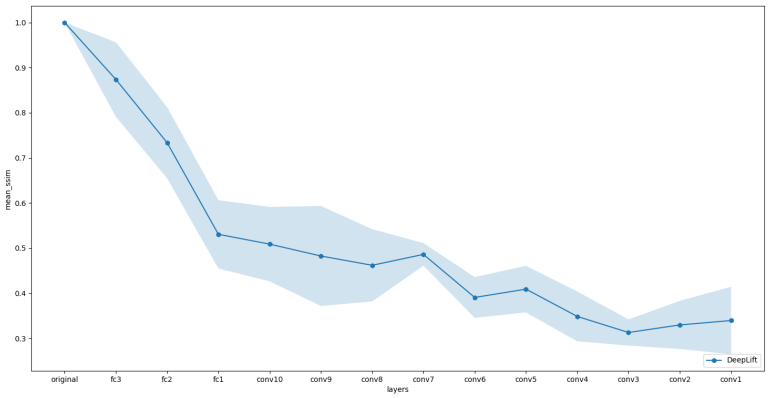


Figure 7.23: Mean SSIM plot for DeepLift method with a blurred baseline

As we can see, in both cases, DeepLift passes the test. The small difference in the range of SSIM values in the two cases is insignificant for testing purposes: a method that shows a mean SSIM value lower than 1 produces reliable explanations. To conclude, we found that satisfying model parameters invariance does not depend upon the choice of reference.

Group 3

The following plots (Fig. 7.24 and Fig. 7.25) shows the explanations generated by methods belonging to the second group: **Integrated Gradients**, **Integrated Gradients with SmoothGrad** and **GradientShap**.

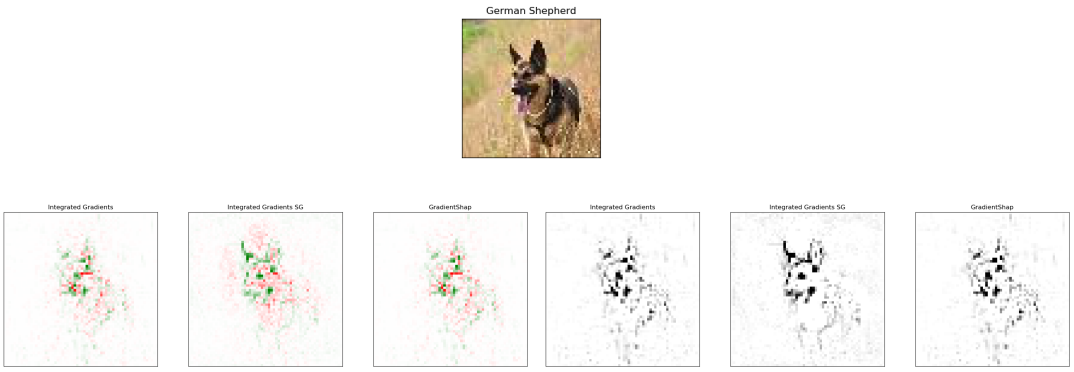


Figure 7.24: All relevances

Figure 7.25: Positive relevances

Similar to DeepLift's heatmap is that generated by Integrated Gradients. Also for this method, we used a blurred version of the input image as baseline, for the reasons which are clear in the next figure. For its implementation, we chose to approximate the integral using a Gauss Legendre quadrature rule, as default, because it is fastest and we set the number of approximation steps to 200.

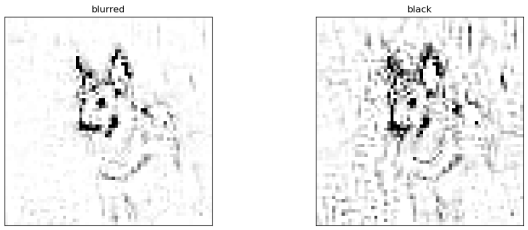


Figure 7.26: (Left) Integrated Gradients explanation with a blurred baseline. (Right) Integrated Gradients explanation with a black baseline.

In addition to DeepLift, we applied Noise Tunnel with SmoothGrad also to Integrated Gradients methods. For completeness, we show here the attributions obtained with the other types provided by Noise Tunnel:

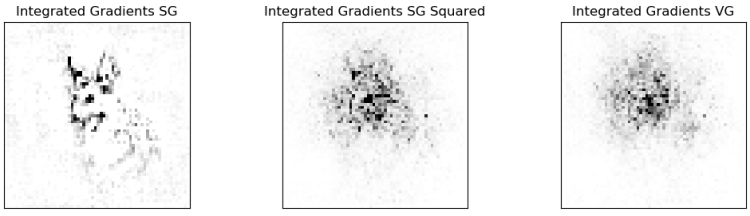


Figure 7.27: From left: Integrated Gradients with SmoothGrad, Integrated Gradients with SmoothGrad Squared, Integrated Gradients with VarGrad.

For Integrated Gradients with SmoothGrad, which operates similarly to DeepLift with SmoothGrad, we used the same value of standard deviation (0.7). As in the previous case, also this time, Noise Tunnel combined with the original method provides a better visualization. Another method present only in Captum that approximates the SHAP values is GradientShap. It computes the expectation of gradients for an input which was chosen randomly between the input and a baseline, extracted from a distribution of reference samples. For the latter, we used a baseline distribution formed by the blurred and the original images.

We begin to see the results of the sanity check for the groups belonging to this third group. Here, we show the heatmaps obtained after a cascading randomization of the network, followed by the mean SSIM plot.

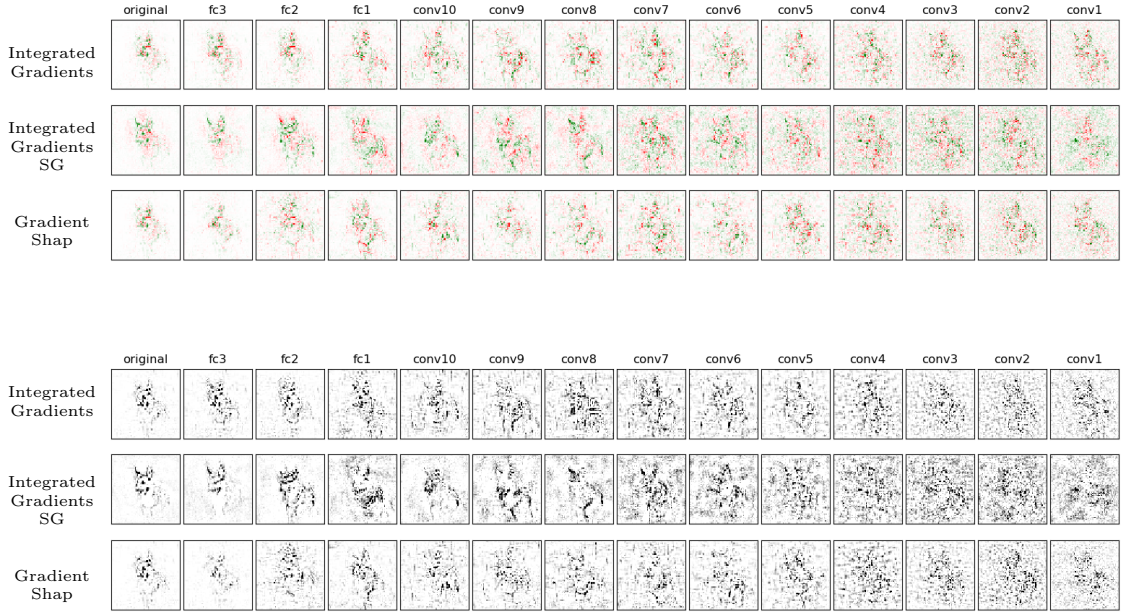


Figure 7.28: Cascading randomization for German Shepherd class. Figure shows the original explanations (first column). Progression from left to right indicates complete randomization of network weights. The last column corresponds to a network with completely reinitialized weights. (Top) Green denotes positive and red negative relevance. (Bottom) Only positive attribute values are displayed.

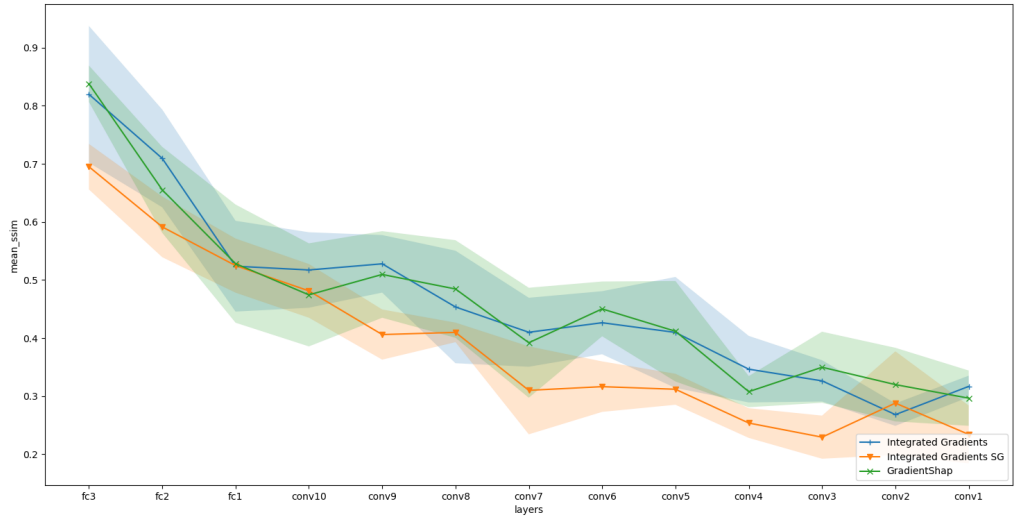


Figure 7.29: Mean SSIM plot for cascading randomization test

We continue with the results of the independent randomization test:

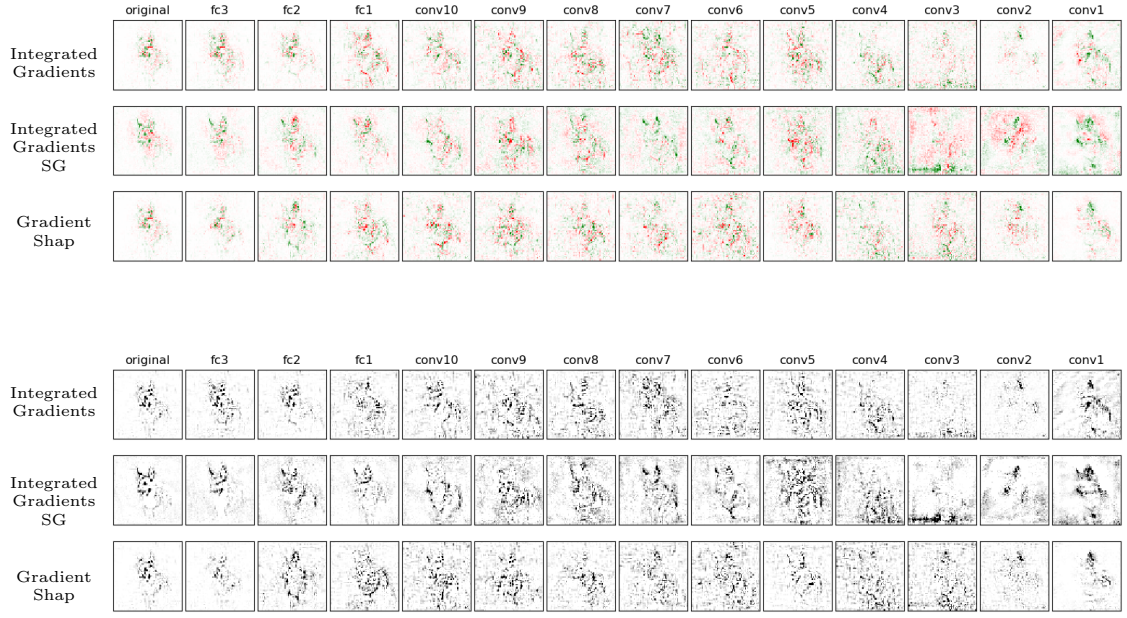


Figure 7.30: Independent randomization for German Shepherd class. Figure shows the original explanations (first column). Progression from left to right indicates a layer randomization of network weights at the time. (Top) Green denotes positive and red negative relevance. (Bottom) Only positive attribute values are displayed.

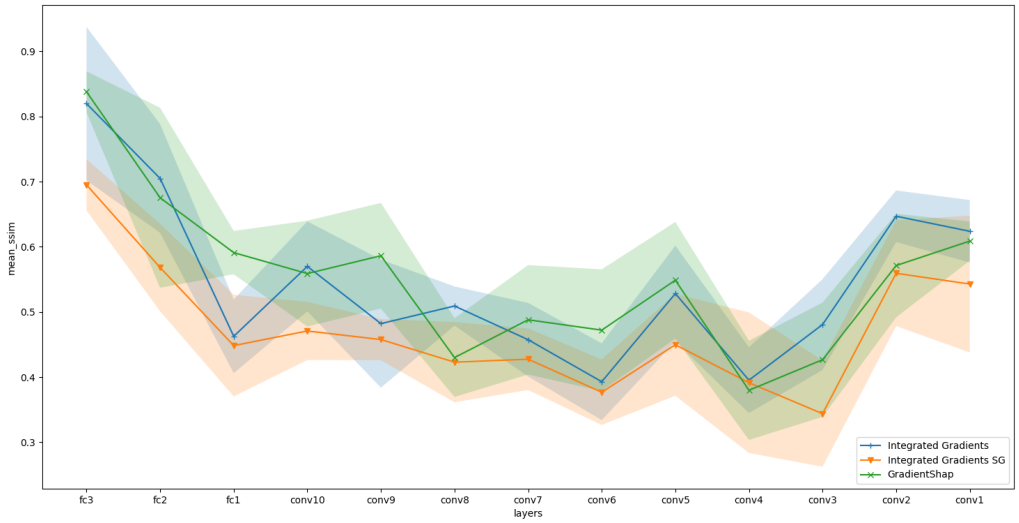


Figure 7.31: Mean SSIM plot for independent randomization test.

We conclude the analysis of this group of methods, showing how also in the case of Integrated Gradients, the choice of the baseline influences the plot of the SSIM. Similarly to what was observed for DeepLift, also in this case, a black baseline makes it more difficult to identify the most important features, if the last layer of the network is destroyed. If with the blurred baseline the SSIM index between the original attribution and that obtained with the last fully-connected layer randomized was equal to about 0.9, now we see that it has dropped to 0.7.

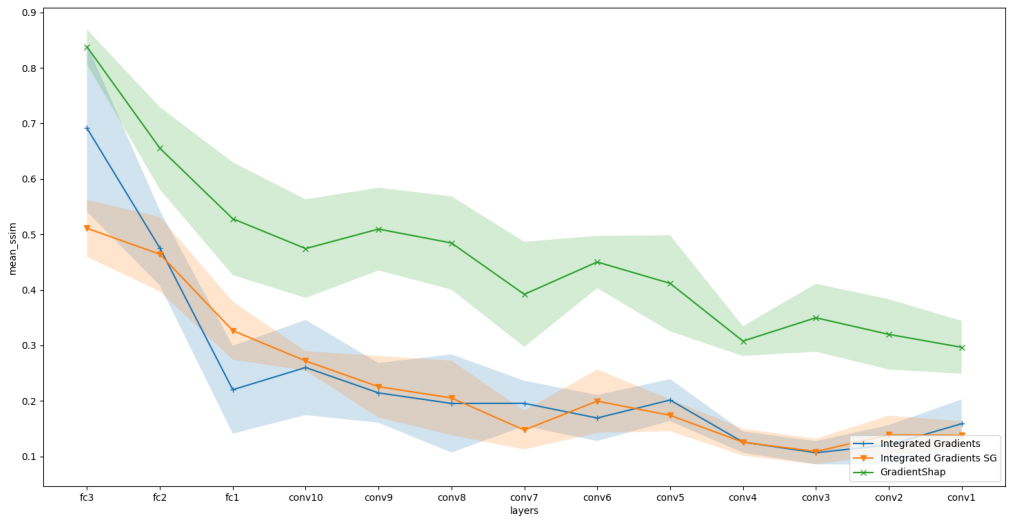


Figure 7.32: Mean SSIM plot for Integrated Gradients with a black baseline.

However, in both cases, this trio of methods pass the test, consistent with the results of [Adebayo et al., 2018] and [Sixt et al., 2019]. As explained by the latter, Integrated Gradients and its variants rely on the gradient directly. For this reason, they do not suffer from the convergence problem.

Group 4

The last group is formed by seven variants of $\alpha\beta$-**LRP** method and **RAP**.

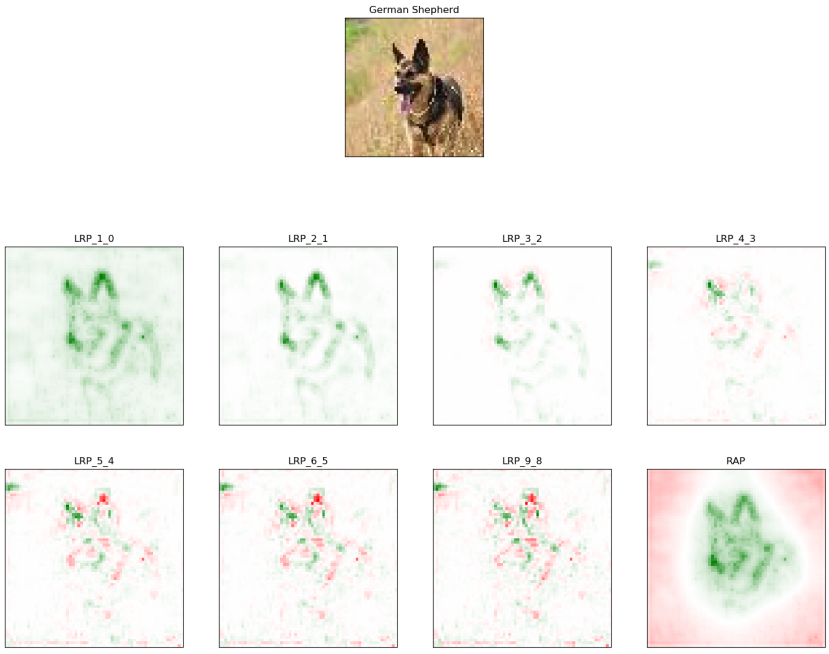


Figure 7.33: All relevances

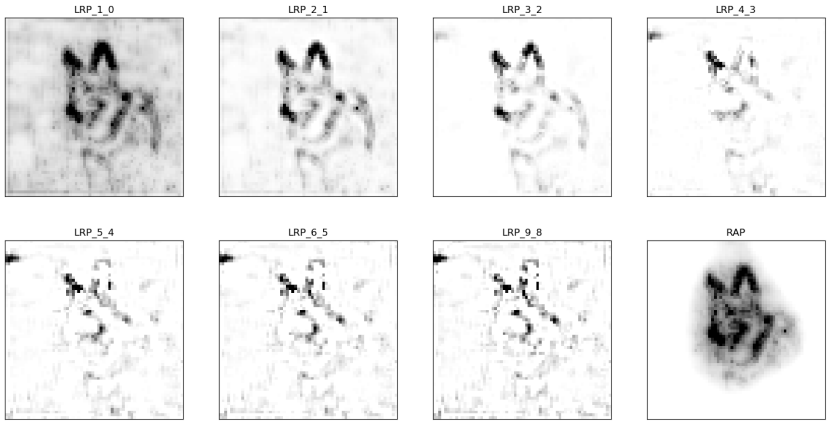


Figure 7.34: Positive relevances

The fourth group of methods are not present in the Captum interface. In order to study them, we used the implementation provided by [Nam et al., 2020], authors of the RAP method. For LRP method, we tested some of its versions with different values of α and β, in order to show the influence of the negative contributions.

As previously done, we analyze the explanations given after a cascading randomization:

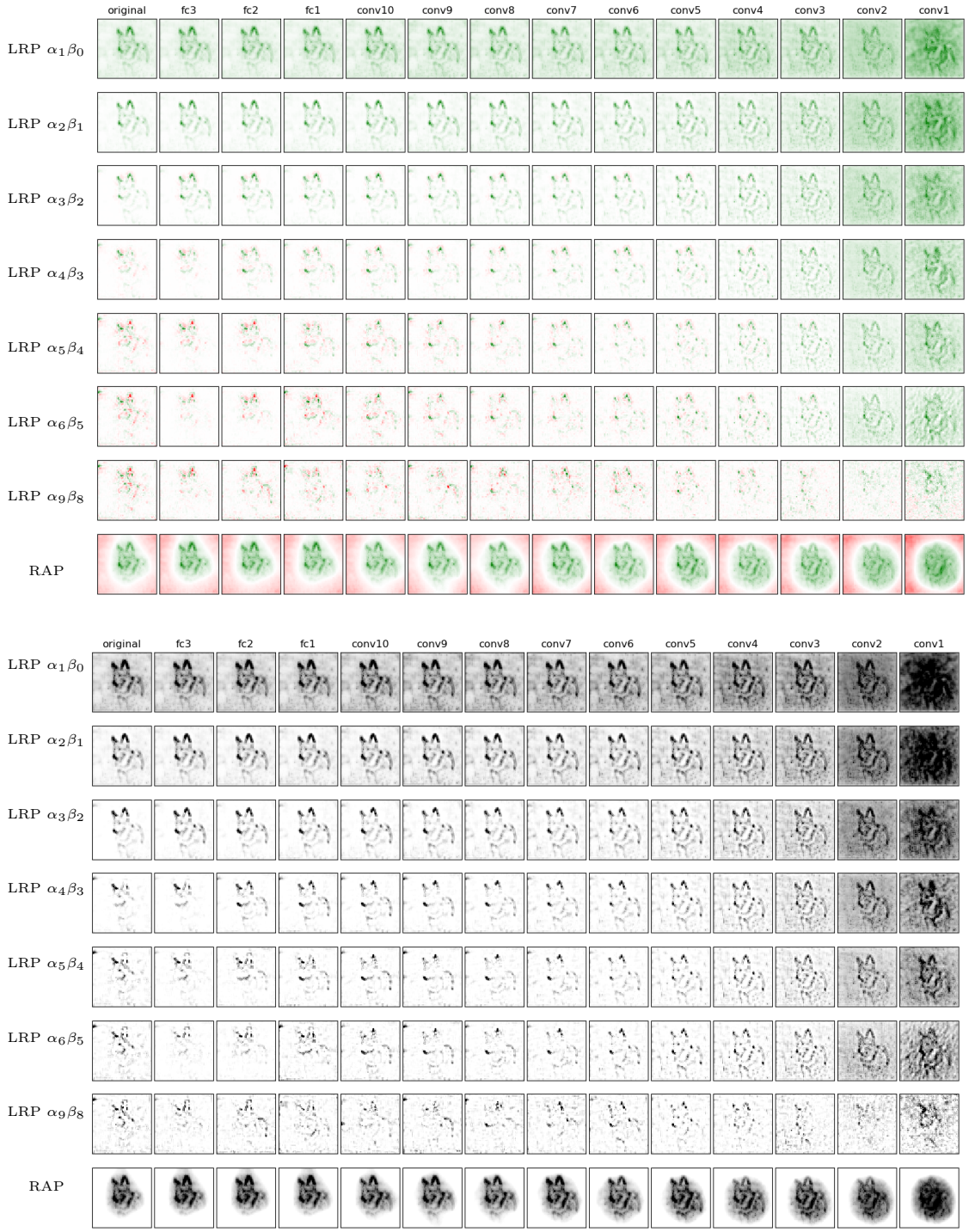


Figure 7.35: Cascading randomization for German Shepherd class. Figure shows the original explanations (first column). Progression from left to right indicates complete randomization of network weights. The last column corresponds to a network with completely reinitialized weights. (Top) Green denotes positive and red negative relevance. (Bottom) Only positive attribute values are displayed.

Here, the heatmaps after an independent randomization are displayed:

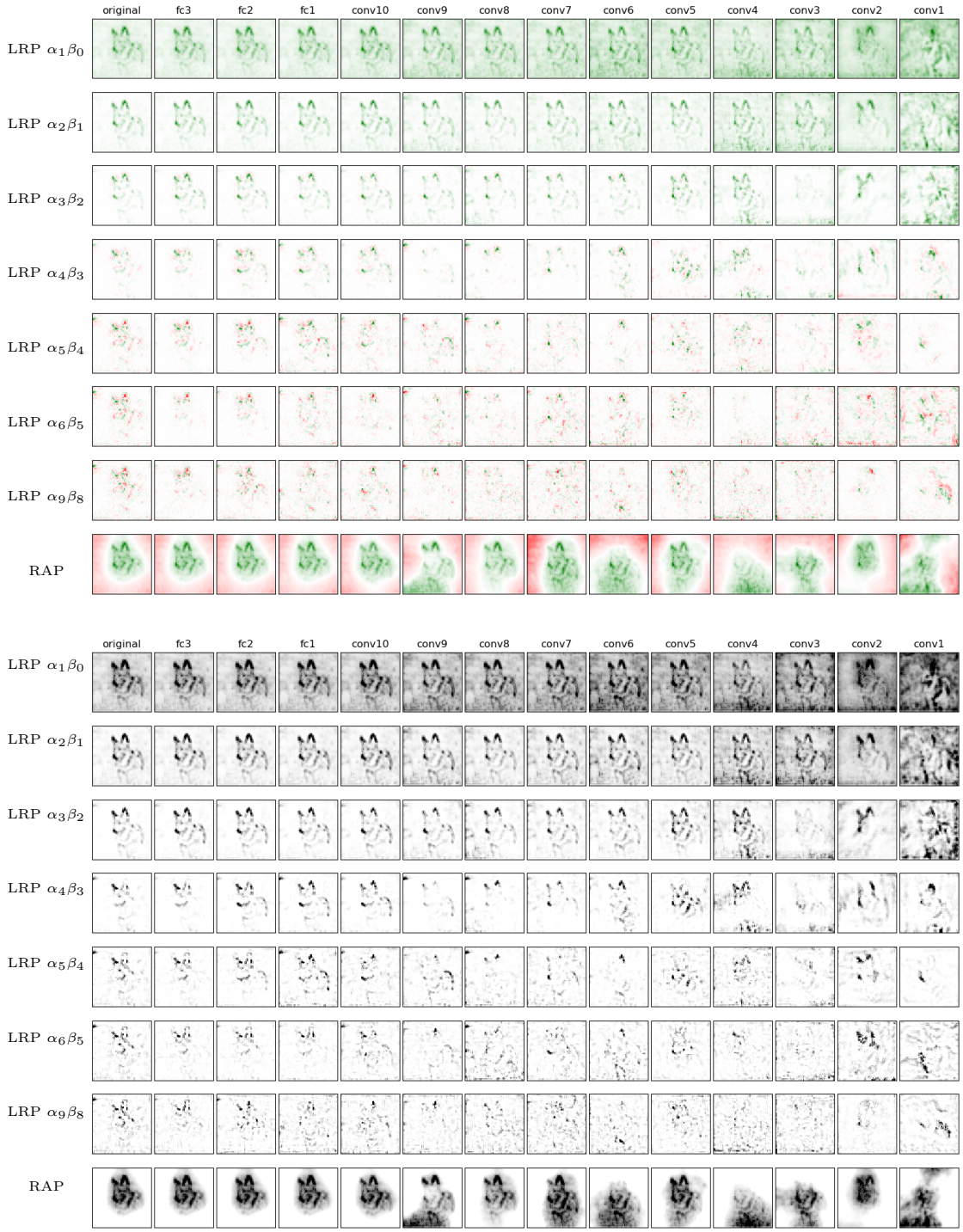


Figure 7.36: Independent randomization for German Shepherd class. Figure shows the original explanations (first column). Progression from left to right indicates a layer randomization of network weights at the time. (Top) Green denotes positive and red negative relevance. (Bottom) Only positive attribute values are displayed.

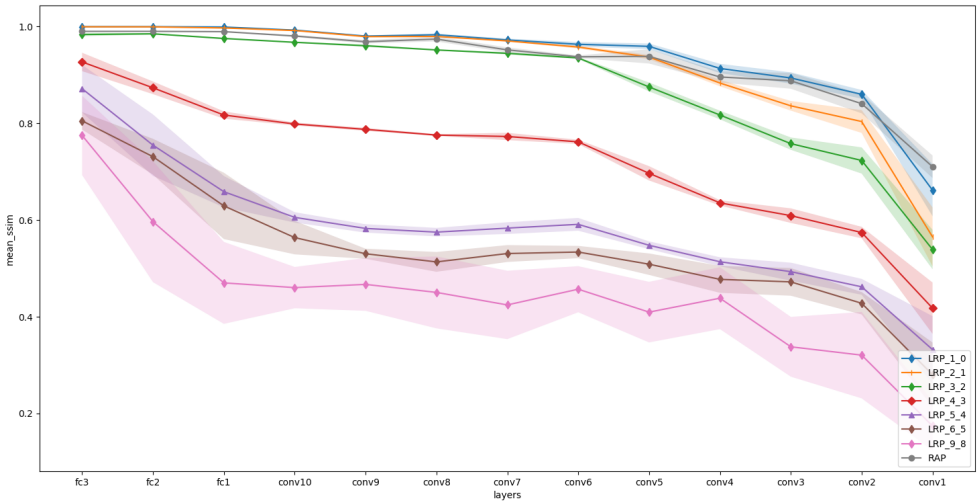


Figure 7.37: Mean SSIM plot for cascading randomization test.

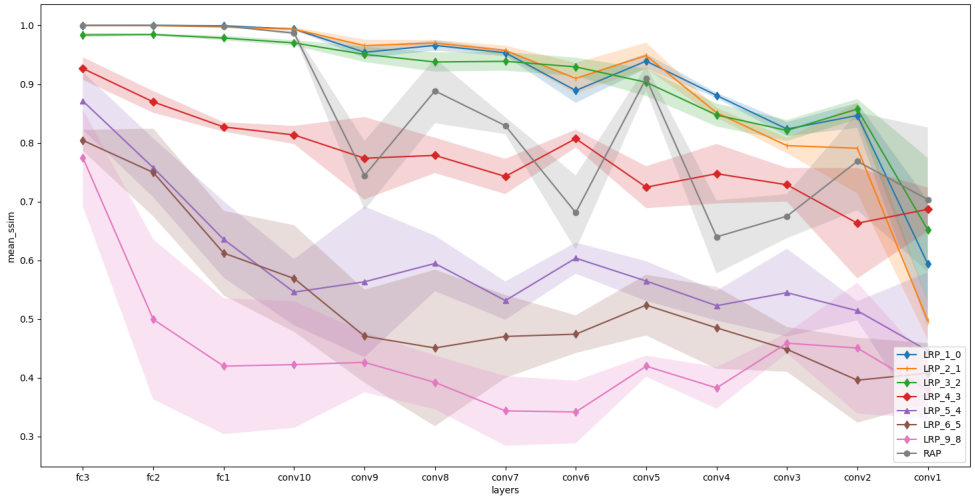


Figure 7.38: Mean SSIM plot for independent randomization test.

As we can see from these last two plots, $LRP\alpha_1\beta_0$, $LRP\alpha_2\beta_1$, $LRP\alpha_3\beta_2$ and RAP methods do not pass the test. They produce similar saliency maps even when convolutional layers are randomized (SSIM close to 1). In contrast, the rest of the group is sensitive to parameter randomization. Hence, our empirical results show that LRP $\alpha\beta$ converges to a rank-1 matrix for the most commonly used parameters $\alpha = 1, 2$. As the β value increases and therefore the influence of negative contributions, the method becomes reliable. The explanation of this behaviour is similar to one provided for DeepLift: the negative and positive relevances can influence each other. This explains also why the RAP method fails the test. As we have studied in Chapter 4, the RAP method considers the negative and positive contributions separately in order to avoid that they cancel each other. Taken individually, these relevances satisfy the conditions of the 1 and so converge to a rank-1 matrix.

7.3 Class-insensitivity

Performance of an explanation method on class discriminativity task has also been used for assessing the explanation methods. In fact, to be a good visualization method, in addition to producing a clean and visually human-interpretable result, it is very important that it also produces discriminative visualizations for the class of interest. Specifically, the method should only highlight the object belonging to the class of interest in an image when there are objects labeled with several different classes. Locating a certain category in the image is essential for gaining user trust. Just think of how important it is in the clinical practice of radiology that the method highlights exactly the region where a tumor is present. A clinical diagnosis is, in many clinical scenarios, not a trivial task and is prone to interpretation errors. Let us consider the clinical case of a patient presenting with a malignant tumor in the liver and a benign tumor in the right lung. The trained network predicts the presence of liver cancer with 60% and right lung cancer with 30%. Obviously the doctor, before communicating the diagnosis to his patient, wants to make sure that the network has correctly identified the affected area, and above all, has learned to distinguish a malignant tumor from a benign one. For this purpose, the doctor applies one of the explanation methods. What is required of the method is to highlight only the region strictly affected by the malignant tumor, completely obscuring the presence of the benign tumor in the lung. If the method also highlights the area related to the benign tumor, the interpretation of the prediction could be more complicated and would increase the risk of a misdiagnosis.

In order to measure whether our methods help distinguish between classes, we selected an image from the Tiny Imagenet validation set, which contain exactly two annotated categories and we created visualizations for each one of them. The image depicts a fruit basket in which a banana and an orange are visible. For the sake of clarity, we again treat the four groups separately.

Group 1

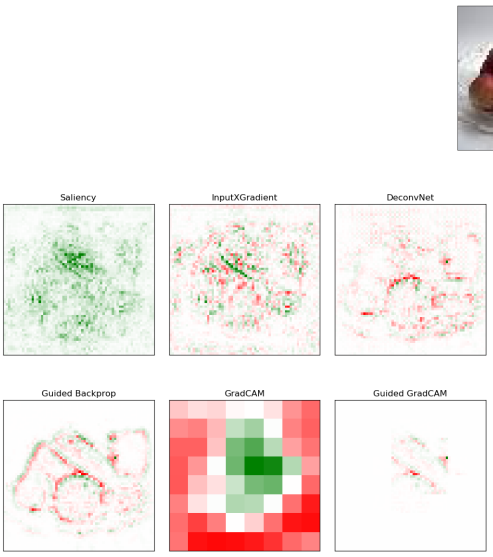


Figure 7.39: Banana class (all).

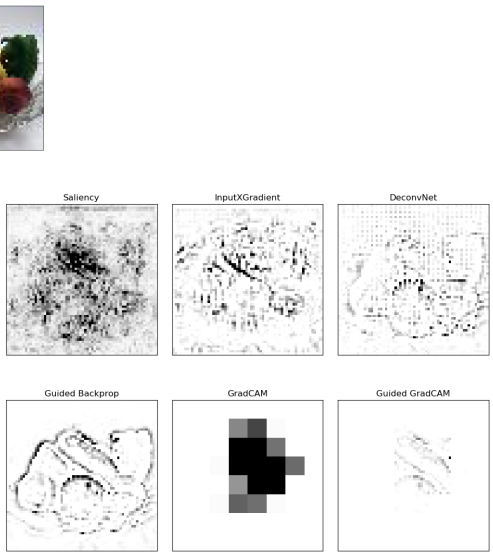


Figure 7.40: Banana class (positive).

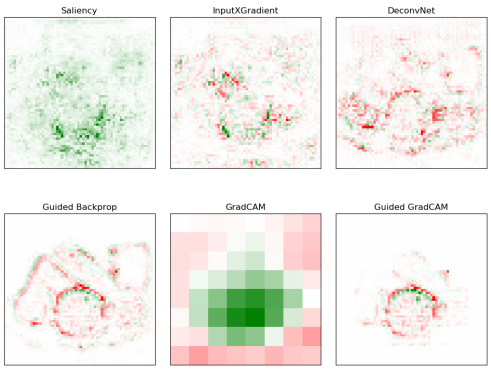


Figure 7.41: Orange class (all).

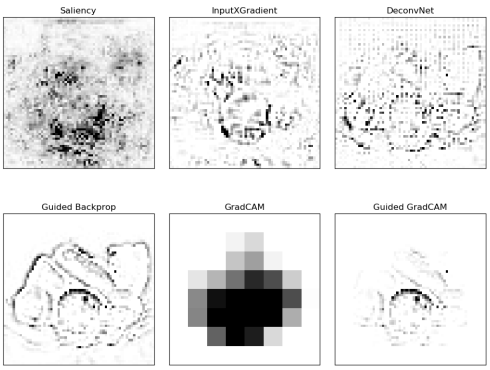


Figure 7.42: Orange class (positive).

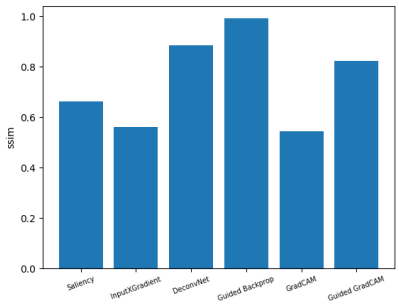


Figure 7.43: SSIM between banana explanation and orange explanation

For Saliency and Input $\odot$ Gradient visualizations, which use the true gradient, the trained weights are likely to impose a stronger bias towards some specific subset of the input pixels, and so they can highlight class-relevant pixels rather than producing random noise. From the Fig 7.43, the outputs of both methods show a SSIM value close to 0.5 between the explanation for banana class and the one for orange class. Contrariwise, as we studied in Chapter 4, Guided Backpropagation and DeconvNet generate more human-interpretable visualizations than Saliency map, but they are less class-sensitive (SSIM very close to 1 in the Fig 7.43). The heatmaps generated by them mainly depend on bottleneck information instead of the neuron-specific information. In other words, the generated explanations are not class-discriminative with respect to class-specific neurons in the output layer. As we can see in the figure, the Guided Backpropagation explanation is selective of any recognizable foreground object in the image, so all object edges are recovered and not only banana edges or orange edges, respectively. In this sense, Guided Backprop and DeconvNet may be unreliable in interpreting how deep neural networks make classification decisions. In contrast, localization approaches like GradCAM, are highly class-discriminative: the *banana* explanation exclusively highlights the banana regions but not *orange* regions in the first figure and vice versa in the second one. GradCAM shows in fact the lowest value of SSIM in Fig 7.43. Also visualizations generated by Guided Grad-CAM method satisfy this property. It in fact combines the low-resolution map of Grad-CAM and the pixel-wise map of Guided Backpropagation to generate a pixel-wise and class-discriminative explanation.

Group 2

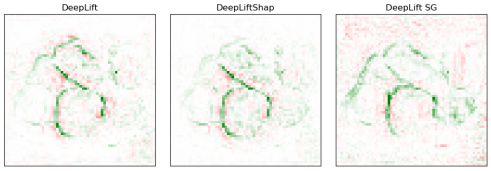


Figure 7.44: Banana class (all).

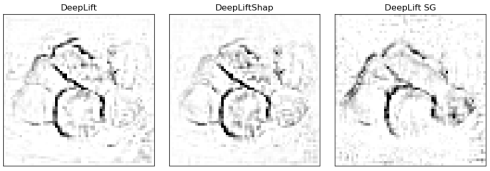


Figure 7.45: Banana class (positive).

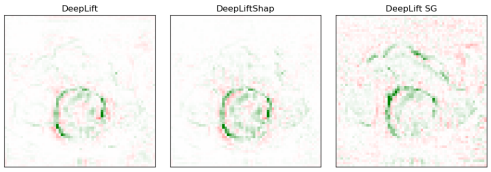


Figure 7.46: Orange class (all).

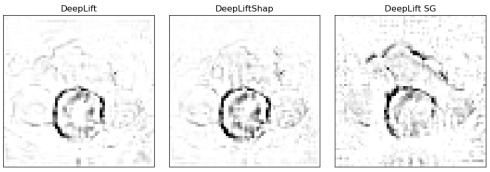


Figure 7.47: Orange class (positive).

Although the explanation for the orange class is very satisfactory in terms of class discriminativity, the heatmap provided to explain the banana class does not allow us to define the methods belonging to the second group as reliable methods. In the top row, in fact, the method should highlight only the edges of the banana, instead it also considers the orange pixels important. Such an explanation could lead to a wrong interpretation and, if we consider the example above in medical application, severe consequences for the patient.

The SSIM plot in Fig. 7.48, in this case, while showing desirable SSIM values (values other than 1), is not sufficient to draw conclusions on the class-discriminativity properties of this three methods.

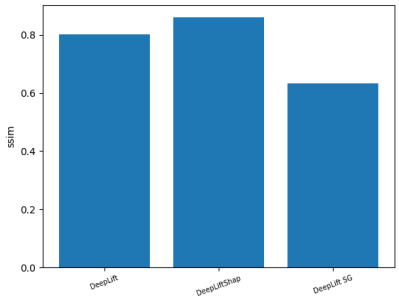


Figure 7.48: SSIM between banana explanation and orange explanation

Group 3

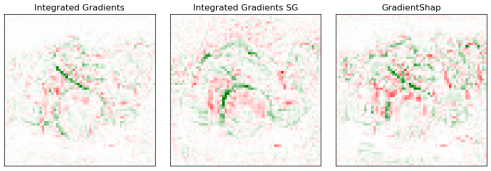


Figure 7.49: Banana class (all).

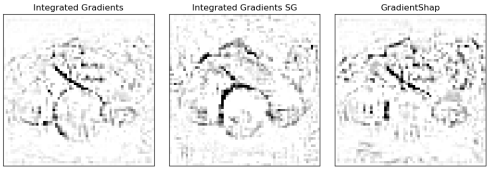


Figure 7.50: Banana class (positive)

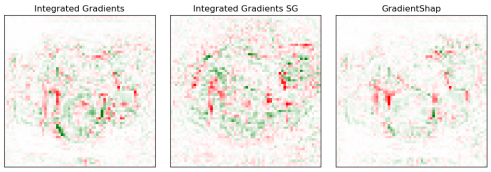


Figure 7.51: Orange class (all).

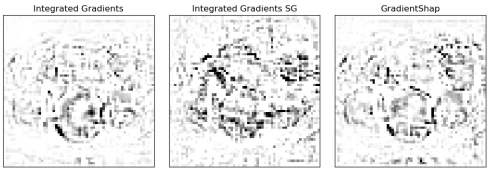


Figure 7.52: Orange class (positive).

The methods belonging to the group 3, Integrated Gradients and its variants do not produce discriminative visualizations for the class of interest. These methods in fact rely on an average of gradients and as we have seen the Saliency method is little class-discriminative. The SSIM plot in Fig. 7.53 shows that the visualizations concerning the explanation for banana class and the one for orange class are different. However, as explained before for DeepLIFT, this metric is not sufficient to define these methods class-discriminative. It seems that the explanations, although different, are not fine-grained in highlighting the class of interest.

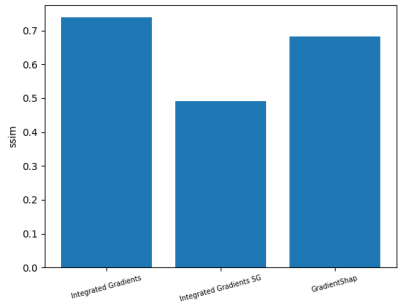


Figure 7.53: SSIM between banana explanation and orange explanation

Group 4

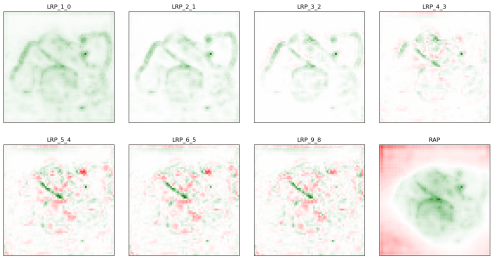


Figure 7.54: Banana class (all).

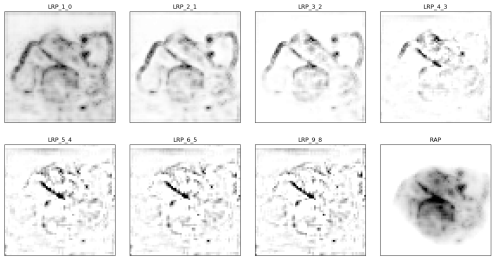


Figure 7.55: Banana class (positive).

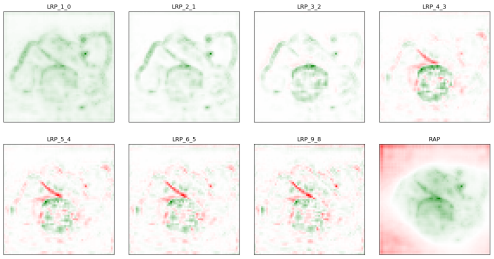


Figure 7.56: Orange class (all).

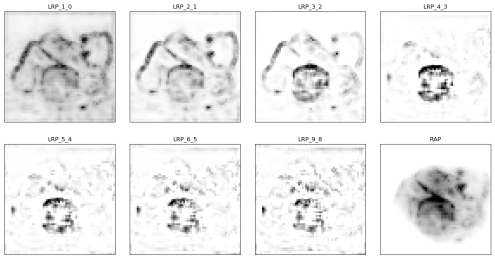


Figure 7.57: Orange class (positive).

The explanations generated by LRP and RAP are visually pleasing and are known to be instance-specific. However, they are not class-discriminative. The experiments carried by [Gu et al., 2018] show that the LRP explanations for different target classes, even randomly chosen classes, are almost identical. Our results are consistent with them: the first two versions of LRP $\alpha\beta$ ($\alpha_1\beta_0$ and $\alpha_2\beta_1$) generate the same heatmap both to explain the banana class and the orange class (SSIM equal to 1 in Fig. 7.58). The essential reason for the problem above is currently unclear. According to [Gu et al., 2018], the reason for this phenomenon lies in the dependency of the method on ReLU masks and Pooling Switches, as it happens for DeconvNet and Guided Backpropagation. These parameters decide the pattern visualized in the explanation, which is independent of class information. In particular, the assigned relevances are different from one class to another, due to different weight connections, but the non-zero patterns of those relevance vectors are almost identical. The problem is that in the explanations of image classification, the pixels on salient edges always receive higher relevance value than other pixels, but those pixels are not necessarily discriminative to the corresponding target class. Different combinations of parameters α and β are shown to modulate the qualitative behavior of the resulting explanation. The higher α and β, the more positive and negative relevance are being created in the propagation phase. By increasing the β value, LRP $\alpha\beta$ produces more class-discriminative explanations. This is confirmed by the Fig. 7.58 that shows a descending trend of the SSIM values into the different combinations of $\alpha\beta$-rule LRP.

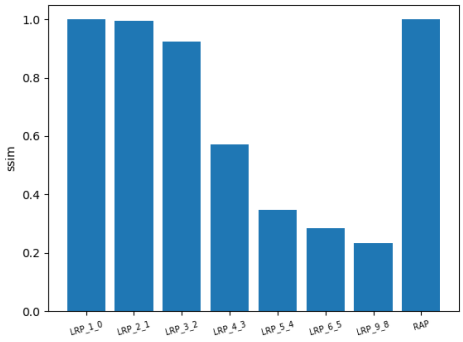


Figure 7.58: SSIM between banana explanation and orange explanation

7.4 Conclusion and Future Research Directions

Despite the explanation methods are hard to evaluate empirically because it is difficult to distinguish errors of the model from errors of the attribution method, determining how explanation methods fail is an important stepping stone to understanding where and how we should use these methods.

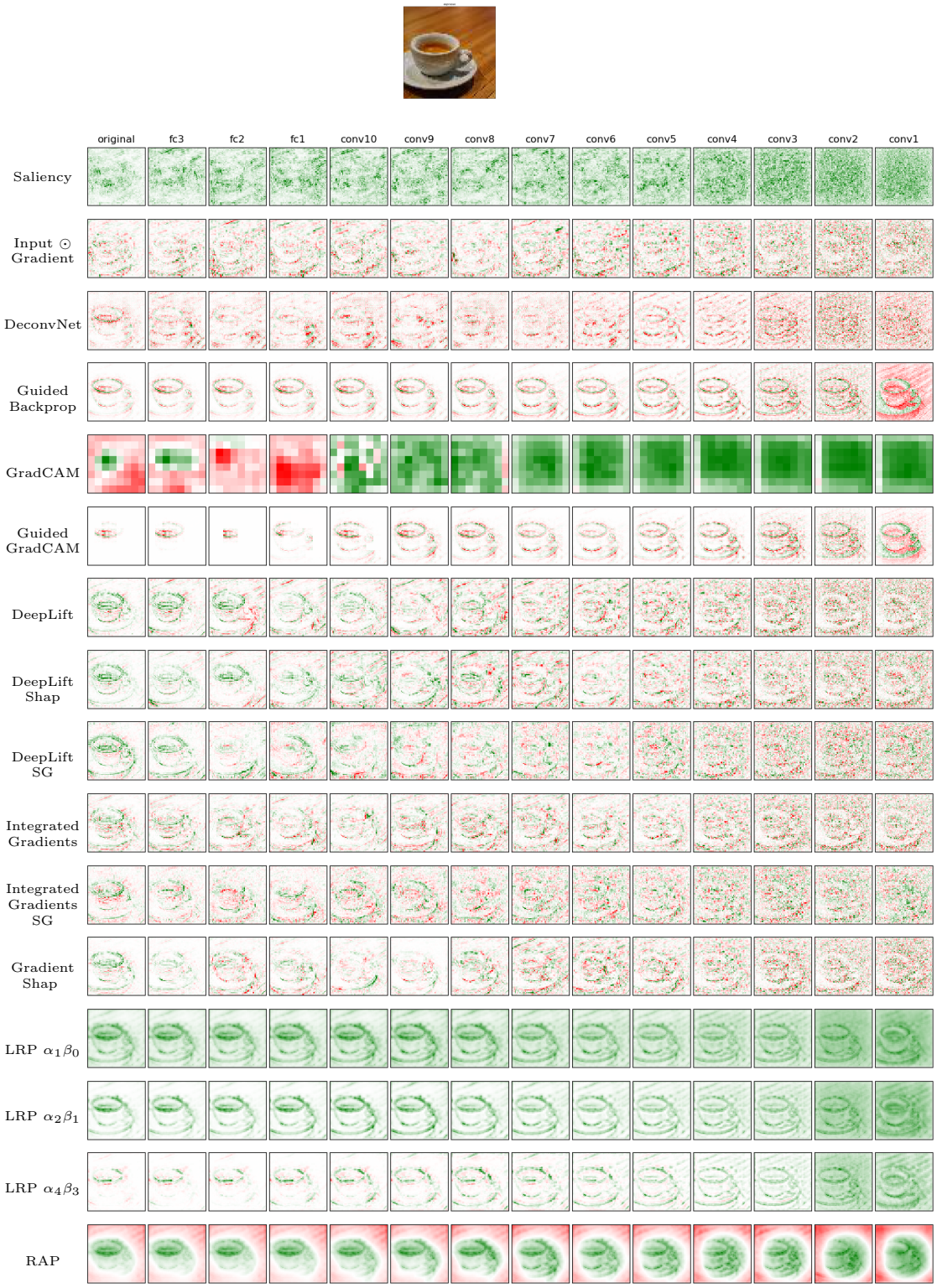
In this thesis, we analyzed several explanation methods, which aim to explain the predictions of deep neural networks for computer vision. A reliable explanation of a prediction reveals the reason why a classifier makes a certain prediction, and it helps users to accept or reject the prediction with greater confidence. Our analysis revealed that most of the attribution methods present in literature have theoretical properties contrary to this goal. Invariance under model randomization in explanation methods gave us a concrete way to rule out the adequacy of the method for certain tasks and unmask its weaknesses. However, explanations that do not depend on model parameters or training data might still depend on the model architecture and thus provide some useful information about the prior incorporated in the model architecture.

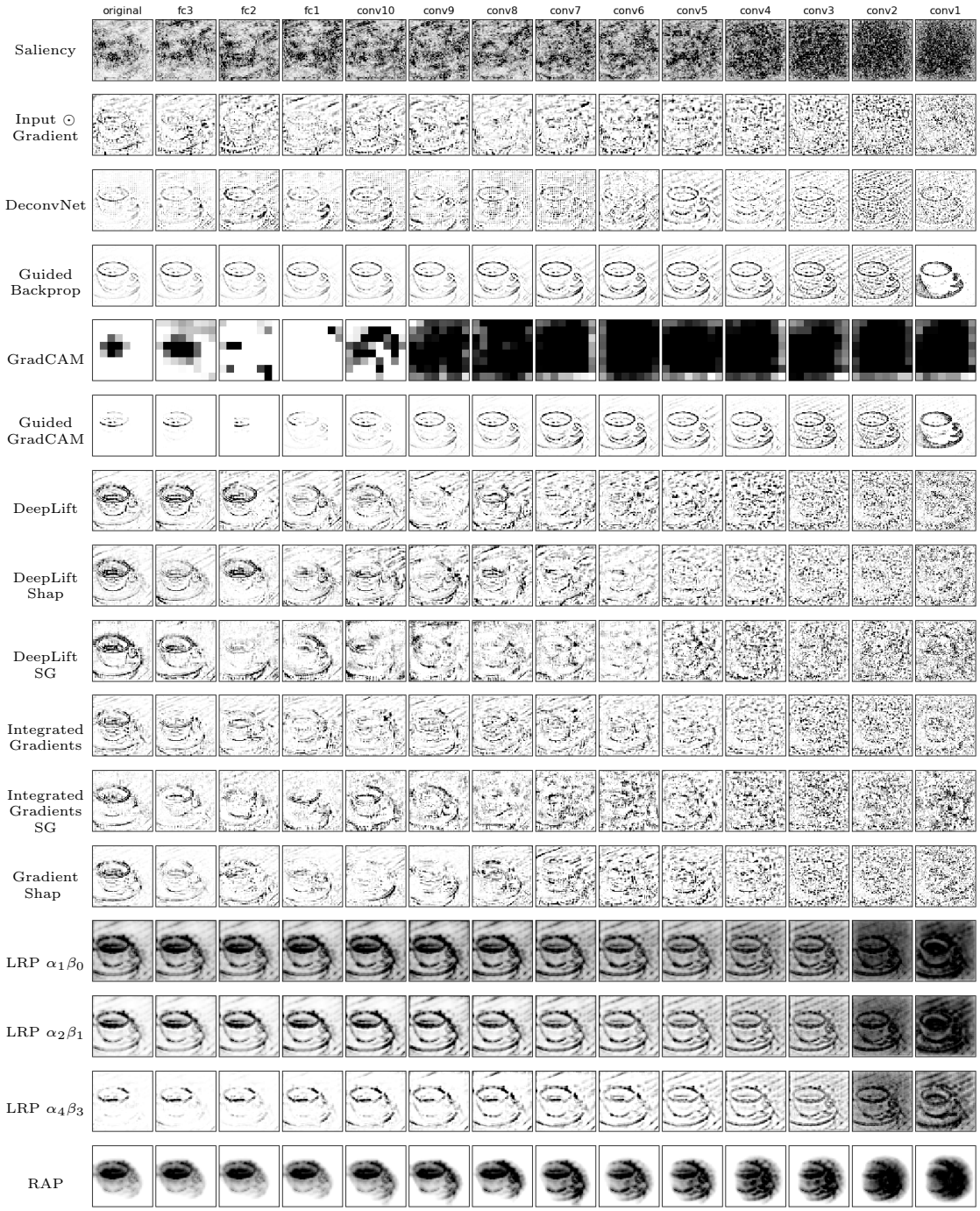
We see several promising directions for future works. A possible direction is to understand how to increase the influence of negative relevance scores in methods that backpropagate only positive ones. As we have seen, they are seems to be crucial to avoid the convergence to a rank-1 matrix, responsible of the model insensitivity, and increase the class-discriminativeness. Furthermore, we can expand this analysis to other explanation algorithms, such as model-agnostic, perturbation-based methods, and perform other tests, like label randomization, useful for investigating the trustworthiness of methods belonging to the XAI field.

Along these lines, we hope that our thesis can give researchers guidance in assessing the scope of model explanation methods and in the design of new model explanations.

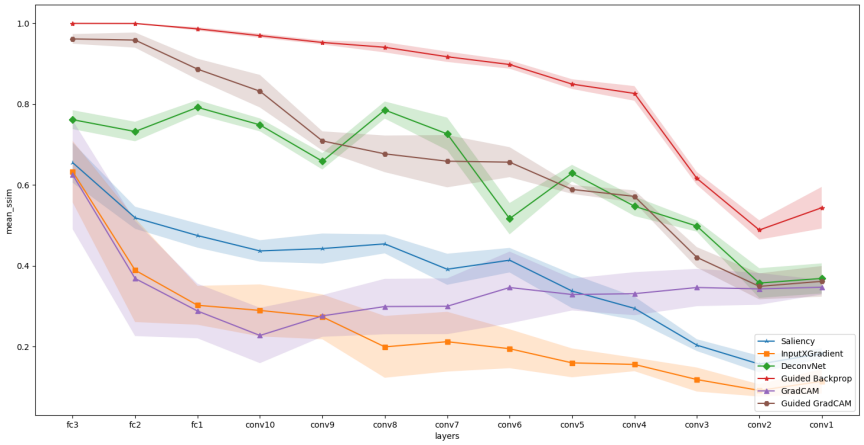
7.5 Additional figures

We now present additional figures rely on cascading randomization of other images.

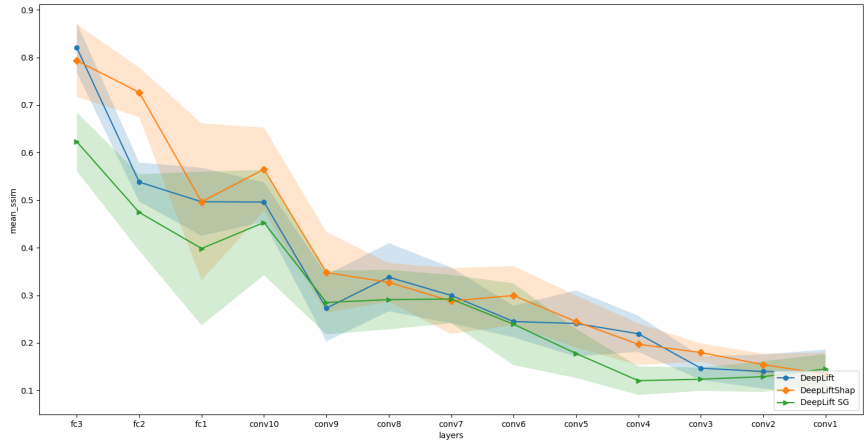



original
fc3
fc2
fc1
conv10
conv9
conv8
conv7
conv6
conv5
conv4
conv3
conv2
conv1
Saliency
Input ⊙ Gradient
DeconvNet
Guided Backprop
GradCAM
Guided GradCAM
DeepLift
DeepLift Shap
DeepLift SG
Integrated Gradients
Integrated Gradients SG
Gradient Shap
LRP $\alpha_1\beta_0$
LRP $\alpha_2\beta_1$
LRP $\alpha_4\beta_3$
RAP

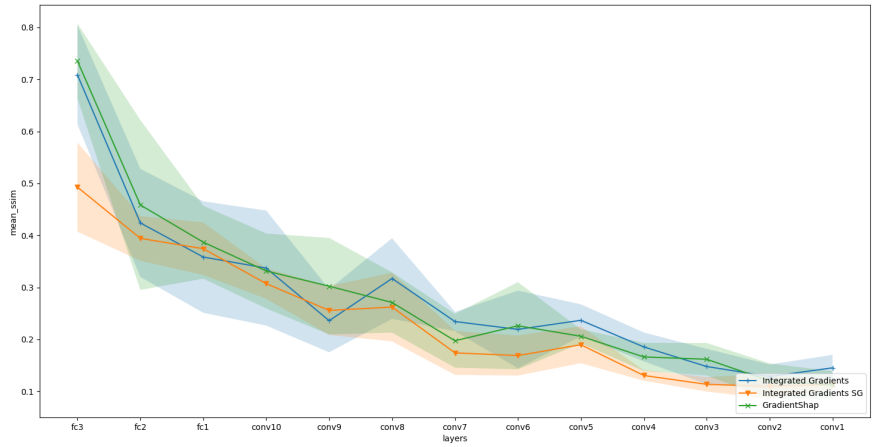
Group 1



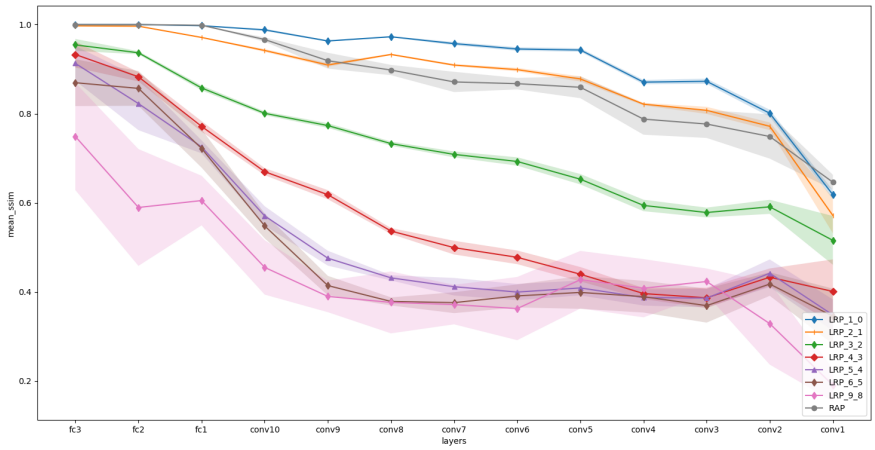
Group 2



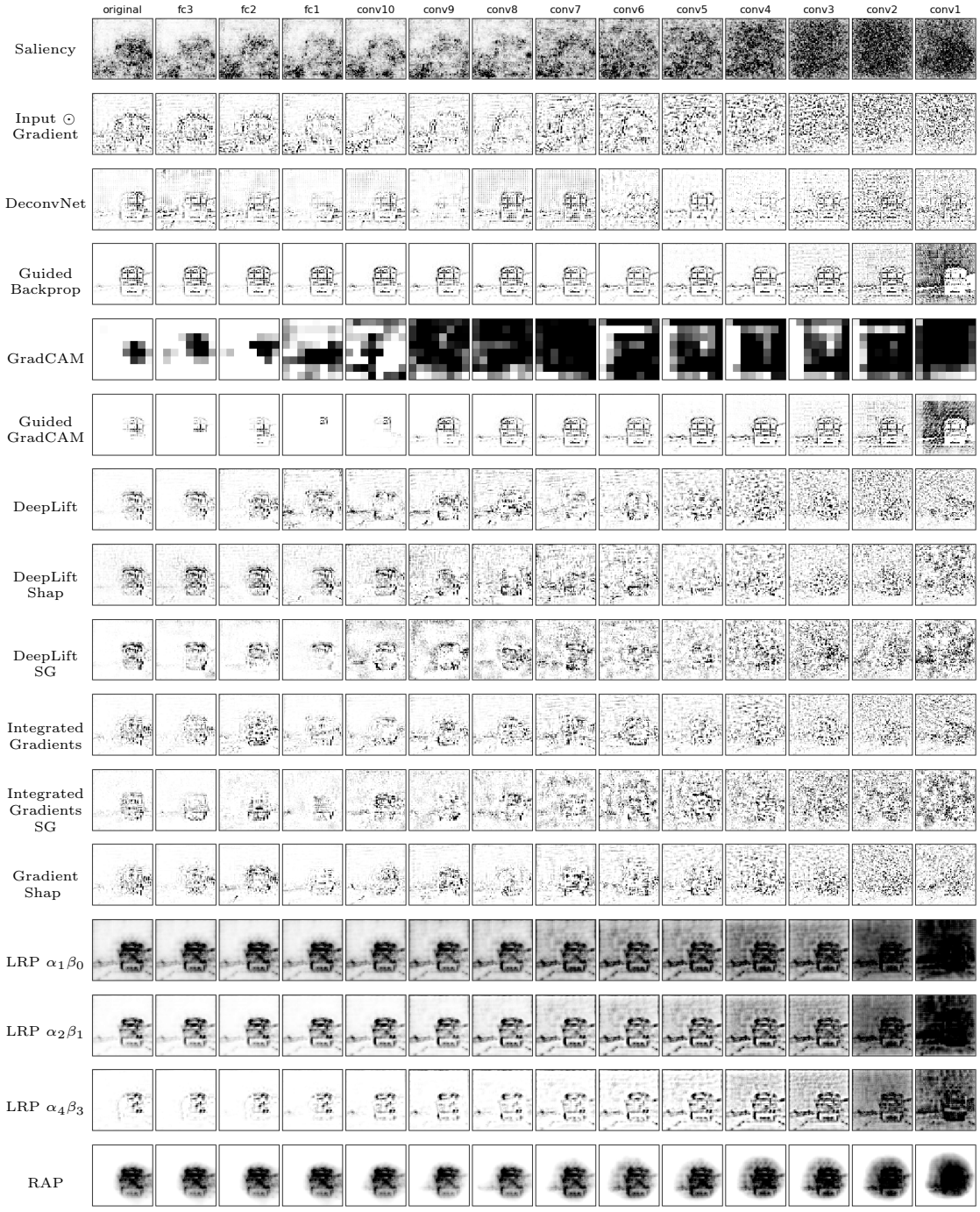
Group 3



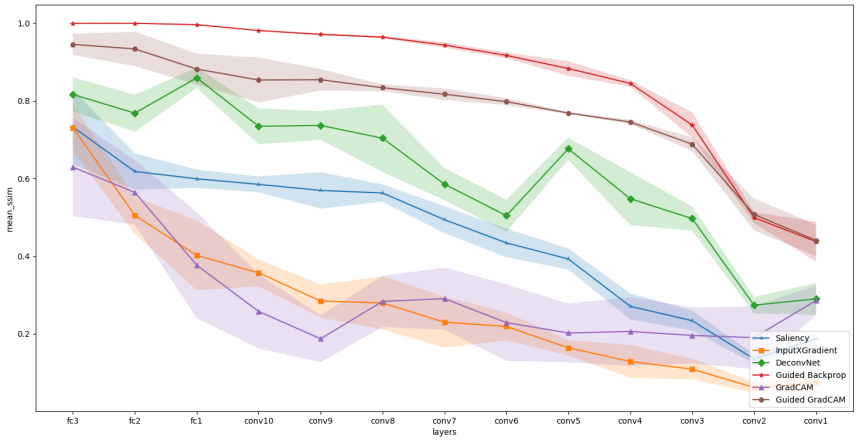
Group 4



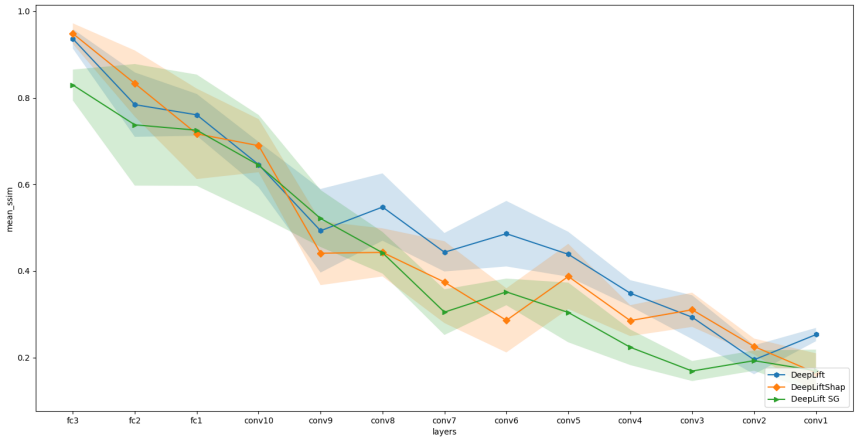

original
fc3
fc2
fc1
conv10
conv9
conv8
conv7
conv6
conv5
conv4
conv3
conv2
conv1
Saliency
Input ⊙ Gradient
DeconvNet
Guided Backprop
GradCAM
Guided GradCAM
DeepLift
DeepLift Shap
DeepLift SG
Integrated Gradients
Integrated Gradients SG
Gradient Shap
LRP $\alpha_1\beta_0$
LRP $\alpha_2\beta_1$
LRP $\alpha_4\beta_3$
RAP


original
fc3
fc2
fc1
conv10
conv9
conv8
conv7
conv6
conv5
conv4
conv3
conv2
conv1
Saliency
Input ⊙ Gradient
DeconvNet
Guided Backprop
GradCAM
Guided GradCAM
DeepLift
DeepLift Shap
DeepLift SG
Integrated Gradients
Integrated Gradients SG
Gradient Shap
LRP $\alpha_1\beta_0$
LRP $\alpha_2\beta_1$
LRP $\alpha_4\beta_3$
RAP

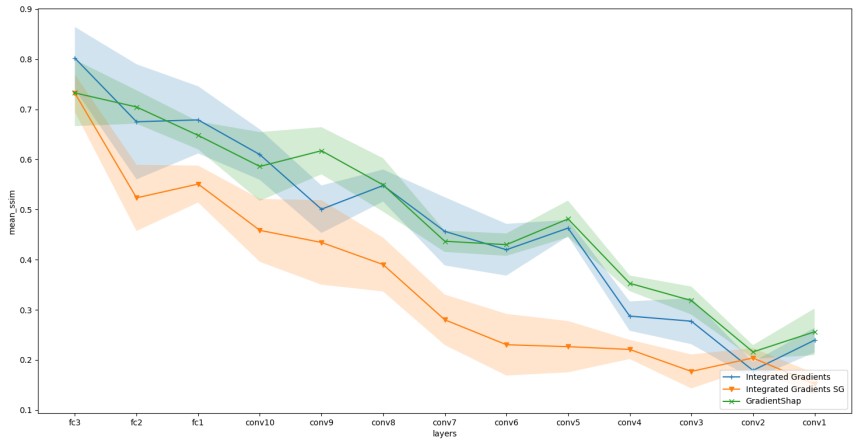
Group 1



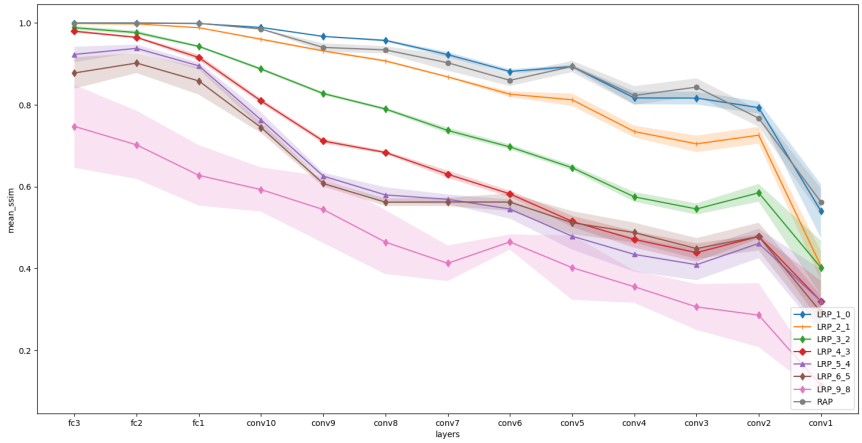
Group 2



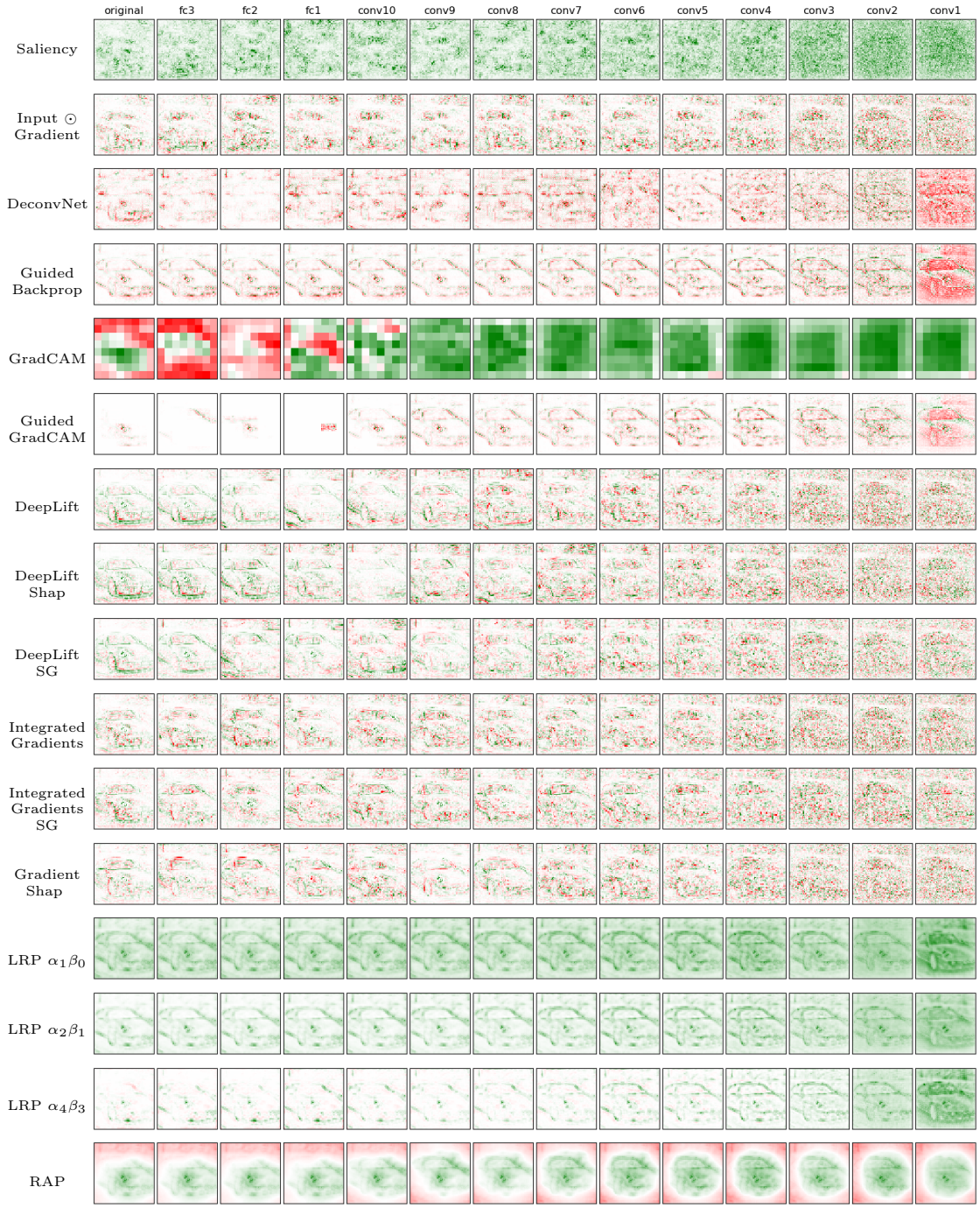
Group 3



Group 4

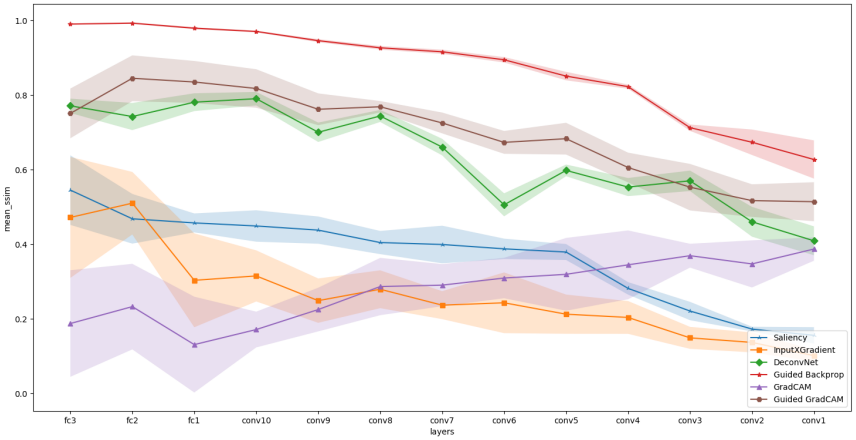



sports car

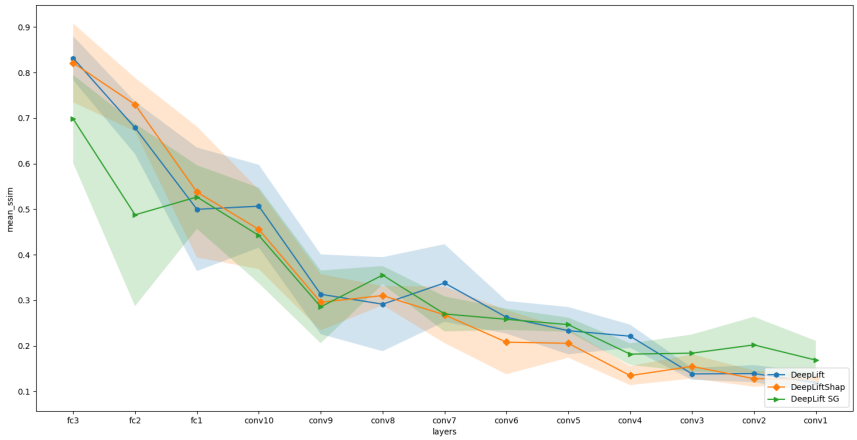

original
fc3
fc2
fc1
conv10
conv9
conv8
conv7
conv6
conv5
conv4
conv3
conv2
conv1
Saliency
Input ⊙ Gradient
DeconvNet
Guided Backprop
GradCAM
Guided GradCAM
DeepLift
DeepLift Shap
DeepLift SG
Integrated Gradients
Integrated Gradients SG
Gradient Shap
LRP $\alpha_1\beta_0$
LRP $\alpha_2\beta_1$
LRP $\alpha_4\beta_3$
RAP


original
fc3
fc2
fc1
conv10
conv9
conv8
conv7
conv6
conv5
conv4
conv3
conv2
conv1
Saliency
Input ⊙ Gradient
DeconvNet
Guided Backprop
GradCAM
Guided GradCAM
DeepLift
DeepLift Shap
DeepLift SG
Integrated Gradients
Integrated Gradients SG
Gradient Shap
LRP $\alpha_1\beta_0$
LRP $\alpha_2\beta_1$
LRP $\alpha_4\beta_3$
RAP

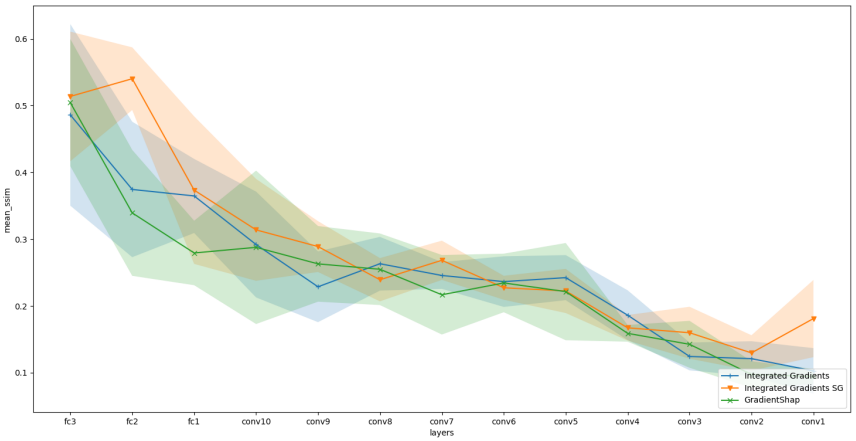
Group 1



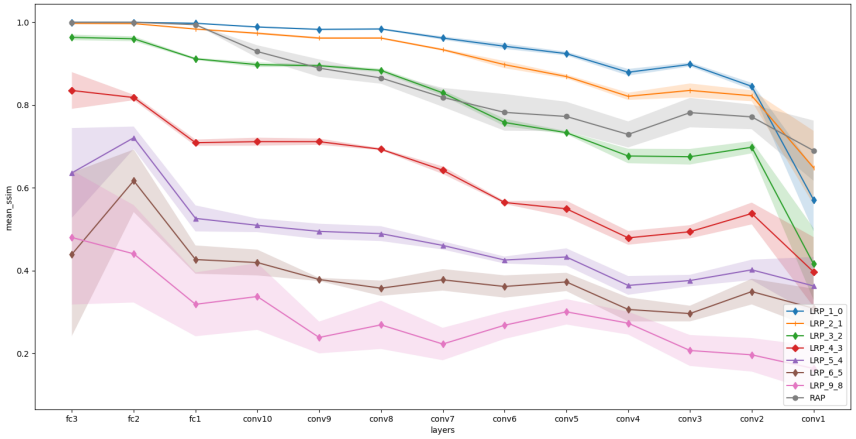
Group 2



Group 3



Group 4



Bibliography

[Adebayo et al., 2018] Adebayo, J., Gilmer, J., Muelly, M., Goodfellow, I., Hardt, M., and Kim, B. (2018). Sanity checks for saliency maps. In *Advances in Neural Information Processing Systems*, pages 9505–9515.

[Alber et al., 2019] Alber, M., Lapuschkin, S., Seegerer, P., Hägele, M., Schütt, K. T., Montavon, G., Samek, W., Müller, K.-R., Dähne, S., and Kindermans, P.-J. (2019). innvestigate neural networks! *J. Mach. Learn. Res.*, 20(93):1–8.

[Ancona et al., 2019] Ancona, M. B., Oztireli, C., and Gross, M. (2019). Explaining deep neural networks with a polynomial time algorithm for shapley values approximation. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 272–281, Long Beach, California, USA. PMLR.

[Bach et al., 2015] Bach, S., Binder, A., Montavon, G., Klauschen, F., Müller, K.-R., and Samek, W. (2015). On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLOS ONE*, 10(7):1–46.

[Chattopadhyay et al., 2018] Chattopadhyay, A., Sarkar, A., Howlader, P., and Balasubramanian, V. (2018). Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 839–847.

[Erion et al., 2019] Erion, G., Janizek, J., Sturmfels, P., Lundberg, S. M., and Lee, S.-I. (2019). Learning explainable models using attribution priors. *arXiv preprint arXiv:1906.10670*.

[Fong et al., 2019] Fong, R., Mandela, P., and Vedaldi, A. (2019). Understanding deep networks via extremal perturbations and smooth masks. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 2950–2958.

[Gu et al., 2018] Gu, J., Yang, Y., and Tresp, V. (2018). Understanding individual decisions of cnns via contrastive backpropagation. In *Asian Conference on Computer Vision*, pages 119–134. Springer.

[Kim et al., 2019] Kim, B., Seo, J., Jeon, S., Koo, J., Choe, J., and Jeon, T. (2019). Why are saliency maps noisy? cause of and solution to noisy saliency maps. *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 4149–4157.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[Lundberg and Lee, 2017] Lundberg, S. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in neural information processing systems*, pages 4765–4774.

[Mahendran and Vedaldi, 2016] Mahendran, A. and Vedaldi, A. (2016). Salient deconvolutional networks. In *European Conference on Computer Vision*, pages 120–135. Springer.

[Montavon et al., 2018] Montavon, G., Samek, W., and Müller, K.-R. (2018). Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15.

[Nam et al., 2020] Nam, W.-J., Gur, S., Choi, J., Wolf, L., and Lee, S.-W. (2020). Relative attributing propagation: Interpreting the comparative contributions of individual units in deep neural networks. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 2501–2508.

[Nie et al., 2018] Nie, W., Zhang, Y., and Patel, A. B. (2018). A theoretical explanation for perplexing behaviors of backpropagation-based visualizations. *International Conference on Machine Learning (ICML)*.

[Ribeiro et al., 2016] Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should i trust you?": Explaining the predictions of any classifier. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144.

[Selvaraju et al., 2019] Selvaraju, R. R., Das, A., Vedantam, R., Cogswell, M., Parikh, D., and Batra, D. (2019). Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128:336–359.

[Shapley, 1953] Shapley, L. S. (1953). A value for n-person games. *Contributions to the Theory of Games*, 2(28):307–317.

[Shrikumar et al., 2017] Shrikumar, A., Greenside, P., and Kundaje, A. (2017). Learning important features through propagating activation differences. In Precup, D. and Teh, Y. W., editors, *Proceedings of Machine Learning Research*, volume 70, pages 3145–3153, International Convention Centre, Sydney, Australia. PMLR.

[Simonyan et al., 2014] Simonyan, K., Vedaldi, A., and Zisserman, A. (2014). Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Workshop at International Conference on Learning Representations*.

[Sixt et al., 2019] Sixt, L., Granz, M., and Landgraf, T. (2019). When explanations lie: Why modified bp attribution fails. *arXiv preprint arXiv:1912.09818*.

[Smilkov et al., 2017] Smilkov, D., Thorat, N., Kim, B., Viegas, F., and Wattenberg, M. (2017). Smoothgrad: removing noise by adding noise. In *Workshop on Visualization for Deep Learning*.

[Springenberg et al., 2015] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2015). Striving for simplicity: The all convolutional net. In *Workshop at International Conference on Learning Representations*.

[Sundararajan et al., 2017] Sundararajan, T., Yan, M., and Ankur (2017). Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 3319–3328. JMLR.org.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9.

[Young, 1985] Young, H. (1985). Monotonic solutions of cooperative games. *Int J Game Theory*, pages 65–72.

[Zeiler and Fergus, 2014] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.

[Zhou et al., 2016] Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A. (2016). Learning deep features for discriminative localization. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2921–2929.