Universidad de la República
Facultad de Ingeniería

PhotoHolmes: Study and implementation of techniques for detecting forgeries in digital images

Memoria de proyecto presentada a la Facultad de Ingeniería de la Universidad de la República por

Julian O'Flaherty, Rodrigo Paganini, Juan Pablo Sotelo, Julieta Umpiérrez

en cumplimiento parcial de los requerimientos para la obtención del título de Ingeniero Electricista.

Tutor

Marina Gardella	École Normale Supérieure Paris-Saclay
Pablo Musé	Universidad de la República
Matías Tailanian	Digital Sense

Tribunal

Matías Di Martino	Universidad Católica del Uruguay
Alicia Fernández	Universidad de la República
Rafael Grompone	École Normale Supérieure Paris-Saclay
Lara Raad	Universidad de la República

Montevideo
Friday 14th June, 2024

PhotoHolmes: Study and implementation of techniques for detecting forgeries in digital images, Julian O'Flaherty, Rodrigo Paganini, Juan Pablo Sotelo, Julieta Umpiérrez.

This thesis was written in LaTeX using the iietesis (v1.1).
It contains 196 pages.
Compiled Friday 14th June, 2024.
`http://iie.fing.edu.uy/`

The past was erased, the erasure was forgotten, the lie became the truth.

1984, George Orwell

This page intentionally left blank.

Agradecimientos

Queremos agradecer a las personas que nos acompañaron durante este emocionante y desafiante proceso. Proceso que comenzó cinco años atrás con un nuevo capítulo de nuestras vidas, y que hoy culminanamos con este proyecto.

Agradecemos en primer lugar a nuestras familias, parejas y amigos, tanto a los que nos acompañan desde el primer día, como a los que fuimos encontrando a lo largo del camino.

A nuestros tutores, Pablo, Marina y Matías, que con dedicación, paciencia y buena voluntad, dedicaron tiempo y conocimentos a guiarnos en este proyecto. Su apoyo continuo fue fundamental para el correcto desarrollo del mismo, que tuvo como fruto un profundo aprendizaje y crecimiento del equipo, crecimiento tanto académico como personal.

Agradecemos a la Facultad de Ingeniería de la Universidad de la República, por los años de formación brindados, por la oportunidad de llevar acabo este proyecto y por todas las experiencias que nos ha dado. Queremos hacer un agradecimiento especial al Instituto de Ingeniería Eléctrica de la UdelaR, y en particular al Departamento de Procesamiento de Señales, cuyos integrantes acompañaron y motivaron los últimos años de la carrera.

Agradecemos nuevamente al Instituto de Ingeniería Eléctrica de la UdelaR, a Pento.ai y a DigitalSense, donde los miembros de este proyecto se desempeñan profesionalmente y quienes han aportado a nuestra formación, la cual fue fundamental para el desarrollo de este proyecto. Este agradecimiento se extiende también a empresas e instituciones pasadas a las cuales pertenecieron los miembros.

Agradecemos a ClusterUY, por proveernos con capacidad de cómputo y almacenamiento.

Por úlitmo, queremos agradecer al tribunal de este proyecto, por su disposición para evaluar nuestro trabajo. En especial, agradecer a Alicia, quien forma parte del mismo y fue quien nos dio el voto de confianza para realizar el proyecto los cuatro juntos.

This page intentionally left blank.

Acknowledgments

We would like to thank the people who accompanied us during this exciting and challenging process. A process that began five years ago with a new chapter of our lives, and that today, we culminate with this project.

First of all, we thank our families, partners and friends, both the ones who have been with us since the first day, and the ones we met along the way.

To our tutors, Pablo, Marina and Matías, who with dedication, patience and good will, dedicated time and knowledge to guide us in this project. Their continuous support was a cornerstone of the project's progression, which allowed the team to learn deeply and grow, both academically and on a personal level.

We thank the School of Engineering of the University of the Republic (UdelaR), for the years of training provided, for the opportunity to carry out this project and for all the experiences it has given us. We would like to give special thanks to the Institute of Electrical Engineering of the UdelaR, and in particular to the Department of Signal Processing, whose members accompanied and motivated the last years of the degree.

We thank again the Institute of Electrical Engineering of the UdelaR, Pento.ai and DigitalSense, where the members of this project work and who have contributed to our formation, which was fundamental for the development of this project. This gratitude is also extended to past companies and institutions to which the members belonged.

We thank ClusterUY for providing us with computing and storage capacity.

Finally, we would like to thank the commitee of this project for their willingness to evaluate our work. In particular, we would like to thank Alicia, who is part of it and who gave us her vote of trust to carry out the project together with the four of us.

This page intentionally left blank.

Resumen

Desde que el ser humano comenzó a pintar, hemos utilizado las imágenes como forma de comunicar información. Tras la invención de la cámara digital, las imágenes se han convertido incrementalmente en una forma de consumir información, llegando al punto en el que consideramos que un hecho es más veraz si hay una imagen que lo valide. Como consecuencia de esta importancia otorgada a las imágenes, surge una necesidad importante en poder identificar las imágenes que hayan sido manipuladas.

Esta tesis presenta un estudio del estado del arte en detección y localización de manipulación de imágenes, y el consiguiente desarrollo de *Photoholmes*, una nueva librería de código abierto en *Python*.

El trabajo comienza sentando las bases de la forensica de imágenes, partiendo de cómo se adquiere una imagen digital y cómo podemos modelar las trazas que dejan las distintas etapas de este proceso, definiendo qué constituye una falsificación y cómo pueden ser detectadas a nivel teórico, y una breve introducción al aprendizaje profundo. Habiendo expuesto estos conceptos básicos del área, presentamos una colección de conjuntos de datos y métricas seleccionadas de la literatura, profundizando en los detalles de cómo las diferentes elecciones impactan en el análisis del rendimiento de un método. A continuación, presentamos una selección de diez métodos de detección de falsificaciones, elegidos por su singularidad, desempeño y relevancia, en donde incluimos una breve descripción de cómo funciona cada método y qué sesgos puede tener su desempeño.

Para garantizar una evaluación justa y reproducible, desarrollamos *PhotoHolmes*. *PhotoHolmes* es una librería de código abierto en Python que compila los métodos, conjuntos de datos y métricas mencionados anteriormente. Además, incluye funcionalidades para ejecutar y evaluar fácilmente los métodos. La librería está diseñada para ser fácilmente extensible, con el objetivo de mantenerla actualizada con el estado del arte a medida que avanza. Utilizando *PhotoHolmes*, realizamos una evaluación exhaustiva de los métodos seleccionados en los conjuntos de datos. Finalmente, como fruto de esta evaluación, se expone un análisis de las fortalezas, debilidades y particularidades de los métodos.

This page intentionally left blank.

Abstract

Ever since humanity began painting pictures, we have used images as a way to communicate information. After the invention of the digital camera, images have increasingly become a part of how information is consumed, reaching a point where we consider something to be more truthful if there is an accompanying image that validates it. Given the importance placed on images, it is crucial to be able to identify images that have been tampered with. This thesis presents a study of the state of the art in image forgery detection and localization, and the consequent development of *Photoholmes*, a novel open-source *Python* library.

We begin the works by laying the groundwork on which image forensics is based, starting from how a digital image is acquired and how we can model the traces left by the different steps, defining what constitutes a forgery and how they can be theoretically detected, and a brief introduction to deep learning. With a basic understanding of the subject matter, we present a collection of datasets and metrics selected from the literature, diving into the details of how the different choices might impact the performance analysis of a method. We later introduced a selection of ten forgery detection methods, chosen for their uniqueness, performance, and relevance, where we include a brief description of how each method works and what biases we should expect when running and evaluating them.

To ensure a fair and reproducible evaluation, we developed *PhotoHolmes*. *PhotoHolmes* is an open-source Python library that compiles the previously mentioned methods, datasets, and metrics. Additionally, it includes functionalities to run and evaluate methods easily. The library is designed to be easily extensible, with the aim of keeping it up to date with the state of the art as it progresses. Using *PhotoHolmes*, we thoroughly evaluated the methods selected across the datasets. By analyzing the results, we identified the methods' strengths, weaknesses, and quirks.

This page intentionally left blank.

Contents

Chapter 1

Introduction

Come, Watson, come! The game is afoot!"

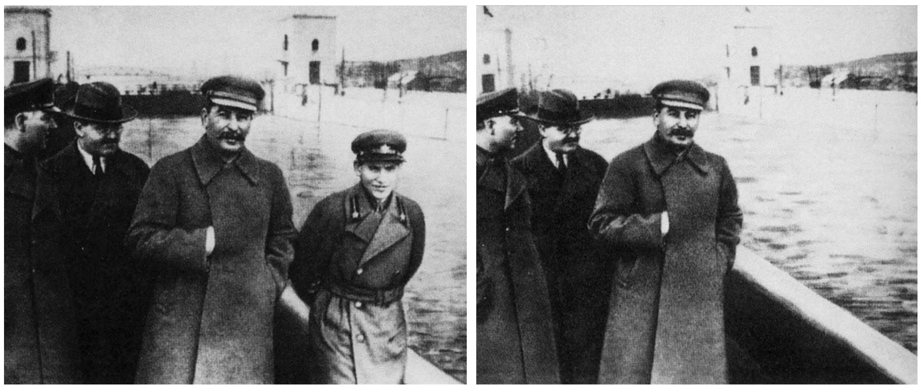
Sherlock Holmes

1.1 Motivation

Images play a crucial role not only in communications but also in the way we as humans perceive the world. There is no room for doubt around the importance of an image when it comes to corroborating a story in the press, as lately, we receive most information through images and videos instead of text. From social networks to news outlets and publicity, images are used to capture the users' attention and transmit a diverse amount of information quickly and effectively. With this comes the surge of the so-called *fake images* that have the malicious intention of spreading *fake news.*

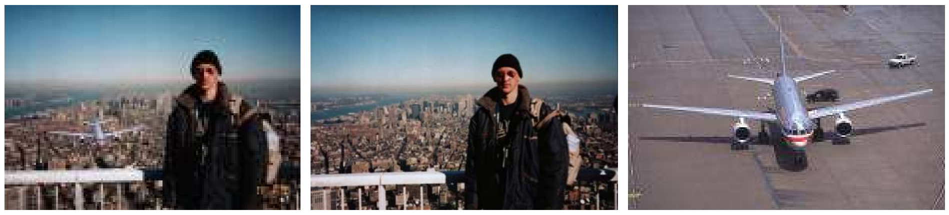
Fake images are replicated in every corner of social networks such as *Twitter*, *Reddit*, and *Instagram*, to name but a few, and probably are the main way to spread misinformation. However, these forged images are not as new as anyone would think. There are many examples of Stalinist propaganda using manipulated images such as the one shown in Figure 1.1 in which, according to M. Jansen and N. Petrov [48], Nikolai Yezhov was erased from the picture after Stalin declared him an enemy of the state.

Another notable and famous example is the one shown in Figure 1.2, usually referred to as the *Tourist Guy.* This viral image depicts a tourist, apparently unaware of an approaching airplane moments before it crashed into one of the Twin Towers on September 9, 2001. It fueled an urban myth claiming it was recovered from a camera found in the rubble. Many observers rapidly found inconsistencies in the image, from the fact that the plane was coming from the north when the airplane that crashed into that tower came from the south or that the man was wearing too many clothes, which was inconsistent with the temperature recorded that day. Later, Péter Guzli came forward and admitted that he had spliced the photo he had taken of himself in 1997 with a plane he found on the internet as a



(a) Original image (b) Forged image

Figure 1.1: Example of a forged image used by Stalinist propaganda. Image (a) shows Stalin walking by a river accompanied by three people. One of them is erased in the image (b). Images extracted from [4].



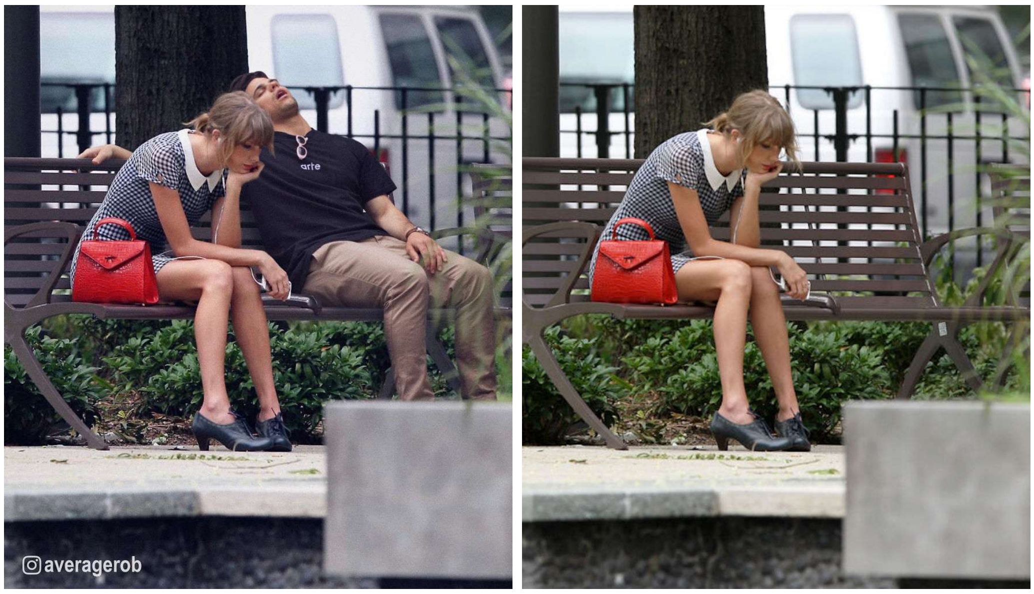
(a) Spliced image (b) Original image (c) Image used for splicing

Figure 1.2: (a) Famous spliced image denominated *Tourist Guy*. (b) Original image. (c) Image used to do the splicing. Images extracted from [6].

joke to his friends and admitted he did not think the picture would spread across the internet.

More recently, with the surge of editing software such as *Photoshop* and social networks such as *Instagram*, the circulation of manipulated images has only increased. Some harmless examples can be found in *Average Rob*'s *Instagram* profile [73] who, back in 2016, posted a lot of pictures splicing himself into images of different celebrities such as Taylor Swift, as shown in Figure 1.3. On top of that, in recent years, an increasing amount of deep learning-based techniques, such as *DALL-E 3* [18], have been released to the public, making it even more accessible to anyone to manipulate an image and change its narrative according to the interest of different people or organizations.

After considering the examples mentioned above, it is evident that certain manipulations, such as the infamous "Tourist Guy" phenomenon, can be identified by visual inspection, trying to find inconsistencies in the context of the image, misalignment of objects, differences in shadows, discrepancies in vanishing points, among others. This is called doing a semantic analysis of an image. However, detecting such manipulations is increasingly challenging as image processing soft-



(a) Spliced image (b) Original image

Figure 1.3: (a) Spliced image of *Average Rob* into an image of Taylor Swift extracted from [73]. (b) Original image extracted from [5].

ware becomes more advanced and individuals with malicious intent have easier access to these tools. Consequently, image processing algorithms have been developed to aid in detecting forged images. These algorithms take advantage of the entire image acquisition pipeline, illustrated in Figure 1.4, to detect forgeries in suspicious images. Each stage of this pipeline leaves distinct traces, enabling the identification of inconsistencies that may reveal potential forgeries.

The pipeline begins with the raw acquisition, during which the image is captured with some noise introduced by the sensors. Subsequently, the image undergoes demosaicing, a process that leaves traces due to the color filter array (CFA) grid and demosaicing algorithm. Color adjustments, such as white balance and color correction, are then applied. Finally, the image is saved, potentially in a JPEG or WebP format, introducing additional traces due to lossy compression. All of these steps will be discussed further in the next chapter.

Because of the relevance of the images themselves, it is vital to develop ways to detect image forgeries. Because of their widespread use, these methods need to be transparent and easy to use by people from areas that have no knowledge of image processing, for example by returning masks that indicate the portion of the image that was manipulated. In addition, the forgery detection community is currently in a very early stage, with few research groups actively developing methods and new techniques. This is evidenced by the lack of development tools or libraries that centralize methods, datasets, and utilities or facilitate using the same methods to perform different performance tests.

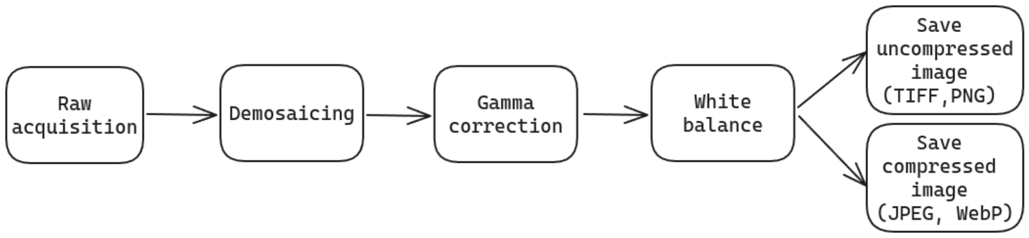

Raw acquisition
Demosaicing
Gamma correction
White balance
Save uncompressed image (TIFF,PNG)
Save compressed image (JPEG, WebP)

Figure 1.4: Full image acquisition pipeline with key steps that leave traces that are useful to later detect potential forgeries.

1.2 Goals

As explained in the previous section, it is vital to have a diverse set of methods available to test suspicious images. What was once solved by semantic analysis is now becoming virtually impossible. Indeed, the advances in editing software and new generative AI-powered tools, such as *DALLE-3* [18], allow users to create realistic forgeries impossible to detect by the naked eye. This is without considering the malicious intentions of some people and organizations that actively try to conceal said forgeries, making the challenge even more difficult. As previously mentioned, the strategy for confronting this is to find inconsistencies within the different traces left by the image acquisition pipeline. However, the nature of the forgeries is rather unpredictable, meaning that no assumption can be made about the kind of forgeries. Therefore, in most cases, inconsistencies are only present in some traces of the imaging pipeline, not all. That is why complementary methods are needed.

Due to the aforementioned reasons, this thesis aims to review different image forgery detection and localization methods and then to build a *Python* library that collects them all. This library will incorporate state-of-the-art methods that exploit the inconsistencies in as many traces as possible to increase the chance of successfully detecting forgeries. The last objective of this work is to evaluate the methods implemented using the tools provided by the library. This evaluation will contemplate different cases regarding types of forgeries and image formats, which will be done using the diverse datasets integrated with the library. Furthermore, such evaluation will be conducted using various metrics provided in the library to highlight the strengths and weaknesses of each method.

1.3 Related Works

Similar ideas to the ones proposed as the goal of this thesis have been found in the literature. For example, Zamploglou *et al.* [93] describes the creation of a *Matlab* toolbox to detect image splicing by testing different methods. However, this work's limitation is its reliance on *Matlab*, which is proprietary software, making it less usable than an open-source language. Related to that work is the Image

Verification Assistant [94] which is a website[1] created by the same authors that allows any user to upload a picture and return the result of many algorithms applied to the image. This one considers the necessity of evaluating an image with different methods to increase the chances of a correct localization of the forgery. However, due to lack of maintenance, new methods are rarely added, so the website does not rapidly keep up with the state of the art.

Another related work is a *GitHub* repository, called *matlab-forensics* containing different implementations of methods in *Matlab* [92]. Another example is the *fake-detection-lab* [91] that started with similar ideas as this project but stopped receiving updates shortly after and rapidly became outdated. The last notable related work is *InVID plugin*[2] [78], a browser plugin developed by the *Agence France Presse* (AFP) to help journalists verify information coming from social networks. Amongst several functionalities, the plugin provides forensic analysis of suspicious images by displaying the results obtained by different algorithms on the image to be tested. This plugin, which is public and free, can be used directly from a browser. This allows people without coding expertise to test images easily. Still, as in the case of the Image Verification Assistant [94], such platforms are not well-suited for benchmark purposes. Besides, the code of the implemented methods is not publicly available.

Despite the timid attempts in the field of forgery detection to develop a unified open-source library, such kinds of libraries have already emerged in other fields. A notable example is the *Anomalib* library [13], a *Python* library specially designed to benchmark and develop anomaly detection methods. Even though said library was designed with another problem in mind, the core design principles lay really close to the ones needed to achieve the goal of this thesis. Therefore, it is taken as a model to design the *PhotoHolmes* library presented in this work. Other remarkable examples are the libraries developed by *OpenMMLab*, which include *MMSegmentation* [24] for image segmentation, *MMPose* [23] for pose estimation, *MMOCR* [53] for text detection, recognition and understanding and *MMDetection* [20] for object detection, to mention a few.

1.4 Thesis outline

The present chapter introduced the subject of this thesis: detecting forgeries on digital images. More specifically, it described the motivation behind the problem and the objectives of this thesis and additionally presented related work in the field. The rest of the thesis is structured as follows.

Chapter 2 provides the necessary background to understand the problem. It starts with Section 2.1, that describes the image acquisition pipeline. Section 2.2 presents definitions related to forgeries, explains the two different challenges we study in this thesis, and describes the different forgery types one can encounter. The final Section 2.3 explains basic machine learning concepts that are used

[1]https://mever.iti.gr/forensics/index.html
[2]https://www.invid-project.eu

throughout this work.

Chapter 3 introduces the data used in this problem. The opening section of this chapter, Section 3.1, explains the purpose of data in most problems and this one specifically. Section 3.2 gives a detailed description of the selected datasets, providing some insights into the processing images suffer when uploaded to different social networks, and introduces novel WebP datasets.

The selected evaluation metrics are discussed in Chapter 4. To achieve this, Section 4.1 gives a brief definition of what is a metric and Section 4.2 begins by explaining why selecting good metrics to evaluate performance is necessary. Then, Section 4.3 gives global definitions used by almost every metric and explains the two considered approaches used when aggregating metrics in evaluating a dataset. Lastly, Section 4.4 presents a thorough explanation of the metrics chosen in this thesis.

Chapter 5 presents the chosen methods. It is constitued by ten sections that correspond to the ten selected methods, describing their methodology and the forgeries they are likely to discover.

The *PhotoHolmes* library is presented in Chapter 6, which comprises the datasets, metrics, and methods presented in the previous chapters. Sections 6.1 and 6.2 present an overview of the library and the design principles. The chapter ends with Section 6.4 describing the structure of the library.

Chapter 7 presents the evaluation of the included methods in the different datasets using some of the presented metrics, and Section 7.4 presents the main takes from the chapter.

Finally, in Chapter 8, we present this project's conclusions and future work.

Chapter 2

Background

"Elementary, my dear Watson."

Sherlock Holmes

This chapter will introduce the background concepts needed to understand the rest of the thesis. We will describe the image acquisition pipeline and the different steps an image goes through before storing it in a digital format. These steps introduce traces that can be exploited to detect tampering. In addition, we shall define what we consider a forgery, how they are created, the different types, and, more importantly, what kind of inconsistencies we can expect in each of them. Finally, we will introduce basic concepts related to the field of machine learning, on which many forgery detection methods are based.

2.1 Image Acquisition Pipeline

The image acquisition pipeline is the complete process of obtaining an image, from the moment incoming light is captured by the camera sensors until the image is stored in a digital format. This process is composed of several steps, each of which introduces artifacts that can be exploited to detect tampering. The main stages in the image acquisition pipeline are the following:

- RAW Acquisition, described in Section 2.1.1.
- Demosaicing, described in Section 2.1.2.
- Color Correction, described in Section 2.1.3.
- Image file formats and compression, described in Section 2.1.4.

2.1.1 RAW Acquisition

The acquisition is the first step of the image pipeline, and it refers to the conversion from light to the earliest digital electric signal present in the camera sensor. We describe first how the process is achieved in Section 2.1.1.1, according to [58]. With this in mind, a full statistical noise model is described in Section 2.1.1.2, based on [12].

2.1.1.1 Light sensors

In order to understand how cameras turn light into digital images, we first need to describe light on a physical level. In modern physics, light is described by a quantum particle called *photon*, which also has properties of an electromagnetic wave, such as frequency or wavelength [58]. Light travels at a constant speed, so it admits a spectral decomposition on the frequency axis or the wavelength axis since $c = \lambda f$ is constant, where c is the speed of light, λ the wave-length and f the frequency. Visible light is essentially a collection of photons with wavelengths ranging from $380\,nm$ to $700\,nm$ (the human visible spectrum). White light is comprised of an equal mixture of all wavelengths. When it impacts an object, this one will absorb energy from certain wavelengths and reflect the light in the remainder of the spectrum. The remainder is a composition of different intensities at different wavelengths. It is due to this phenomenon that, when the reflected light reaches our eyes, we see color. An optimal representation of light with color can be achieved by a tuple of three numbers describing, for example, the intensity at the red ($700\,nm$), green ($546.1\,nm$), and blue ($435.8\,nm$) colors. The sum of these three colors at different intensities will result in a colored dot at a given intensity.

Hence, a digital image can be represented as a matrix of colored light cells, or *pixels*, each being a tuple of three numbers: the light intensity at red, green, and blue. The combination of these three colors creates a colored dot, and a matrix of colored dots composes a digital image. To capture this information, one must be able to measure the light intensity reflected from the object of interest, at different wavelengths (according to the three different primary colors).

Acquiring a RAW digital image requires cameras to be equipped with sensors that convert light to digital signals. There are mainly two mechanisms to do so: charge-coupled devices (CCDs) and complementary metal-oxide semiconductors (CMOS) [58]. In either of them, light is captured by photo-diodes, which create an electric charge in response to an incoming photon. For these to appropriately measure intensity, the electric charge is accumulated in a potential well throughout a given time called exposure time. In this section, we focus on the measurement of light intensity in general. Sensing intensity at different wavelengths can be achieved through a matrix of optical filters preceding the photo-diodes and with a series of operations called "demosaicing", which will be covered in Section 2.1.2.

More specifically, a digital camera will acquire an image with a system comprised of a lens, followed by an integrated circuit that contains a dense matrix of photo-diodes and a conversion circuit that outputs the digital signal in the form of voltage. It is in this conversion system that CCDs and CMOS sensors differ: the first will have the potential well charge be carried into another potential well that later converts the signal, while the latter will convert the signal at the location of the photo-diode.

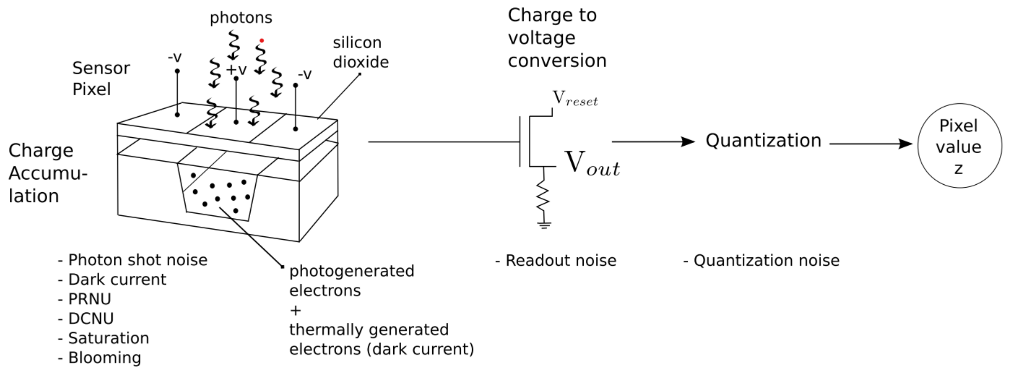


Figure 2.1: Diagram showing the pixel acquisition stages, indicating the noise sources. Extracted from [12].

2.1.1.2 Sensor noise model

In the interest of performing image forensics, this stage of the camera acquisition pipeline is where one can begin to model the digital image's noise. Noise in a signal is essentially the errors on the final measured signal with respect to the actual signal. Noise is usually modeled as a random variable following a certain distribution. An image's noise model is specific to the image itself, as it depends not only on the camera that captured the image but also on the specific parameters used during the acquisition process, which are related to the scene and the user's settings. Therefore, a digital image's noise model may be useful information to reveal tampering.

Errors or noise come from several sources in the RAW acquisition of an image (see Figure 2.1). First, a variable $S_i \sim Poisson(\lambda_i)$ is introduced to model what is called *random shot noise* and *dark current*. *Shot noise* refers to the random variations of the incoming photons in a pixel, so the same pixel may read a different value at the same light intensity, and *dark current* is the term for the thermally generated electrons stored in the potential well. Both of these can be modeled as independent Poisson variables. Therefore, the variable S_i, which is a linear combination of both, is also a Poisson variable. A Poisson distribution implies that the noise levels are signal-dependent, as the variance depends on the mean value: the intensity of the light incident on the pixel at each exposure. Moreover, the previous noise model also has a spatial dependency introduced by manufacturing details on each sensor, resulting in a variance in their gain. This is often referred to as *Photo Response Non-Uniformity* (PRNU), which is a random variable that depends both on the signal and the spatial location of the pixel and is unique to each camera.

Added to this, a second noise source N_R refers to *readout noise*, which is Gaussian distributed as it is associated with the thermal noise in the readout circuitry. This can be modeled equally for the whole image in the case of CCDs, or as column-dependent in the case of CMOS sensors, due to their difference in the readout mechanism.

Finally, another noise source is quantization noise, denoted by Q. It corresponds to the error produced by the digital representation of the voltage value and can be modeled by a uniform distribution.

Putting all of this together, and denoting g_i the total gain of the sensor signal, the complete model for the final pixel i voltage noise (RAW image before demosaicing) is:

$$Z_i \sim g_i S_i + N_R + Q. \tag{2.1}$$

This model can be simplified in some ways. The first way is to observe which is the most predominant noise. In this regard, it has been established that photon shot noise is more relevant under high illumination, while the readout noise is dominant under low illumination. Another way of simplifying this is by neglecting the quantization noise and approximating the Poisson noise as Gaussian, thus modeling the complete noise model as Gaussian as explained in [12].

Finally, the non-uniformity of the noise model is very relevant in several image processing applications but is of little use in blind image forgery detection methods as it is too weak compared to other types of traces.

2.1.2 Demosaicing

For capturing color in a real image, most cameras have a juxtaposition of sensors with different chromatic filters (red, green, and blue), meaning each pixel "truly" measures the intensity at only one of the three primary colors. With a sensor capturing one color and its neighboring sensors capturing the others, when captured, a RAW image has only one color intensity for each pixel. A three-color image can be reconstructed by applying an interpolation algorithm. This interpolation is usually referred to as *demosaicing*.

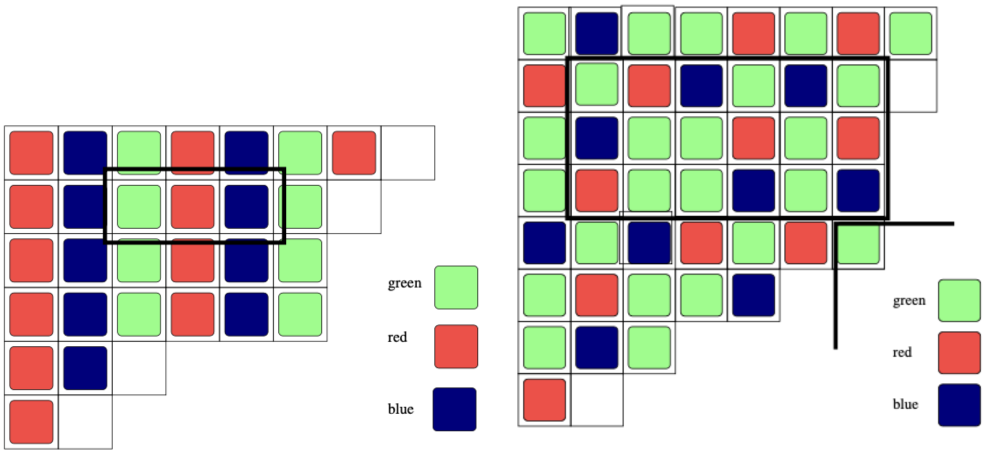
The different color sensors' arrangement configuration is called chromatic arrays or color filter arrays (CFA). There have been various configurations proposed, starting from the simple *striped* arrays (see Figure 2.2(a)) to more complex configurations such as *X-Trans* array which implements a hexagonal pattern (see Figure 2.2(b)).

One of the most used chromatic arrays is the *Bayer filter*, also referred to as *Bayer mosaic*, which B.E. Bayer invented at Kodak [17]. Due to the high popularity of this configuration, it deserves to be explained more deeply.

2.1.2.1 Bayer filtering

The Bayer filter favors the green channel, assigning half of the sensors to this color, whereas the red and blue channels are assigned a quarter of the sensors each [17]. The mosaic consists of the repetition of a 2×2 matrix (see Figure 2.3), where half of the pixels sample the green color, and are placed diagonally. Naturally, this means the Bayer array can follow four basic configurations, as shown in Figure 2.4.

This design is based on the principle that the Human Visual System (HVS) is more responsive to green detail than to red and blue details [17]. By sampling the



(a) Striped array (b) X-Trans array

Figure 2.2: (a) *Striped* array: One-third of the pixels are allocated to each color channel, resulting in an imbalanced sampling between the two directions. There is a significant undersampling in one direction, whereas the sampling is done correctly in the other direction. (b) *X-Trans* array: This arrangement is said to minimize *moiré effects* and, in turn, increase resolution by eliminating the need for a low-pass filter. Images extracted from [58]

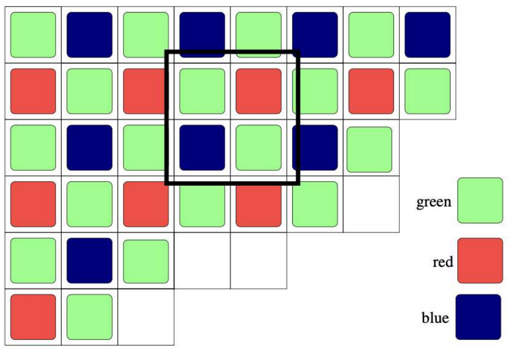


Figure 2.3: *Bayer array*: arrangement of color sensors in a Bayer filter pattern. The repeating 2x2 grid showcases the distribution of red, green, and blue sensors. Image extracted from [58].

green channel twice as much as the other two channels, the Bayer array provides a higher quality perception to the human observer by matching its visual system.

The raw output of Bayer-filter cameras is referred to as a Bayer pattern image and this image is interpolated using *demosaicing*.

2.1.2.2 Demosaicing algorithms

Let S be the matrix of sampled pixels in the chromatic array layout shown in Figure 2.5. Given a pixel (x, y), the entry $S(x, y)$ has a label associated, which can be $r_{(x,y)}$, $g_{(x,y)}$ or $b_{(x,y)}$, depending on which color was sampled at pixel (x, y), where r stands for red, g for green and b for blue. The demosaicing problem

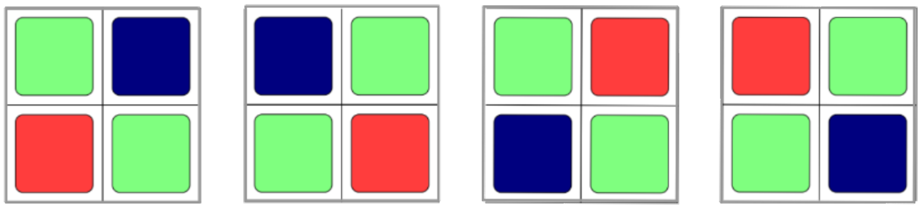


Figure 2.4: Bayer array configurations. From left to right: GBRG, BGGR, GRBG and RGGB. Image extracted from [58].

$r_{1,1}$	$g_{1,2}$	$r_{1,3}$	$g_{1,4}$	$r_{1,5}$	$g_{1,6}$	
$g_{2,1}$	$b_{2,2}$	$g_{2,3}$	$b_{2,4}$	$g_{2,5}$	$b_{2,6}$	
$r_{3,1}$	$g_{3,2}$	$r_{3,3}$	$g_{3,4}$	$r_{3,5}$	$g_{3,6}$	
$g_{4,1}$	$b_{4,2}$	$g_{4,3}$	$b_{4,4}$	$g_{4,5}$	$b_{4,6}$	$\cdots$
$r_{5,1}$	$g_{5,2}$	$r_{5,3}$	$g_{5,4}$	$r_{5,5}$	$g_{5,6}$	
$g_{6,1}$	$b_{6,2}$	$g_{6,3}$	$b_{6,4}$	$g_{6,5}$	$b_{6,6}$	
			$\vdots$			$\ddots$

Figure 2.5: The top-left portion of a CFA image obtained from a Bayer array, where the letter r, g, or b denotes the color filter at the pixel position, and the sub-indexes denote the pixel position. Extracted from [67].

consists of estimating the missing colors for each pixel. More specifically, if we define $\tilde{R}(x,y)$, $\tilde{G}(x,y)$ and $\tilde{B}(x,y)$ as the channels extracted from the chromatic filter:

$$\tilde{R}(x,y) = \begin{cases} S(x,y) & if \text{ label}[S(x,y)] = r_{x,y} \\ 0 & otherwise \end{cases}, \tag{2.2}$$

$$\tilde{G}(x,y) = \begin{cases} S(x,y) & if \text{ label}[S(x,y)] = g_{x,y} \\ 0 & otherwise \end{cases}, \tag{2.3}$$

$$\tilde{B}(x,y) = \begin{cases} S(x,y) & if \text{ label}[S(x,y)] = b_{x,y} \\ 0 & otherwise \end{cases}, \tag{2.4}$$

then the goal of demosaicing algorithms is to find the interpolated channels $R(x,y)$, $G(x,y)$ and $B(x,y)$ by estimating the missing samples in $\tilde{R}(x,y)$, $\tilde{G}(x,y)$ and $\tilde{B}(x,y)$, respectively.

Several demosaicing algorithms are designed to interpolate RAW pattern images [22, 37, 55]. Some of these are brilliant in their simplicity and achieve acceptable results, while others are highly complex but manage to represent details more finely. To get a glimpse of how demosaicing algorithms operate, we present a list of some traditional algorithms and a brief description of them, following [67]. Since the Bayer pattern is one of the most widely used CFA patterns, the demosaicing algorithms presented below are based on this array.

- Kernel-based: This methods treat each channel separately and apply linear operations to interpolate the missing values as follows:

$$R(x,y) = \sum_{u,v=-N}^{N} h_r(u,v)\tilde{R}(x-u,y-v) \tag{2.5}$$

$$B(x,y) = \sum_{u,v=-N}^{N} h_g(u,v)\tilde{G}(x-u,y-v) \tag{2.6}$$

$$G(x,y) = \sum_{u,v=-N}^{N} h_b(u,v)\tilde{B}(x-u,y-v) \tag{2.7}$$

  where $h_r(\cdot)$,$h_g(\cdot)$ and $h_b(\cdot)$ are linear filters of size $(2N+1)\times(2N+1)$ for the red, green, and blue channels, respectively. The size and values of the filters vary depending on the type of interpolation. For bilinear and bicubic interpolations, with $N=1$ and $N=3$, the filters are shown below.

$$h_{bilinear} = \frac{1}{4}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad h_{bicubic} = \frac{1}{256}\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -9 & 0 & -9 & 0 & 0 \\ 0 & -9 & 0 & 81 & 0 & -9 & 0 \\ 1 & 0 & 81 & 256 & 81 & 0 & 1 \\ 0 & -9 & 0 & 81 & 0 & -9 & 0 \\ 0 & 0 & -9 & 0 & -9 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{2.8}$$

- Smooth Hue Transition [22]: This method is based on the hypothesis that color hue varies smoothly in natural images and thus can be considered constant in a small region. Here, the definition of *hue* is the ratio between the chrominance (red and blue channel intensities) and the luminance (green channel intensities). This algorithm first estimates $G(x,y)$ using bilinear interpolation. Then, the hue channels $R(x,y)/G(x,y)$ and $B(x,y)/G(x,y)$ are biliearly interpolated from $\tilde{R}(x,y)/G(x,y)$ and $\tilde{B}(x,y)/G(x,y)$, respectively. After this is done, the missing red and blue pixels can be calculated by doing a pointwise multiplication between $G(x,y)$ and the aforementioned hue channels.

- Median filter [37]: First $\tilde{R}(x,y)$, $\tilde{G}(x,y)$ and $\tilde{B}(x,y)$ are bilinearly interpolated. Then, the pairwise differences of the interpolated channels are calculated as red minus green, red minus blue, and green minus blue, and these differences are filtered using a sliding window that converts the value of the central pixel of the window into the median value of the window, commonly referred to as median filtering. In this algorithm, the filtered differences at every pixel are used as estimations of $R(x, y)-G(x, y)$, $R(x, y)-B(x, y)$ and $G(x, y) - B(x, y)$. Hence, one can operate by applying a sum or difference between these and $S(x, y)$ to obtain $R(x, y)$, $G(x, y)$ and $B(x, y)$. Whether a sum or a difference applies depends on the estimated channel and the true color intensity $S(x, y)$ at the pixel.

- Gradient-Based [55]: This interpolation method uses the gradient information of $S(x, y)$ to interpolate missing values of $\tilde{G}(x, y)$ in order to avoid interpolating across edges. After this is done, bilinear interpolation obtains $\tilde{R}(x, y)$ and $\tilde{B}(x, y)$. This type of interpolation technique is called adaptive.

There are other demosaicing algorithms, and there will be more in the future, as this step is crucial in the image processing pipeline and directly impacts the final image color and detail quality.

2.1.3 Color Correction

When we take a picture, the image is not always a perfect representation of what we see with our eyes. This is because the camera sensor does not have the same capabilities as the HVS. The HVS is capable of adapting to different lighting conditions and can perceive a wide range of colors and intensities. On the other hand, the camera sensor has a limited dynamic range and color gamut and is not capable of adapting to different lighting conditions. This is why we need to apply color correction techniques to the images we capture, to make them look more like what we see with our eyes.

Color correction is the process of adjusting the colors of an image to make them look more natural. This process can involve several steps, such as white balance and gamma correction.

2.1.3.1 White Balance

White balance is a color-correcting process that aims to counteract the effect of a colored light source on an image, as a means to obtain an image under neutral light. In other words, white balance makes objects that are white in real life appear white in the image.

White balancing can happen automatically, as is the case for most images taken by our phones, or can be done manually if the image is taken with more specialized devices. The HVS is really good at detecting white colors under various lighting conditions, but it is a challenge to replicate this behavior automatically in digital images.

An example of an automatic white balance algorithm is the *White Patch Retinex* [33]. This algorithm works under the assumption that the brightest patch in an image corresponds to a white spot. Essentially, it works as a sort of channel-wise normalization to obtain white color at the most intense point in the image. However, this algorithm estimates the highest intensity in each channel not by the

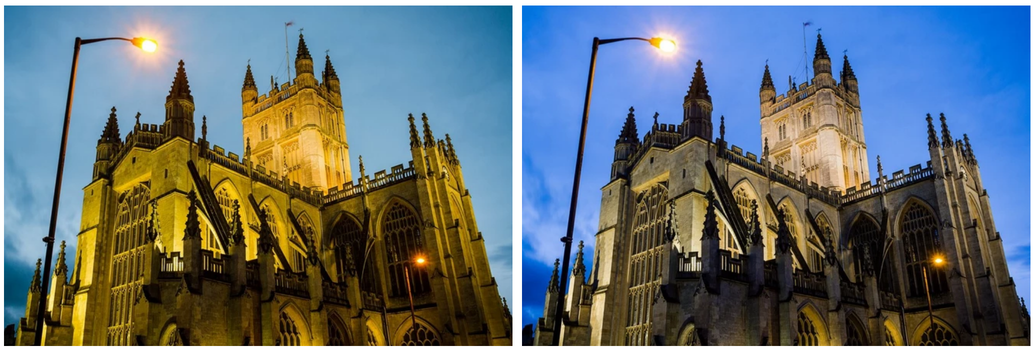

(a) Image with Canon auto White Balance (b) Image with custom White Balance

Figure 2.6: (a) A scene photographed using the Canon auto white balance setting. (b) The same scene with a custom white balance setting. Images extracted from [35].

maximum value, but rather the value at the top k^{th}-percentile, and thus corrects the image by dividing by these estimates and clipping, leading to a more robust output.

Figure 2.6 shows an example of how white balance affects an image.

2.1.3.2 Gamma Correction

As described in Section 2.1.1, the intensity is proportional to the number of photons that reach a given sensor during the exposure time. This means that if twice the amount of photons reaches the sensor, the intensity read by the sensor increases twofold. However, human perception does not follow this linear response. The HVS, under normal lighting conditions[1], follows a power law response [77]. This type of response makes human perception more sensitive to relative differences in darker tones than brighter tones.

If we were to see the captured image right off the sensor, we would have washed out images, with many details lost in the high-intensity bands. To address this, the image goes through a process called *gamma correction* that transforms the image in a way that is more suitable to our perception, allocating more bits to the bandwidths we can perceive [69].

2.1.4 Image file formats and compression

Once an image has undergone several processing steps, it is stored in a file format that a computer can read. The file format is a standard way of organizing and storing data, and it is essential to understand the different types of file formats and their characteristics to work with images. On top of that, some compression formats leave traces that are of great assistance in the context of image forgery detection. In this section, we discuss the most common image file formats and their compression methods.

[1]This response is not valid in pitch black darkness and extremely bright environments

2.1.4.1 Tag Image File Format (TIFF)

Tag Image File Format [11], commonly referred to as TIFF, is a computer file used to store raster graphics and image information. The TIFF makes it possible to store the image either uncompressed or using LZW lossless compression [25], while also being able to attach any extra information needed to the file. The file specification's copyright is held by *Adobe*. This image format is used for high-quality photographs, high-resolution scans, or as container files, as they allow the storage of multiple JPEG compressed images in one file. In the image forgery domain, TIFF images are used for their ability to store images in raw format and uncompressed.

2.1.4.2 JPEG compression

JPEG compression has become the most popular way of compressing an image to transmit it in a reasonable quality [58]. This form of compression (which is the short form of the Joint Photographic Experts Group) was born in 1992 when a group of experts presented a new method to compress images. The encoding is based on the steps mentioned below:

1. Color space conversion from RGB to a luminance/chrominance space (e.g. YCrCb). This is done leaning on the fact that according to the HVS the human eye is much more sensitive to luminance changes than to chromatic changes and allows for the second step explained below.

2. Subsampling of the chrominance channels in both directions (commonly by a factor of 2).

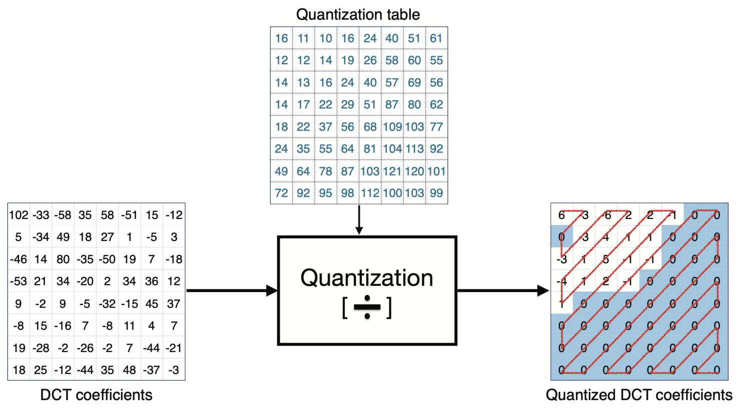
3. Division on 8×8 non-overlapping blocks. In the case of the luminance channel, this maps to a 8×8 window and, in the case of chrominance channels, into 16×16 blocks (which were subsampled by a factor of 2).

4. Discrete Cosine Transform 2D (DCT type II) of each block to concentrate the energy of each block in a few coefficients. Let B be a block of size $N \times N$ of any channel, $B(x, y)$ the pixel value of that block at coordinates (x, y) and D the DCT of that block, the value of the DCT block in the frequency pair of coordinates (k, l), $D(k, l)$, can be defined as follows:

$$D(k,l) = \frac{2}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} B(i,j)C(i)C(j) \cos\left(\frac{(2i+1)k\pi}{2N}\right) \cos\left(\frac{(2j+1)l\pi}{2N}\right)$$

   with $C(i) = \frac{\sqrt{2}}{2}$ if $i = 0$ and $C(i) = 1$ otherwise.

5. Quantization of each DCT coefficient. This is the most crucial step to accomplish the compression. Based on the HVS, the human eye fails to detect changes in high frequencies, and so the compression scheme uses that by highly quantizing those high frequencies. To do so, a quantization table Q is



Quantization table

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

DCT coefficients

102	-33	-58	35	58	-51	15	-12
5	-34	49	18	27	1	-5	3
-46	14	80	-35	-50	19	7	-18
-53	21	34	-20	2	34	36	12
9	-2	9	-5	-32	-15	45	37
-8	15	-16	7	-8	11	4	7
19	-28	-2	-26	-2	7	-44	-21
18	25	-12	-44	35	48	-37	-3

Quantized DCT coefficients

6	-3	-6	2	2	-1	0	0
0	-3	4	1	1	0	0	0
-3	1	5	-1	-1	0	0	0
-4	1	2	-1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 2.7: Quantization step in JPEG compression. Including a quantization table, the DCT coefficients, and then the quantized DCT coefficients with the zig-zag followed to achieve entropy encoding. Extracted from [70].

defined, where each entry of the table is represented by $Q(k,l)$, which contains a natural number representing the amount of quantization each DCT coefficient will suffer. Afterwards, the (k,l) entry of the DCT block is quantized as $round\left(\frac{D(k,l)}{Q(k,l)}\right)$. The quantization table is later saved in the image metadata to allow the decoding of the image, by multiplying the values by $Q(k,l)$.

The higher the quantization coefficient $Q(k,l)$, the higher the level of compression, as $\frac{D(k,l)}{Q(k,l)}$ has a smaller dynamic range (for a fixed $round(.)$ function to integer numbers). As an example, note that the least harsh compression in the JPEG standard is associated with a table Q consisting of all ones.

6. Zig-zag of the quantized coefficients to encode by entropy coding. This ordering process rearranges the coefficients within each block by following a zig-zag pattern that begins at the top-left corner and weaves through the block to the bottom-right. This specific pattern is designed to place lower frequency components, which typically hold more significant visual information and are less likely to be zero, at the start of the sequence. The high-frequency components, often reduced to zero or near-zero values due to aggressive quantization, are placed toward the end. The coefficients are encoded using entropy coding techniques.

The last two steps are illustrated in Figure 2.7, which shows the DCT coefficients after quantization, an example of a quantization table, and then the quantized DCT coefficients.

It is in the quantization step of the method that the quality factor of the compression comes into play. The quality factor is a number that ranges from 1 to 100 (the higher, the better quality) linked to the quantization table that saves the information regarding how much each coefficient was quantized.

2.1.4.3 Portable Networks Graphics (PNG) compression

Portable Network Graphics [41], known by its abbreviation and file extension PNG, is a computer file format for storing, transmitting, and displaying images. Nowadays, it is widely used in web development. Unlike JPEG, PNG compression is lossless, allowing for the full reconstruction of the image after transmission. In the domain of image forensics, PNG compression is not a relevant step of the image processing pipeline, since the lossless compression does not destroy or add any traces to the image.

PNG is also highly regarded for its support of transparency. The transparency is managed through what is known as an alpha channel, an additional layer of information that is integrated into the image file. It specifies the opacity of each pixel in the image to be used when overlayed upon another, ranging from completely transparent to completely opaque. The output image is the pixel-weighted average between the image and the background it was overlayed on, according to the value of the alpha channel.

2.1.4.4 WebP compression

WebP is an image compression format [40] developed to improve web performance by reducing the file size of images, thus minimizing the data transferred in web applications. It employs both lossy and lossless compression methods. In its lossless mode, WebP can achieve file sizes that are approximately 26% smaller than equivalent PNG files. However, its lossy compression, which is particularly relevant in the context of image forensics, can reduce file sizes by approximately 25-34% compared to JPEG images while maintaining a similar Structural Similarity Index (SSIM), a measure of image quality [85].

The procedure for WebP compression format is based on *block prediction*, and uses the same principle as existing video encoders (VP8 in particular). The image is divided into non-overlapping blocks, and an encoder will attempt to predict the block values from the previously processed blocks. The residue between the prediction and the original image is then compressed, in a similar fashion to JPEG compression (Section 2.1.4.2), by computing the DCT of each block, quantizing, and entropy encoding the coefficients. The lossy step in this procedure is only the quantization step, although the encoder allows for similar results with further compression than the traditional JPEG compression.

Lossy WebP compression is noteworthy not only for its efficiency in reducing file size but also for its support of transparency, a feature not typically associated with lossy image formats. This can result in files that are up to three times smaller than their PNG counterparts when transparency is required. This characteristic is especially significant in web contexts where transparency and compression can greatly enhance performance without compromising visual quality.

In the domain of image forensics, we are interested in lossy WebP as this image compression method introduces specific artifacts that can serve as forgery detection hints. However, they can also alter the previously existing traces left by image manipulation operations. If WebP artifacts cover the traces that the methods rely on to try to detect forgeries, the detection process may be hindered or even rendered ineffective.

2.2 Forgeries

The main subject of this work is the detection of forgeries in digital images. Therefore, it is important to understand what a forgery is, how it is made, and how it can be detected. In this section, we will explore these topics.

2.2.1 Defining a forgery

In image forensics, the main type of data are images and, in particular, forged images. A forgery is a local manipulation of an image with the intent to change the original captured scene. The use of the word *local* is key in this definition since we consider the forged regions in an image to be small compared to the size of the image. This works as an important prior for both developing methods and evaluating them.

While working with this problem, we will encounter annotated data. In this case, the data will be an image and the corresponding ground truth mask. The ground truth mask is an image of the same size as the original image. If the image is pristine this ground truth mask will be an image full of zeros. If the image is forged, the ground truth mask will be an image with ones in the area in which the image was tampered with. Throughout this work, we will refer to these masks as ground truth masks or forgery masks.

2.2.2 Making a forgery

Traditionally, forgeries in digital images were done manually, using software like *Adobe Photoshop* [1], by copying and pasting parts of an image, painting over regions whose content wants to be hidden, or even pasting some parts of images on top of another one.

Recently, there has been an outburst of text-to-image and image-to-image generation models, such as *DALL-E 3* [18], Midjourney [61] or Stable Diffusion XL [65] to name a few. Concretely, these are deep learning generative models that allow to, in one way or another, generate artificial imagery (as opposed to that generated by a camera pipeline). The images generated by these models are outstanding, and currently, the challenge of detecting whether or not an image is artificially generated is very relevant. However, this work focuses on identifying image acquisition traces, which may not be present in artificially generated images. Moreover, as better described in Section 2.2.3, the strategy will be to identify anomalous behavior within these traces, and hence completely artificially generated images are excluded from the detection methods in this study.

Having said this, we will consider a dataset that creates local in-painting of images using AI generation tools to see how the detection methods fare.

2.2.3 Detecting a forgery

In Section 2.1 we discussed how an image, after being captured by a camera, flows through a processing pipeline that converts the reading from the photo-electric sensors to a digital image. The resulting images have numerical patterns, that should be consistent throughout the image. These patterns are what we refer to as *traces*, which include noise from the sensors and artifacts from compression or demosaicing, among others.

When a modification is applied to an image, the traces are also modified. Here is where the locality of forgeries becomes key to the problem. As mentioned before, the traces are mostly uniform throughout the image, but this uniformity is altered when a part of an image is modified. There is now a new region of the image that has distinct traces from the rest of it. Let us frame the problem as a segmentation problem, where we want to segment image regions with different trace distributions.

Although it may seem like a simple problem, the reality is that trace uniformity is a crucial assumption that rarely occurs because patterns from the image scene itself are added to the traces generated by the pipeline. For instance, pixel saturation in images with extreme luminosity or darkness, or part of an image featuring a large textured region (rapid luminosity changes), can alter the traces in specific zones of the image.

One thing to keep in mind is that, while the approach described before is theoretically correct and yields the most explainable results, one can brute force the problem by training a supervised machine learning model on pairs of forged images and masks, as some deep learning-based methods do. These approaches perform well most of the time on popular datasets but are prone to errors when used in other contexts. In addition, deep learning models lack the explainability that methods that work upon the trace distributions have, which can be an important requirement in some applications of image forensics. Nonetheless, understanding how the problem can be solved enriches the research behind these methods, regardless of whether the approach is by brute force or not.

The approaches described before are statistical approaches, where detection or localization is achieved by analyzing the pixel statistics of the image. Another approach to forgery detection is by using context information within the image, which is what we call a semantic evaluation or a *semantic cues* analysis. Illumination details, noticeable cropping and resizing, duplicates within the image, all of these and more constitute semantic clues. These are elements that methods could use to detect forgeries. However, these clues are not reliable enough in light of the recent development of tools that are making forgeries almost invisible in this regard, and because forgers are paying close attention to avoid such clues.

In addition, humans could perform a richer semantic analysis by including their knowledge of the world, something that methods will not do. Some images, such as the *Tourist Guy* presented in Chapter 1, can be analyzed more in-depth by using external information, such as dates, context or even finding the original images. All in all, in this work when we refer to semantic evaluation or semantic cues, we will be referring to analyzing the suspicious area within the context of the image but without taking into account real-world knowledge that methods' are unaware of.

2.2.4 Two different challenges

By now, it should be clear that the goal of this project is to detect whether a digital image was tampered with or not. At first thought, one could assume that the subject matter is to solve a classification problem in which the target is between two classes, tampered and untampered. This problem, from now on, will be referred to as *detection problem*.

Detection outputs can be binary predictions (1 for tampered and 0 for pristine) or scores. The latter is a number ranging from 0 to 1, that can be interpreted as the probability of an image being forged.

The detection problem, although it is of vital importance, is not the only goal of this work. As was described in Chapter 1, one of the reasons that motivate this project is to provide tools for people not familiar with advanced image processing, for them to visualize which parts of an image can be trusted and which may have been tampered with. The aforementioned problem from now on will be called *localization problem*. If the method tackles the localization problem, it should output an image of the same size as a suspected image, which will be referred to as a localization map.

The localization map can be of two types, either a binary mask or a heatmap. The binary mask has two possible values for each pixel pixel, either zero, which means no tampering, or one, which means it is tampered. The heatmap is a type of localization output that allows each pixel to take a continuous value between zero and one, which should correlate with how likely a pixel is to be tampered. From this point on, we will refer to the heatmap outputs as *heatmaps*, and to binary masks simply as *masks*.

It is worth mentioning that some of the methods that tackle the localization problem also provide a detection prediction. In the cases where the method only tackles the localization, the highest values in the heatmap highlight the most suspicious regions, but no confidence is provided to judge whether it constitutes a forgery.

2.2.5 Falsification types

One might think that there are as many types of forgeries as the imagination of forgers allows. While this is true to some extent, in the field of forgery detection, falsifications are often grouped into a few categories. In this section, we will explore these categories, each of which has unique traits that can either aid detection or, conversely, make detection very difficult.

2.2.5.1 Copy-move

Copy-move forgeries consist on cloning a region of an image on itself. The copied region can also be resized and rotated when moved. Figure 2.8 provides an example

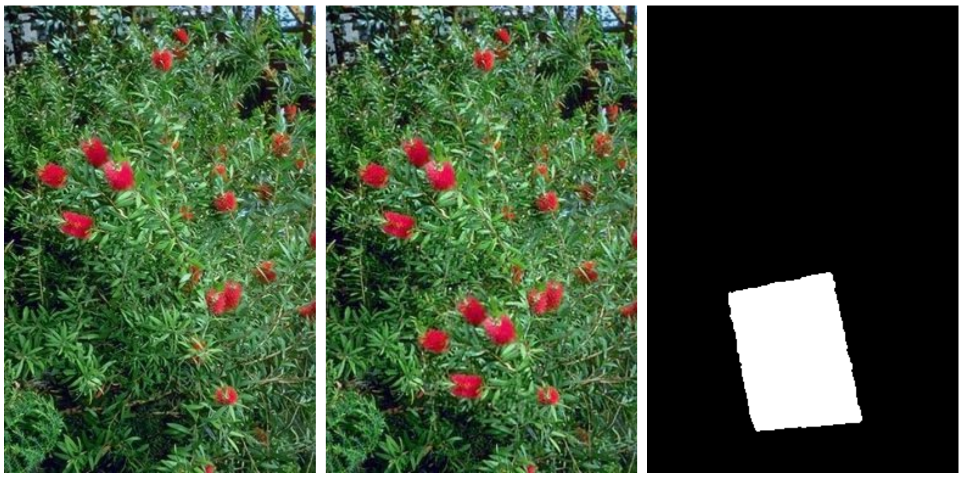

(a) Original image (b) Copy-move forgery (c) Forgery mask

Figure 2.8: Example of a copy-move forgery (b) extracted from the Casia-V1 dataset [32]. Notice how the flowers in the top left corner of the image (a) are duplicated in the forgery (b). On image (c) you can see the forgery mask.

alongside the forgery mask.

From the forger's perspective, this kind of forgery has the upside that the forged region went through the same image generation pipeline and compression history as the rest of the image, making them difficult to detect with noise-based algorithms. However, if the part used for copy-move is resized, some noise-based methods might be able to detect the forgery because the noise level will be modified.

The downside of this type of forgery is that the forged region has an exact replica in the image, so one could try and find the pair to check if it was tampered with. Sometimes, the human eye can spot the forged regions, but this is not always the case as transformations can be applied to the copied region, like rotation or scaling, or simply because the region is not easily recognizable. At the trace level, though noise-based methods rarely work on this kind of forgeries, there are still some things that can be exploited to detect forgeries, such as misalignment of the Bayer filter or the JPEG grid.

Some methods that are dedicated to this type of forgery, which seek to find two identical objects in the image. For example, in [8] a method based on detecting copy-move forgeries with Scale Invariant Feature Transform (SIFT), which identifies and describes local features in images robustly against scaling and rotation, and then clusters these features into similar groups, and when features in the same cluster are spatially separated they are reported as possible copy-move forgeries. This type of method is not the subject of this thesis.

2.2.5.2 Image Splicing

Image splicing is a technique that consists of pasting one or more regions of an image onto another. This is often the type of forgery one is familiar with, by

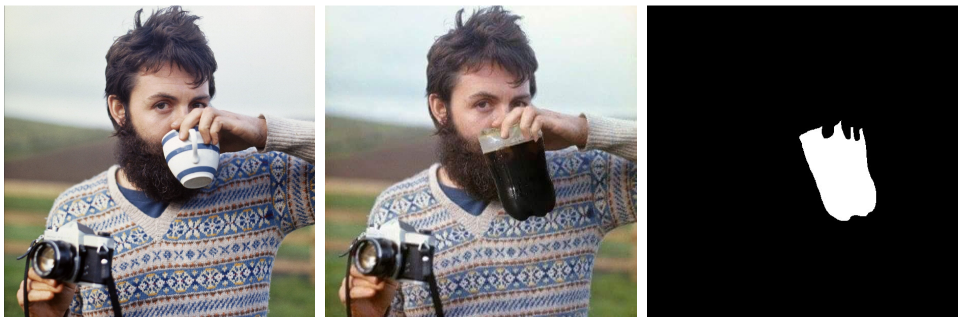


Figure 2.9: A forged image of Paul McCartney drinking fernet from a makeshift glass (b) alongside the original image (a) and the forgery mask (c). The forged image source is unknown, the original image was extracted from Paul's twitter [7]

encountering it on the Internet or social media (at least before the boom of generative models). In Chapter 1, Figure 1.2 and Figure 1.3 are two examples of this type of forgery. In Figure 2.9 we can see another example of image splicing, where Paul McCartney's teacup has been replaced by the bottom of a plastic bottle with Fernet with a cola soft drink.

In Figure 2.10, we can also see another example of image splicing with all the images involved in the forgery.

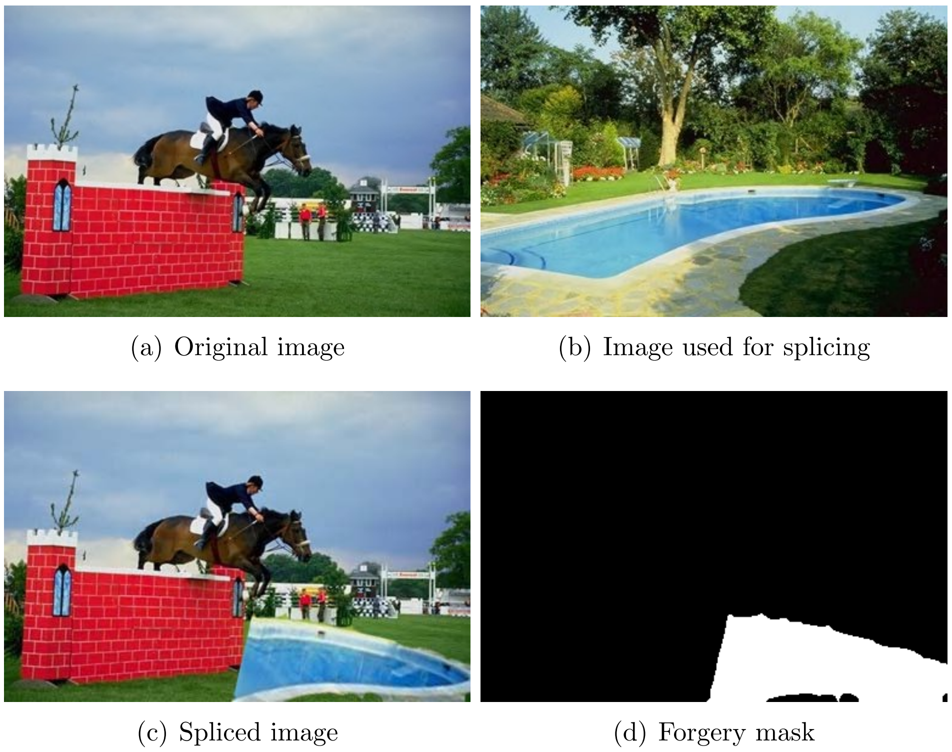
Unlike copy-move forgeries, the resulting image has regions that may have been subjected to different camera pipelines. Most methods try to exploit this by modeling pipeline traces, be it by camera noise, compression, or demosaicing, and looking for parts of the image where those differ from the rest.

2.2.5.3 Image Inpainting

Image inpainting is the task of reconstructing regions of an image. In the context of image manipulation, image inpainting is used to erase regions of an image and reconstruct them with other content. Several techniques that can be used for image inpainting. A classical approach is to replicate the image content to complete the scene. Modern approaches like Photoshop's Generative Fill [2] or ClipDrop [3] make use of deep-learning generative models to reconstruct parts of the image. They analyze the surrounding areas of the region to be filled to generate new and fitting content, as opposed to simpler copy-and-paste methods. This approach allows for more natural and cohesive image restorations or alterations. In Figure 2.11, an example of image inpainting is shown.

2.3 Machine Learning

The last piece of background we need before continuing with this dissertation is in the field of *machine learning*, as it is how most recent methods try to solve the problem at hand.


(a) Original image (b) Image used for splicing

(c) Spliced image (d) Forgery mask

Figure 2.10: Example of an image splicing forgery. In (a) the background image, (b) the donor images, (c) the forged image, and (d) the forgery mask. Extracted from the Casia-V1 dataset [32].

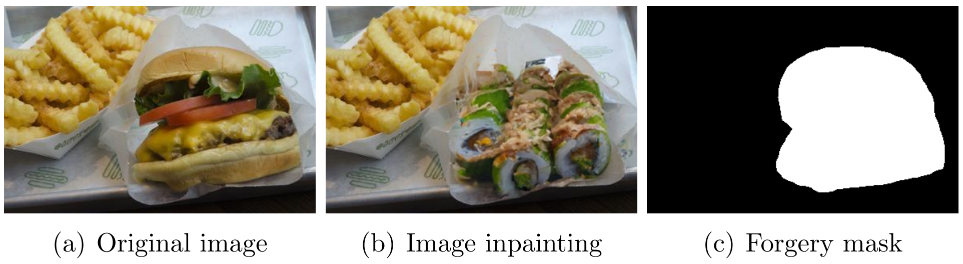

(a) Original image (b) Image inpainting (c) Forgery mask

Figure 2.11: An example of image inpainting. The tampered image (b) was made using DALL-E 2. The image was extracted from the AutoSplice dataset [49].

2.3.1 Introduction

When engineering a solution for a problem, one of the first steps is obtaining a model through physics and mathematical modeling. This model, which is a set of equations, depends on a set of parameters that represent properties or characteristics of the system being modeled. By adjusting these parameters, either through measurements or analytically, we can make the model approximate the phenomena featured in the problem, which can be used to make predictions, decisions, or an analysis.

In recent years, using *Machine Learning* (ML) to solve problems has become increasingly popular, where instead of manually describing the problem through mathematical equations, a generic parametric multi-purpose model is postulated and the parameters are fitted to a set of data using an optimization algorithm.

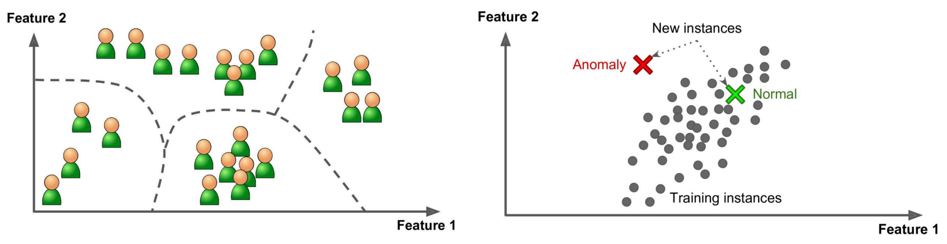
Machine learning is the field of study that attempts to give computers the capacity to "learn" from given data without being explicitly programmed [43]. More concretely, this means that the solution is first outlined by a much more general mathematical model $f(x;\theta)$, where θ is a set of parameters that can be adjusted. This model's parameters are fitted to a set of data using an optimization algorithm, which aims to minimize (or maximize) a certain metric, commonly referred to as *loss function*.

Machine learning is a vast field with a lot of different algorithms and techniques. One of the most common ways to classify machine learning techniques is in the following categories: *supervised*, *unsupervised*, and *reinforcement*. More recently, the term *self-supervised learning* has been added to this classification. Each of these types of learning was designed to tackle a different kind of problem, stemming from how the data is received and dealt with. Supervised learning deals with labeled datasets, aiming at extracting specific information from the data to make a prediction that matches the provided label. Unsupervised learning deals with unlabeled data, where the aim is to find relations between data points in the dataset. In self-supervised learning, the model is trained on a task using the data itself to generate supervisory signals, rather than relying on external labels. Lastly, reinforcement learning aims to optimize behavior, where the model is rewarded or penalized according to the events it triggers.

2.3.1.1 Supervised Learning

In a *supervised* scenario, the data is labeled, meaning the data fed to the model consists of an input $\mathbf{x} \in \mathcal{X}$ and an expected output $y \in \mathcal{Y}$. To put the problem in mathematical terms, we can say there is an unknown *target function* $g : \mathcal{X} \to \mathcal{Y}$ mapping the input data to its correct output.

The goal of supervised learning is to approach the target function g with an explicit known function f from a set of labeled input data $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$ [10]. To tackle this, a variety of hypothesis f functions are explored, whose parameters, θ, need to be adjusted to their optimal values for the specific problem. An optimizing algorithm will then find the values of said parameters $\hat{\theta}$ according to a set of data $\{\mathbf{X}, \mathbf{y}\}$, by minimizing (or maximizing) a *loss function* $\mathcal{L}(\mathbf{X}, \mathbf{y}, \theta)$. This means



(a) Clustering algorithm (b) Anomaly detector

Figure 2.12: Visual representations of some of the unsupervised learning algorithms. Extracted from [43].

that the final hypothesis $f_{\hat{\theta}}$ is the optimal solution to approximate g in terms of the loss function $\mathcal{L}$ and data available. The stage where a model optimizes to certain data, or *fits* certain data, is called *training*.

2.3.1.2 Unsupervised Learning

In contrast to supervised learning, in *unsupervised* learning, the data is not labeled, meaning that the model only has access to the input data $\mathbf{X}$. The goal of unsupervised learning is to find patterns in the data without the need for an output y. Different types of problems can be solved with unsupervised learning, but for this thesis, we will focus on clustering and anomaly detection.

Clustering aims to group the input data $\mathbf{X}$ into a set of classes. An easy example is to have a distribution of properties of coins, such as their weight or radius, and attempt to find how many types of coins there are and how to classify them. An illustration of this task is shown in Figure 2.12(a). Regarding image forensics, a good example of this can be found in some methods that achieve forgery localization by treating an image as a set of pixels $\mathbf{x}$ and trying to identify which group of pixels is tampered with and which is not. The optimal way of performing this is to apply clustering over what is known as a *feature* map of the image, not directly to the set of pixels. Feature maps of images are an important concept in ML and Computer Vision. They are essentially a transformation of the input image of size $M \times N \times 3$ into a $M \times N \times k$ volume, where each pixel is transformed into a vector in $\mathbb{R}^k$ that (hopefully) captures the relevant characteristics of it.

Anomaly detection attempts to model the "normal" class and identifies an anomaly as anything that is sufficiently odd to this class, as depicted in Figure 2.12(b). An example of solving this task is by attempting to represent normal data through a probabilistic model and setting a low probability threshold to indicate an anomaly. Naturally, finding anomalies has a lot to do with the tasks involved in forgery detection, where anomalies in traces indicate the presence of a region foreign to the image pipeline.

One commonly used technique in unsupervised learning models is dimensionality reduction, which aims to transform data $\mathcal{X}$ into another space $\mathcal{Z}$ with smaller dimensions than the original but that still captures the relevant information of the data. This is useful as more dimensions in a model mean training is a harder task that requires more data and also, reducing the dimensionality of the data can help to speed up the process of training the model. An example of this type of algorithm is the Principal Component Analysis (PCA) [43].

2.3.1.3 Self-supervised Learning

Self-supervised learning is a unique approach that falls between supervised and unsupervised learning. In self-supervised learning, the model is trained using the data itself to generate labels, eliminating the need for manual labeling. The goal is to design a pretext task where the model predicts certain parts or properties of the input data based on other parts of the same data. This approach to learning has recently gained popularity given the ability to learn from massively available unlabeled data, the more notable examples being the generative text models. Since they learn to generate text, the text itself is used both as input and target to the model.

Another self-supervised strategy that is commonly used is *contrastive learning*, where the label is whether two or more data points relate to each other. OpenAi's CLIP model [71] is an example of this strategy, where images and their captions are used to train an image and text encoder simultaneously. A batch of image-caption pairs is fed to the encoders, and then every image encoding is compared with every caption encoding. The objective is to cluster the encoded representation if the image-caption pair is correct and disperse them if they do not match.

2.3.1.4 Reinforcement Learning

In *reinforcement learning* the model learns by interacting, using declared rules, with an environment. The model receives a reward or penalty for each action it takes, and the goal is to learn a policy that maximizes the reward. This type of learning is commonly used in games, where the model learns to play a game by playing it and receiving a reward or penalty for each action it takes.

2.3.2 Data and biases

ML is a data-driven solution, and the parameters of the model are adjusted to fit the data, so the data quality is crucial for the model's performance. For instance, the data needs to be representative of the problem and include all the possible variations of the problem in order for the model to learn to tackle it correctly. In some cases, data is not as abundant as needed, and so the model may not be able to learn the problem correctly. This happens in image forensics, where it is hard to obtain labeled data with actual in-the-wild forgeries (a problem we will discuss further in Chapter 3), and researchers opt to simulate the required data, making it hard for the model to have insight of the real problem.

Behind this difficulty is the concept of *generalization* [43]: how well the model performs beyond the training dataset, in the real world. Even when the dataset is representative enough, an appropriate dataset needs to be large enough for the

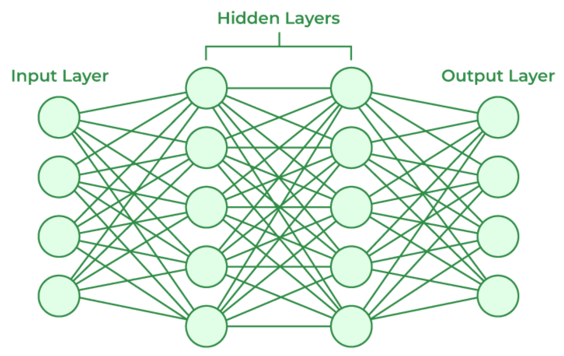


Figure 2.13: Graph representing the structure of a very simple neural network. The vertical alignment of dots represents a layer. This diagram contains an input layer, a couple of *hidden* layers in the middle (as an example), and an output layer. Image extracted from [39].

model to "see" the variability of the problem. In fact, the more complex the problem is, the more data is required to train a model that generalizes well.

Another issue to avoid in the quest for generalization is *overfitting*. It refers to learning patterns from the data instances themselves rather than patterns from the general data of the problem. A way to estimate this is by the gap between the performance on train data compared to the performance on unseen data. To be able to measure this, available data needs to be split before training into a train and a test dataset, so that the model only trains on certain data and is later validated on new unseen data in the test dataset.

Regarding this thesis, the lack of data is not a problem as there are several popular datasets accessible for research purposes that will be used for benchmarking. However, it is important to be cautious about datasets used in the training of the machine learning-based methods, as it is likely they will be biased to perform better within these, as we will discuss further in Chapter 3.

2.3.3 Neural Networks

One of the most famous ML algorithm families is the *Neural Networks* (NNs). Essentially, these algorithms take an input and transform it into a predicted output by transferring the data over a stack of *layers*, where each layer is comprised of an affine function followed by an activation function (a non-linear transformation) [43]. The concatenation of affine and activation transformations of the input data produces an output that can be a number, vector, or feature image, which can be used for regression or classification purposes.

The nomenclature of Neural Networks comes from a basic graph diagram of layers (Figure 2.13) and the propagation of the input from one to the next, which can be said to resemble the connections of neurons in the human brain.

Earlier, when discussing the supervised learning paradigm in Section 2.3.1.1, we mentioned that models have a defined architecture and a set of parameters to tune to the data. With the development of computing power, the number of model parameters has increased, as has their complexity. This advancement leads to the term *Deep Learning* (DL), which refers to models with a large number of layers and great expressive power. This term is more than just a term for a collection of architectures, as several training difficulties and techniques are related to this field specifically. Within common DL models come the Convolutional Neural Networks (CNNs) [43], where each affine transform is a linear convolution with a fixed kernel. The more recent Transformers [81] paradigm treats image input and outputs as a sequence of inputs by dividing them into blocks and applying a series of *attention* mechanisms to them.

In this chapter, we have presented the different steps in the image acquisition pipeline, such as RAW acquisition, demosaicing, color correction, and image formats and compression. We have also defined what constitutes a forgery, how it is made, how to detect it, and the different categories into which it is classified. Additionally, we have briefly introduced the generalities of machine learning and relevant concepts in this field.

In the next chapter, we will delve into the importance of data in image forensics and the datasets chosen for this work. We will also discuss the importance of selecting good datasets and how they can influence the results of our work. Finally, we will provide a comprehensive report of all the datasets selected for this work.

This page intentionally left blank.

Chapter 3

What we are looking at: the data

"It is a capital mistake to theorize before one has data."

Sherlock Holmes

As stated in Chapter 1, one of the goals is to evaluate state-of-the-art methods in forgery detection. To achieve that goal, data is needed. This chapter will begin by explaining the importance of data itself in diverse fields, and then delving into the importance of benchmark data in image forensics. Finally, we will present the datasets that were chosen for this work and the reasons behind their selection.

3.1 Importance of data

When it comes to data, its importance cannot be overstated. Data is the cornerstone of any research in any field, and the same applies to image forensics. As with many other fields within image processing, data is used to adjust parameters in algorithms, either manually or through machine learning techniques, and to evaluate the performance of the proposed methods. Understanding the role of data in each of these tasks is crucial to reach good results.

Our main and central data type is images, particularly forged ones since we want to tackle the problem of detecting and localizing forgeries. Within this domain, several datasets have been proposed in the literature. The data distribution within each of them may vary depending on forgery types, compression types and rates, and differences in image acquisition. Analyzing the characteristics of each dataset allows for a more in-depth understanding of the strengths and weaknesses of each method.

3.1.1 The generalization problem

As mentioned before, data is essential in the world of research, and image forensics is no exception. When selecting the data for these tasks, it is important to consider what is called the *generalization problem*, introduced in Chapter 2. This problem arises when the data used for training or tuning the parameters of a method is not the same as the data the method will encounter in the real world. Selecting a good evaluation dataset is crucial to reflect the method's real-world performance.

As mentioned in Chapter 2, some of the recent works in image forensics are based on machine learning techniques. These models are *data-driven*, meaning they are optimized to recognize patterns in a certain dataset, which is referred to as the training set.

Most classical forgery detection algorithms, which rely on filters, thresholds, and mathematical constraints, have tunable parameters that can greatly impact the method's performance. These parameters are usually adjusted using a training set to find the best values.

All of these algorithms and models have one thing in common: They perform well in the training set. However, they may not perform as well in the wild. This may happen because the parameters were adjusted to maximize the performance in the training set, and the training set may not be representative of the data in the real world, or it may not have sufficient data to cover all the possible scenarios and learn the patterns that are present in real cases.

This poses a clear problem because it is hard to sample the variety of forgeries out in the wild, and since forgers will not necessarily disclose their methodologies, detection methods will be blind to newer or unknown forgery types. Although the *generalization problem* is common to all data-driven models in general, it is especially problematic in this field where there is a mismatch between the available research data and the data encountered in the real world. This mismatch might also be referred to as domain shift. Generally, to simulate real-world performance, the model is evaluated in a test set that ought to be different from the training set. In this work, the benchmark datasets will act as the test set, covering a wide variety of forgeries and image processing pipelines.

Several antidotes to the lack of generalization can be applied during training. However, this work focuses on evaluating the methods and the importance of selecting good benchmark datasets to reflect the methods' real-world performance.

3.2 Benchmark datasets in image forensics

As mentioned before, benchmark data is crucial for evaluating methods' performance and comparing their results against those of other methods.

The benchmark datasets will contain forgery masks indicating the tampered region. As it was explained in Section 2.2, a forgery mask is a binary image with the same dimensions as the image, where the value 1 indicates a tampered pixel, and the value 0 indicates an untampered pixel. The mask is used to compare the predicted tampered region with the ground truth. In the case of pristine images, the masks are all zeros, indicating the absence of forgery.

Given the discussions presented above, there is no room for doubt about the importance of carefully selecting datasets to ensure a fair evaluation of methods. This involves selecting datasets that encompass the full spectrum of forgery types outlined in the previous chapter and incorporating datasets with as many processing pipelines as encountered in the wild[1].

The first aspect is vital because establishing which type of forgery a particular method excels in, compared to others, is essential. For instance, some methods excel in detecting almost any type of forgery, while others focus on identifying how a specific trace left by the imaging pipeline varies across different areas of an image. The second aspect is also crucial because certain methods are designed to detect compression artifacts. Evaluating them solely on uncompressed datasets would be unfair, as they were not designed to operate under those conditions. Using compressed images is also relevant because it allows us to see how robust methods are to compression in general.

Given all of the aforementioned requirements, seven datasets were chosen, which will be described below. The first six of the chosen datasets also contain pristine images that allow for a proper evaluation of the detection problem. In the case of miniTrace, where only tampered images are available, the detection evaluation lacks true negative images. All seven datasets include ground truth masks, so they will all be used to evaluate localization.

An image and its corresponding mask for each of the selected datasets for this work is shown in Figure 3.1.

3.2.1 Columbia

The Columbia Uncompressed Image Splicing Detection Evaluation Dataset [45] contains spliced images, which are not realistic at all and could be easily detected by semantic evaluation. This means a person can identify the suspicious area by just looking at the image and considering the context. One could argue that detecting forgeries of this type does not add value to a method, as they can be easily identified by the human eye. However, the dataset allows us to see how well each method solves a simpler problem before confronting it with more challenging datasets. The authors of this dataset ensure that images are uncompressed and that their sizes range from 757×568 to 1152×768.

3.2.2 CASIA 1.0

CASIA 1.0 [32, 64] contains both splicing and copy-move forgeries, which are not so easy to identify to the naked eye and are JPEG compressed. The images in this dataset are very small: they do not exceed the size of 374×256. Given that the dataset is already separated into two different types of forgeries, in this work, we are going to consider them as two different datasets, CASIA 1.0 Splicing, and CASIA 1.0 Copy-Move, to assess the performance of the methods in each kind of forgery by itself.

[1]Given that in most datasets, the processing history of the images is not available, we are going to focus on covering a variety of compression types which is something we can get from the image itself.

3.2.3 COVERAGE

COVERAGE [86] is the most popular dataset for evaluating copy-move forgeries. The images in this dataset are stored in TIFF format, yet they were captured using the front camera of an iPhone 6, which typically saves images in JPEG format. Consequently, the images underwent compression at some stage. Despite the iPhone 6 front camera's resolution being 1280×960, the average size of the images in the dataset is 400×486. This indicates that the images underwent subsampling, cropping, or both. The pristine images consistently feature a repetition of a certain object. For the forged images, one of these objects is cut and pasted elsewhere, with the pasted object sometimes easily noticeable and other times not. This dataset helps determine whether a method merely searches for similar parts within the image to detect a copy-move forgery or if it looks for inconsistencies in traces, such as the demosaicing grid or the JPEG grid.

3.2.4 DSO-1

DSO-1 [29] is a dataset that contains spliced images of size 2048×1536 in which the subjects used for the splicing are humans. At first glance, the splices are hard to catch; however, most of the time, a semantic evaluation of the illumination (or other details) shows which subject is spliced. Even though the image format is PNG, the dataset creators do not provide users with the full editing history of the images, so we can not assume that they are uncompressed.

3.2.5 Korus

The Korus Dataset [50, 51] is also named realistic tampering. As the title suggests, this dataset contains forgeries that are almost impossible to detect through semantic evaluation. It has uncompressed images containing splicing, copy-move, and object removal. The images are all of size 1920×1080.

Although the dataset is uncompressed and distributed in TIFF format, the image `DSC07222` was spliced with a JPEG image. This will be taken into consideration when evaluating methods.

3.2.6 AutoSplice

This novel dataset incorporates generative inpainting. Jia *et al.* [49] introduce the utilization of *DALL-E2* to generate forged images guided by a text prompt. These images are JPEG compressed, and the dataset includes variations with three JPEG quality factors: 100, 90, and 75, but the 90 and 75 versions include only forged images. This diversity facilitates the quantification of how well methods can handle varying degrees of JPEG compression. The image sizes range from 256×256 to 4232×4232.

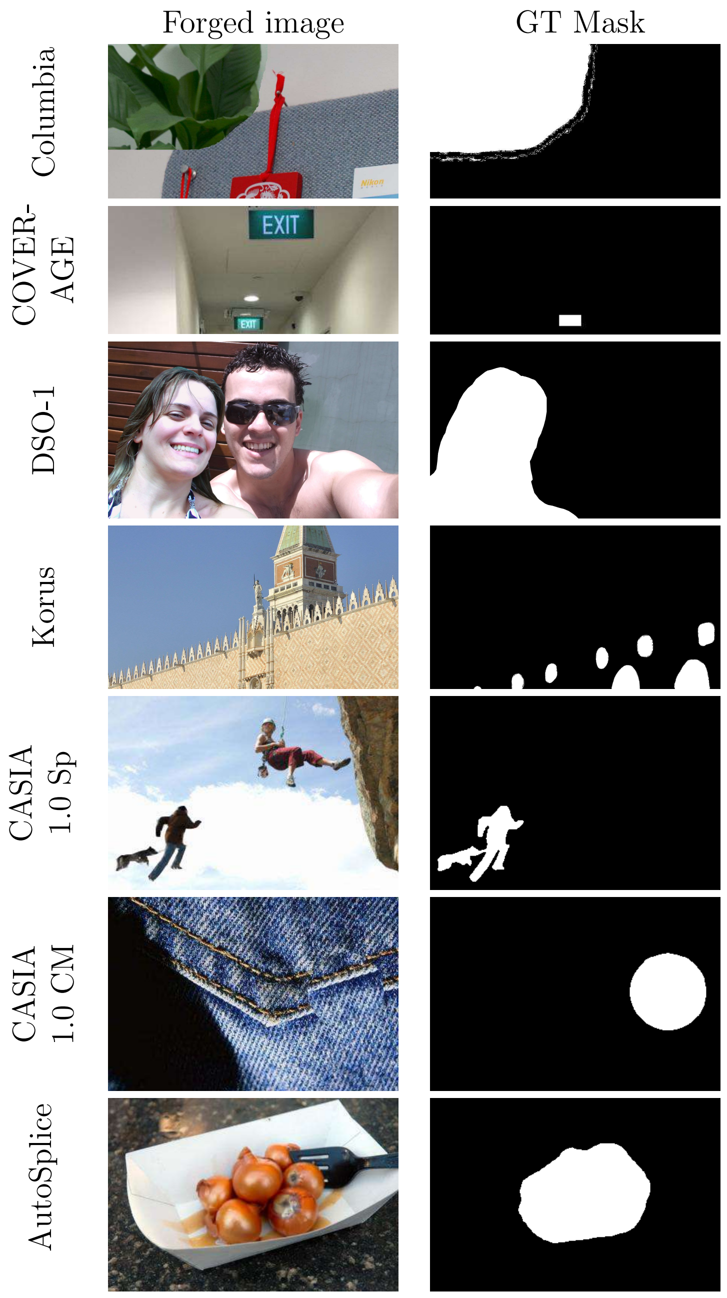


Figure 3.1: Examples of each chosen dataset, including a forged image and the corresponding ground truth mask. Some images were cropped for visualization purposes. All images were extracted from each original dataset.

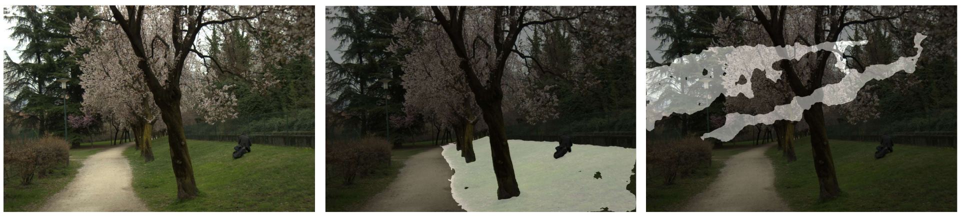

(a) Source Image (b) Endogenous mask overlay (c) Exogenous mask overlay

Figure 3.2: An example image from the miniTrace dataset (a) and its endogenous (b) and exogenous (c) masks to illustrate the mask selection procedure. The endogenous mask follows the edges of the grass patch in the image, while the exogenous mask spreads randomly across multiple objects. Extracted from [15].

3.2.7 miniTrace dataset

All of the previously described datasets encompass a wide variety of forgery types, incorporating different types of compression. From now on, these will be referred to as popular datasets. However, evaluating these datasets solely is insufficient for several reasons. Firstly, some methods use these datasets to train their networks. In such cases, it is natural to expect that these methods may outperform others that did not incorporate these datasets into their training process. Additionally, if the evaluation is exclusively conducted using these selected datasets, it raises the question of whether the methods solely rely on semantic cues or if they focus on the various traces left by the imaging pipeline.

The answer to this question distinguishes a more effective method from a less effective one, as the latter case ensures better generalization to in-the-wild forgeries, given that relying just on semantic cues may lead to overfitting on popular datasets. To address this concern, Bammey *et al.* [15] proposed the creation of the Trace dataset, providing a means for non-semantic evaluation.

In Trace, the forged and pristine regions differ only in the traces left behind by the imaging pipeline. This approach enables the creation of extensive datasets without the need to invest time in making forgeries appear realistic. The concept involves selecting and processing a raw image using two distinct imaging pipelines. The results are then merged, forming a single image with two areas, each corresponding to one of the two pipelines. The merging of these images is accomplished using a mask.

The authors propose two types of masks, resulting in the code outputting two forged images for each image-pipelines pair. The first forged image corresponds to the utilization of what the authors term an *endomask*. This *endomask* is constructed by segmenting the image and then choosing a random pixel whose corresponding segment becomes the *endomask* of the image. In this case, the forgeries are constrained by the natural borders of the image. An example for this type of mask can be found in Figure 3.2(b).

The second mask is named *exomask*, which has no commonalities with any edges of the image. It is essentially a randomly selected *endomask* from another image. The only condition for this random selection of *endomask* from another image is that it has a similar area to the *endomask* of the image in question.This type of mask can be viewed in the example of Figure 3.2(c).

The authors of [15] emphasize the importance of considering that raw images inherently contain noise, and all pixels in the image are sampled from the same CFA pattern. To control the level of noise and eliminate the CFA pattern, the authors opt to downsample the image by a factor of 2. This downsampling allows for adding any desired noise and selecting any CFA pattern for mosaicing the image. Consequently, the image processed with the two different pipelines is the downsampled version.

Bammey *et al.* [15] use the RAISE dataset [28], especially the RAISE-1K containing 1000 raw images to create six datasets, each with a different pipeline inconsistency:

- Raw Noise Level dataset: The authors add random noise to both images and then process them with the same pipeline. The random noise variance follows a linear relation, $\sigma^2 = A + Bu$ where A and B are constants chosen randomly for each area (the authors bound A and B to ensure the images look natural and are not overly corrupted by noise), and u is the noiseless[2] image after down-sampling.

- CFA Grid dataset: This dataset changes the origin of the CFA grid inside the forged part of the image, this new pattern has a chance of 1/4 of being aligned with the main pattern. The rest of the pipeline remains the same for both areas.

- CFA Algorithm dataset: The dataset involves changing the demosaicing algorithm in the different pipelines and also randomly choosing a new grid pattern, this new pattern has a chance of 1/4 of being aligned with the main pattern. The rest of the pipeline remains the same for both areas.

- JPEG Grid dataset: It changes the compression grid origin and then randomly selects a compression quality factor, which is the same in both pipelines. In this case, the chance of getting an alignment between grids is 1/64.

- JPEG Quality dataset: This dataset randomly chooses two quality factors, one for each pipeline and for one of the images it also chooses a new grid pattern, this new grid has a chance of 1/64 of being aligned with the main grid. The rest of the pipeline remains the same for both areas.

- Hybrid dataset: Consists of introducing two differences between pipelines to account for the evaluation of methods that do not rely solely on one trace as a detection cue. The dataset combines noise, demosaicing, and/or JPEG compression.

[2]The image is called noiseless because it is the down-sampled version with reduced noise.

The authors provide the resulting dataset and the code to generate it. As previously stated, the authors use the RAISE-1K dataset [28] that contains 1000 images. However, to reduce the complexity of the analysis and given that this will not be the only dataset used for evaluation, it was decided to create a new dataset called *miniTrace* that uses 200 images from the full RAISE dataset.

One could simply sample 200 images from the original Trace dataset; however, two problems might arise. First, one criterion for sampling could be to get a subsample with the same mask size distribution as the original. In this case, the image from which the exomask was extracted might not be sampled in the new dataset. This is not a major problem because the original image from which the exomask is extracted is not relevant for the evaluation.

However, one might want to sample in a way that represents the distribution of the traces. This means that we maintain the same proportion of images with each camera pipeline. For example, in the noise datasets, this would mean that the new subsample has the same representation of noise levels as the original Trace. However, the sample satisfying this might not have the same representation of demosaicing algorithms, JPEG grid origins, or every other variant as the original Trace had. The problem lies in the almost impossibility of obtaining a sample that accurately represents all datasets. Adding this to the fact that the original code is available made us opt to generate the dataset ourselves.

The RAISE dataset comprises 8156 raw images that can simultaneously belong to multiple categories, such as outdoor, indoor, landscape, nature, people, objects, and buildings. For instance, an image can naturally fall into categories like outdoor, landscape, and nature concurrently. Maintaining the distribution of these categories is crucial to correctly sampling 200 images from the available 8156.

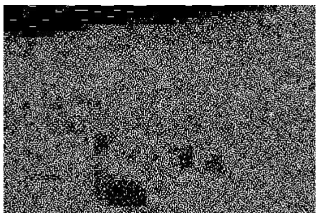
The challenge lies in achieving multilabel stratified sampling. The solution is found in the iterative stratification algorithm proposed by Sechidis *et al.* [75]. In each iteration, this greedy algorithm selects the label with fewer remaining examples, as those with more examples have more iterations to achieve a balanced distribution. For each example of a selected label, the algorithm determines the subset placement based on the following criteria: firstly, it chooses the subset that will need the greatest number of elements to represent the label accurately. In the case of a tie, it selects the subset that will require the highest total number of elements. If there is still a tie, the choice is made randomly. This process continues until all elements are assigned to one of the subsets. The chosen implementation for this algorithm is from [79].

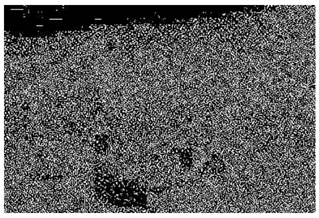
From the 200 sampled images, the code provided by Bammey *et al.* [15] was executed, providing a dataset for non-semantic evaluation.

3.2.8 Social networks versions

In a recently published paper by Wu *et al.* [89], some popular datasets were uploaded to social networks such as *Whatsapp* and *Facebook*, which allows researchers to see if the proposed methods are resistant to the new traces left by the social networks processing pipelines which are unknown to the public. Columbia, CASIA

(a) Original image


(b) *Whatsapp* Residual


(c) *Facebook* Residual

$$\begin{bmatrix} 6 & 4 & 4 & 6 & 9 & 11 & 12 & 16 \\ 4 & 5 & 5 & 6 & 8 & 10 & 12 & 12 \\ 4 & 5 & 5 & 6 & 10 & 12 & 12 & 12 \\ 6 & 6 & 6 & 11 & 12 & 12 & 12 & 12 \\ 9 & 8 & 10 & 12 & 12 & 12 & 12 & 12 \\ 11 & 10 & 12 & 12 & 12 & 12 & 12 & 12 \\ 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 16 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \end{bmatrix}$$

(d) Original Q-table

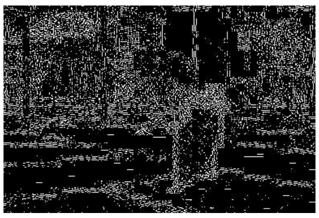
$$\begin{bmatrix} 3 & 2 & 2 & 3 & 4 & 6 & 8 & 10 \\ 2 & 2 & 2 & 3 & 4 & 9 & 10 & 9 \\ 2 & 2 & 3 & 4 & 6 & 9 & 11 & 9 \\ 2 & 3 & 4 & 5 & 8 & 14 & 13 & 10 \\ 3 & 4 & 6 & 9 & 11 & 17 & 16 & 12 \\ 4 & 6 & 9 & 10 & 13 & 17 & 18 & 15 \\ 8 & 10 & 12 & 14 & 16 & 19 & 19 & 16 \\ 12 & 15 & 15 & 16 & 18 & 16 & 16 & 16 \end{bmatrix}$$

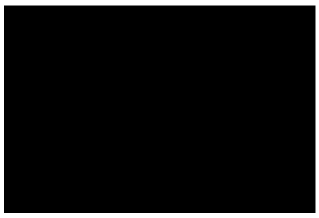
(e) *Whatsapp* Q-table

$$\begin{bmatrix} 3 & 2 & 2 & 3 & 4 & 7 & 9 & 11 \\ 2 & 2 & 3 & 3 & 5 & 10 & 11 & 10 \\ 3 & 2 & 3 & 4 & 7 & 10 & 12 & 10 \\ 3 & 3 & 4 & 5 & 9 & 16 & 14 & 11 \\ 3 & 4 & 7 & 10 & 12 & 20 & 19 & 14 \\ 4 & 6 & 10 & 12 & 15 & 19 & 20 & 17 \\ 9 & 12 & 14 & 16 & 19 & 22 & 22 & 18 \\ 13 & 17 & 17 & 18 & 20 & 18 & 19 & 18 \end{bmatrix}$$

(f) *Facebook* Q-table

(g) Original image


(h) *Whatsapp* Residual


(i) *Facebook* Residual

$$\begin{bmatrix} 6 & 4 & 4 & 6 & 9 & 11 & 12 & 16 \\ 4 & 5 & 5 & 6 & 8 & 10 & 12 & 12 \\ 4 & 5 & 5 & 6 & 10 & 12 & 12 & 12 \\ 6 & 6 & 6 & 11 & 12 & 12 & 12 & 12 \\ 9 & 8 & 10 & 12 & 12 & 12 & 12 & 12 \\ 11 & 10 & 12 & 12 & 12 & 12 & 12 & 12 \\ 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 16 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \end{bmatrix}$$

(j) Original Q-table

$$\begin{bmatrix} 3 & 2 & 2 & 3 & 4 & 6 & 8 & 10 \\ 2 & 2 & 2 & 3 & 4 & 9 & 10 & 9 \\ 2 & 2 & 3 & 4 & 6 & 9 & 11 & 9 \\ 2 & 3 & 4 & 5 & 8 & 14 & 13 & 10 \\ 3 & 4 & 6 & 9 & 11 & 17 & 16 & 12 \\ 4 & 6 & 9 & 10 & 13 & 17 & 18 & 15 \\ 8 & 10 & 12 & 14 & 16 & 19 & 19 & 16 \\ 12 & 15 & 15 & 16 & 18 & 16 & 16 & 16 \end{bmatrix}$$

(k) *Whatsapp* Q-table

$$\begin{bmatrix} 6 & 4 & 4 & 6 & 9 & 11 & 12 & 16 \\ 4 & 5 & 5 & 6 & 8 & 10 & 12 & 12 \\ 4 & 5 & 5 & 6 & 10 & 12 & 12 & 12 \\ 6 & 6 & 6 & 11 & 12 & 12 & 12 & 12 \\ 9 & 8 & 10 & 12 & 12 & 12 & 12 & 12 \\ 11 & 10 & 12 & 12 & 12 & 12 & 12 & 12 \\ 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 16 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \end{bmatrix}$$

(l) *Facebook* Q-table

Figure 3.3: Original images and the residual after uploading it to *Whatsapp* and *Facebook* and the corresponding quantization tables. Images extracted from the CASIA-V1 dataset [32].

1.0, and DSO-1 are the only datasets previously mentioned in this version. Evaluating the methods on these datasets is of high importance, given that with the correct metrics, it allows for the quantification of how well or poorly a method can generalize in-the-wild forgeries, especially in the context of the different processing an image undergoes when uploaded to any social network. From the two available social networks, we chose to report the results on the *Facebook* datasets; however, we did explore the difference between the original images and both their *Whatsapp* and *Facebook* versions.

For the *Whatsapp* case, we found out that for large images, the unknown pipeline applied by the social network must include resizing operations because some of the new images are smaller than the originals. We did not encounter this behavior on the *Facebook* images. We also computed the residual between the original image and the version through each social network, which are shown in Figures 3.3(b) and 3.3(c). The residuals are calculated as the absolute value of the difference between the original image and its corresponding social network version. While these residuals might look fairly similar, upon closer inspection, one can find some differences, as expected, because each social network has its own processing pipeline. Figures 3.3(d), 3.3(e) and 3.3(f) show the different JPEG quantization tables, the main difference between the original and the social networks version is the harsher compression that the latter employ for the higher frequencies with the *Facebook* version being harsher than the *Whatsapp* ones.

We also found that when images are lightweight enough, *Facebook* does not add any artifacts. This only occurs for images belonging to CASIA 1.0 [32], as shown in Figures 3.3(g), 3.3(h) and 3.3(i) which showcase an image that is modified when uploaded to *Whatsapp*, whereas it remains unchanged when uploaded to *Facebook*. Accordingly, the *Facebook* Q-table shown in Figure 3.3(l) remains the same as the original shown in Figure 3.3(j). The difference between the two examples is the weight of each original image. The first is 41.320 bytes, whereas the second is 20.915 bytes.

3.2.9 WebP datasets

One thing we noticed when looking for forgery datasets is the lack of datasets with WebP files, despite this being an increasingly popular format. Indeed, pushed by *Google*, it is intended to become the standard format for images on the web. That is why we decided to create two WebP datasets based on existing datasets. To do so, we used *OpenCV* `imwrite` function and saved the images with an 80 quality factor. This value was chosen as it is the standard value encountered when uploading an image to the web.

We chose uncompressed datasets because we wanted to avoid adding double compression given that, as explained in Chapter 2, WebP works similarly to JPEG. The goal motivating this kind of dataset is to see how robust methods are to this kind of compression. In order to achieve that, we used Columbia and Korus datasets. Columbia is a fairly easy dataset that poses a simple problem to the methods and allows us to assess the methods' performance before jumping into more challenging datasets. The comparison between the original and the WebP version of Columbia will allow us to see how robust the methods are to this type of compression. On the other hand, Korus is a challenging dataset in which many methods might achieve a poor performance, but in those that do a good job, we want to find out if these challenging forgeries can still be detected when the image underwent this compression and actually assess the methods performance over in-the-wild forgeries.

Dataset	Types of forgery	Nb. of images (forged + pristine)	Format	Social network version	WebP version
Columbia [45, 89]	Splicing	363 (180 + 183)	TIFF	✓	✓
COVERAGE [86]	Copy-move	200 (100 + 100)	TIFF	✗	✗
DSO-1 [29, 89]	Splicing	200 (100 + 100)	PNG	✓	✗
Korus [50, 51]	Splicing, copy-move object removal	440 (220 + 220)	TIFF	✗	✓
CASIA 1.0 [32, 64, 89]	Splicing, copy-move	1023 (923 + 100)	JPEG	✓	✗
AutoSplice [49]	Generative inpainting	5894 (3621 + 2273)	JPEG	✗	✗
miniTrace [15]	Alterations to acquisition pipeline	2400 (2400 + 0)	PNG	✗	✗

Table 3.1: Summary of the main characteristics of the datasets, such as the type of forgery they feature, the number of images (both pristine and forged) included in each of them, the images' format, and whether their social network and WebP versions are also incorporated.

3.2.10 Summary of datasets

All of the previously presented datasets are summarized in Table 3.1. This table shows the number of images each dataset has, how many of those are forged and how many are pristine, and indicates whether the dataset has a social network version or a WebP version.

In this chapter, we presented the importance of data and the generalization problem that might arise in data-driven approaches. We also presented the chosen datasets used in this work, six of which we will refer to as popular datasets, and the seventh, miniTrace, will allow us to do non-semantic evaluation. Additionally, two possible variants for the popular datasets were shown. First, the social network versions, for which we analyzed the impact of unknown processing pipelines used by various social networks on an image. Second, we introduced the WebP versions of two datasets, which will enable us to evaluate the performance of methods on this increasingly popular image format.

The next chapter will introduce another fundamental part of this thesis: the metrics that will allow us to evaluate different methods on the chosen datasets.

This page intentionally left blank.

Chapter 4

Measuring and qualifying the predictions: the metrics

"They can only see the mere show, and never can tell what it really means."

Sherlock Holmes

The previous chapters introduced the image forgery detection problem and the datasets that can be used to tackle the problem. However, one fundamental ingredient is missing: how do we decide which method is good, and how do we compare them? This is the vital puzzle piece that is described in this chapter. To do so, the importance of good metrics will be discussed first, followed by global definitions and discussions on how to aggregate metrics. The chapter will end with a rigorous report of all the metrics used in this work.

4.1 What is a metric?

Mathematically speaking, a metric is a function $m : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, that maps $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^n \times \mathbb{R}^n$ to a real value $m(\mathbf{x}, \mathbf{y})$, which captures some relationship between $\mathbf{x}$ and $\mathbf{y}$. In the context of evaluation, $\mathbf{x}$ represents the predicted output obtained by a model, and $\mathbf{y}$ represents the labeled target, commonly referred to as ground truth. The use of metrics during the evaluation phase comes from the need to have a quantifiable measure of how a method is performing. Having quantified performance allows for comparing a method's configurations and different methods.

Having said this, metrics often emphasize certain dimensions of performance while overlooking others. With this in mind, one should select one or more metrics that address all aspects of the problem to be evaluated.

4.2 The importance of good metrics

Metrics are the backbone of any scientific evaluation, providing a quantifiable amount against which to compare and benchmark solutions. Paired with the data used, the metrics can offer crucial insights into the strengths and weaknesses of the evaluated subjects. Forgery detection is no different, yet there is no consensus or a common benchmark that all researchers use when publishing. Unfortunately, in many cases, the choice is made to favor a particular method.

Let us give a quick example of how choosing a good metric matters. Suppose we have a dataset of wheel images where 95% of them depict normal wheels and the other 5% flat wheels, and we want to build a classifier that predicts whether a wheel is flat or not. Imagine we build a classifier that predicts all wheels as being normal. Even if it is not working properly, its percentage of correct classifications (commonly called accuracy) is 95%, which is a really high value. In this case, accuracy is not a good metric for evaluating the classifier because it does not capture the fact that the classifier does not detect any flat wheel. If we had used a metric that captures the number of flat wheels detected, we would have noticed that the classifier is not working as expected.

In the case of image forgery detection, we aim to classify whether something is tampered or pristine. Going back to the two problems presented in Section 2.2.4, in the case of the localization problem, we classify whether a pixel is tampered or not; when detecting, we classify the whole image as tampered or pristine.

4.2.1 Characteristics of Good Metrics

For a metric to be effective in its purpose, it should fulfill the following requirements:

- **Relevance:** The metric should aim to evaluate the core objective of the problem.
- **Consistency:** The metric should yield identical results under identical conditions.
- **Sensitivity:** The metric should detect small variations in the prediction with respect to the ground truth. In the case of images, a one-pixel difference should be enough for the metric to pick it up.
- **Scalability:** As the amount of data grows, the metric must maintain its effectiveness, ensuring that it remains relevant in a variety of scenarios.
- **Interpretability:** The value of a metric lies not only in its calculation but also in its communication. It must be easy to understand what the metric is evaluating and the implications of the measurement.

Returning to our previous example, let us define our core objective of detecting flat tires. While accuracy is relevant to our problem, it lacks scalability and sensitivity, especially with a very imbalanced dataset. Missing a flat tire does not impact the metric in a meaningful way when our dataset mainly consists of normal tires. In the case of image forgeries, some of the problems to look out for are the relative size of forgeries with respect to the image size when evaluating localization and the amount of pristine and tampered images in the dataset when evaluating detection.

4.2.2 Consequences of Poor Metrics

As illustrated in the example, a poor selection of evaluation metrics can lead to a false sense of good performance, masking problems or not evaluating what is desired. In the case of image forensics, many metrics fail to capture the localization quality of a prediction if the forged area is small. Suppose we gave an image in which 10% of its pixels are tampered. If we take, for instance, the accuracy of a mask that predicted all pixels as untampered, then we would have a method with a 90% accuracy. This is a fairly common problem since forgeries are usually small compared to the whole image.

Another common error is selecting a limited number of metrics to evaluate the results or metrics that only evaluate one aspect of the problem. When developing a method, one tends to overfit the selected metrics as they measure performance and serve as comparisons to other methods. In turn, if our metric selection is poor, the end result is unlikely to generalize to other metrics or real-world scenarios.

4.3 Global definitions

Before defining the metrics used in this thesis, we present some general notions and details about how the metrics are calculated.

4.3.1 True and False, Positives and Negatives

In the problem of data classification, many metrics are based on the definitions of True and False Positives, or True and False Negatives [68]. A single classification can be considered Positive or Negative, where the meaning of “positive” is arbitrarily assigned in a given problem. In the case of image forensics, a forged image (or a forged patch or pixel when evaluating localization) is labeled as positive, and untampered data is considered negative. On the other hand, the classification is True or False according to whether or not the prediction matches the ground truth, respectively. This concept can be better understood with the diagram shown in Figure 4.1.

Using these simple definitions, we can calculate all the metrics we will present later on in the chapter, often creating an easier way to understand the metric.

		Predicted class	
		Tampered	*Untampered*
Actual class	*Tampered*	True positive	False negative
	Unpatmpered	False positive	True negative

Figure 4.1: Confusion matrix: rows represent the actual classes while columns represent the prediction. The matrix has four possible values corresponding to the four possible combinations of predicted and actual classes.

4.3.2 Aggregating metrics

When evaluating a method on a dataset, we need to define how we aggregate the results into a single value that reflects the performance of the method in the chosen dataset. There are many ways to aggregate results that preserve different aspects of a data distribution, yet we will only work with two of them.

The first and simplest way to aggregate results is to store every prediction and calculate the metric over all data points, whether pixels or patches, when evaluating localization or the whole image if we are interested in detection. For example, let us assume for a moment that we want to evaluate a metric defined as $\frac{TP}{TN}$. If we were to aggregate the metric in this way, we would accumulate the TP and TN for every image of the dataset. Then, with the accumulated values, we would compute the metric using the aforementioned definition in order to obtain a single number that summarizes the method's performance. We will refer to this way of aggregating the metrics as *dataset-level* score. This aggregation technique is used by *Torchmetrics* [31] in most metrics in their module. In the case of localization, this aggregation type gives us a pixel-wise performance since the metrics are calculated over all the pixel predictions of the method under evaluation.

The second approach we consider is the average performance over the dataset. To calculate this, we compute the metric for each image and store the result. Then, we compute the average performance on the dataset. So, following the same example as before, in this case, we would calculate the $\frac{TP}{TN}$ for each evaluated image and accumulate said values to later divide them over the number of images in the dataset to obtain a single number that summarizes the method's performance. We will refer to this way of aggregating the metric as the average *image-level* score. In the case of detection, where the method's output is a single value, this aggregation is not possible for most metrics and is not useful either. However, in the case of localization, this aggregation captures a method's image-wise performance, that is, how well we can expect a method to work on a single image.

A natural question to ask is why make the distinction between the two if, intuitively, the results should be equal or extremely similar. To see that this is not the case, let us look at the following example: two images, one of size 4096x4096 and another of size 10x10. Also, suppose a method works perfectly on the large image but poorly on the small one. If we aggregate the results using the mean, the final result will land somewhere between the performance of both images. However, if we accumulate all the pixel results and then calculate the metric, the number of correctly labeled pixels will be far larger than the number of incorrectly labeled pixels. Then, the final result will be close to the perfect performance of the large image.

The example above is a bit extreme, but it illustrates that how we aggregate the results can have a big impact on the final score.

4.4 Metrics

This section introduces some of the classic metrics used in image forensics. These metrics will be later considered in Chapter 7 to benchmark state-of-the-art methods.

4.4.1 Recall (rec) or True Positive Rate (TPR)

It is the ratio between the True Positives and all the positive-labeled data. Following a probabilistic interpretation, this would be the probability that a Positive labeled data be predicted as such,

$$rec = TPR = \frac{TP}{P} = \frac{TP}{TP + FN}. \tag{4.1}$$

4.4.2 False Positive Rate (FPR)

It is the ratio between the False Positives and all the Negative labeled data,

$$FPR = \frac{FP}{N} = \frac{FP}{TN + FP}. \tag{4.2}$$

4.4.3 Precision ($prec$)

It is the ratio between the True Positives and all the predicted Positive, reflecting the probability that data predicted as forged is actually forged,

$$prec = \frac{TP}{TP + FP}. \tag{4.3}$$

4.4.4 F1 score

It is frequently useful to combine precision and recall into a single metric. The most common way of doing so is by taking their harmonic mean. This metric is known as the F1 score. Unlike the conventional mean, which assigns equal importance to all values, the harmonic mean heavily emphasizes lower values [43]. This metric is calculated as:

$$F_1 = \frac{2}{\frac{1}{prec} + \frac{1}{rec}} = 2 \times \frac{prec \times rec}{prec + rec}. \tag{4.4}$$

It is also common to express metrics in terms of the confusion matrix, as follows:

$$F_1 = \frac{2TP}{2TP + FN + FP}. \tag{4.5}$$

4.4.5 Matthews correlation coefficient (MCC)

The MCC is a metric that measures the quality of a binary prediction, which can be used both for the classification task and for the localization task [60], which is defined by:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}. \tag{4.6}$$

This definition shows that the MCC goes from -1 (worst) to 1 (best). The MCC is often regarded as a measure of the quality of a confusion matrix. The difference between this metric and precision or recall is that the MCC takes into account both true and false, positive and negative rates, with some authors [21] crowning it as the best binary classification metric. Another important characteristic of the MCC is that it is a correlation, meaning it does not matter which class is defined as positive or negative.

4.4.6 Intersection over Union (IoU)

Intersection over Union, also known as the Jaccard Index, is a metric that measures the quality of a spatial prediction.This metric allows us to measure the localization quality of our method. If A denotes the ground truth region and B the predicted forged region, IoU is defined as

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}. \tag{4.7}$$

The IoU can also be written in terms of the confusion matrix:

$$IoU = \frac{TP}{TP + FP + FN}. \tag{4.8}$$

4.4.7 Reciever Operating Characteristic curve (ROC)

The receiver operating characteristic (ROC) curve is a common metric used in binary classification. It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the TPR against the FPR [43].

As explained above, prediction often occurs by outputting a probability or score value, which is compared to a threshold to determine the predicted binary

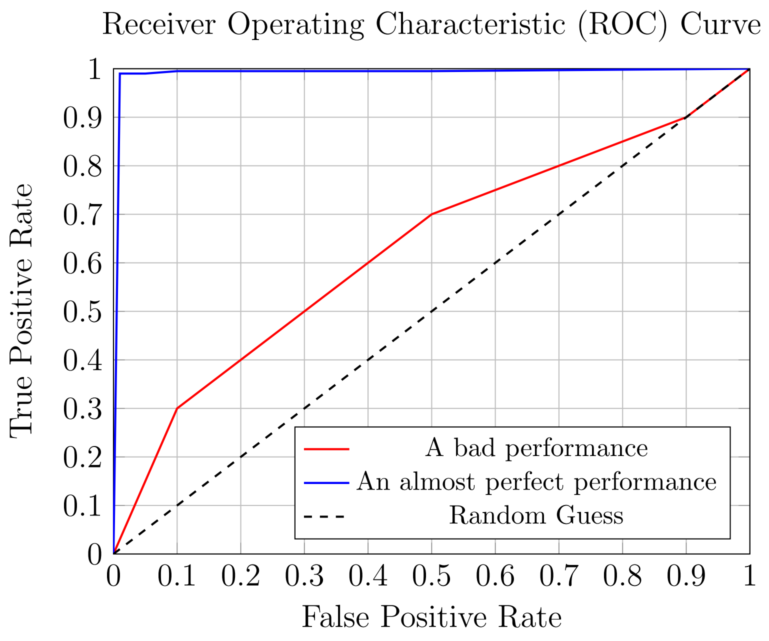


Figure 4.2: Example of ROC curve showing a random classifier, a classifier achieving a barely acceptable performance, and a classifier achieving an almost perfect performance.

class. Different threshold values serve different purposes, as they can lead to over-prediction when too low or missing detections when set too high. Varying this threshold can plot a set of operation points (a TPR and FPR tuple). Ideally, this curve should be a square, meaning that when $TPR \approx 1$ for every value of FPR except at $FPR = 0$, where it two should be zero. The benefit of this metric (or others similar) is that it contemplates the variety of operation points of score or probability outputting methods, allowing for a fair comparison.

Figure 4.2 shows an example of a ROC curve.

4.4.8 Area Under the ROC curve (AUROC)

One way to compare classifiers numerically using the ROC curve is to measure the area under the curve, commonly referred to as AUROC (Area Under the ROC curve). A perfect classifier will have an AUROC equal to 1, whereas a purely random classifier will have an AUROC equal to 0.5 [43].

4.4.9 Weighted Metrics

Many of the aforementioned metrics assume that the prediction is a binary result; however, as previously mentioned, some methods output a heatmap instead of a binary output. For instance, metrics like F1 score, MCC score, and IoU are designed for binary outputs. One approach to handle heatmaps is to binarize them using a threshold, often set to 0.5 by default. Alternatively, an optimal threshold can be determined to maximize a specific metric on a chosen dataset.

Some authors present another solution by proposing the definition of weighted metrics. This allows for comparing methods that output heatmaps and those that output binary outputs. Gardella *et al.* [38] and Bammey *et al.* [15], by interpreting the heatmap at each pixel as the probability of the pixel being forged, define weighted true positives (4.9a), weighted false positives (4.9b), weighted true negatives (4.9c), and weighted false negatives (4.9d):

$$TP_w = \sum_x H(x)M(x), \tag{4.9a}$$

$$FP_w = \sum_x H(x)(1 - M(x)), \tag{4.9b}$$

$$TN_w = \sum_x (1 - H(x))(1 - M(x)), \tag{4.9c}$$

$$FN_w = \sum_x (1 - H(x))M(x). \tag{4.9d}$$

Here, $H(x)$ is the heatmap value (ranging from 0 to 1) and $M(x)$ is the binary ground truth mask.

With this definition, the F1 score, the MCC score, and the IoU defined in the previous subsections have their weighted version.

These metrics can be implemented as either dataset or average image-level scores. Of course, they can be used for either localization or detection. The only thing to keep in mind is that given the reasons outlined above, the dataset-level version should be used for detection, whereas for localization, the average image-level is preferred. However, both yield reasonable results.

For the detection case, the weighted metrics serve to compare methods with binary classification outputs to detection scores. To this end, the definition given in 4.9 is modified in the sense that the prediction and the target are both single numbers, so the weighted scores correspond to weighted TP, TN, FP, FN according to the detection score given by the method.

In this chapter, we have discussed the importance of metrics in the evaluation of forgery detection methods. We have seen the importance of selecting good metrics and the consequences of selecting bad ones. We presented some global definitions used later in the chapter and two ways of aggregating metrics. Lastly, we presented the most common metrics used to evaluate forgery detection methods.

In the following chapter, the key element missing in terms of forgery detection will be described, namely the methods. We will discuss different forgery detection methods, from those that marked the beginning of forgery detection to the most recent and advanced ones, from methods that aim to detect forgeries by exploiting *different types of traces with a classical signal processing approach to those relying on machine learning techniques to detect forgeries.*

This page intentionally left blank.

Chapter 5

Putting the traces together: the methods

"I see no more than you, but I have trained myself to notice what I see."

Sherlock Holmes

In the previous chapters, we introduced the context of the problem to be solved and discussed the selection of data and metrics, showing the importance this has in the evaluation process. In the present chapter, we will analyze the core of the field of forgery detection: the methods. This section is structured in the following way: for each method selected, there is a brief summary of its procedure, followed by a more detailed description, followed by the target forgery trace or type the method searches for.

In recent years, the growing field of image forensics has proposed a myriad of methods trying to solve the problem of detecting forgeries in digital images. This wide array of methods comprises solutions that target specific types of forgeries or exploit a specific type of trace inconsistency left in forged images, such as noise, CFA, and JPEG. There are also versatile methods, often using deep learning, designed to detect a wider range of forgeries by simultaneously examining multiple traces.

In this chapter, ten of these methods are discussed, each of which has been chosen on the basis of its performance, uniqueness, and complementary to the other methods on the list. These methods will be the ones implemented in the *PhotoHolmes* library. For each of these methods, we explain how they work and analyze which type of forgery they target. In the case of deep learning methods, we will list the datasets used during training and how that might introduce biases in our evaluation later on.

5.1 Splicebuster

The Splicebuster method [27] aims to extract a set of blockwise features that capture camera-processing traces. Using these features and statistical analysis tools, the method computes two clusters, grouping tampered pixels in one cluster and pristine ones in the other. The assumption is that, under forgery, an image's features will follow different distributions in the tampered and pristine regions. Finally, the method delivers a heatmap pointing to the most suspicious zones of the image.

This is a model-driven approach and thus offers some advantages. Namely, its results are interpretable and not data-dependent. Interpretability can be of interest in some applications, and data independence relieves the possibility of biases in the method (although it also limits its performance). These are some of the reasons why it was selected for the *PhotoHolmes* project.

5.1.1 Method

The first stage of this method consists of extracting the image's features from its high-frequency components. The image's residual is obtained by high-pass filtering. More specifically, the residual is computed by estimating the third-order derivatives in both the horizontal and vertical directions, obtaining two residuals r^h and r^v, which are then quantized in three levels. These quantized residuals, which we denote as $\hat{r}^h$ and $\hat{r}^v$, respectively, are then subject to the following post-processing.

For both $\hat{r}^h$ and $\hat{r}^v$, the method computes the local co-occurrences of four pixels along the horizontal and vertical directions. These co-occurrences can be represented as four 4-dimensional tensors:

$$\begin{cases} C_{h,h}(k_0,k_1,k_2,k_3) := \sum_{i,j} \mathbb{1}(\hat{r}^h_{i,j}=k_0, \hat{r}^h_{i,j+1}=k_1, \hat{r}^h_{i,j+2}=k_2, \hat{r}^h_{i,j+3}=k_3) \\ C_{h,v}(k_0,k_1,k_2,k_3) := \sum_{i,j} \mathbb{1}(\hat{r}^h_{i,j}=k_0, \hat{r}^h_{i+1,j}=k_1, \hat{r}^h_{i+2,j}=k_2, \hat{r}^h_{i+3,j}=k_3) \\ C_{v,h}(k_0,k_1,k_2,k_3) := \sum_{i,j} \mathbb{1}(\hat{r}^v_{i,j}=k_0, \hat{r}^v_{i,j+1}=k_1, \hat{r}^v_{i,j+2}=k_2, \hat{r}^v_{i,j+3}=k_3) \\ C_{v,v}(k_0,k_1,k_2,k_3) := \sum_{i,j} \mathbb{1}(\hat{r}^v_{i,j}=k_0, \hat{r}^v_{i+1,j}=k_1, \hat{r}^v_{i+2,j}=k_2, \hat{r}^v_{i+3,j}=k_3). \end{cases} \tag{5.1}$$

With these four flattened tensors, a single vector concatenating $C_{h,v} + C_{v,h}$ with $C_{h,h} + C_{v,v}$ is created. A PCA is then applied to reduce its dimensionality.

The process previously described is applied to 128×128 overlapping blocks, with stride 4 if the image size is below a specified threshold and stride 8 if above. This way, we obtain a flat vector for every block.

The resulting feature map is used to fit a two-class mixture model using an Expectation-Maximization (EM) algorithm. Two types of mixtures can be considered: Gaussian-uniform or Gaussian-Gaussian. The concept behind this is that the forged region can have either a uniform or a Gaussian distribution, depending on the size of the forgery applied (small vs. relatively large, respectively), while the pristine region is expected to have a Gaussian distribution. According to the Splicebuster authors, modeling small forgery traces by a uniform distribution contemplates the fact that intra-class variability is mostly accounted for by the image content.

Once the parameters of the mixture models have been computed, a heatmap is built. For the Gaussian-uniform mixture model, the heatmap is obtained as the Mahalanobis distance of the block's feature histogram and the Gaussian distribution. In the case of a Gaussian-Gaussian mixture model, the heatmap is the quotient of both Mahalanobis distances. The heatmap is later normalized to take values from 0 to 1.

A pseudo-code of the method is presented in Algorithm 1, and the method's output on an example is shown in Figure 5.1.

Algorithm 1 Splicebuster pseudo-code

```
Input: $Image$ (in grayscale), $stride$
Output: $heatmap$
  $blocks \leftarrow tiled\_image(Image, block\_size = 128, stride = stride)$
  $features \leftarrow$ empty list
  for $b$ in $blocks$ do
      $r^h, r^v \leftarrow \textbf{residual}(b)$
      $\hat{r}^h, \hat{r}^v \leftarrow \textbf{quantize}(r^h, r^v, threshold)$
      $C_{h,h}, C_{h,v}, C_{v,h}, C_{v,v} \leftarrow \textbf{co-occurrence}(\hat{r}^h, \hat{r}^v)$
      $feat\_reduced \leftarrow \textbf{PCA}((C_{h,h} + C_{v,v}, C_{v,h} + C_{h,v}))$
      $features.append(feat\_reduced)$
  end for
  $\mu, \sigma, \pi \leftarrow \textbf{expectation\_maximization}(features)$
  $heatmap \leftarrow \textbf{mahalanobis\_distance}(feat\_reduced, \mu, \sigma, \pi)$
```

5.1.2 Target forgeries

The EM algorithm is rooted in the assumption that the method has a forgery. This implies that the base model is built to localize potential forgeries but not for forgery detection.

The feature extraction begins from the co-occurrences of the image's residual, as the authors had found it convenient for image forensics. The residuals can track traces of many sorts, so this method is not targeted to any trace in particular. Additionally, as this method has no training nor parameter-tuning of any sort, it is not biased toward a particular dataset.

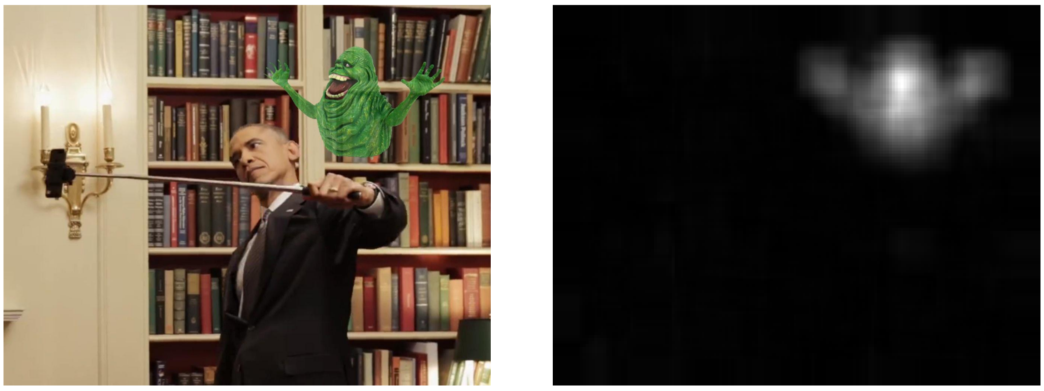

(a) Original image (b) Predicted heatmap

Figure 5.1: Result of running Splicebuster on an image. The original image (a) was extracted from [27], and the predicted mask (b) is the result of our implementation of the method.

5.2 Noisesniffer

The Noisesniffer method described by Gardella *et al.* [38] exploits the effects that the acquisition pipeline described in Chapter 2 has on the noise model of a digital image by inspecting whether or not the noise model is consistent across the image.

Many methods that apply this principle while searching for noise-related traces follow the same procedure: They find blocks useful for noise estimation, make a local estimation of the noise level, and search for inconsistencies in this domain. The issue with these methods is that the noise level estimation might be unreliable, so the consequent anomaly detection is inaccurate.

In contrast, Noisesniffer differs in the last step: it finds blocks that are optimal for noise estimation and computes the noise variance within these blocks, but then the consistency is expected in the *spatial distribution* of the selected blocks. By doing so, this method relieves the burden of having to model the noise levels with such accuracy. Anomalies are detected following the *a contrario* theory of Desolneux, Moisan, and Morel [30], which allows forgery detections to be based on sufficient statistical evidence.

5.2.1 Method

The noise level is best estimated in homogeneous regions where noise dominates over the signal. The overall procedure is to sort the blocks in a homogenous region by their variance to distinguish the lower m-percentile. If this class has a suspicious concentration within the homogeneous regions, it is likely produced by a difference in the noise model and, hence, introduced by forgery.

To do so, the input image is divided into overlapping blocks of size $N \times N$. Blocks that contain saturated pixels are discarded to avoid unreliable noise estimations. Then, for each channel, blocks are grouped in bins of a fixed size according to their mean intensity. As explained in Chapter 2, the noise model is signal-dependent, so it is not appropriate to compare noise levels for different intensities.

For a given channel and intensity bin, the following steps are performed: first, the DCT type II (described in Section 2.1.4.2) is computed for each block; then, the variance in low and medium frequencies is calculated for each block. In each bin, blocks are arranged in ascending order according to the variance computed in the previous step, and only a percentile n of them is kept. The blocks kept in that step are the most homogeneous ones.

After that, the variances of each of those blocks are computed, and the blocks are ordered in ascending order, from which only the ones with the $m\%$ lowest variance are kept. If more than $m\%$ of blocks have zero variance, the bin is declared invalid. Ultimately, this part of the method obtains the group of most homogeneous blocks, L, and a subset V conformed by the blocks within L having the lowest variance.

In the absence of tampering, the spatial distribution of blocks of V and L should be the same, and this is adopted as the *null hypothesis* H_0. However, some fluctuations in that spatial distribution are bound to happen due to randomness, so the question is whether the spatial distribution of blocks can be observed by chance or not. If the answer is the latter, that region could have been tampered with. Here is where the *a contrario* approach comes into play, where the selection of the subset V of L is modeled as uniform, and thus, a Number of False Alarms (NFA) can be computed for every region.

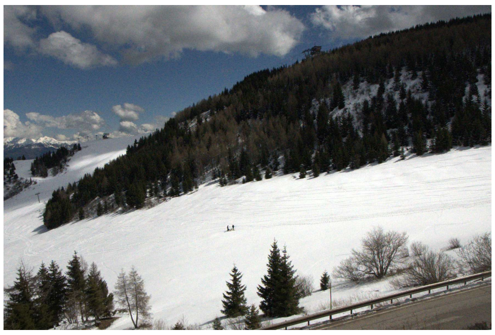
A region-growing algorithm is used to establish which regions are forgeries. This algorithm starts from a square tessellation of the image. Firstly, the authors describe a criterion in order to determine which cells are most significant (or suspicious), namely that the proportion of cells inside of V with respect to L is superior to m (which is suspicious with respect to a uniform Binomial model). From those cells, the algorithm iteratively adds contiguous cells that make the region more meaningful in the sense of the NFA.

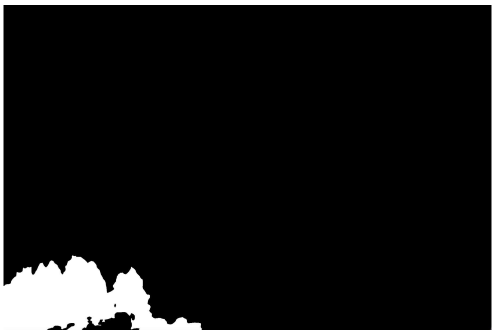
This algorithm results in a mask of the fully-grown regions that indicate forgery. An example of the method's output is shown in Figure 5.2, along with a representation of the spatial distribution of the L and V blocks. A high-level description of the method is in Algorithm 2.

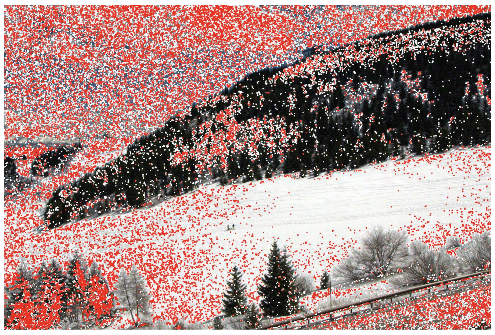
5.2.2 Target forgeries

This method is responsive to noise-related traces. Some inherent limitations are that textured regions can be misinterpreted as noise or that JPEG compression and image downsampling can reduce the noise-based traces and inhibit detection.

An assumption made in this method is that forged regions have a suspiciously *low* variance with respect to the remainder of the image. This means forgeries with a higher variance than the original image are undetected, or in some cases (if the forgery region is large enough), it can lead to a false attribution (the complementary mask) as a prediction. Other limitations were also pointed out in the original paper. A small forgery region or saturated forgery region can make a forgery pass


(a) Forged image


(b) Ground truth


(c) Distributions

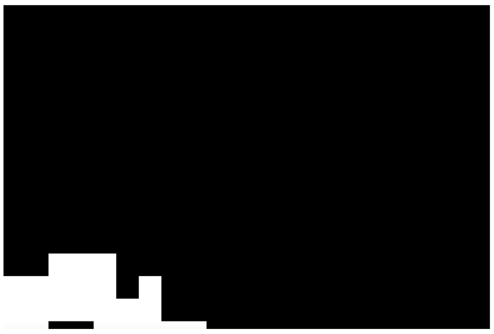

(d) Predicted mask

Figure 5.2: The method performance on a forged image. Along with the original image (a) and the ground truth mask (b), the distributions (c) with the output predicted mask (d) are shown. The distributions (c) are illustrated by painting the blocks in L in white, and on top of these, the blocks in V are painted in red. Images extracted from [38].

under the radar. Alternatively, a homogeneous forgery amongst a textured image can produce a false detection.

The main reason this method is selected for the project is that it is a classical noise-based model. This means it covers noise-based traces and has an interpretable pipeline that can justify detection (or, in some cases, even explain false detection).

5.3 DQ

Double quantization methods identify forgery through inconsistent traces left by double JPEG compression. As described previously in Chapter 2, the JPEG algorithm quantizes each DCT coefficient in a particular scale. Therefore, double JPEG compression implies a double quantization in the frequency domain. Double quantization of any signal in any context has the peculiarity to induce periodic peaks in the signal distribution, as long as the quantizations are done with different scales [66]. In the JPEG compression context, this effect can occur over the DCT coefficient distribution when an image undergoes double JPEG compression

Algorithm 2 Noisesniffer pseudo-code

```
Input: $Image$
Output: $mask$
  function compute_blocks($I_{blocks}$)
      for $c$ in $channels$ do
          for $b$ in $intensity\_bins$ do
              $coefs \leftarrow \mathbf{DCT\_2D}(I_{blocks}[b, c])$
              $\sigma^2_{LF} \leftarrow \mathbf{sum}((coefs[low\_frequencies])^2)$
              $L[c, b] \leftarrow \mathbf{sort\_ascending}(coefs, sort\_by = \sigma^2_{LF})$[low indices]
              $\sigma^2_L \leftarrow \mathbf{var}(L[c, b])$
              $V[c, b] \leftarrow \mathbf{mth\_percentile}(L[c, b], sort\_by = \sigma^2_L)$
          end for
      end for
      return $V$, $L$
  end function

  $I_{blocks} \leftarrow \mathbf{tiled\_image}(Image)$
  $V, L \leftarrow \mathbf{compute\_blocks}(I_{blocks})$
  $mask \leftarrow \mathbf{region\_growing}(I_{blocks}, V, L)$
```

with different quality factors, as these are associated with different quantization scales in the DCT coefficients domain.

A forged image could be created following this procedure: the original image is in JPEG format, decompressed to have some sort of forgery applied to it, and then re-compressed for distribution. Should the forgery be some sort of inpainting or splicing, there is a good chance the forged area does not have the same compression history as the rest of the image. For instance, the re-compression could have a different quality than the original image, resulting in the double quantization effect. If this effect is present but is not uniform across the image, it is a sign of tampering.

Lin *et al.* [56] propose a classical image processing method for detecting forgery through this procedure. It is based on the observation that double JPEG compression with different JPEG qualities will result in a double quantization with different quantization steps. In [66], it is shown that the double quantization of a signal produces periodic peaks, as illustrated in Figure 5.3. This effect could be useful for detecting double JPEG compression, which could then reveal tampering if it is inconsistent. Furthermore, the quantization of the JPEG algorithm occurs at the DCT coefficient level, meaning this method can work on a JPEG image without decompressing it. Figure 5.4 shows an example of how a specific channel's histogram of the DCT coefficient can show a double-quantization effect.

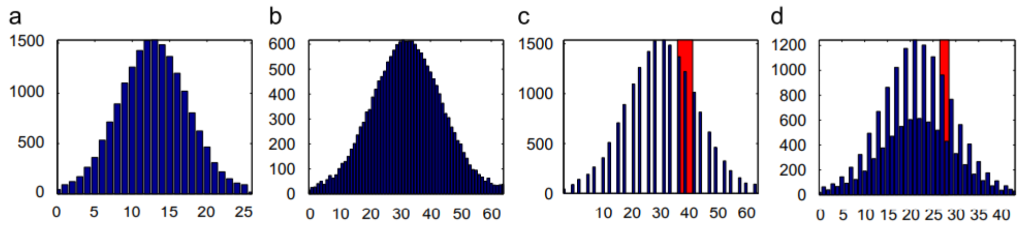


Figure 5.3: An illustration of the double-quantization effect. The left two figures are histograms of single quantized signals with steps 5 (a) and 2 (b), respectively. The two right figures are histograms of double quantized signals with steps 5 followed by 2 (c) and 2 followed by 3 (d). Note the periodic artifacts in the histograms of double-quantized signals. The shaded rectangles show one period of the histograms. Figure extracted from [56].

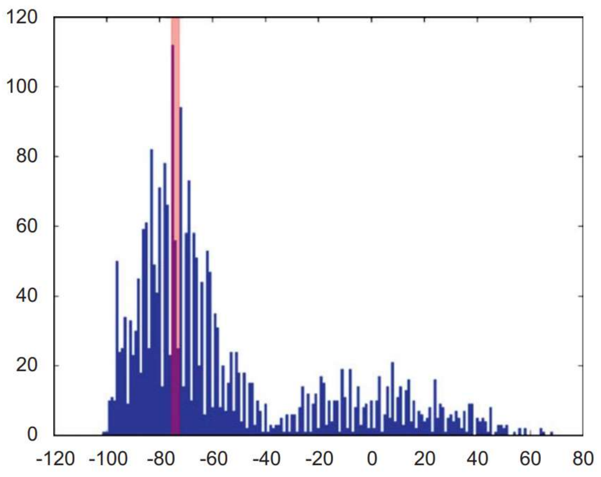


Figure 5.4: A typical DCT coefficient histogram of a tampered JPEG image. The shaded rectangle shows one estimated period of the histogram. Figure extracted from [56].

5.3.1 Method

The method operates on the DCT coefficients of the JPEG image. If the image is not in JPEG format, a workaround is to convert it to JPEG with quality 100.

The first step is to build a histogram of the DCT coefficients of the blocks across the image for each of the 64 DCT frequency components and for each YUV color channel, which leads to $64 \times 3 = 192$ histograms. Note that in this implementation, histograms have bin size 1, so each bin counts the occurrences of each of the DCT coefficients. This method exploits each of these histograms by producing a probability map for each block being tampered.

To achieve this, two distribution models are provided, one for tampered blocks that undergo single quantization and one for untampered blocks that are untampered and double quantized. A Naive Bayes approach permits deriving from these a tampered probability function for each value of the histogram bins. Finally, this last function can be interpreted as a probability map for the image's blocks, namely, the probability of each block being tampered given its DCT coefficient value (at a certain frequency and channel). An overall illustration of this procedure can be seen in Figure 5.5. The maps of each channel and of a set of frequencies are then aggregated to produce a final heatmap.

We shall describe this procedure in further detail. The probability maps, called Block Posterior Probability Maps (or $BPPMs$), correspond one to one to the histograms obtained, meaning there is a probability map for each frequency bin of the 64 used in the DCT, and also for each of the three color channels. The main logic behind constructing this BPPM from the histogram h is as follows.

First, since the histogram showcases periodic peaks, with a period we denote as p, we estimate the period p using different techniques we will not delve into. Next, we consider the peak period surrounding c_{DCT}, meaning $c_{DCT} \in [s_k, s_k + p)$ with $h(s_k)$ being a peak value. This allows for the definition of the probabilities

$$\begin{cases} P_u(c_{DCT}) := P(c_{DCT}|untampered, T_k) = \frac{h(c_{DCT})}{\sum_{i=0}^{p-1} h(s_k+i)} \\ P_t(c_{DCT}) := P(c_{DCT}|tampered, T_k) = \frac{1}{p}, \end{cases} \tag{5.2}$$

where T_k is the event "$c_{DCT} \in [s_k, s_k + p)$". These formulas model the probability mass functions conditioned to T_k, for the case of tampered and untampered blocks. Once again, the hypothesis is that tampered blocks suffer single JPEG compression whilst untampered blocks undergo double JPEG compression. An interpretation behind these estimations is that single JPEG compressed blocks have uniform distribution (as there was a single quantization of the DCTs), while double JPEG compressed blocks will have a higher chance to take peak values in the histogram (a consequence of double quantization).

The authors claim that a Bayesian approach can derive the probability map for each frequency and channel, earlier referred to as $BPPM$, defined as

$$BPPM = \frac{P_t}{P_t + P_u}. \tag{5.3}$$

This derivation is not presented in the original work [56], but applying Bayes theorem, it is easy to see that for (5.3) to hold, it is necessary that the prior probability of a block being tampered is 1/2. While fixing such a value is not evident, in our opinion, this choice seems to be too high.

By averaging the histogram's BPPMs over the frequencies and channels, a heatmap for the entire image is produced, with values ranging from 0 to 1. It is important to point out that when averaging, the high-frequency range is not considered since it may degrade the estimates. Indeed, higher frequencies are too heavily quantized and exhibit a dynamic range that is too limited to present double-quantization artifacts.

the authors' implementation, besides the heatmap, includes a predicted mask for forgeries. This mask is obtained using an SVM trained on a set of features extracted from the heatmap, which include an optimal threshold, class variances, and tampered-class connectivity. We did not include this mask prediction stage in *PhotoHolmes*' implementation of the method since the training dataset and procedure are not provided.

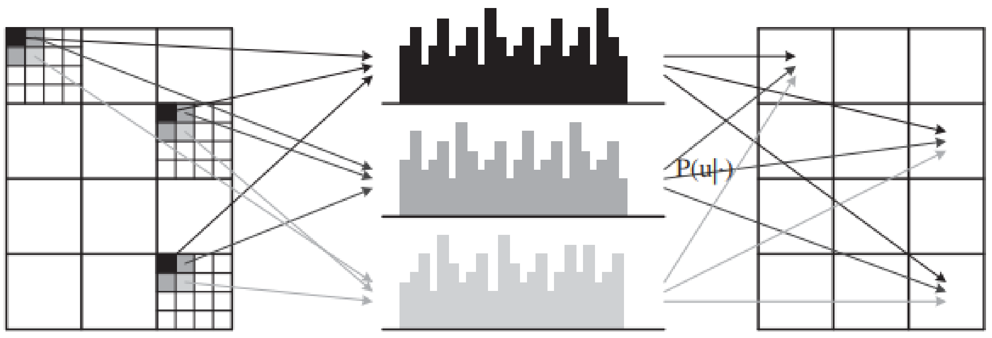


Figure 5.5: How the DCT coefficient histograms are built (for each YUV channel) and how the histograms vote for a Block Posterior Probability Map (or BPPM). A histogram is obtained for each channel and each of the 64 frequencies, from which the peak period can be estimated and a probability map can be derived. Averaging all of these together yields the BPPM. Figure extracted from [56].

Figure 5.6 shows an example of this method's output. The method is summarized in Algorithm 3.

5.3.2 Target forgeries

The method identifies anomalies in the double JPEG compression effect, so it is intended to work within a sub-domain of JPEG traces: those with pronounced double compression of different compression qualities. The paper also claims the results are better when the double compression goes from low to higher.

5.4 ZERO

As described in the acquisition pipeline in Chapter 2, JPEG compression is computed in non-overlapping 8×8 blocks that form a JPEG *grid*. The ZERO method [62, 63] works by predicting the JPEG compression grid origin for every pixel in the image, or as the paper describes it, each pixel "votes" for a main grid origin. If the vote map presents an anomaly with enough significance, it indicates forgery.

The grid votes are obtained by observing a pattern that JPEG compression leaves. Concretely, the compression algorithm quantizes the DCT coefficients of non-overlapping 8×8 blocks of images, setting many of the coefficients in the higher frequency range to zero. With this in mind, the method identifies the presence of a JPEG grid when a significant number of DCT zeros are observed for a given grid origin.

Similar to the Noisesniffer method presented in Section 5.2, the ZERO algorithm includes a statistical validation step according to the *a contrario* approach, which associates the NFA to each tampering detection. The detections are obtained by a threshold of the NFA.

Algorithm 3 DQ pseudo-code

```
Input: $Image$
Output: $heatmap$
  function calculate_tampered_probability($coefs, hist, p$)
      $BPPM \leftarrow$ empty_map($shape = $ shape($coefs$))
      for $c\_dct$ in $coefs$ do
          $b \leftarrow$ bin($h, c\_dct$)
          $s \leftarrow$ previous_peak($b, hist$)
          $period\_range \leftarrow [s : s + p - 1]$
          $P_u \leftarrow h[b]/$sum$(h[i \in period\_range])$
          $P_t \leftarrow 1/p$
          $BPPM[s] \leftarrow P_t/(P_t + P_u)$
      end for
      return $BPPM$
  end function

  $dct\_coefs \leftarrow$ read_jpeg($Image$)
  $BPPMs \leftarrow$ empty list
  for $c$ in $channels$ do
      for $f$ in $low\_frequencies$ do
          $coefs \leftarrow dct\_coefs[channel = c, frequency = f]$
          $hist \leftarrow$ histogram($coefs, bin\ size = 1$)
          $p \leftarrow$ detect_period($hist$)
          $BPPM \leftarrow$ calculate_tampered_probability($coefs, hist, p$)
          $BPPMs$.insert($BPPM$)
      end for
  end for
  $heatmap \leftarrow$ mean($BPPMs$)
```

5.4.1 Method

The outline of the ZERO algorithm is described in Algorithm 4. First, the luminance channel is computed from the RGB image, following the YCrCb conversion used in the JPEG standard.

Each pixel can belong to 64 different overlapping 8×8 blocks, which are associated with the different possible grid origins, as shown in Figure 5.7.

For each pixel, the method counts the number of zeros in the DCT coefficients corresponding to each grid origin. A zero in this context is a DCT coefficient with an absolute value smaller than 0.5, which is considered to be zero due to a rounding error introduced by image decompression.

Each pixel then "votes" for the grid origin where more zeros are observed. The grid origin receiving the most votes is considered as the most likely original grid. However, if another grid origin also receives a significant number of votes, it may

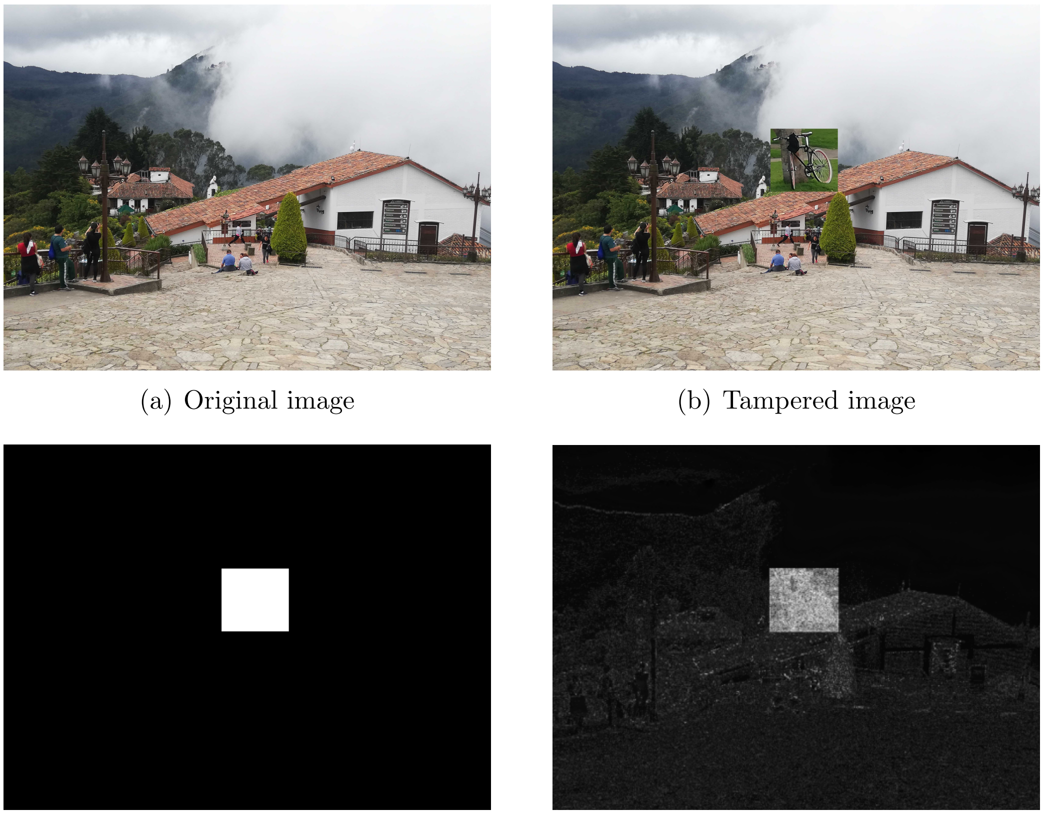

(a) Original image (b) Tampered image (c) Predicted mask (d) Predicted heatmap (BPPM)

Figure 5.6: Result of running our implementation of DQ on an image. The original image (a), the tampered image (b), the predicted mask (c), and the predicted heatmap (BPPM) (d).

indicate a forgery, particularly in the pixels that voted for this alternative grid.

It is important to mention that blocks with constant values along either the vertical or horizontal direction are excluded from the voting process. This is because constant blocks, by their nature, will have many zeros in their DCT coefficients. Allowing them to vote could result in a misleading emphasis on a particular grid origin that is not actually the true grid.

Finding the main grid cannot be reduced to simply choosing the grid with the most votes since JPEG compression and, therefore, a main JPEG grid may not be present. To address this issue, an *a contrario* statistical validation is used to ensure that the detection of a grid is not merely caused by noise or stochasticity. In this step, a null model is defined that considers possible correlations between neighboring pixels.

Considering the complete model, the NFA is defined for a certain grid origin to reject the null hypotheses if it is below a certain threshold. A low NFA means that the number of votes received by this grid is rare under the null hypothesis and has a very low probability of occurring just by chance, so it is statistically significant. Then, the most significant grid that scores below a certain threshold is considered

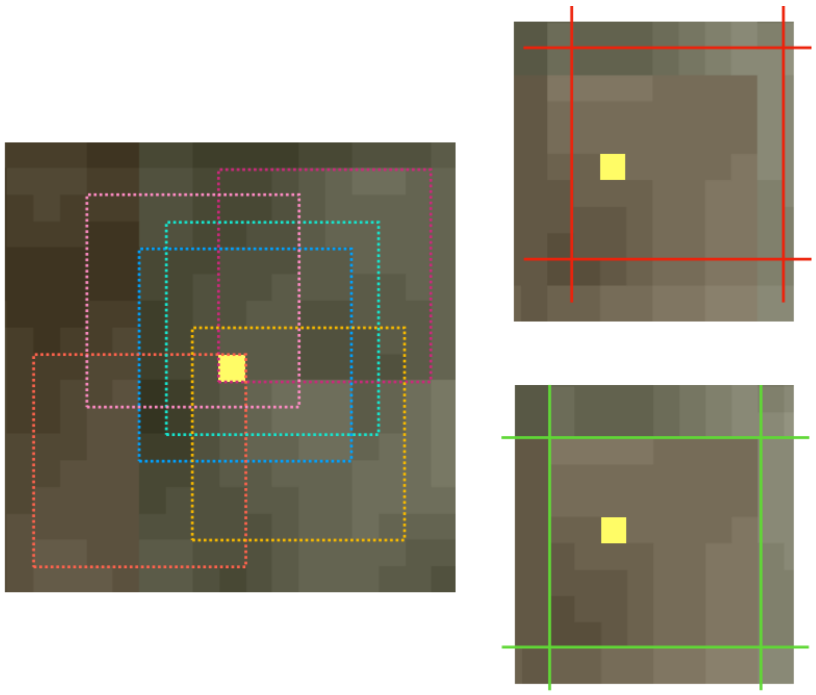


Figure 5.7: Each pixel (yellow) belongs to 64 different 8×8 blocks of the image. Six of them were drawn in different colors on the left. The top right shows (in red) the position of a patch not aligned with the grid. The bottom right shows (in green) the position of the patch containing the pixel matching the JPEG grid. Extracted from [63].

the main grid. This threshold is by default equal to 1 because in the *a contrario* approach, it is the expected number of false detections under the null hypothesis, which is a pretty acceptable standard to establish. This grid detection procedure can be performed globally on the entire image or locally on an image window.

Putting all of this together, a forgery can be detected by comparing a locally detected grid in a region to the main grid detected. If these differ (or one of them is a no-vote), it is an indicator of forgery. The ZERO method applies this concept with a region-growing algorithm to return a forgery prediction mask. To obtain a more homogeneous forged region, the morphological *closing* operation is performed on the predicted mask to obtain the final mask of the algorithm, F. A view of a given image, vote map, and forgery mask F is shown in Figure 5.8.

The method also has an additional feature: identifying regions with missing JPEG compression overall. If one region of the image lacks JPEG compression and the rest does not, this is taken as another indicator of forgery. To do this, the image is compressed with quality 99, which introduces some lossy compression (but a minimal amount) with the grid origin at $(0, 0)$. Then, the grid vote map and forgery detection algorithms are performed, but the pixels that previously voted for the main grid are set to a no-vote. This unveils forgery by producing the predicted mask M if a group of pixels suspiciously votes for the origin $(0, 0)$ after the second compression when it is not the main grid previously detected. If this

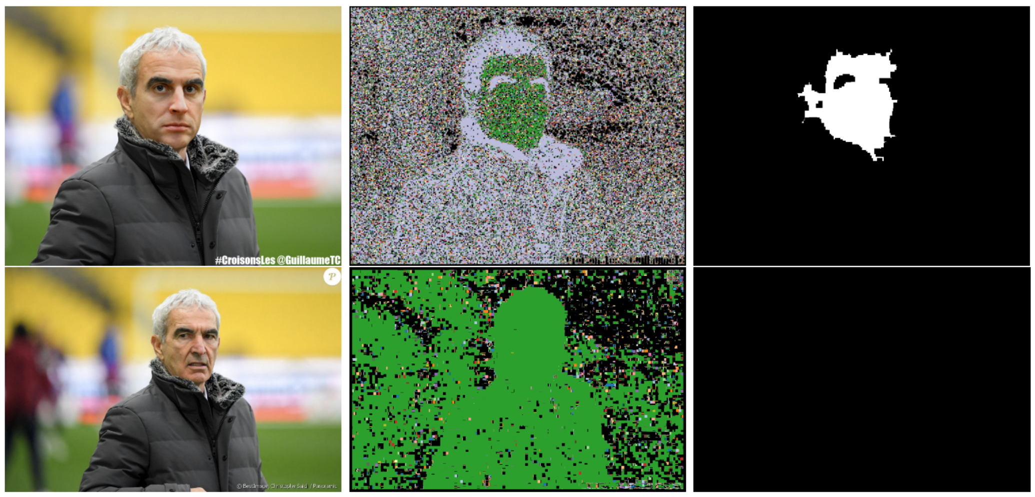


Figure 5.8: Top: forged image from the well-known Twitter account GuillaumeTC, its vote map and forgery mask F. Bottom: original image found online, its vote map and forgery mask F. Extracted from [62].

feature is also computed, a way to merge M with F to produce a final mask is by computing element-wise logic or function between the two: a pixel is forged if any of both methods detect it as such.

5.4.2 Target forgeries

ZERO method targets forgeries that misalign JPEG grids, which means it is intended for JPEG compression traces (either in the forged region, the main, or both). Cases of tampering in which both the forged region and the main grid are aligned, which happens with 1 in 64 chance, will fall under the radar. The authors also report a poor performance when images undergo too harsh of compression, as it produces several blocks with horizontal or vertical uniformity that are disregarded in the procedure.

The method's threshold is chosen to have NFA < 1, which provides a prudent criterion that avoids over-detecting forgeries but has good evidence in the case of detection. Viewing an example such as the one presented in Figure 5.8, it is clear by looking at the vote maps that the method offers great interpretability of its results, as it provides an explanation for the forgery detection (or lack thereof).

5.5 CAT-Net

CAT-Net [54] is an end-to-end fully convolutional neural network designed to detect compression artifacts in images. CAT-Net combines both RGB and DCT streams, allowing it to learn image acquisition and compression artifacts jointly. Each stream considers multiple resolutions to accommodate the various shapes and sizes of the spliced objects. This comprehensive deep-learning method is used

Algorithm 4 ZERO pseudo-code

```
Input: $Image$, [$Image\_quality\_99$]
Output: $mask$, $missing\_grid\_regions$
  function grid_votes_map($luminance$)
      for $p$ in $pixels$ do
          $zeros\_amount \leftarrow 0$
          $best\_grid \leftarrow None$
          for $g$ in $grids$ do                                  ▷ 64 possible grids
              $grid\_pixels \leftarrow luminance[g]$
              $coefs \leftarrow$ DCT_2D($grid\_pixels$)
              $zeros\_amount \leftarrow$ sum(abs($coefs$) $< 0.5$)
              if $zeros\_amount > most\_zeros$ then
                  $zeros\_amount \leftarrow most\_zeros$
                  $best\_grid \leftarrow$ grid_codification($g$)
              end if
          end for
          $votes[p] \leftarrow best\_grid$
      end for
      return $votes$
  end function

  $L \leftarrow$ luminance($Image$)
  $votes \leftarrow$ empty array($size = L$)
  $votes \leftarrow$ grid_votes_map($L$)
  $main\_grid \leftarrow$ detect_main_grid($votes$)
  $F \leftarrow$ detect_forgeries($votes, main\_grid$)
  if $Image\_quality\_99$ is an input then              ▷ Missing grids computation
      if $main\_grid \neq$ NO VOTE then
          $votes \leftarrow$ grid_votes_map($Image\_q99[luminance]$)
          $votes[votes = main\_grid] \leftarrow$ NO VOTE
          $M \leftarrow$ detect_forgeries($votes, main\_grid, grid \leq 0$
      end if
  else
      $M \leftarrow None$
  end if
  $mask \leftarrow M$ elementwise-or $F$
```

for localizing spliced regions in JPEG and non-JPEG images.

The RGB stream processes the semantic and color information of the image, which is often altered during image splicing, while the DCT stream analyzes the compression artifacts, which are usually introduced when an image is saved in a compressed format like JPEG. By analyzing both color information and compression artifacts, CAT-Net can better discern inconsistencies associated with splicing or copy-move compared to using only one of the streams.

Multiple-resolution analysis refers to the processing of image data at various scales or resolutions. This is crucial in image splicing or copy-move detection because forged objects can come in different sizes and shapes. By analyzing the image at multiple resolutions, CAT-Net can adapt to various scales, making it more versatile and accurate in detecting spliced regions, regardless of their size.

5.5.1 Method

The method uses HRNet [84] as the backbone architecture for both streams. The choice of such architecture relies on its multi-resolution analysis. Using pooling layers is avoided since pooling reinforces content and suppresses noise-like signals, undesirable for a fine-grained analysis [19]. The architecture for this method is shown in Figure 5.9. In Figures 5.10 and 5.11, the basic blocks of the network and the JPEG stream are shown in detail.

Several considerations are made in order to aid the DCT stream to focus on JPEG compression artifacts. Firstly, the DCT stream is encoded into a binary volume, which is a representation not far from DCT histograms other methods use for analyzing compression artifacts (see Section 5.3) whilst maintaining local information of the coefficients. The JPEG quantization tables are also an input of the DCT stream. Furthermore, the DCT stream weights are initialized by pre-training them for the task of detecting double quantization.

On the other hand, the RGB stream is an HRNet that was initialized by pretraining on the ImageNet [52] classification problem.

The network is designed to work with JPEG images, but it can also process non-JPEG images by treating them as if they were JPEG images with a quality factor of 100. This is achieved by placing a JPEG encoder at the beginning of the network to convert non-JPEG images into JPEG format.

Figure 5.12 shows an example image and the output corresponding output heatmap.

5.5.2 Target forgeries and dataset biases

The network architecture is developed to be sensitive to JPEG artifacts, so this network should be expected to perform better under the presence of these, although it may also find other traces through the RGB stream.

Within the datasets that were used for training and running experiments is CASIA v2 [32]. Although this dataset is not one of the datasets selected for this work, its images are very similar to those in CASIA v1.

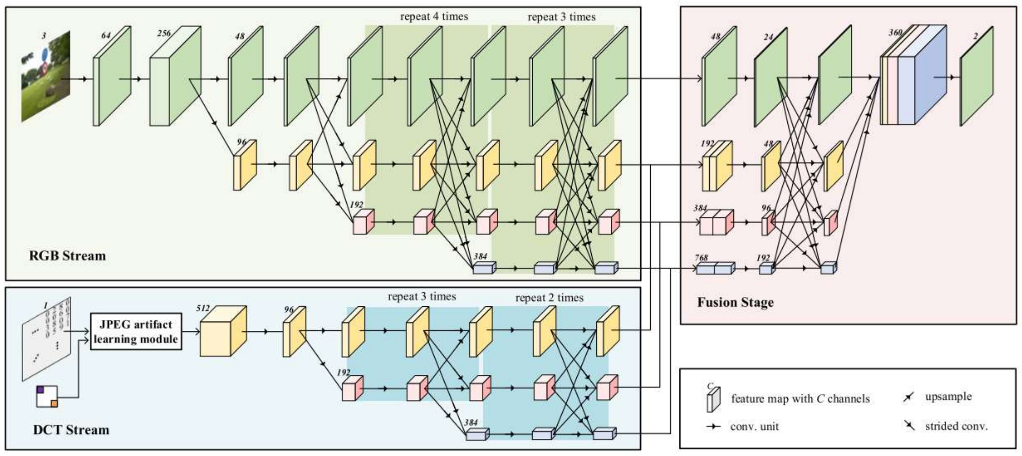


Figure 5.9: CAT-Net architecture includes an RGB stream, a DCT stream, and a final fusion stage. The RGB stream takes the RGB pixels, while the DCT stream takes the Y-channel DCT coefficients and a Y-channel quantization table as input. Extracted from [54].

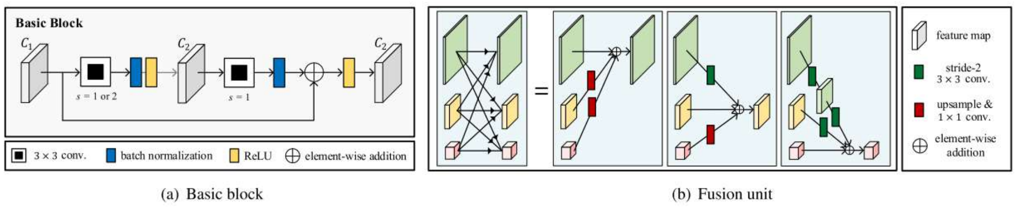


Figure 5.10: Elements in the CAT-NET network. A convolutional unit mainly consists of four consecutive basic blocks. The fusion unit fuses multi-resolution feature maps by summing them after matching resolutions. Extracted from [54].

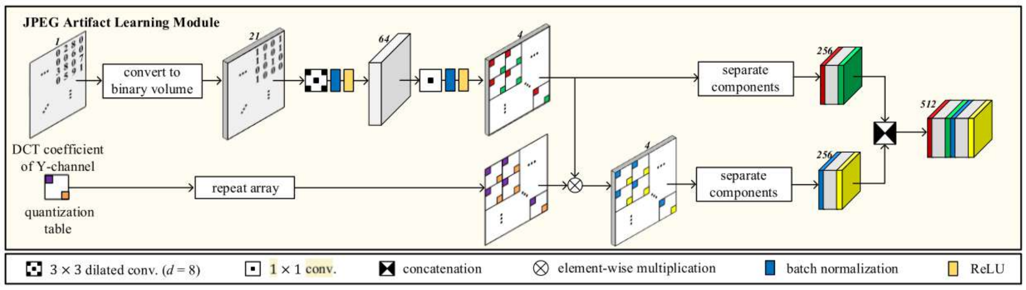


Figure 5.11: JPEG artifact learning module architecture. Extracted from [54].

5.6 EXIF as Language

An image file contains not only the pixel values but also a lot of extra metadata accompanying the image: camera model, exposure time, focal length, JPEG quantization details, and more. This is called the Exchangeable Image File Format or EXIF. This method [96] is a more modern approach to Huh *et al.* [46], where a

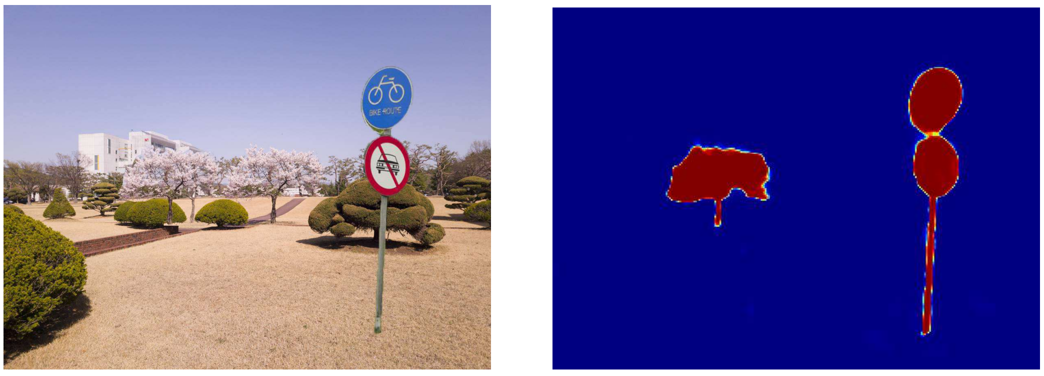


Figure 5.12: Result of running CAT-Net on an image. The original image (a) was extracted from [54], (b) shows the predicted heatmap.

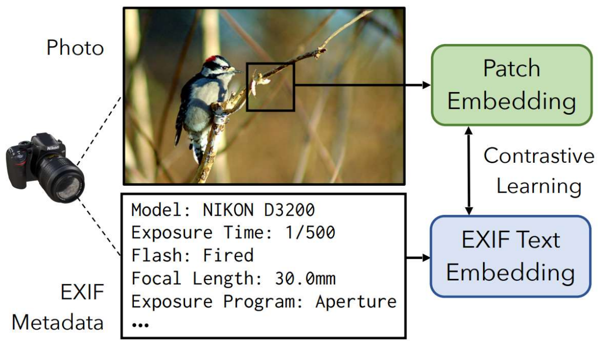


Figure 5.13: Illustration of the multi-modal embedding employed in EXIF. Extracted from [96].

cross-modal model is trained to understand the relationship between what is being "said" about the image (in its EXIF metadata) and the image content itself. This can later aid forgery detection by inspecting whether or not this information is consistent across the image.

Interestingly, the method operates blindly by learning to map the EXIF information with the image content into a shared embedding space through contrastive learning on a cross-modal model, as illustrated in Figure 5.13. After adequate training, the EXIF information can be captured in the embedding space directly from the image. Once this is done, all that remains is to search for inconsistencies in this embedding space across the image, as these indicate that a region of the image does not match the original EXIF information.

5.6.1 Method

The method consists of training both an image and text encoder using contrastive learning, obtaining a single, cross-modal embedding space. The paper draws inspiration from OpenAI's CLIP [71], changing out the natural language for the EXIF

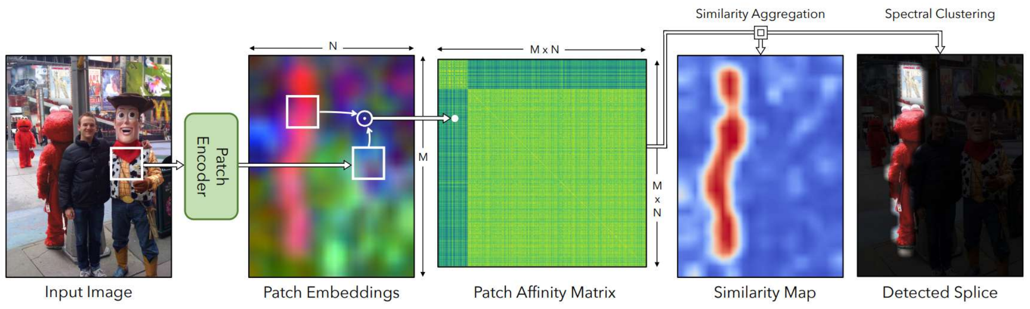


Figure 5.14: Outputs of the different components in the EXIF pipeline. From left to right: the original image, from which patches are drawn to map into the embedding space. These patch embeddings are used to compute the Affinity Matrix, from which a detection heatmap (through Mean Shift) or a detection mask (through Normalized Cuts) is inferred. Extracted from [96].

information concatenated as a string.

This training scheme results in two encoders, image, and text, that operate in the same embedding space. In other words, patches from the same image should be close to one another in the embedding space, while patches from images with different EXIF information should be farther apart. This allows the method to use the image encoder to detect images that have been spliced without relying on the EXIF information attached to the image. If patches are taken from the same image cluster in two or more regions of the embedding space, that means that the image is a splicing of images that share different EXIF data.

The procedure to predict forgery from the EXIF embeddings is shown in Figure 5.14. First, in order to find the inconsistencies the method extracts an "Affinity Matrix", by computing the cross product of all patch against patch combinations. The matrix is then clustered by rows into similarity aggregations that can reveal the tampered region. This last part has two approaches: Mean Shift [47], which yields a heatmap, and spectral clustering via Normalized Cuts [76], which yields a mask.

The encoder is a ResNet pre-trained to ImageNet [52], and DistilBERT [74] is used as a text encoder for the training procedure.

5.6.2 Target forgeries and dataset bias

By searching for consistency with image metadata, this method is not oriented into any particular trace of the ones discussed in Chapter 2, although it will likely oversee copy-move type of forgeries (as the EXIF data should be consistent).

The method trained on datasets comprised of pristine images and their metadata, none of which included the datasets that we selected for the benchmark (Section 3.2).

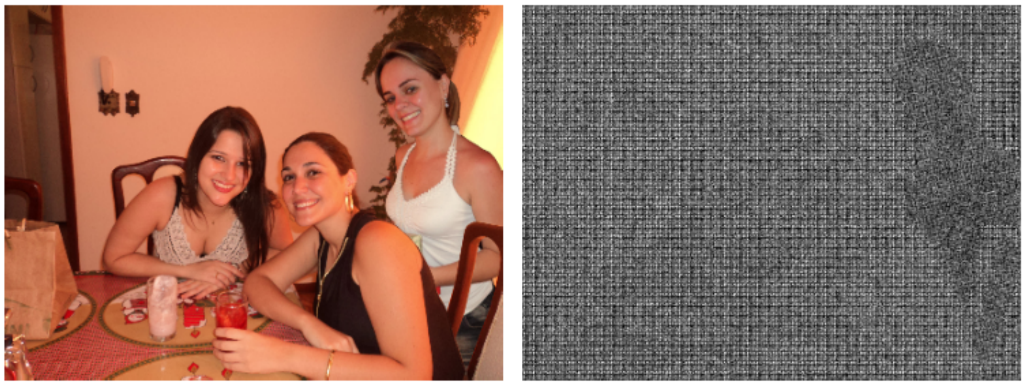


Figure 5.15: A forged image (left) where the woman on the right has been introduced by splicing, and its noiseprint (right). The inconsistencies caused by the manipulation are visible in the extracted noiseprint. Figure obtained from [26].

5.7 TruFor

The method described in [42] is a forensic framework that solves a detection problem in order to say if an image has been manipulated or not and a localization problem. The method not only returns a heatmap highlighting suspicious areas but also returns a reliability map highlighting which areas might be error-prone. To do so, the authors propose a combination of two inputs, the original RGB image and a noise fingerprint called *Noiseprint++*. Through this combination, forgeries are detected as the deviation from the regular pattern of non-tampered images.

5.7.1 Noiseprint++

The Noiseprint++ is a noise fingerprint that comes from retraining the original Noiseprint [26] previously developed by the same authors but contemplating more varied scenarios. Concretely, it is a deep learning model that produces a fingerprint intended to capture not only the camera noise model but also the editing history of the image (comprised of a combination of basic processing operations such as resizing, compression, and illumination changes). Figure 5.15 illustrates how a noiseprint may reveal a splicing of two images from different camera models.

This model is a CNN trained to differentiate different camera noise models and edit history following a self-supervised contrastive learning paradigm. This is done by taking a Siamese network that takes pairs of patches as input and training it to produce distant outputs when the patches do not share the same camera model and editing history and close outputs when they do. Figure 5.16 illustrates this training paradigm. It is important to note that the spatial non-uniformity in the camera noise model (see Section 2.1.1.2) is contemplated, meaning patches of different spatial position in the image are also regarded as "different". This is useful for revealing forgery as different camera models or editing history within the same image is a clear sign of splicing or other sorts of tampering, as discussed

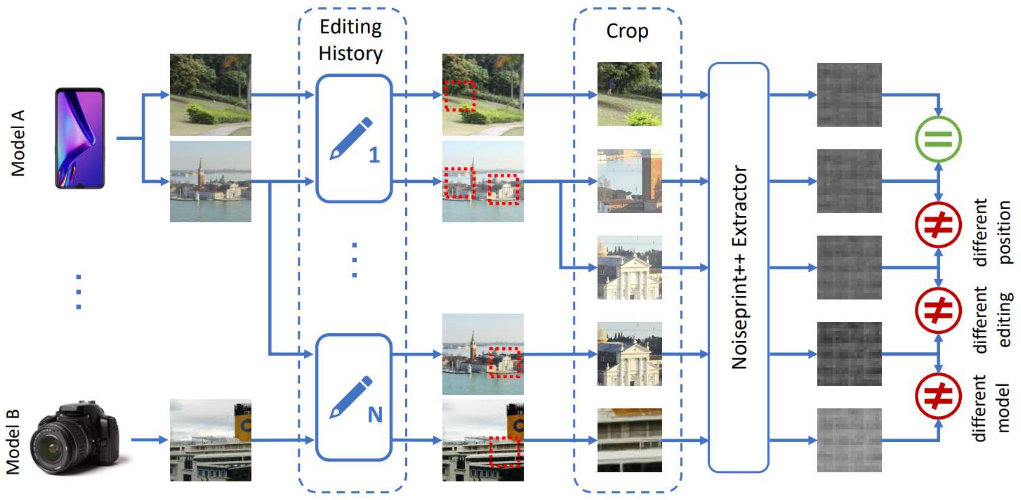


Figure 5.16: Noiseprint++'s training paradigm, represented in high-level. The objective is to minimize the distance between patches with the same camera model and editing history and maximize it otherwise. Extracted from [42].

in Chapter 2.

It is also interesting to note that, for this training procedure, the data required is not from a forgery dataset but rather a pristine dataset that includes camera model information. This means Noiseprint++ does not have a bias towards a certain type of forgery; rather, it aims to capture the image pipeline altogether and emphasize forgery traces in this process.

The weights of this network are initialized to a noise-extracting CNN (using a successful denoiser CNN), which directs the model toward capturing the relevant data in the higher frequencies range.

Noiseprint++ can then be used as a forensics feature input for image forensics. The original Noiseprint ran on the head of a classical method previously presented by the authors (Splicebuster [27]), but in TruFor, this is an input to the rest of the network.

5.7.2 Method

The full *Trufor* framework is illustrated in Figure 5.17. An RGB image is the framework input from which a noise feature is extracted with the *Noiseprint++* module. Both the RGB image and the noise feature are the inputs of an encoder that extracts dense features used in the next steps of the framework. Those features serve as input to an anomaly decoder from which an anomaly map is extracted. This anomaly map is the heatmap that allows users to identify forged regions. Those same features are also an input of a confidence decoder whose output is the confidence map, representing the confidence level the model assigns for each pixel predicted in the anomaly map. The pipeline also combines the output of both decoders with a pooling module to get a compact descriptor, going through

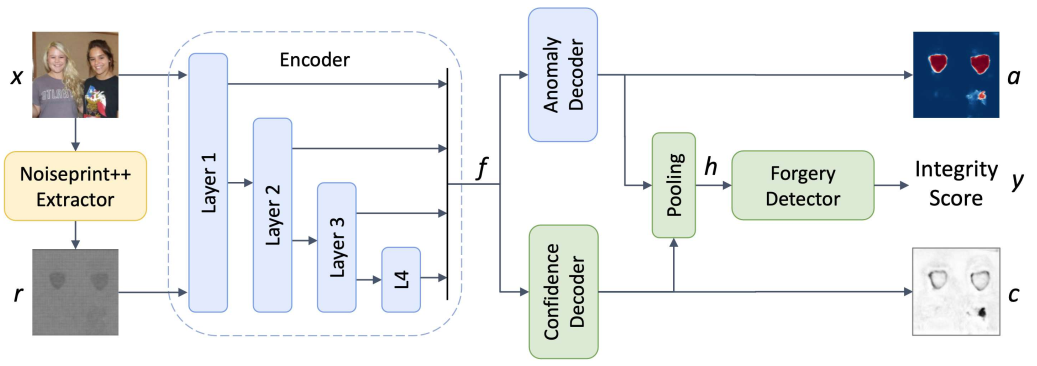
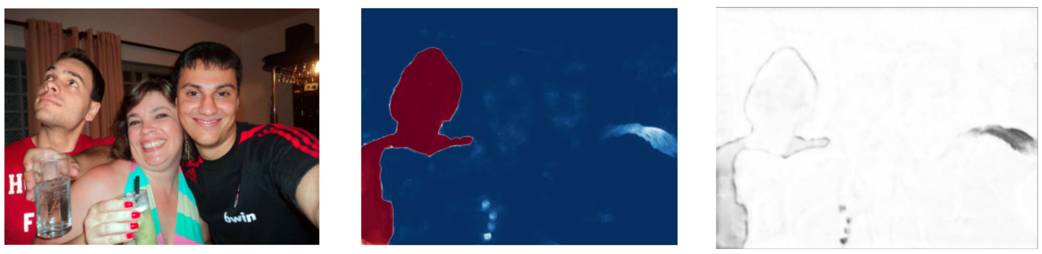


Figure 5.17: *Trufor* full framework, extracted from [42]. The input image is x, from which the Noiseprint++ residual r is extracted. Both of these are fed to the encoder, which, after decoding, produces the anomaly map a, the confidence map c, and the integrity score y.



(a) Original image (b) Predicted heatmap (c) Confidence map

Figure 5.18: Result of running TruFor on an image. The original image (a) was extracted from [42], (b) shows the predicted heatmap, and (c) shows the confidence map.

a forgery detector module to predict an integrity score. This detection score is necessary to answer the detection problem.

As it was discussed in Section 5.7.1, the *Noiseprint++* module is constituted by a noise-extracting CNN. The encoder is based on a CMX architecture [95] that relies on SegFormer modules [90]. Both decoders have the architecture of the multilayer perceptron of SegFormer [90] and were trained with a dataset including pristine and forged images with their respective ground truth masks. In the case of the confidence decoder, the true class probability map (computed from the anomaly map and the ground truth) was used as a mask in order to capture the confidence of the anomaly prediction. The pooling module is in charge of generating an 8-component feature vector that is fed to two fully connected layers that predict the integrity score. Those fully connected layers that conform to the forgery detector are trained with the same dataset as the decoders.

The output heatmap and confidence map for an example image are shown in Figure 5.18.

5.7.3 Target forgeries and dataset biases

The method is trained on three distinct modules: Noiseprint++ on one side, the encoder on another, and a block that consists of the anomaly decoder, the confidence map decoder, and the forgery detector.

As mentioned before, the Noiseprint++ is trained using self-supervised contrastive learning. This form of blind training performed over pristine images favors generalization, as it is not biased towards any forgery dataset. Moreover, no pristine images from the forgery datasets selected for the benchmark are used in this training (Section 3.2).

This makes TruFor a particularly interesting method to include, as it is deep learning-based but has components to aid generalization, which, as discussed previously (Section 3.1), is especially relevant in media forensics. This is part of the reason why it was chosen to be integrated into the *PhotoHolmes* library, added to the fact that it also outperforms other self-supervised methods proposed by these authors, like Noiseprint. Having said this, the remainder of the network, this being the encoder, decoders, and forgery detectors, are trained in a supervised manner on popular forgery datasets, so the overall method is not free of having a certain degree of data bias.

The datasets used for training the encoder and anomaly decoder, and the confidence decoder and forgery detector, are not used in the benchmarking of *PhotoHolmes* (Section 3.2).

Regarding the specific traces targeted by the model, it is trained using both the RGB image and Noiseprint++. Consequently, it is not possible to specify exactly which forgery traces the overall model detects. However, Noiseprint++ is designed to identify camera models, potentially capturing noise-based traces, CFA algorithm traces, and JPEG artifacts, which include considerations of editing history and compression processing.

An important comment to make about the method selection for this library is that the Multimedia Forensics group, led by Verdoliva at the University Federico II of Naples, presents several methods that feature a sort of unsupervised fingerprint that can be used for forensics, such as Noiseprint [26], Comprint [59] and TruFor [42]. Although this methodology is interesting and each of these methods tackles different forgery traces, the *PhotoHolmes* team decided to include only TruFor, which is the one reporting the best performance.

5.8 PSCC-Net

The method described in [57], referred to as PSCC-Net, consists of a neural network implementing a *coarse to fine* approach. The method returns a mask locating forgeries in the input image together with a label that indicates whether the image has been manipulated or not.

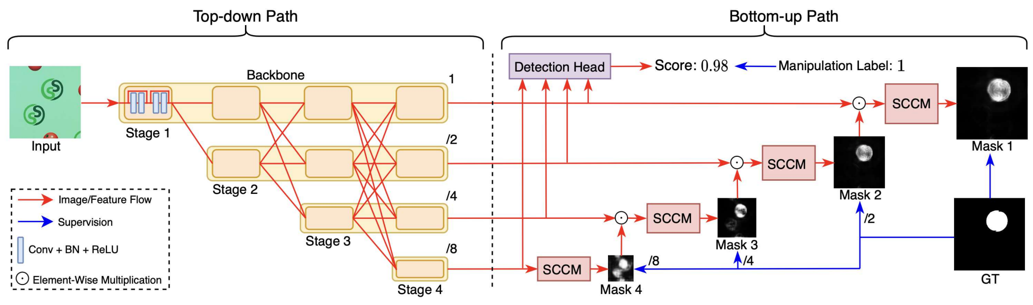


Figure 5.19: PSCC-Net full framework, extracted from [57], showing the top-down and bottom-up paths along with the Detection Head. The output mask is *Mask 1* and the detection score from the Detection Head. There is also an illustration of how the output mask prediction improves by learning on the outputs on the lower scales.

5.8.1 Method

Figure 5.19 depicts the architecture presented in [57]. The network consists of two different sections, first the *top-down path* constituted by an HRNet backbone, and then the *bottom-up path* that combines the features at different scales in a detection head and a sequence of "Spatio-Channel Correlation Modules". The top-down path architecture is the HRNetV2p-W18 described in [82], initialized with ImageNet [52] pre-trained weights. The main goal of this part is to compute features at different scales that serve as inputs to the same levels of the *bottom-up path*, as well as the detection head that indicates if the image is pristine or not. In contrast to other encoder-decoder and no-pooling networks, HRNet performs dense cross-connections between different scales in order to handle scale variation better, progressively incorporating local and global features at each scale.

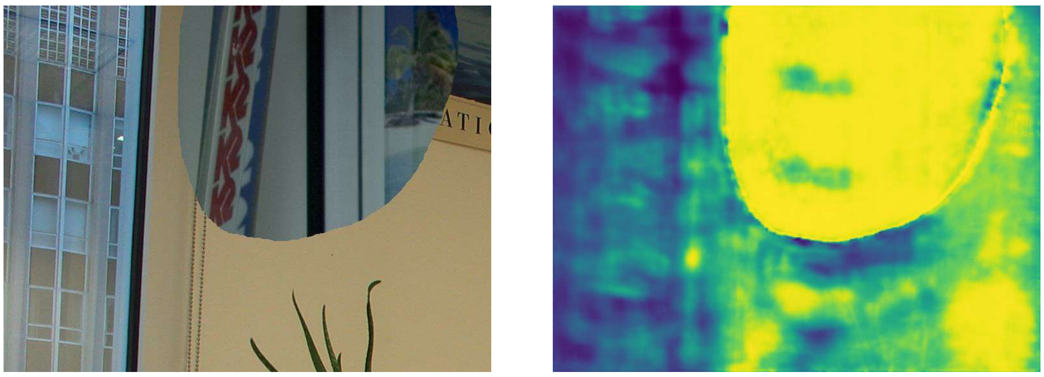
In the bottom-up path, the authors add to every level a Spatio-Channel Correlation Module (SCCM) that tries to lay hold of spatial and channel-wise correlations by applying attention to both spatial and channel-wise levels. Here, a *coarse to fine* approach is used, obtaining an increasingly more precise resolution of the masks as shown in Figure 5.19.

An example image and its resulting heatmap are shown in Figure 5.20.

5.8.2 Target forgeries and dataset biases

The full architecture is trained over synthetic datasets that include splicing, removal, copy-move, and pristine images. The overall architecture design is not directed towards a specific type of forgery trace.

To the interest of this project, it was trained on a fraction of Columbia [45] and CASIA [32] datasets, among others not used in our benchmarking (Section 3.2). It could be expected that the method will have a favorable bias towards Columbia and CASIA.



(a) Original image (b) Predicted heatmap

Figure 5.20: Result of running PSCC-Net on an image. The original image (a) was extracted from [57], (b) shows the predicted heatmap.

5.9 Adaptive CFA forensics

The method in [16] is a blind approach to identifying local mosaic inconsistencies. This is done by means of an end-to-end CNN, which is set to detect inconsistencies in the positioning of the Bayer matrix (see Section 2.1.2). More specifically, the first part of the network is trained to predict the position of each pixel modulo (2,2), which is reflective of the Bayer matrix position. Having done this, training the remainder of the network for forgery prediction is a straightforward process as an inconsistent Bayer matrix position reveals forgery, at least when it comes to detecting copy-move or splicing forgeries with strong demosaicing artifacts. An interesting feature of this model is that it can be trained directly on unlabeled and possibly forged images, and it is designed for lightweight retraining on new datasets.

5.9.1 Method

An end-to-end CNN architecture is presented that bases its layers on demosaicing features. The architecture consists of the concatenation of four modules: a spatial network, a pixel-wise causal network, a blocks preparation module, and a blockwise causal network.

The first module is the spatial module, shown in Figure 5.21, intended to capture the spatial traces that a demosaicing algorithm leaves. The second module enables the capturing of complex causal relations without adding more spatial dependencies to the convolutions and is named the pixel-wise module, shown in Figure 5.22. The following section in the pipeline performs the processing of the pixel-wise output into blocks by grouping the pixel to make more reliable predictions and permutating the blocks to keep a balance between the four possible CFA patterns (see Section 2.1.2.1). This module is shown in Figure 5.23. Each block is spatially represented by one pixel due to the average pooling. Finally, the

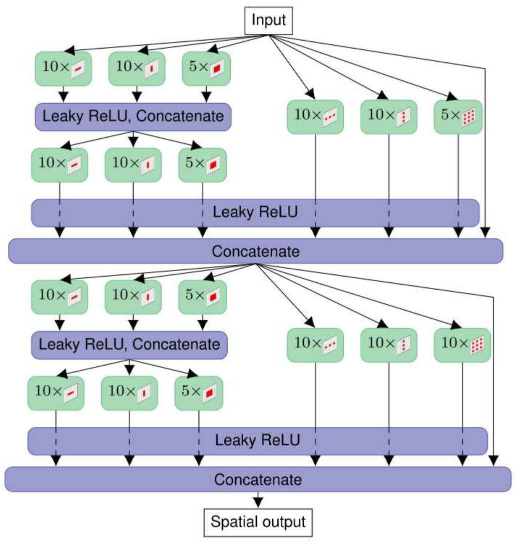


Figure 5.21: Spatial module of the network for Adaptive CFA, comprised of layers with full and diluted convolutions, and skip connections. Extracted from [16].

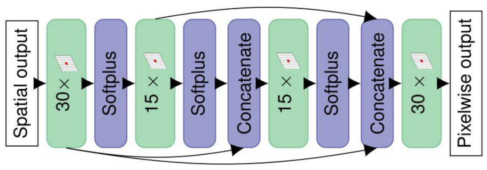


Figure 5.22: Pixelwise causal module for Adaptive CFA, achieved by a sequence of 1×1 convolutional layers. Extracted from [16].

blockwise module processes the blockwise features to make a forgery prediction. This module is shown in Figure 5.24.

The training procedure is to first train the spatial and pixel-wise networks, with an auxiliary head of 4 pointwise convolutions, to predict a pixel's position modulo (2,2). After this is achieved, the auxiliary module is removed, and the pixel-wise outputs are processed into blocks. The remaining procedure is to train the blockwise network, taking the processed blocks as inputs. This sequence reduces training time and empowers the network to detect the demosaicing algorithm.

Figure 5.25 shows the output of this method on an example image.

5.9.2 Target forgeries and dataset biases

This method is trained to detect CFA traces and is directed to detecting demosaicing inconsistencies, which can be produced mainly by splicing or copy-move. Note that in 1/4 of the cases, the forged and pristine region's grid would be aligned, preventing detection. It is also important to note that the paper indicates that post-processing and compression severely hinder performance, so it will likely not perform well under datasets with JPEG compression or other post-processing.

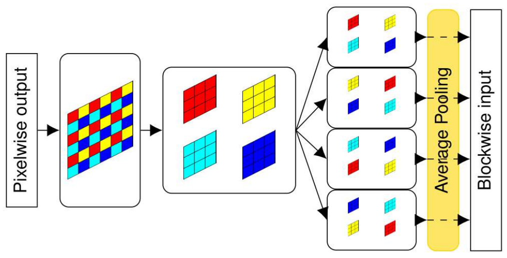


Figure 5.23: Processing module for Adaptive CFA that transforms the image into blocks, which is done to make predictions more reliable. The image is split into four images according to the four modulo (2,2) positions (illustrated as the red, yellow, light blue, and dark blue matrices). A concatenation of these images in different positions produces 4 new features (the 4 blocks that are vertically aligned on the right). After average pooling, the blockwise output is obtained. Extracted from [16].

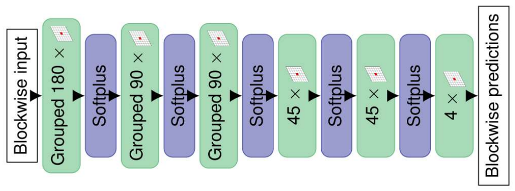
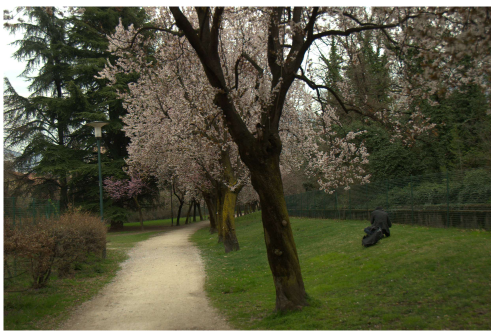


Figure 5.24: Blockwise module of the Adaptive CFA network. After the blockwise processing, each resulting pixel maps to one block. Pointwise convolutions are computed over these features to process the data in each block. The first three layers process the blocks independently through group pixel-wise convolutions, and the remainder of the network merges the features through full-depth pixel-wise convolutions. Extracted from [16]


(a) Original image

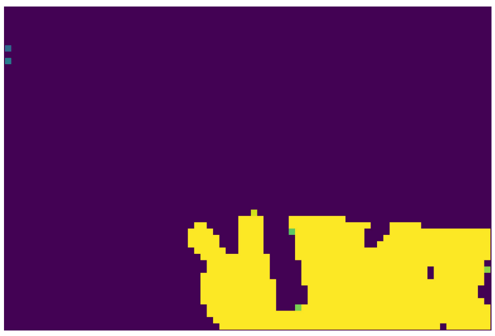

(b) Predicted heatmap

Figure 5.25: Result of running Adaptive CFA on an image. The original image (a) was extracted from [16], (b) shows the predicted heatmap.

An important issue to address is that although the capacity to adapt the model by retraining on test data is alluring, the *PhotoHolmes* team opted not to do this on the benchmark process. This decision was made primarily because it could result in an unfair comparison to other methods, which may also benefit from retraining.

Having said this, the training data, which is RAW data synthetically demosaiced with different methods, is not present in the benchmark data (Section 3.2).

5.10 FOCAL

Forensic Contrastive Clustering [88] (or FOCAL) comprises a feature extractor module, followed by a clustering module that obtains the predicted mask. The concept behind this method is that the feature extraction should be trained separately from the classifier, and more importantly, the classifier be an unsupervised online learning algorithm.

This paper argues that the classification of image regions as either genuine or forged depends heavily on the context within the image. Specifically, a spliced region should not be considered tampered with if it aligns with its original context within the image. While traditional supervised classifiers are trained to predict based on features alone, without considering feature distributions, this paper contends that a more effective approach is to segment features within each image context independently. This is what FOCAL resolves by utilizing a trained feature extractor along with an unsupervised clustering method to produce the predicted mask.

5.10.1 Method

The extractor module, the only trained component of this pipeline, comprises a ViT [83] and an HRNet [84]. The output features are trained using contrastive learning, which employs an InfoNCE++ (a modified version of [80]) loss with the features represented by a set of keys and a query. The loss is computed image by image and later averaged over a batch. A representation of this procedure is shown in Figure 5.26, which illustrates that as a result of the contrastive training, the feature vectors are close when they belong to the same mask class (forged or pristine) and distant when they are not.

Once the feature extractor is trained, all that is left is to cluster it into the two classes in order to predict a mask. For this task, the HDBSCAN [34] clustering algorithm is used. Importantly, this last component is performed image by image, which ensures there is no "contamination" learned between different images, and the forgery is predicted by taking only into account the feature context of the given image. The class with the lesser amount of pixels is regarded as the forged class, under the assumption that forged regions are usually the lesser portion of the image.

Finally, since clustering is applied to a set of features of any depth, this method can be extended by concatenating other sorts of features and clustering them

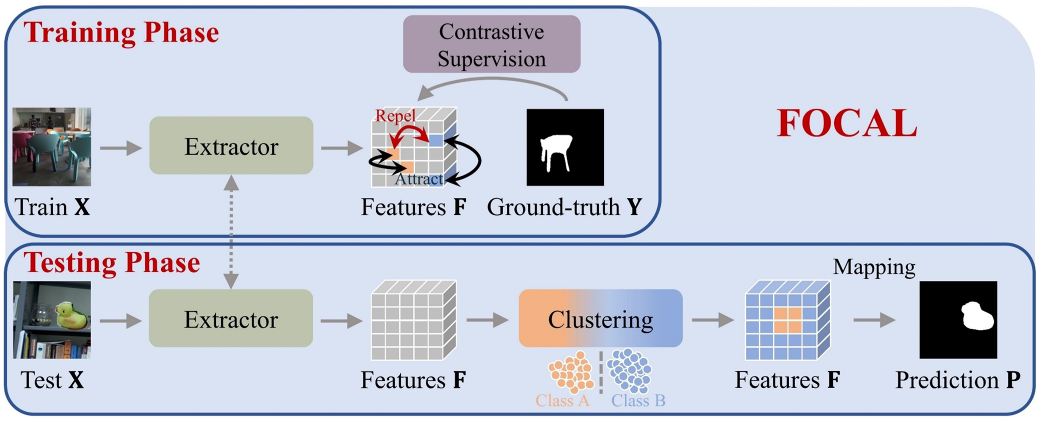


Figure 5.26: FOCAL training and prediction framework. The training phase uses the ground truth mask to train the feature extractor using contrastive learning. For inference, the model extracts the features and clusters them into two groups, resulting in the output mask. Extracted from [88].

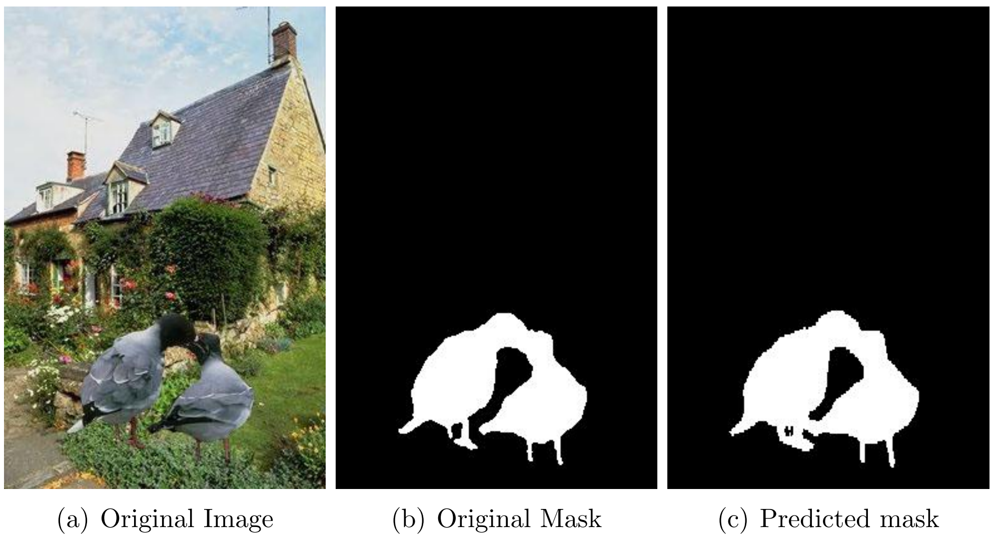


Figure 5.27: Results of running focal on a forged image (a), extracted from CASIA 1.0 SP [32]. FOCAL predicts a mask (c) almost identical to the ground truth (b).

altogether. Naturally, this only makes sense for features that capture the image-processing pipeline in a certain way.

In Figure 5.27, we show the results of running this model on a forged image.

5.10.2 Target forgeries and dataset bias

FOCAL is not trained or designed to target a specific type of forgery. Within the model, the dataset-sensitive component is the feature extractor. The extractor was trained on the same datasets as CAT-Net, meaning amongst the *PhotoHolmes*' benchmarking datasets (Section 3.2), the use of CASIA v2 might result in a favorable bias towards CASIA v1 [32]. Although the feature extractor was trained for the purpose of clustering (not quite a mask prediction), it could still be expected for the method to have a positive bias towards this dataset.

This chapter provided a brief overview of a variety of methods that attempt to detect and localize forgeries and that have been chosen to be implemented in the PhotoHolmes library and benchmarked. The ten selected methods follow both classical and data-driven approaches. Some expose forgery by capturing specific traces left from noise in the camera model, demosaicing, and compression, whereas other methods achieve forgery detection or localization by recognizing patterns learned from data.

In the next chapter, we introduce the PhotoHolmes library, which builds on the concepts we have discussed so far in this manuscript. We present the library structure and principles behind its development, how it enables researchers to benchmark a method easily, and how it allows for contribution and extension of itself.

Chapter 6

The PhotoHolmes library

"pip install photoholmes"

Future users

The previous chapters have provided a thorough exploration of the problem, the relevant data, metrics, and methods available in the literature to address the issue. The increasing number of publications around image forgery detection highlights the need for a standardized library integrating methods, metrics, and datasets. Ideally, such a library should allow both to benchmark methods in popular forgery datasets and test suspicious images using different methods without going through each method's implementations separately.

Considering most methods have strengths and weaknesses related to the forgery type and acquisition traces they pursue, being able to run an image through different methods with ease results in the possibility of quickly creating robust forgery reports on a suspicious image.

With these considerations in mind, we created PhotoHolmes. The goal of this chapter is to introduce and elucidate the purpose and functionalities of PhotoHolmes. The chapter will begin by explaining the overall structure of the library and subsequently delve into a detailed explanation of its various modules.

6.1 What is *PhotoHolmes*?

PhotoHolmes is a novel *Python* library for digital image forgery detection, which includes a diverse array of state-of-the-art methods, as well as a variety of datasets, metrics, and useful tools for evaluation and inference. The idea behind the library arises from the survey of the state of the art carried out, where a lack of consistent tooling, coding standards, and structure was noticed across different repositories. This lack of standardization makes it difficult to compare methods and reproduce results, relying on the authors to produce documentation or to answer questions directly.

In an effort to supply the community with a tool that can help overcome these issues, we created *PhotoHolmes*. The library is designed to be modular, reproducible, extensible, and easy to use. As we mentioned when discussing the related works section in Chapter 1, it is important to note that this is not the first project of its kind. Matlab-forensics [92], for instance, is one of the earliest libraries that compiled state-of-the-art image forgery detection methods, but it is not being actively maintained and is built on *Matlab*, a licensed programming language. A more recent example is *fake-detection-lab* [91], a *Python* project that had the same objective as *PhotoHolmes* but has stopped receiving updates after being published.

The library is available in two remote git repositories:

- The university's Gitlab: used for developing the project
  https://gitlab.fing.edu.uy/photoholmes/photoholmes/-/tree/main
- Photoholmes' Github: a cleaner repository created for its publication
  https://github.com/photoholmes/photoholmes

6.2 Design principles

To ensure the library fulfills its purpose and is maintained over time, the library is designed with four main principles: modularity, reproducibility, extensibility, and usability.

Modularity. The library is designed to be modular, where each module tackles a specific aspect of the image forgery detection pipeline. This simplifies maintenance work and makes for cleaner and simpler code.

Reproducibility. The library is designed to allow reproducibility, enabling the user to easily replicate the experimental results of the different methods. Furthermore, it provides a transparent open-source implementation and detailed documentation of the methods.

Extensibility. The library is designed to be extensible, allowing contributors to quickly expand the functionalities of the library by adding new methods, datasets, metrics, and tools. This flexibility is facilitated through a wrapper-based code architecture complemented by detailed guides on integrating new functionalities.

Usability. The library is designed to be easy to use. This is accomplished through a Command Line Interface (CLI), which allows the user to invoke the library without writing code. Classes are also available for seamless library integration into the users' code.

6.3 Design choices

Following the aforementioned principles, we designed and wrote the library following an Object Oriented Programming (OOP) paradigm. In OOP, one can define a parent class that sets a basic structure to follow, from which new classes can be defined, inheriting properties and methods. The second reason OOP was chosen is because the language chosen, *Python* [36], is an interpreted OOP programming language.

Choosing *Python* as the programming language has its roots in the popularity the language has gained in the computer vision and data science community in general, coming to a point where most of the research carried out today is written in *Python.* Coincidentally, it is also the language the team is most familiar with. *Python*'s strong foothold in the data science community comes from its simple syntax and its extensive third-party libraries support. Libraries such as *Numpy* [44] and *PyTorch* [14] enable quick and efficient numerical computations, allowing it to reach speeds close to a compiled language like C, but with a much simpler syntax.

The last important design choice refers to the deep learning framework used. There are many deep learning frameworks, but there are two that stand out: *Tensorflow* [9] and *PyTorch* [14]. Until a few years ago, *Tensorflow* was the most widely used deep learning framework, and some image forensics methods were implemented using it, like the original Noiseprint [26]. Lately, the tide has turned in favor of *PyTorch* [87], and most recent research has been trained using this framework, which is why we chose only to support *PyTorch.*

6.4 Structure

The library is subdivided into 7 different modules, each of them with a specific purpose. The modules are the following:

- **Datasets**: contains the code implementation for loading the different datasets that can be used to benchmark the methods.
- **Preprocessing**: contains different preprocessing operations that can be applied to the images before using the methods.
- **Methods**: contains the methods that can be used to detect forgeries.
- **Postprocessing**: contains different postprocessing functions that can be used to post-process the outputs of the methods.
- **Metrics**: contains the different metrics that can be used to evaluate the performance of the methods.
- **Benchmark**: contains the benchmark class that allows the user to benchmark a method with a list of metrics in different datasets.
- **CLI**: contains the CLI that allows the user to use the library from the command line.

Following the modularity principle, each module tackles a specific aspect of the forgery detection pipeline. The *Datasets*, *Preprocessing*, *Methods*, and *Metrics* are designed to work in unison, but each of the modules can be used independently from each other. *Postprocessing* groups useful functions used at the end of a method's pipeline, so the *Methods* module depends on it. The *Benchmark* and the *CLI* are both designed to run image forgery pipelines, so naturally, they both make use of all the modules.

6.4.1 Datasets

The *Datasets* module contains a compilation of popular datasets that are used to evaluate methods. In the library, a Dataset is a class with instructions to find and load the images of a dataset.

Following OOP principles, we define a *BaseDataset* to which we add common attributes and methods that any dataset might use. In particular, the data loading logic is implemented in a way that is reusable for all datasets, needing only to override simple properties and methods to define the folder structure. Some methods, like the ones that define the folder structure, need to be overwritten when creating a new Dataset, while others should be overwritten when the default implementation is not fit for the dataset, like mask binarization. An important note is that the *BaseDataset* inherits from *Pytorch*'s dataset, meaning any *PhotoHolmes*' dataset can be used within that framework. The code of the base class and an example of how to create a dataset within *PhotoHolmes* can be found in Appenddix A.1.1 and Appendinx A.1.2 respectively.

From the survey of the state of the art, we identified three types of image data that methods use: the image itself, the DCT coefficients, and quantization tables (`qtables`) of JPEG images (Section 2.1.4.2). With that in mind, our datasets can load the three types of data, specified either through the `load` parameter or by the preprocessing pipeline's input, which will be defined in the next section. Another important parameter is `tampered_only`, which allows the user to specify whether they want all the images to be loaded or only those where a forgery does exist.

The first release of *PhotoHolmes* includes 7 benchmarking datasets: Columbia [45,89], CASIA 1.0 [32,64,89], DSO-1 [29,89], Korus [50,51], AutoSplice [49] and Trace [15]. On top of the original versions, we include the social media versions of Columbia, DSO-1, and CASIA v1, as well as a WebP compressed version of Korus and Columbia. As mentioned in Chapter 3, the selected datasets cover a wide range of forgery types and image formats, which we deemed important to benchmark the diverse array of included methods. The most relevant information about the included datasets is summarized in Table 3.1 on Section 3.2.10.

Following the design principles from which we built *PhotoHolmes*, using the included datasets is a straightforward process. For example, to use the Columbia dataset and get the first image of the dataset, the following code snippet can be used:

```python
from photoholmes.datasets.columbia import ColumbiaDataset
from photoholmes.utils.image import plot

# Load the dataset
dataset_path = "data/Columbia"
dataset = ColumbiaDataset(
    dataset_path=dataset_path,
    preprocessing_pipeline=None,
    tampered_only=True,
    load=["image"]
)

# Get the first image
data, mask, image_name = dataset[0]
image = data["image"]
plot(image)
```

On top of the dataset definitions, the module contains a registry that lists the available datasets, as well as a factory that allows the user to easily load any of the available datasets. Using the factory, the code to load the Columbia dataset becomes:

```
from photoholmes.datasets import DatasetRegistry,
   DatasetFactory

# Load the dataset
dataset_path = "data/Columbia"
dataset = DatasetFactory.load(
    DatasetRegistry.COLUMBIA,  # can use "columbia" instead
    dataset_path=dataset_path,
    preprocessing_pipeline=None,
    tampered_only=True,
    load=["image"]
)
```

Given the simple extensibility achieved with how the module was designed, we expect to continue growing the *PhotoHolmes* dataset registry as new datasets are proposed.

6.4.2 Preprocessing

Most forgery detection methods work on transformations of the image data rather than the image itself. Some transformations are simple, for example, a grayscale transformation, while others require more complex operations like computing the DCT volumes in CAT-Net (see Section 5.5). To give structure to these transformations, we define the *Preprocessing* module.

Within this module, we define a `BasePreprocessing` class. This class is extremely simple, only requiring children classes to implement the `__call__` method, as can be seen in Appendix A.2.1. In order to allow preprocessing operations to be mixed and matched in different pipelines, each preprocessing operation expects a dictionary as input and outputs a dictionary. The preprocessing operations can modify, add, or remove entries in this dictionary as long as they are composed in a compatible fashion. In the first release of *PhotoHolmes*, we include the preprocessing operations:

- `ZeroOneRange`: changes the image pixel values from $[0, 255]$ to $[0, 1]$.
- `Normalize`: applies standardization to the image by subtracting the mean and dividing by the standard deviation.

- `RGBtoGray`: converts an image from the RGB colorspace to grayscale.
- `GraytoRGB`: converts the image from grayscale to RGB.
- `RoundToUInt`: rounds the input float tensor and converts it to an unsigned integer.
- `ToNumpy`: converts tensors to *numpy* arrays.
- `ToTensor`: converts a *numpy* array to a *Torch* tensor.
- `GetImageSize`: adds the size of the image to the dictionary.

Some methods require preprocessing operations outside this list, but given their specificity, we opted to define them within the method's module. Each method has a `preprocessing.py` file where custom preprocessing operations can be defined, and more importantly, the pre-processing pipeline is defined.

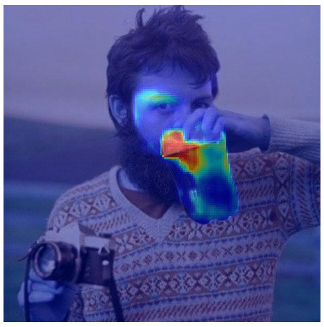
The `PreprocessingPipeline` is a class that sequentially runs a list of preprocessing operations on the input data and leaves it ready for the method to intake. The pipeline is designed to be easy to use, simplifying the composition of transforms and controlling the input to the model. The `PreprocessingPipeline` requires three parameters: the list of preprocessing operations, the input keys that need to be included in the initial dictionary provided to the pipeline, and the output keys that the method expects. The input keys are used to validate the pipeline input, avoiding obscure errors when used incorrectly, and are also used by the *Datasets* to load only the necessary image information. The output keys are used to filter out any extra keys that were left over during the preprocessing operations. The implementation of the class is in Appendix A.2.2.

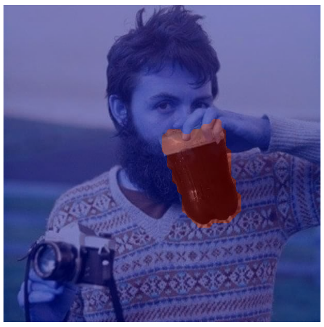
Here is a code snippet that implements a simple preprocessing pipeline. It expects an image, converts it to a numpy array, and then converts it to grayscale.

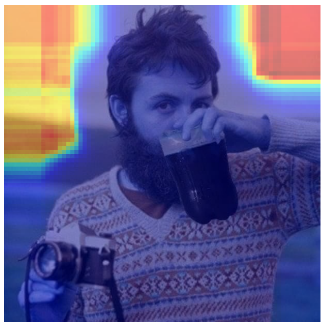
```python
from photoholmes.preprocessing import ToNumpy, RGBtoGray,
    PreProcessingPipeline
from photoholmes.utils.image import read_image

pipeline = PreProcessingPipeline(
    transforms=[ToNumpy(image_keys=["image"]), RGBtoGray()],
    inputs=["image"],
    outputs_keys=["image"]
)

image = read_image("example_image.jpeg")
result = pipeline(image=image)
```

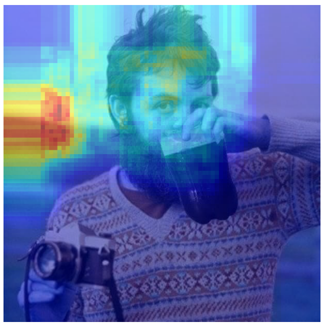
6.4.3 Methods

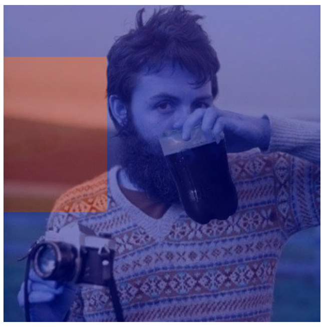
The *Methods* module is the core of the library, and all the modules are designed around it. As we mentioned in the introduction (Section 6.1), the implementations of forgery detection methods are diverse in code structure, programming style,

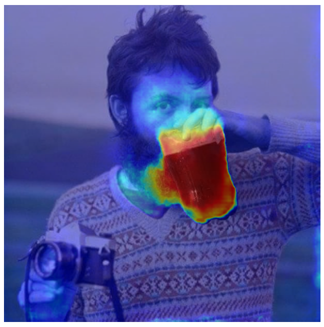

(a) CAT-Net 5.5

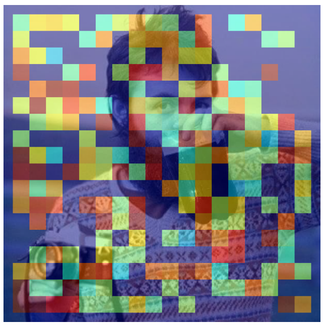

(b) FOCAL 5.10

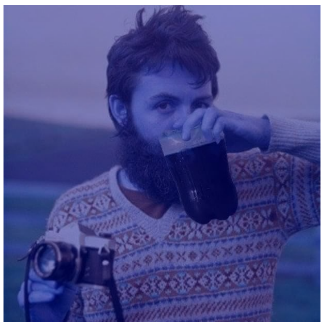

(c) Splicebuster GU 5.1

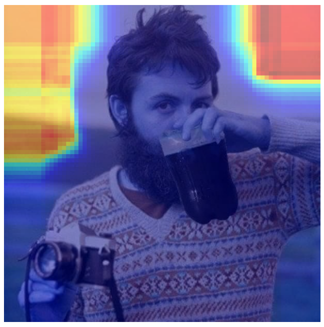

(d) Splicebuster GG 5.1

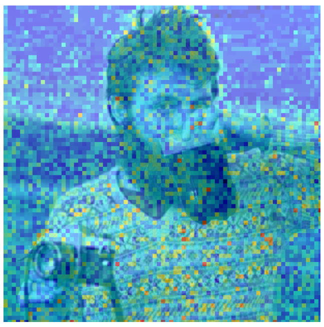

(e) Noisesniffer 5.2

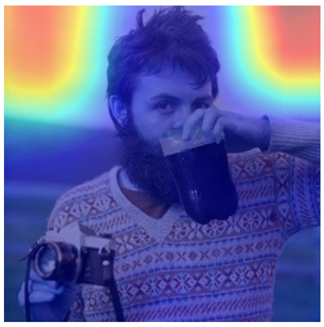

(f) Trufor 5.7)


(g) AdaptiveCFA 5.9


(h) ZERO 5.4


(i) PSCC-Net 5.8


(j) DQ 5.3


(k) EXIF MS 5.6

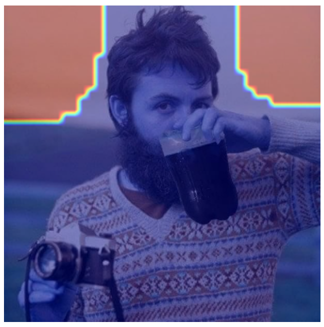

(l) EXIF Ncuts 5.6

Figure 6.1: Results of running all the methods included in *PhotoHolmes* on the image presented in Figure 2.9, showing the overlay of each method's output over the original image, the former being predicted masks or heatmaps accordingly.

inputs, outputs, and documentation, making it difficult to run quick inference or evaluation unless the authors provide specific scripts for it.

To address this issue, following OOP principles, we designed a *BaseMethod* class that all methods inherit from, and that ensures compatibility with the rest of the modules provided in the library. Additionally, we defined *BaseTorchMethod*, meant to be used by those methods that rely solely on neural networks. This way, the methods are compatible with *PyTorch* and can be used for retraining and other experiments. As with the *Datasets*, the *BaseMethod* includes default implementations of some functionalities like loading from a configuration file, and it requires the user to implement the `benchmark` function[1]. This function will be used by the *Benchmark* module we will later introduce, simplifying a method's evaluation process. Another notable function[2] is `to_device` that allows the user to move the method into a device optimized for tensor acceleration, such as a GPU, especially useful for deep learning methods. The code for both base classes is included in Appendix A.3.1.

Apart from the base definitions, the first version of *PhotoHolmes* contains the implementation of ten image forgery detection methods: Adaptative CFA [16], Noisesniffer [38], ZERO [62], DQ [56], CAT-Net [54], Splicebuster [27], EXIF as a language [96], PSCC-Net [57], TruFor [42] and FOCAL [88]. For details about any of these methods, refer to Chapter 5. In Figure 6.1, you can see the results of running all the methods on the forged image in Figure 2.9.

To provide an example of how the methods module can be used, we provide a code snippet to run CAT-Net on an image. In a few lines of code, we instantiate the method, change the device to GPU for a faster inference, and run the method on the image.

```
from photoholmes.methods.catnet import CatNet,
    catnet_preprocessing
from photoholmes.utils.image import read_image,
    read_jpeg_data

path_to_image = "path_to_image"
image = read_image(path_to_image)
dct, qtables = read_jpeg_data(path_to_image)

# Preprocess data
image_data = {"image": image, "dct_coefficients": dct, "
    qtables": qtables}
input = catnet_preprocessing(**image_data)

# Declare the method and use .to_device if you want to run it
     on cuda or mps instead of cpu
arch_config = "pretrained"
path_to_weights = "path_to_weights"
method = CatNet(
```

[1]This function is a method of the object. We chose not to use the word method for clarity since we were discussing forgery detection methods.

[2]Also a method of the object

```
    arch_config=arch_config,
    weights=path_to_weights,
)
device = "cuda"
method.to_device(device)

# Use predict to get the final result
output = method.predict(**input)
```

Just as in the *Datasets* module, the *Methods* module includes both a factory and a registry that simplifies loading a model. The registry contains a list of all the available models, while the factory loads the method and associated preprocessing pipeline.

```
from photoholmes.methods import MethodFactory, MethodRegistry
from photoholmes.utils.image import read_image,
   read_jpeg_data

path_to_image = "path_to_image"
image = read_image(path_to_image)
dct, qtables = read_jpeg_data(path_to_image)
image_data = {"image": image, "dct_coefficients": dct, "
   qtables": qtables}

# Load the method
method, preprocessing = MethodFactory.load(
    MethodRegistry.CATNET,
    {"weights": "path_to_weights", "arch_config": "pretrained
   "},
)
device = "cuda"
method.to_device(device)

# Preprocess the input
input = preprocessing(**image_data)

# Use predict to get the final result
output = method.predict(**input)
```

Newer releases of *PhotoHolmes* will include more methods according to the demand of the forgery detection community.

Not all the methods included in *PhotoHolmes* have a commercial license. In an effort to include as many methods as possible while respecting the original author's rights, the decision was reached to include the original License inside the method's folder and to log warning messages advising the user to check whether the specific method's license is within their scope of use.

6.4.4 Postprocessing

Postprocessing is a common step in image forgery detection, as many methods employ sliding window predictions or other sub-sampling strategies that yield a prediction smaller than the input image or have an output that has to be rescaled to a different dynamic range. Given that most methods employ at least one of these functions, a module was created to centralize them for reusability.

Another commonly applied post-processing is casting types and moving data across devices (ie. GPU to CPU). While the method itself might not need this for prediction, they are useful to integrate with other parts of the *PhotoHolmes* library or even third-party libraries.

Having identified the two main uses for postprocessing, the first version of *PhotoHolmes* includes the following postprocessing functions:

- `to_device_dict`: moves dictionary values to the specified device.
- `to_tensor_dict`: converts dictionary values to tensors.
- `to_numpy_dict`: converts dictionary values to NumPy arrays.
- `zero_one_range`: rescales the output to $[0, 1]$.
- `resize_heatmap_with_trim_and_pad`: zero-pads or trims the heatmap to match the original image size.
- `upscale_mask` and `simple_upscale_heatmap`: interpolates the mask or heatmap to match the original image size.

Unlike the previously introduced modules, *Postprocessing* doesn't have a base class that gives structure. This decision was taken to simplify the image forgery pipeline, choosing to have the method output a comparison-ready mask rather than having to include instructions or a pipeline to transform the output. However, some methods may need custom post-processing, as is the case for Splicebuster. In these cases, there are no restrictions regarding the structure of the postprocessing, but we choose to have the code inside a `postprocessing.py` file.

6.4.5 Metrics

As covered in Chapter 4, metrics are essential to evaluate a method's performance. Unlike the previous modules, the *Metrics* module builds on top of the popular *Torchmetrics* [31] library. This means that any metric implemented in *Torchmetrics* is compatible with *PhotoHolmes.* This library includes implementations for the most popular metrics and is easily extensible with new metrics.

Most metrics use a different naming than the one we used in this thesis, so we decided to create wrappers with our naming conventions for all the metrics. The module also includes implementations for the FPR (Section 4.4.2), the mAuroc, and the weighted metrics introduced in Section 4.4.9. Table 6.1 summarizes the metrics included in the first version of *PhotoHolmes.*

As was the case with the *Datasets* and *Methods* modules, by following the design principles upon which we built *PhotoHolmes*, using one of the metrics included in *PhotoHolmes* is a straightforward process. For example, to utilize the weighted IoU on its first version, the following code snippet can be employed:

Implementation	Dataset-level score	Average image-level score
Torchmetrics	ROC, AUROC, TPR, Precision, F1, IoU, MCC	-
Custom	$\mathrm{F1}_w^{v2}$, IoU_w^{v2}, MCC_w^{v2}, FPR	$\mathrm{F1}_w^{v1}$, IoU_w^{v1}, MCC_w^{v1}, mAUROC

Table 6.1: Breakdown of the metrics included on the first release of *PhotoHolmes*.

```python
from photoholmes.metrics import IoU_weighted_v1
import torch

iou_weighted_v1_metric = IoU_weighted_v1()

# Generate random data
data = [
    (torch.rand(256, 256), torch.randint(0, 2, (256, 256)))
    for _ in range(10)
]

# Update the metric for each image
for pred, mask in data:
    iou_weighted_v1_metric.update(pred, mask)

# Compute the final value
iou_weighted = iou_weighted_v1_metric.compute()

print("IoU_weighted_v1:", iou_weighted)
```

The *Metrics* module also contains a registry of the available modules, as well as a factory that allows the user to load any of the registered metrics easily. Unlike the previous factories, the *Metrics* factory can receive a list as input, returning the collection of metrics requested in one simple call.

```python
from photoholmes.metrics import MetricFactory, MetricRegistry

# load metrics
metrics = MetricFactory.load(
    [
        MetricRegistry.IoU,
        MetricRegistry.F1,
        MetricRegistry.mAUROC
    ]
)

...
# evaluating predictions
metrics.update(preds, mask)
```

6.4.6 Benchmark

This module aims to solve one of our initial problems, which was the lack of uniform and reproducible evaluations of the state of the art. The *Benchmark* module, which consists of a single *Benchmark* object, is designed to work seamlessly with any method, dataset, and metric that was implemented using *PhotoHolmes*.

While one could build a custom benchmark script to run a method over a dataset, our module includes some useful functionalities that simplify the process. The most useful one is saving the outputs in a compressed *npz* format, making it possible to resume a benchmark process if it was interrupted or quickly re-run the outputs on a different set of metrics without running the method. It is important to note that the benchmark process can take a long time, especially if the method in place is slow and the dataset has a lot of images. Since batching requires the images to be of the same size, it is not possible to optimize inference speeds with this technique since resizing and cropping can destroy useful traces in the image.

The *Benchmark* object is used in two steps. Firstly, we need to instantiate the class with the configurations the benchmark will follow. These configurations include the device to run on, whether to store outputs, re-use stored outputs and where to save them, and controlling the verbosity of the logging. Once we have our benchmark instance, we can call the `run` method, providing it a method, a dataset, and the set of metrics to run on. It is in this function that the `benchmark` method required by the *BaseMethod* (see Section 6.4.3) is used. Figure 6.2 illustrates the described end-to-end benchmarking pipeline.

During the state-of-the-art review, we identified three types of method outputs: heatmaps, binary masks, and detection scores. As such, we introduce this notion to our benchmark process, expecting a method to output at most one of each category. This way, we create an interface for the benchmark and the methods to interact. Each one of these output types is evaluated on a different set of metrics, so if our method has the three types of output, at the end of the benchmark process, we are left with three metric reports. A metric report is a JSON file, stored within the output folder, where the metric results for the method are dumped.

Once finished, the benchmark results will be in the output folder selected when creating the *Benchmark* objects, which defaults to `output`. Inside this folder, the following structure is present:

```
output/
└─{method}/
   └─{dataset}/
      ├─metrics/
      │  └─{timestamp}_{dataset_mode}/
      │     ├─{output_type_1}_report.json
      │     └─{output_type_2}_report.json
      └─outputs/
         └─...
```

The following code snippet provides an example of using the *Benchmark* module by concatenating all of the other modules included in *PhotoHolmes*. The examples showcase the benchmarking of DQ in Columbia with AUROC and F1 by using the corresponding factories.

```
from photoholmes.datasets import DatasetFactory,
    DatasetRegistry
from photoholmes.metrics.factory import MetricFactory
from photoholmes.methods import MethodFactory, MethodRegistry
from photoholmes.benchmark import Benchmark

# Load the dataset
dataset = DatasetFactory.load(
    DatasetRegistry.COLUMBIA,
    dataset_path=columbia_dataset_path,
    load=["image", "dct_coefficients"],
    preprocessing_pipeline=dq_preprocessing,
)

# Load the metrics
metrics = MetricFactory.load(["auroc", "f1"])
print(metrics)

# Load the method
dq, dq_preprocessing = MethodFactory.load("dq")

# Create the Benchmark object
benchmark = Benchmark(
    save_method_outputs=True,
    save_extra_outputs=False,
    save_metrics=True,
    output_folder="example_output",
    device="cpu",
    use_existing_output=False,
    verbose=1,
)

# Run the benchmark
benchmark.run(method=dq,
    dataset=dataset,
    metrics=metrics
)
```

In the repository, there is a *Jupyter Notebook* inside the `notebooks` folder with a step-by-step guide on how to benchmark a method.

6.4.7 Command Line Interface (CLI)

As mentioned in the introduction, one of the design principles of the library was usability. Following this principle, a Command Line Interface (CLI) was developed to ease the user experience. In the first version of the library, the CLI contains three commands: `run`, `download_weights`, `adapt_weights`.

The `run` command allows the user to run a method in a single image and

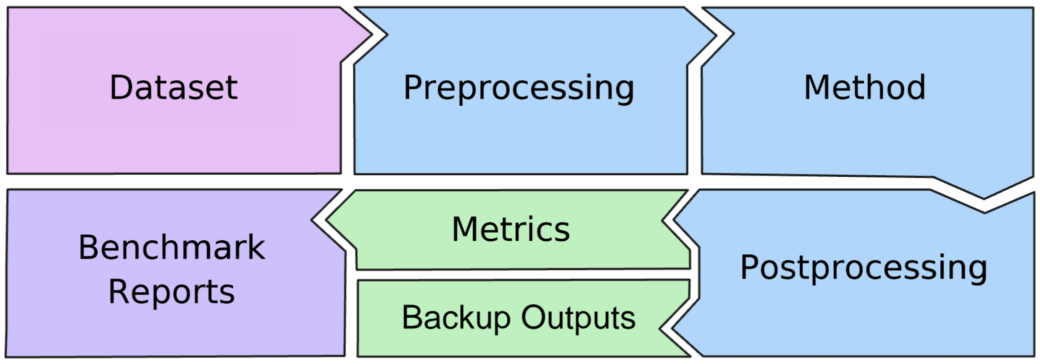


Figure 6.2: Benchmark class flow diagram. Everything starts by choosing a dataset and a method, then according to the chosen method, the dataset is preprocessed with the corresponding preprocessing. Then, outputs can be visualized, and chosen metrics are computed. The metrics are then stored as benchmark reports.

check the results without writing code. Each method has its sub-command and can expect more arguments (for instance, the path to the pre-trained weights in the case of learning-based methods), but they all share the following arguments and options:

- Arguments
  - `image_path`: path to the image to run the method on.
- Options
  - `output-folder`: path to a folder where to save the method outputs. If no path is provided, then the outputs are not saved.
  - `overlay`: flag that if set, a plot with the mask or heatmap overlayed on the image is included.
  - `show-plot` / `no-show-plot`: whether to show results as a matplotlib plot.
  - `device`: torch device to run the methods on. Only available in methods that use neural networks.

In Figure 6.3, we present the output of running CAT-Net on the forged image from Figure 2.9.

The `download_weights` provides a simple interface for a user to download the model's weight for a deep learning method. This command takes a method as an argument and has the option to choose the folder where the weights are downloaded. As mentioned before, some of the methods included in *PhotoHolmes* have their weights licensed to be used only in research contexts. When this is the case, the CLI will display a warning and require the user's input on whether they accept those terms or not, protecting the original author's rights.

Lastly, the `adapt_weights` script modifies original model weights to align with *PhotoHolmes'* methods implementations. Some methods have been adjusted to

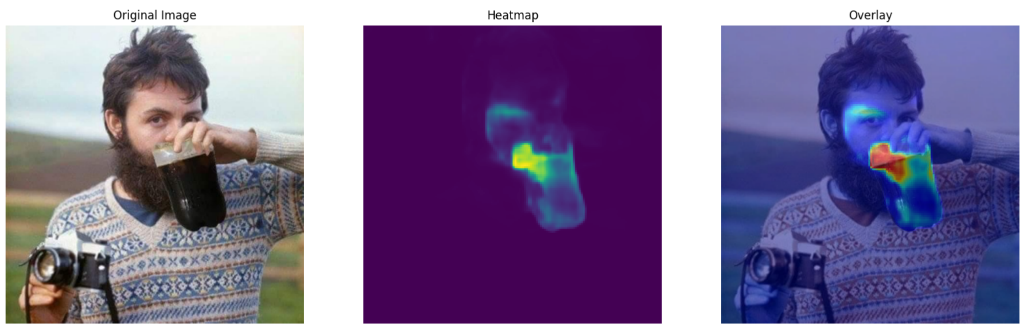


Figure 6.3: Output of running `photoholmes run catnet <image_path> --overlay` using the photoholmes CLI. The forged image is the one presented in Figure 2.9.

eliminate unnecessary structures within the architecture and remnants from applying transfer learning. This streamlines the model, making it more efficient and suitable for the intended tasks. For example, EXIF as Language [96] inherits from *OpenAI*'s Clip model [72], yet overrides some of its properties and leaves some structures unused. Our scripts remove any unused modules for a cleaner implementation.

In this chapter, we have introduced the PhotoHolmes library, a novel Python library for digital image forgery detection. The library includes a diverse array of state-of-the-art methods for detecting forgeries in digital images, as well as a variety of datasets and metrics for benchmarking these methods. The library is designed to be modular, reproducible, extensible, and easy to use. It is subdivided into 7 different modules, each of them with a specific purpose. These modules are Datasets, Preprocessing, Methods, Postprocessing, Metrics, Benchmark, and CLI.

In the next chapter, we exploit one of the PhotoHolmes library's capabilities, specifically the utility of the benchmark module. Using this tool, we have performed a thorough analysis of the different methods in the library, comparing their performance in different situations and metrics. This evaluation has enabled us to provide a detailed evaluation of each method, highlighting their strengths and weaknesses, as well as identifying the specific contexts in which they are most effective.

This page intentionally left blank.

Chapter 7

Evaluation

"There is nothing like first-hand evidence."

Sherlock Holmes

The preceding chapter introduced the PhotoHolmes library, providing users with the capability to benchmark various methods using a diverse range of datasets and metrics. In this chapter, we conduct a detailed evaluation of the ten methods included in PhotoHolmes, which cover a wide range of approaches to forgery detection in digital images, in order to highlight their strengths and weaknesses. The chapter is structured as follows: first, we provide a brief overview of the methods selected for analysis. Next, we describe the metrics and datasets used. Lastly, we present the evaluation and its results. The evaluation is split into two parts: the non-semantic evaluation, which identifies specific traces targeted by each method, if any, and the popular datasets evaluation, which benchmarks the methods against widely used datasets. By doing the non-semantic evaluation first, the trace-related information found allows us to further explain some results in the popular datasets evaluation.

7.1 Recapitulation and evaluation notes

As mentioned throughout this thesis, the objective of this review of the state of the art in forgery detection is to understand current technologies' capabilities, strengths, and weaknesses, as well as what follow-up research can be done to mitigate these shortcomings. Before conducting the evaluation, we will briefly recap the methods and datasets mentioned in the previous chapters and discuss the selection of metrics for this evaluation. Regarding the latter, some clarifications are made on the decisions that the team made.

7.1.1 Benchmarked methods

The methods chosen for study in this work were presented in Chapter 5 and are summarized in Table 7.1. These ten methods were chosen based on their per-

Method	Target traces	Deep-learning	Outputs		
			Heatmap	Mask	Detection
Adaptive CFA [16]	CFA	✓	✓	✗	✗
Noisesniffer [38]	Noise	✗	✗	✓	✓
ZERO [62]	JPEG	✗	✗	✓	✓
DQ [56]	JPEG	✗	✓	✗	✗
CAT-Net [54]	JPEG	✓	✓	✗	✗
Splicebuster [27]	Multiple	✗	✓	✗	✗
EXIF as language [96]	Multiple	✓	✓	✓	✓
PSCC-Net [57]	Multiple	✓	✓	✗	✓
TruFor [42]	Multiple	✓	✓	✗	✓
FOCAL [88]	Multiple	✓	✗	✓	✗

Table 7.1: Summary of the target traces of each method included in the first release of *PhotoHolmes* as well as the kind of output they provide. The outputs can be continuous heatmaps representing probability, binary masks, and detection scores.

formance, relevance, and complementarity, as the goal is to have a diverse set of methods to contemplate the different types of forgeries covered in Chapter 2. We also divide the methods into four categories: demosaicing-based, noise-based, JPEG-based, and multi-purpose methods, as can be seen in the table, along with the types of output the method yields.

All of them solve the localization problem, either through a heatmap or a mask, with some cases returning both. To be able to compare the results to the ground-truth mask, we need the heatmaps and masks to match the original image size, so postprocessing might be needed to get the output to the correct size. In the cases where the authors included that operation in their code, we followed it, but in other cases, we had to opt for either a simple upscale using interpolation, padding, or both. For example, when the method yields a heatmap where each pixel corresponds to a DCT block of dimensions 8×8, the upscale is done by a factor of 8 in both directions so that all pixels corresponding to that block have the same value. Then, if the image dimensions differ with this resize by less than 8, it is padded with zeros or trimmed as necessary to match the dimensions.

Additionally, certain methods tackle the detection problem, yielding a detection score alongside the localization output. For ZERO and Noisesniffer that generate a mask, we decide to generate a detection output as follows: if any pixel in the mask appears as forged, then the detection is set to 1; conversely, if the mask consists entirely of zeros, the detection output is 0.

Some important notes have to be made about some of the methods. Firstly, we performed two localization evaluations for the method ZERO (Section 5.4).

One of them, which we will refer to as “ZERO” from now on, is the basic method that only searches for misalignments in the JPEG grid. The other one corresponds to an extension of the method proposed by the authors, which also searches for missing grids, called “ZERO with missing grids” throughout this chapter. In the case of detection, we only considered ZERO since the authors reported that the missing grids version tends to introduce false positives.

By looking at the table, it is easy to see that EXIF as Language (Section 5.6) is the only method that generates all three types of output. Regarding localization, this method returns the heatmap when running the Mean-Shift algorithm over the features and a mask when using NCuts clustering, so we include both evaluations in the reports. However, in the case of detection, a modification was made to the original implementation. The authors proposed to use the mean of the Mean Shift heatmap as a detection score, but we found this could be misleading, especially in images where the forgery is small proportionally to the image where a few high-confidence predictions can be diluted by numerous pixels with low probability. The authors themselves noted that the values were usually quite low and opted to analyze the score only by determining whether it was high or low. Still, they do not provide any threshold to establish a fair comparison and even discourage the usage of the score itself. To address this, we propose a different approach based on how the detection output is handled in ZERO and Noisesniffer.

We also remind the reader that although DQ (Section 3) outputs only a heatmap, the original implementation additionally outputs a mask. This mask is obtained by segmenting a set of features extracted from the heatmap with an SVM, which we could not carry out due to a lack of implementation details. Regarding this chapter, it is important to consider that the final output from the SVM may achieve better results than the implementation used in the benchmark.

Before moving on to evaluation, we must define the term *hallucination*. In this work, *hallucination* will specifically refer to false positive localizations, but only in instances where the model identifies a forgery in a region of the image that is entirely authentic or in an image that contains no forgery whatsoever. This term will not apply to cases where the model correctly detects a forgery but erroneously extends this detection to small adjacent areas, nor does it cover false negatives, where the model fails to identify actual forgeries.

7.1.2 Metrics

As presented in Chapter 4, the metrics need to be carefully selected to be able to correctly describe the methods’ performance and ensure a fair comparison. In this evaluation, we need metrics that can measure the performance of the two problems, localization and detection, and also take into account the different types of outputs that the methods can have (both heatmaps and masks). To this end, from all the metrics described in this chapter and later implemented in the *PhotoHolmes* library, we selected seven of them to report in this work. The selected metrics are the weighted Matthews Correlation Coefficient (MCC_w), Intersection over Union (IoU_w), and $\mathrm{F1}_w$, in both average image-level (v_1) and dataset-level (v_2) versions, and the Area Under the ROC (or AUROC).

As in the previous section, there are some comments to be made on implementation decisions of the metrics used in this evaluation. Regarding the AUROC, it is important to note that if the prediction is a mask, the ROC curve is a single point, and thus, the AUROC lacks meaning. Having said that, the implementation of the AUROC included in *Torchmetrics* outputs a value for these cases by generating a ROC with three points: one point being the $(0, 0)$ which implies an empty mask, the (FPR, TPR) where FPR and TPR are calculated with the mask as is, and the $(1, 1)$ which is obtained when the mask is all ones. Although this workaround exists, it will not be used for masks in this work because, as already mentioned, it makes no theoretical sense to calculate the AUROC for these cases. With this consideration in mind, the AUROC is reported only for methods that output a heatmap for localization. Although AUROC does not permit a comparison between all evaluated methods, it is still included because it is one of the go-to metrics authors use to evaluate the methods' performance.

As was presented in Section 4.3.2 and Section 4.4.9, there are two versions of the weighted metrics, the average image-level and the dataset-level. The image-level metrics will be used to evaluate the localization problem, as it better captures how the methods perform in each image, while the dataset-level version will be used for the detection problem since the image-level does not make sense when the output is a single number.

From the previously mentioned weighted metrics, the MCC is the least dependent on the proportion of positive values to negatives in the ground truth [21]. Unlike IoU and F1 Score, which only penalize false negatives (FN) and false positives (FP), MCC takes into account the true negatives (TN), thus providing a balanced assessment of a method's performance across all categories, considering both positive and negative classifications. This makes the MCC particularly effective in penalizing the overestimation of forged areas. In the case of an overestimation, the IoU and the F1 will only take into account the appearance of more false positives; however, the MCC will also take into account the decrease in true negatives, which will act as a double penalty. This is why the MCC will be regarded as the most complete and rigorous metric reported when evaluating the localization problem. Coincidentally, when running detection on a dataset that only has tampered images, the MCC will not be reported as it is undetermined due to a lack of true negatives and false positives.

Finally, a design decision regarding the implementation of the weighted metrics should be noted. When evaluating pristine images, where the ground-truth mask is all zeros, if the prediction is only zeros, then the TP, FN, and FP are all zero, resulting in an indetermination of the definition of these metrics. Unfortunately, *Torchmetrics*, which is our framework of reference for the metrics, does not have a standard strategy for handling these cases. In the case of the F1 score, it reports a 0, in the case of the IoU it reports a NaN, and in the case of MCC, it yields a 1. This lack of standardization left us with a choice of how to handle this case when designing our custom-weighted metrics.

For the weighted MCC, we chose to follow *Torchmetrics*'s MCC by returning a 1, the highest value, if both the prediction and target are all zerosand thus are identical. In the case of the IoU and F1 score, to avoid getting NaN values during evaluations, we opted for *Torchmetric*'s F1 solution, returning a 0 if both the prediction and ground truth are all zeros.

7.1.3 Datasets

The last piece of the puzzle is the datasets we are going to use, which, as described in Chapter 3, fall under two categories. The first category consists of non-semantic datasets, which make up miniTrace, a set of synthetic datasets created by applying different pipelines (Section 2.1.1) to raw images in a controlled environment, resulting in images that have distinct image traces. This allows us to analyze what type of traces, if any, each method exploits to make its prediction. As mentioned in Section 3.2.7, the datasets within miniTrace are divided into exogenous and endogenous datasets, so the tables presented in this chapter report two values for each entry. Exogenous datasets will be reported in blue and the endogenous in green.

The second category is the popular datasets, which are datasets that are used in the literature and contain a mix of tampered and pristine images. These datasets are Columbia, CASIA 1.0 v1, Coverage, DSO-1, Korus, and AutoSplice. When evaluating these datasets, the tables will report three values: the original dataset with all images (tampered and pristine) in blue, the original dataset with only the tampered images in green, and the dataset with only tampered images that went through *Facebook* in purple whenever it is available. This distinction is made for several reasons:

- We want to study how well a method can locate forgeries; that is why we are using datasets with just tampered images.
- It is important to analyze whether a method is able to detect whether a forgery is present in an image or not. To this end, the comparison between using tampered and pristine images and only tampered images is needed.
- We want to evaluate how sharing forged images over social media might affect the methods' performance, for which a comparison with the *Facebook* datasets is required. However, the *Facebook* versions only include tampered images, so the results can only be compared with the results of running the method only on the tampered images in the original dataset.

One last detail is that Autosplice's 90-quality and 75-quality versions (see Section 3.2.6) only include tampered images, so only these results will be available.

7.2 Non-semantic evaluation

This section reports and analyzes the results obtained from the non-semantic evaluation of the methods on the miniTrace dataset for both localization and detection

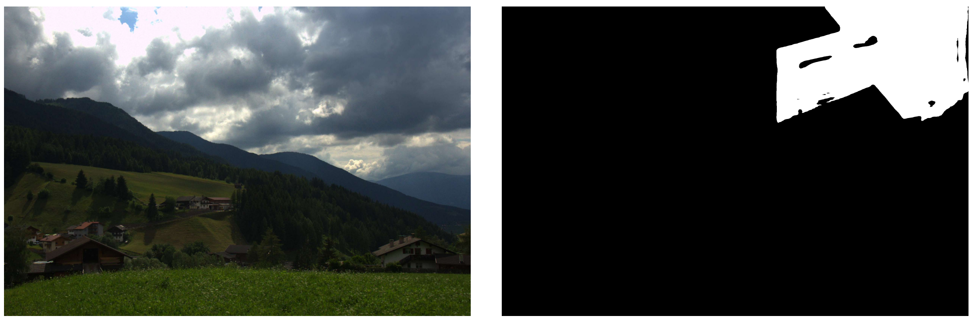

(a) Forged Image (b) Ground Truth Mask

Figure 7.1: Non-semantic sample image (a) with it's exo-mask (b).

problems. The outputs of every method on a set of images from the miniTrace dataset are shown in Figure 7.2.

7.2.1 Localization

Figure 7.2, which shows a single example from the dataset, reflects the localization performance results obtained across the entire miniTrace dataset. These results are reported in Tables 7.2, 7.3, 7.4 and 7.5.

The first notable observation to be made is that Adaptive CFA, Noisesniffer, and ZERO stand out for their effectiveness in detecting the inconsistencies within the traces they were designed to exploit, as can be visualized in Figure 7.3. This is especially evident for ZERO and Adaptive CFA, which achieve the highest scores in JPEG and CFA traces, respectively, with a significant margin ahead of the second best. This exceptional performance is reflected in the figure, wherein the CFA datasets, only Adaptive CFA can capture both the algorithm and grid inconsistencies, ZERO gets the clearest mask in JPEG, and Noisesnifffer is the only one that gets a mask close to the original in the noise traces.

Regarding ZERO, the performance in the JPEG Grid dataset is unsurprising, given that the method is designed to detect misalignments in the JPEG grid. The excellent performance on the JPEG Quality dataset might come as a surprise, but is explained by the choice made by the Trace authors (Section 3.2.7) to not only modify the quality factor in between camera pipelines but also to include a different grid origin to simulate splicing. This enables ZERO to detect this inconsistency in the image. It should also be noted that when there is no JPEG grid misalignment ZERO does not report any forgeries, in other words, ZERO does not have any false positives. This affirmation can be distilled from the tables and observed in the figure.

An additional analysis regarding ZERO comes from a comparison between ZERO and ZERO with missing grids. Firstly, ZERO with missing grids obtains the same results as ZERO in the Noise and CFA datasets, which means that even with this new step, ZERO does not detect anything in datasets without JPEG compression. Regarding the JPEG and Hybrid datasets, we can observe a decrease

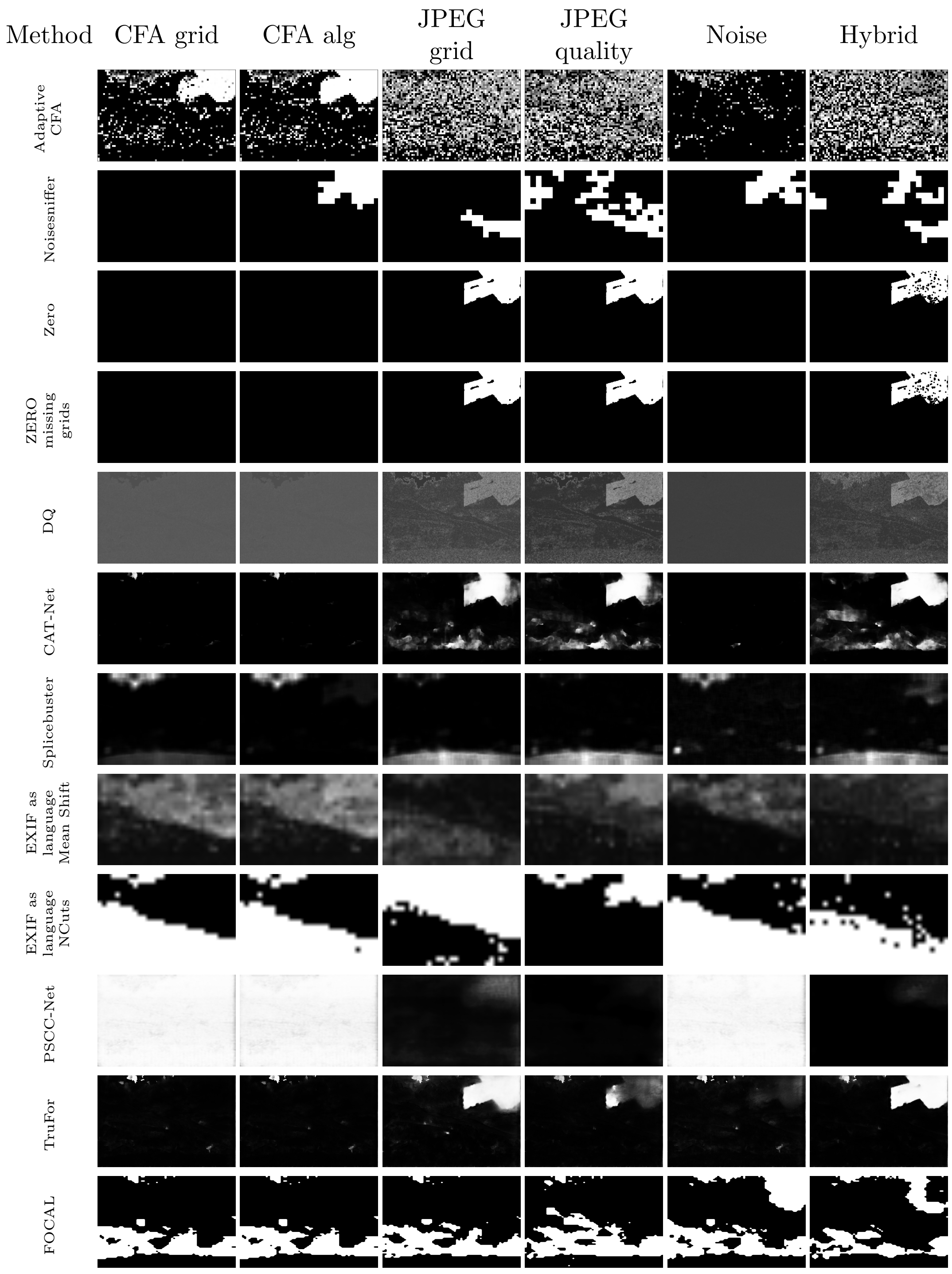


Figure 7.2: Outputs in miniTrace datasets of all of the evaluated methods. Original image and corresponding mask are shown in Figure 7.1.

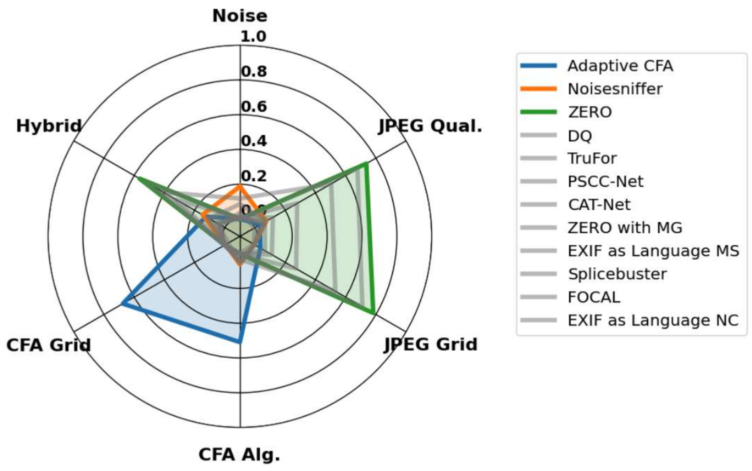


Figure 7.3: A spider diagram illustrating the $\text{MCC}_w^{v_1}$ on miniTrace dataset for the top three performing methods, according to Table 7.2. For each method, it shows the absolute value of MCC on every axis, where each axis is a different dataset. Each method of the top three performing methods is plotted with a different color, and a gray shade is used for the remaining methods. The purpose of this figure is to show how different traces are best exploited by each of the corresponding classical methods.

in performance when considering the missing grids, which might mean that when adding the detection of the missing grids, the mask grows into non-forged areas. However, even with the diminished performance, ZERO with missing grids is still the second-best performer in the aforementioned datasets.

Adaptive CFA not only performs remarkablely in CFA datasets but also outperforms all methods in those datasets. This was predictable as its architecture is designed specifically to detect this type of inconsistency, paired with the fact that most multipurpose methods do not take into account demosaicing inconsistencies. As mentioned in Section 5.9, Adaptive CFA is trained upon a synthetic demosaicing pipeline with several known demosaicing algorithms, which mimic the demosaicing process used to generate miniTrace. It is yet to be seen how this method performs in more complex, unknown demosaicing algorithms that may be found in the wild. However, the performance in the Hybrid dataset is a bit low as this dataset includes manipulations of the CFA algorithm and/or grid, but it might also have a JPEG compression. This is coherent with the observation made in Section 5.9 that Adaptive CFA lowers its performance in the presence of JPEG compression.

In the case of Noisesniffer, a noteworthy remark is how, as expected, it excels in detecting noise traces, judging by the $\text{MCC}_w^{v_1}$ (Table 7.2) score on the Noise datasets, where it gets the best score out of all methods. Despite being the highest score, it is lower when compared to ZERO on its target forgery. As mentioned in Section 5.2.2, the authors state that the method cannot detect forged areas with higher noise levels than the original image, which the miniTrace Noise dataset contains examples of. This known limitation might result in the difference in performance observed compared to its counterparts.

However, the case of $\mathrm{IoU}_w^{v_1}$ and $\mathrm{F1}_w^{v_1}$, shown in Tables 7.3 and 7.4 respectively, suggest a different result as EXIF as Language with NCuts surpasses Noisesniffer on the Noise endogenous datasets. As explained before, the IoU and F1 do not penalize as hard as the MCC predictions with larger forged areas than the ones in the ground truth. This means that EXIF as Language with NCuts is probably estimating larger forged areas than the real ones, whereas Noisesniffer is not, and that is likely why in terms of $\mathrm{MCC}_w^{v_1}$ the results are better.

This difference between $\mathrm{MCC}_w^{v_1}$ and the other metrics also appears for PSCC-Net, where the results of $\mathrm{IoU}_w^{v_1}$ and $\mathrm{F1}_w^{v_1}$ are better than $\mathrm{MCC}_w^{v_1}$. Looking at the latter, it can be inferred that despite being trained as a multipurpose method, PSCC-Net performs best when JPEG traces are involved.

Another relevant observation is the good performance of TruFor across Noise and JPEG traces as well as with the hybrid dataset, getting almost always third place. This is unsurprising, taking into account that TruFor uses Noiseprint++ as a feature, which was trained to distinguish between different processing pipelines (including different compressions), cameras, and spatial positions (Section 5.7.1). In the case of CAT-Net, the best performance is achieved with JPEG traces, which can be attributed to its DCT stream, which enables it to detect compression-related inconsistencies.

Analyzing FOCAL's performance on the miniTrace datasets is surprising. Its poor overall performance seems to indicate that the method isn't exploiting any trace inconsistencies when detecting a forgery. This suggests that it mainly relies on exploiting the semantic inconsistency of the image.

Finally, Table 7.5 shows the results of the localization performance with the AUROC metric. As mentioned before, reported results are only on methods whose output is a heatmap. Among those methods, TruFor thrives in Noise, JPEG, and hybrid datasets, whereas Adaptive CFA excels within the CFA ones. The results reinforce those obtained previously; these methods detect trace inconsistencies very well in the presence of their target traces. One noteworthy observation should be made regarding DQ. Although the performance of DQ in terms of the weighted metrics was not exceptional, the performance on the AUROC metric in JPEG datasets is little short of remarkable. The most likely explanation for this is that DQ outputs a probability map with relatively low values, and these low values yield low-weighted metrics. However, it seems that the highest output values are in the correct places, so with the correct thresholds, the final masks are good, hence the good results in AUROC terms. An example of this behavior can be seen in Figure 7.2, where on JPEG grid, JPEG quality, and Hybrid, the heatmap is correct, but the image is grayed out due to the small dynamic range it has. As brought up previously, the mask the original method generates using the SVM may solve this issue, resulting in a better performance.

Method	Noise	JPEG Quality	JPEG Grid	CFA Alg	CFA Grid	Hybrid
Adaptive CFA	0.001	0.033	0.035	**0.509**	**0.675**	0.126
	0.020	0.025	0.036	**0.513**	**0.658**	0.119
Noisesniffer	**0.187**	0.074	-0.006	0.062	-0.005	0.148
	0.137	0.037	0.003	0.033	-0.002	0.100
ZERO	0.000	**0.737**	**0.783**	0.000	0.000	**0.565**
	0.000	**0.663**	**0.697**	0.000	0.000	**0.572**
ZERO with missing grids	0.000	0.678	0.711	0.000	0.000	0.533
	0.000	0.624	0.643	0.000	0.000	0.536
DQ	0.000	0.066	0.066	0.000	0.000	0.056
	0.000	0.058	0.060	-0.002	-0.002	0.050
CAT-Net	0.006	0.279	0.272	0.002	0.002	0.322
	0.002	0.316	0.314	0.002	-0.002	0.340
Splicebuster	0.086	0.071	-0.002	0.051	-0.005	0.103
	0.077	0.074	0.022	0.071	0.013	0.102
EXIF as Language Mean Shift	0.016	0.051	-0.004	0.046	0.000	0.034
	0.029	0.046	0.011	0.056	0.023	0.053
EXIF as Language NCuts	0.008	0.052	-0.004	0.056	-0.023	0.049
	0.049	0.061	0.040	0.064	0.010	0.128
PSCC-Net	0.007	0.118	0.111	0.002	0.002	0.121
	0.003	0.118	0.116	-0.001	-0.002	0.120
TruFor	0.118	0.504	0.532	0.033	0.004	0.408
	0.076	0.524	0.552	0.022	-0.001	0.398
FOCAL	0.035	-0.002	-0.008	-0.001	0.002	0.033
	0.032	0.017	0.018	0.016	0.012	0.045

Table 7.2: Localization performance in terms of the mean weighted MCC score ($\text{MCC}_w^{v_1}$) in the miniTrace database, for both, the exogenous datasets and the endogenous datasets. In **bold**, the highest score in each dataset, and underlined, the second highest one.

7.2.2 Detection

Although we previously stated that detection evaluation requires a dataset with both tampered and pristine images, we will make an exception for this dataset to highlight some interesting points. It is important to keep in mind that evaluating without pristine images does not penalize false positives on the metrics, so a method with really low precision but high recall will appear on top. As mentioned in Section 7.1.2, the dataset-level IoU and F1 scores are the only metrics used to evaluate detection performance, reported on Table 7.6, and Table 7.7 respectively.

Looking at the detection results, what first stands out is the excellent performance of EXIF as Language. Yet, this apparent success is misleading. The method systematically flags images as forged, and since the dataset contains only manipulated images, the predictions are always correct. This behavior stems from our decision to derive the detection score from the mask prediction, also employed in ZERO and Noisesniffer, where we consider a forgery is detected if any value in the predicted mask is 1. However, this approach does not fully apply to the case of EXIF as Language. Unlike ZERO and Noisesniffer, which in some cases

Method	Noise	JPEG Quality	JPEG Grid	CFA Alg	CFA Grid	Hybrid
Adaptive CFA	0.014	0.120	0.121	**0.459**	**0.603**	0.184
	0.026	0.116	0.121	**0.457**	**0.584**	0.180
Noisesniffer	**0.192**	0.116	0.043	0.107	0.041	0.158
	0.141	0.083	0.049	0.076	0.042	0.119
ZERO	0.000	**0.702**	**0.740**	0.000	0.000	**0.578**
	0.000	**0.641**	**0.673**	0.000	0.000	**0.575**
ZERO with missing grids	0.000	0.642	0.657	0.000	0.000	0.540
	0.000	0.605	0.612	0.000	0.000	0.534
DQ	0.096	0.127	0.128	0.101	0.101	0.124
	0.096	0.123	0.124	0.100	0.100	0.121
CAT-Net	0.005	0.187	0.186	0.002	0.002	0.243
	0.003	0.230	0.227	0.005	0.002	0.262
Splicebuster	0.070	0.076	0.037	0.055	0.033	0.092
	0.071	0.084	0.053	0.077	0.045	0.095
EXIF as Language Mean Shift	0.056	0.085	0.049	0.096	0.062	0.070
	0.064	0.086	0.059	0.102	0.075	0.080
EXIF as Language NCuts	0.132	0.164	0.111	0.168	0.102	0.176
	0.176	0.187	0.159	0.190	0.140	0.230
PSCC-Net	0.117	0.117	0.108	0.125	0.123	0.122
	0.117	0.114	0.113	0.145	0.122	0.119
TruFor	0.132	0.431	0.453	0.067	0.043	0.357
	0.092	0.456	0.478	0.060	0.043	0.351
FOCAL	0.095	0.074	0.072	0.073	0.076	0.092
	0.104	0.097	0.096	0.092	0.089	0.114

Table 7.3: Localization performance in terms of the mean weighted IoU score ($\text{IoU}_w^{v_1}$) in the miniTrace database, for both, the exogenous datasets and the endogenous datasets. In **bold**, the highest score in each dataset, and underlined, the second highest one.

generate empty masks, EXIF's clustering method always generates two classes, one corresponding to forged pixels and the other to authentic pixels, so there will always be a forged class. This is why the detection score is always 1 and should be disregarded.

Regarding ZERO and Noisesniffer, some interesting comments can be made. As mentioned before, the detection score of these methods will be 1 if at least one pixel is marked as forged in the output mask and 0 if no pixels are detected. Taking this into account, Noisesniffer seems to detect a lot of tampered images, even in cases such as JPEG grid or CFA grid datasets, for which the localization shown in Tables 7.2, 7.3 and 7.4 was poor. This behavior can be explained by the appearance of false detections, which are explained by the authors to happen because of two reasons. First, in the presence of highly textured areas, as in those cases, the method will detect the flat regions as forgeries. Secondly, some false detections might be caused by what the authors name as wrong attributions. Wrong attributions happen when the forgery's size is large enough that small pristine areas with lower noise levels are detected as local anomalies, which are flagged by the method. Due to the lack of pristine images, we cannot determine

Method	Noise	JPEG Quality	JPEG Grid	CFA Algorithm	CFA Grid	Hybrid
Adaptive CFA	0.023	0.197	0.198	**0.531**	**0.692**	0.265
	0.039	0.191	0.197	**0.531**	**0.676**	0.260
Noisesniffer	**0.255**	0.168	0.075	0.154	0.072	0.222
	0.199	0.126	0.082	0.116	0.070	0.173
ZERO	0.000	**0.747**	**0.785**	0.000	0.000	**0.608**
	0.000	**0.683**	**0.697**	0.000	0.000	**0.604**
ZERO with missing grids	0.000	0.699	0.711	0.000	0.000	0.576
	0.000	0.660	0.662	0.000	0.000	0.567
DQ	0.163	0.207	0.209	0.170	0.169	0.202
	0.163	0.202	0.204	0.169	0.169	0.199
CAT-Net	0.009	0.276	0.270	0.005	0.005	0.327
	0.006	0.319	0.319	0.007	0.004	0.350
Splicebuster	0.116	0.124	0.070	0.096	0.060	0.147
	0.118	0.132	0.088	0.124	0.077	0.150
EXIF as Language Mean Shift	0.101	0.146	0.090	0.159	0.110	0.123
	0.111	0.145	0.103	0.165	0.127	0.136
EXIF as Language Ncuts	0.189	0.221	0.176	0.227	0.160	0.234
	0.225	0.240	0.212	0.243	0.193	0.284
PSCC-Net	0.189	0.182	0.170	0.198	0.196	0.189
	0.188	0.176	0.173	0.254	0.193	0.183
TruFor	0.198	0.542	0.564	0.114	0.077	0.458
	0.145	0.561	0.583	0.103	0.077	0.446
FOCAL	0.149	0.118	0.116	0.117	0.122	0.146
	0.151	0.137	0.135	0.131	0.129	0.163

Table 7.4: Localization performance in terms of the mean weighted F1 score ($F1_w^{v_1}$) in the miniTrace database, for both, the exogenous datasets and the endogenous datasets. In **bold**, the highest score in each dataset, and underlined, the second highest one.

Method	Noise	JPEG Quality	JPEG Grid	CFA Alg	CFA Grid	Hybrid
Adaptive CFA	0.496	0.536	0.542	**0.812**	**0.935**	0.608
	0.507	0.525	0.543	**0.824**	**0.926**	0.599
DQ	0.534	0.810	0.828	0.551	0.547	0.791
	0.534	0.782	0.795	0.530	0.535	0.777
CAT-Net	0.501	0.670	0.654	0.502	0.502	0.733
	0.499	0.710	0.702	0.498	0.499	0.751
Splicebuster	0.603	0.598	0.522	0.556	0.491	0.640
	0.585	0.570	0.522	0.554	0.485	0.613
EXIF as Language Mean Shift	0.610	0.717	0.518	0.661	0.535	0.649
	0.600	0.647	0.527	0.654	0.585	0.654
PSCC-Net	0.497	0.699	0.706	0.699	0.531	0.653
	0.498	0.695	0.699	0.542	0.527	0.645
TruFor	**0.727**	**0.920**	**0.915**	0.601	0.548	**0.913**
	0.640	**0.932**	**0.944**	0.565	0.535	**0.912**

Table 7.5: Localization performance in terms of AUROC in the miniTrace database, for both, the exogenous datasets and the endogenous datasets. In **bold**, the highest score in each dataset, and underlined, the second highest one.

Method	Noise	JPEG Quality	JPEG Grid	CFA Alg	CFA Grid	Hybrid
Noisesniffer	<u>0.850</u> <u>0.820</u>	<u>0.915</u> <u>0.925</u>	0.850 0.855	<u>0.850</u> <u>0.825</u>	<u>0.780</u> <u>0.780</u>	<u>0.880</u> <u>0.885</u>
ZERO	0.000 0.000	0.825 0.800	<u>0.865</u> <u>0.835</u>	0.000 0.000	0.000 0.000	0.700 0.695
EXIF as Language	**1.000** **1.000**	**1.000** **1.000**	**1.000** **1.000**	**1.000** **1.000**	**1.000** **1.000**	**1.000** **1.000**
PSCC-Net	0.572 0.577	0.148 0.156	0.154 0.151	0.652 0.651	0.647 0.647	0.229 0.217
TruFor	0.141 0.106	0.673 0.679	0.654 0.668	0.118 0.112	0.100 0.100	0.612 0.608

Table 7.6: Detection performance in terms of the weighted IoU score over the full dataset ($\text{IoU}_w^{v_2}$) in the miniTrace database, for both, the exogenous datasets and the endogenous datasets. In **bold**, the highest score in each dataset, and <u>underlined</u>, the second highest one.

which of the two cases occurs the most, so we cannot draw any conclusions about the reliability of the method.

On the other hand, ZERO has both 0.000 scores for localization in Noise and CFA datasets while also having a 0.000 in detection in the same datasets, meaning that it does not detect anything. This implies that ZERO only flags forgeries when it is confident, which, in other words, means that ZERO is a high-precision, low-recall method.

Lastly, PSCC-Net and TruFor are the only two methods in which the detection output is a probability. Looking at the results, they appear to be fairly complementary, one succeeding where the other falls short. Compared to localization, TruFor maintains its good detection performance on JPEG traces yet appears to fall behind on noise detection. Given that detection and localization are performed separately, we can assume that the method can pick up noise traces on the image but is not confident enough to flag the traces as a forgery. In contrast, PSCC-Net, whose localization performance was a little worse than TruFor's, greatly outperforms it in detection. Again, the lack of pristine images means we cannot judge if the method is better at detecting or just simply tends to predict more false positives, but given that the JPEG-related scores are low, it appears to be able to pick up noise and CFA trace inconsistencies really well when detecting but not when localizing.

7.3 Popular datasets evaluation

This section reports and analyzes the results obtained from the evaluation of the methods on popular datasets for both localization and detection problems.

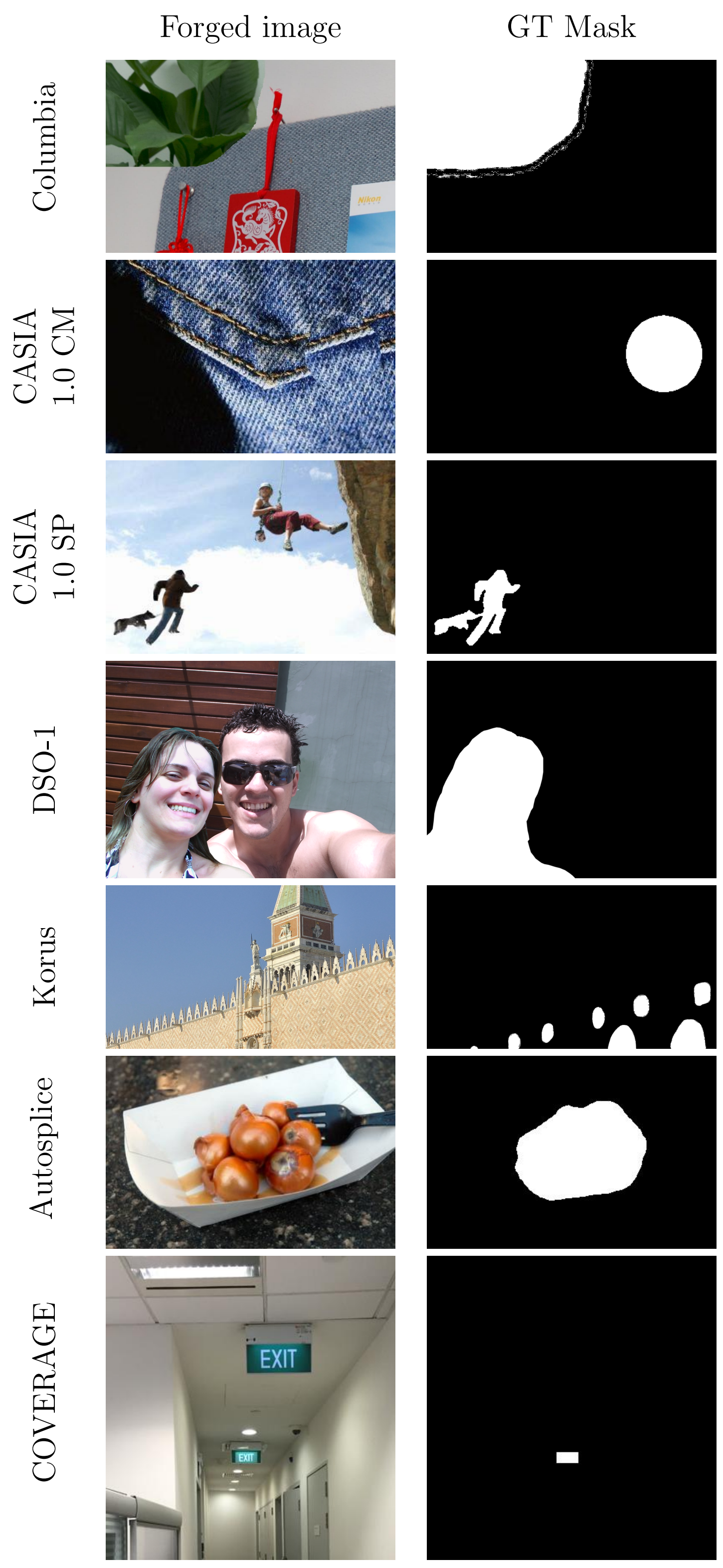


Figure 7.4: Image samples from the popular datasets.

Method	Noise	JPEG Quality	JPEG Grid	CFA Alg	CFA Grid	Hybrid
Noisesniffer	0.919	0.956	0.919	0.919	0.876	0.936
	0.901	0.961	0.922	0.904	0.876	0.939
ZERO	0.000	0.904	0.928	0.000	0.000	0.824
	0.000	0.889	0.910	0.000	0.000	0.820
EXIF as Language	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
	1.000	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
PSCC-Net	0.727	0.258	0.267	0.789	0.786	0.372
	0.732	0.270	0.262	0.788	0.786	0.356
TruFor	0.247	0.805	0.791	0.211	0.182	0.760
	0.191	0.809	0.801	0.202	0.182	0.756

Table 7.7: Detection performance in terms of the weighted F1 score over the full dataset ($F1_w^{v_2}$) in the miniTrace database, for both, the exogenous datasets and the endogenous datasets. In **bold**, the highest score in each dataset, and underlined, the second highest one.

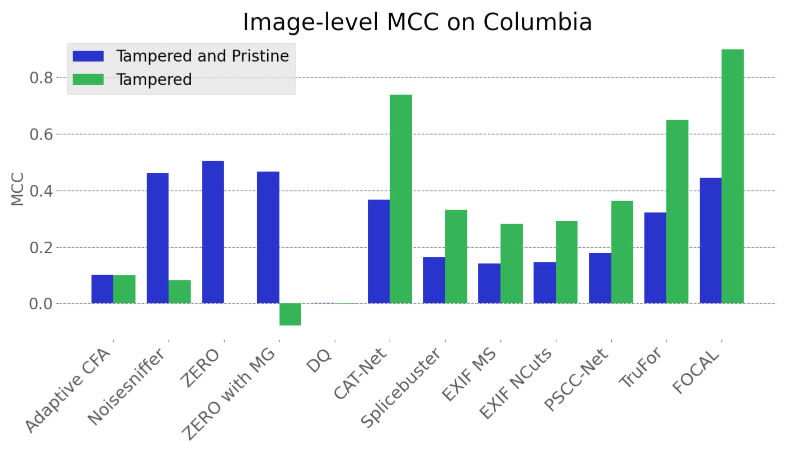


Figure 7.5: A bar plot of the $MCC_w^{v_1}$ performance for the Columbia Dataset, according to the values in Table 7.8. The performance on tampered only images is shown in green, while the tampered and pristine performance is shown in blue.

7.3.1 Localization

Results of the localization performance for evaluation in popular datasets with $MCC_w^{v_1}$, $IoU_w^{v_1}$, $F1_w^{v_1}$ and AUROC are presented in Tables 7.8, 7.9, 7.10 and 7.11 respectively. Given the diversity of the datasets, this section will be structured differently than the previous ones by describing different aspects of the problem in different sections.

7.3.1.1 Forgery or no forgery

Earlier in this chapter, we brought up the fact that one critical aspect is to evaluate whether the method is able to detect the absence of a forgery. Although this is typically done with the detection score, we want to measure the quality of the localization mask on pristine images too, as it reflects the tendency of a method to hallucinate. This is relevant because, in the wild, the number of pristine images is much higher than the number of tampered ones. Within the literature, it is common to evaluate the performance of localization methods only on tampered images, overlooking this aspect of the problem. This section will present both the tampered-only and the tampered-and-pristine performances, which will capture how the methods behave on any given image. For this purpose we present the $\text{MCC}_w^{v_1}$, $\text{IoU}_w^{v_1}$, $\text{F1}_w^{v_1}$ in Tables 7.8, 7.9 and 7.10 respectively, where as mentioned in Section 7.1.3, the values in blue correspond to evaluating on the full dataset, while the green correspond to evaluating on forged images only. To complement the following analysis, Figure 7.5 shows $\text{MCC}_w^{v_1}$ for the methods in the Columbia dataset.

First, we will begin by evaluating the performance on pristine images. This puts methods like Splicebuster, FOCAL, and EXIF as Language, which cluster the features into two classes, at a disadvantage, so we expect them to perform poorly. The other methods should potentially be able to tackle this task correctly, especially in the case of Trufor, which runs an anomaly detector. Unfortunately, this does not seem to be the case, as most methods perform better when evaluated only on tampered images, as seen in Table 7.8.

However, this is not the case for ZERO and Noisesniffer. In most datasets, both of these methods achieve a better score when they are evaluated with both pristine and tampered images than when evaluated only on tampered, which is indicative of their ability to detect the absence of forgery. In other words, these methods rarely give false positives but fail to detect many of the forgeries when the traces they are designed for are not present. When comparing ZERO with its missing grids version, the basic version is better in this sense, given that, as stated in Section 7.1.1, the missing grids version tends to have more false positives.

This is particularly interesting for Noisesniffer. During the non-semantic evaluation, we were not able to conclude how many false detections it gave. This evaluation featuring pristine images shows that the method's reliability, although not as high as ZERO's, is impressive nonetheless, rarely flagging pristine areas as forged. There is one exception, the DSO-1 dataset, where Noisesniffer achieves a better score when evaluating only tampered images. As the authors expressed, the method is susceptible to highly textured images, so we believe this might be the source of the increase in false positives.

7.3.1.2 Images through *Facebook*

Following the datasets presented in Section 3.2.8, we evaluated the methods on a set of images that were passed through *Facebook*, where the images were subjected to an unknown processing pipeline that might affect the performance of methods. An example of DSO-1 is presented in Figure 7.6, where most methods that had a good performance on the original image fail when run on the *Facebook* version. It is also interesting to note that this is not the case with CASIA 1.0, as seen in Figure 7.7, where passing the image through Facebook barely modifies it, as images were already heavily compressed and small in size. Columbia is the middle point, shown in Figure 7.10, where the changes are noticeable, but the methods

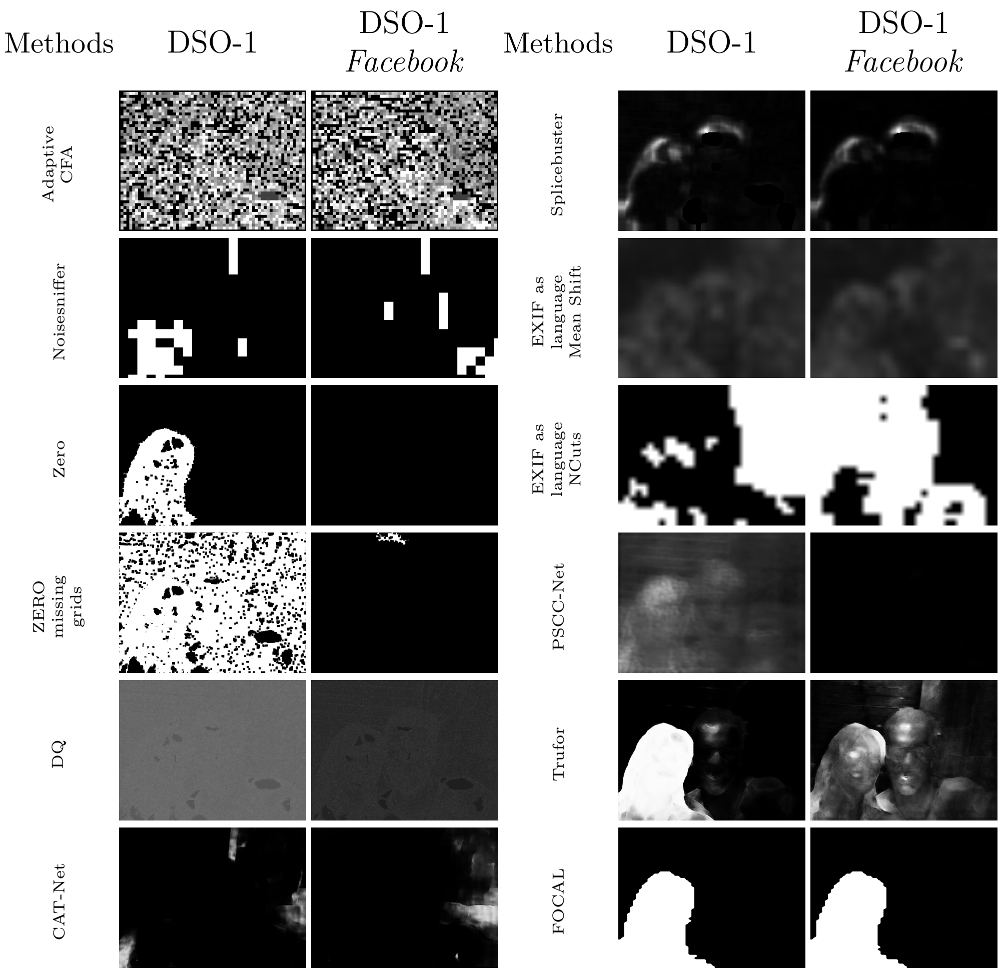


Figure 7.6: Outputs in all DSO-1 variants of all of the evaluated methods. Original image and corresponding mask are shown in Figure 7.4.

do not fail as with DSO-1.

Tables 7.9, 7.10, and 7.8 show the contrast in performance on the original datasets against the *Facebook* datasets. The former is reported in green and the latter in purple. Intuitively, and judging by the examples provided before, one might expect that the methods would perform better, or fairly similar, in the original dataset than in the dataset that was passed through *Facebook*, as the original traces of tampering are more evident in the original dataset.

However, this is not the case for CAT-Net and PSCCNet. In both cases, the performance improves dramatically for the Columbia *Facebook* dataset. A possible explanation for this phenomenon comes from the fact that Columbia is an uncompressed dataset, along with the fact that CAT-Net and PSSCNet are both deep-learning-based methods. Uncompressed data falls outside of the methods' known data distribution, so when presented with an uncompressed image, they hallucinate. When passed through *Facebook*, the images become more similar to the image seen during training, and their predictions become more accurate. This also seems to apply to FOCAL, but the improvement is marginal.

As for the CASIA 1.0 datasets, the original dataset was already JPEG compressed, so the methods' performance in the dataset passed through *Facebook* is a little worse than in the original dataset. An example of the impact of *Facebook* on a CASIA 1.0 image, or lack thereof in this case, is presented in Figure 3.3 on Section 3.2.8, where the example image of CASIA v1 remains untouched after passing through *Facebook*.

Going back to the first example of the section, we observe that the performance of the methods on DSO-1 falters the most out of all the datasets passed through *Facebook*. Given the big size of the images, it makes sense since they are heavily compressed when shared over social networks to reduce the cost of transmission. This also brings to light a problem with the DSO-1 dataset, which is the lack of details regarding the images' processing history. Still, when comparing the behavior of the methods on the original and Facebook versions of this dataset, we suspect that the images, before being stored in PNG format, suffered some kind of compression. This suspicion is fueled by the performance of ZERO in the original DSO-1 dataset. Of course, this is not a definite argument as the method may fail, but, as shown before in Section 7.2.1, ZERO delivers very few, if any, false alarms. Hence, either DSO-1 triggers ZERO to make a lot of false alarms, or there is indeed a compression history in DSO-1 images.

Lastly, FOCAL is the method that gets impacted the least by the *Facebook* compression, with an impressive example in the DSO-1 Figure 7.6. The first detail is that FOCAL and the OSN datasets come from the same research group, which does not ensure good performance per se, but it is reasonable to expect they developed their solution with this problem in mind. Secondly, as seen during the non-semantic evaluation, FOCAL does not appear to base its prediction on any particular image trace but rather relies on the semantic information of the image, so the modification of traces by the *Facebook* pipeline barely impacts its performance.

7.3.1.3 Splicing forgeries

Splicing forgeries can be detected by inconsistencies in noise, JPEG, and CFA, as stated in Section 2.2.5.2. In this part of the evaluation, we consider the dataset obtained through the splicing of images, which are Columbia, CASIA 1.0 splicing, and DSO-1. Example outputs of the methods on an image of each dataset are presented in Figures 7.10, 7.7, and 7.6. This section will only focus on localization results, thus considering just the tampered-only case, as the performance impact when pristine images are involved was already covered earlier in Section 7.3.1.1.

When it comes to the Columbia dataset, the best-performing methods are FOCAL, CAT-Net, and TruFor, with FOCAL leading significantly over the other two. Adaptive CFA, Noisesniffer, and DQ perform poorly on the task, while EXIF as Language, PSCC-Net, and Splicebuster perform similarly in the middle. ZERO, keeping the trend observed, gets 0.000 in all metrics because the dataset is uncompressed, meaning there are no JPEG traces in the images. That being

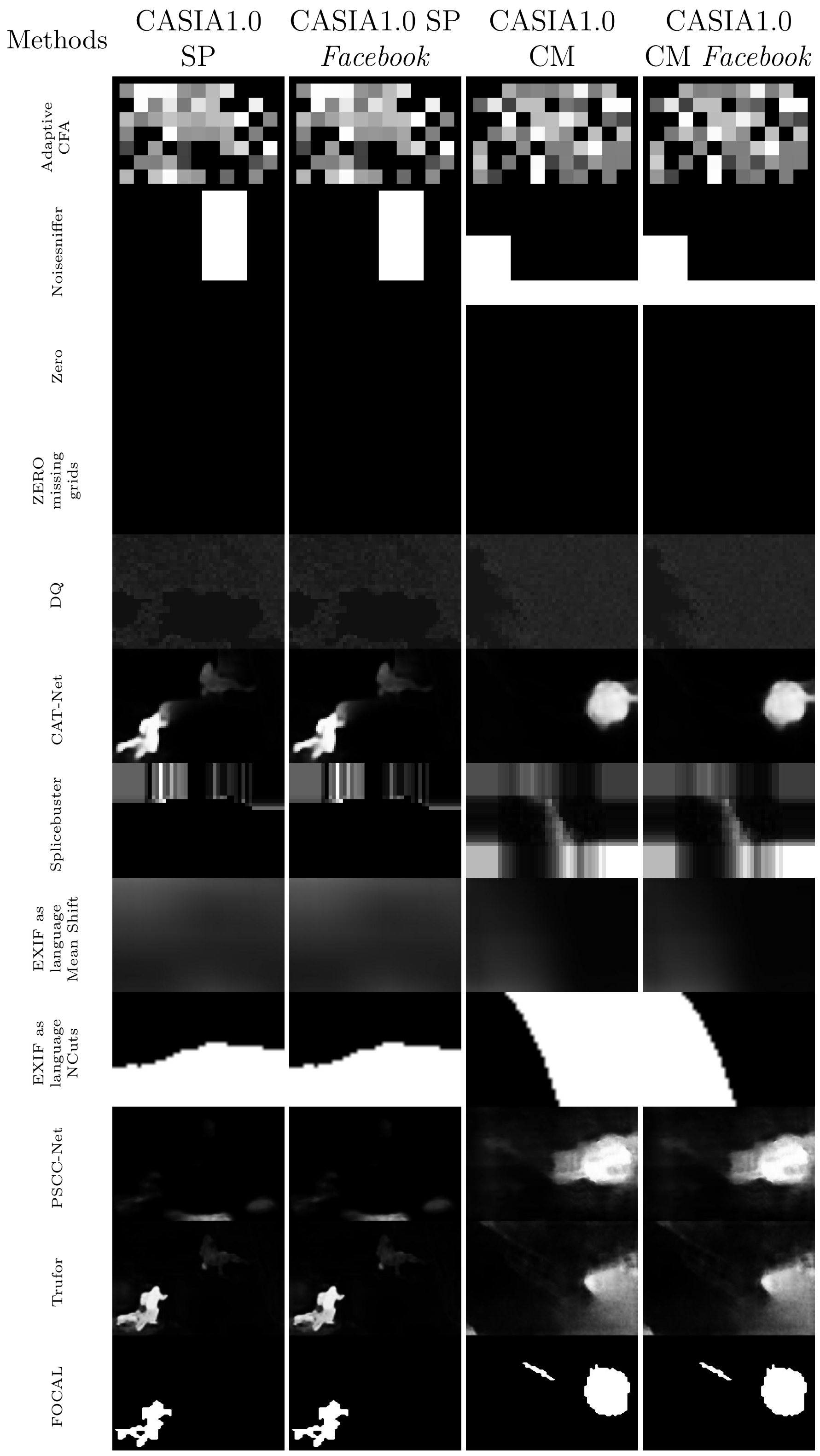


Figure 7.7: Outputs in all CASIA 1.0 variants of all of the evaluated methods. Original images and corresponding masks are shown in Figure 7.4.

said, ZERO with missing grids makes some detections poorly localized, aligning with the author's comment on a loss of reliability compared to the basic method.

On the CASIA 1.0 Splicing dataset, FOCAL achieves the best result, followed by CAT-Net and TruFor close behind. The image features in this dataset are heavily JPEG compressed, making it natural for a method like CAT-Net, which contains a neural network specially designed for processing DCT coefficients, to achieve a really high performance. FOCAL is trained to detect splicing among other forgeries, and one of the datasets used is CASIA 2.0, which features images similar to version 1.0. Thus, a good result in this dataset can be expected.

In the case of ZERO, it manages to detect some forgeries, but the results are still poor, which is not surprising given the high compression level of these images, a weak spot detected by the method's authors (Section 5.4). Except for FOCAL, which achieves a medium-high score, the rest of the methods perform poorly on these datasets, seeming to indicate a weakness to heavy compression. As mentioned in the previous section, FOCAL does not seem to be affected by compression when shared through *Facebook*, and the performance on this dataset reinforces that idea.

Finally, on the DSO-1 dataset, TruFor achieves the best results. The second best, though far behind, is FOCAL. Closely in third place is CAT-Net, which, paired with TruFor's results, contributes to support the claim that the images on the dataset were compressed at some point. This becomes even more apparent when evaluating ZERO, where the low performance (but not 0.000) in only tampered images, along with the observed reliability of the method, seem to indicate that JPEG traces are present on the images. On the topic of ZERO, ZERO with missing grids outperforms the basic ZERO, which might also be explained by the unknown processing history of the images, making missing grids more likely to appear. With the exception of Adaptive CFA and DQ, which perform badly, the rest of the methods achieve similar scores.

7.3.1.4 Copy-move forgeries

Copy-move forgeries, as covered in Section 2.2.5.1, are particularly interesting in their nature, where parts of the image are pasted onto the image itself. This process might introduce inconsistencies on the JPEG and CFA grids, while noise inconsistencies can be found when the cloned part is resized. In this part of the evaluation, we will consider the two copy-move datasets, CASIA 1.0 Copy-Move and COVERAGE. Once again, we will only look at tampered images.

In the case of CASIA 1.0, copy-move results for all methods are much lower than those in the case of CASIA 1.0 Splicing. The top three performers are FOCAL, CAT-Net, and TruFor. PSSC-Net is not far behind TruFor. Like its splicing counterpart, CASIA 1.0 Copy-Move has heavily compressed images, so the top performers are the same. The rest of the methods perform poorly in this dataset, likely related to the high compression of the images.

The results for the COVERAGE dataset are low for most of the methods, with the top performer being FOCAL. Far behind come TruFor, CAT-Net, and PSCC-Net, with the rest of the methods achieving really low results. Once again,

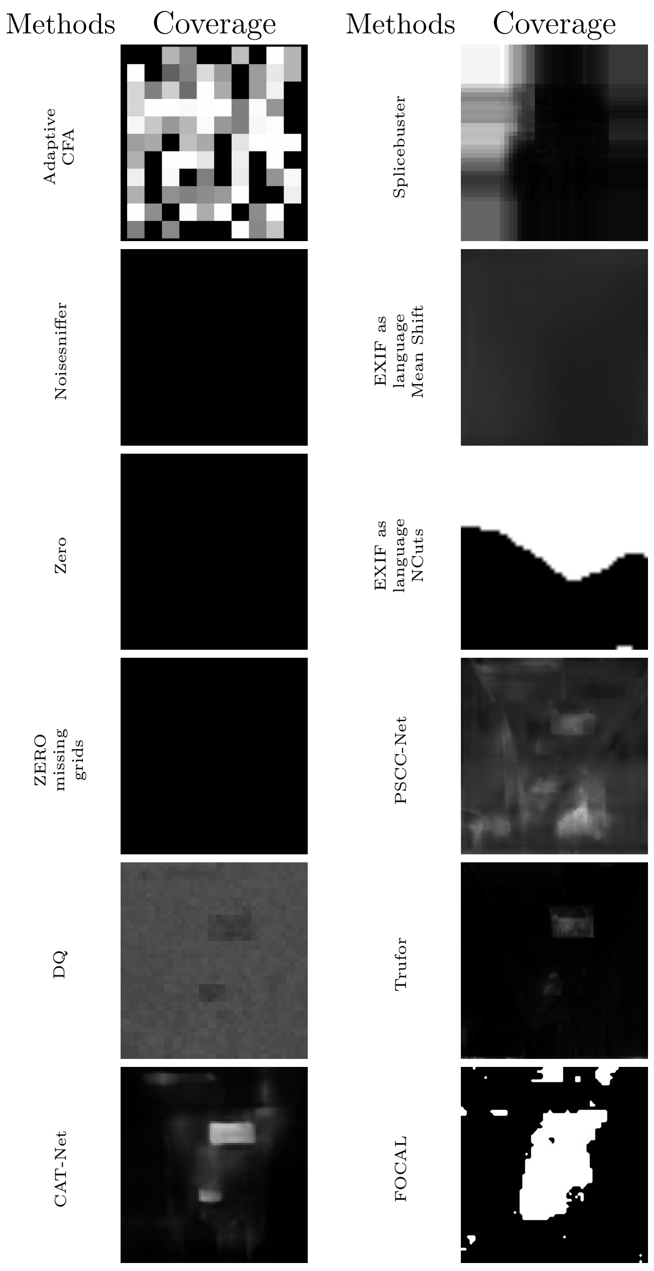


Figure 7.8: Outputs in Coverage of all of the evaluated methods. Original image and corresponding mask are shown in Figure 7.4.

ZERO detects forgeries on a seemingly uncompressed dataset, where the images are saved in TIFF format. However, as covered in Section 3.2.3, the images were taken with an iPhone 6 that saves the images in JPEG format, which is what might be allowing ZERO to detect grid misalignments.

Taking the results of both datasets into account, certain details merit attention. First, Noisesniffer is not able to detect much, most likely due to the noise distribution being spatially uniform across the image. Secondly, Adaptive CFA barely gets any results, possibly meaning that the more complex demosaicing algorithms involved in in-the-wild images prevent the method from detecting forgery.

7.3.1.5 Korus forgeries

The Korus dataset introduced in Section 3.2.5, also known as the Realistic Tampering dataset, contains forgeries that are almost impossible to detect by the naked eye. Unfortunately, most methods also struggle with this dataset, as seen in the example in Figure 7.9. Although the dataset contains splicing and copy-move images, it also contains more complex forgeries, such as object removal, and this is why it has its section in this evaluation.

The best performances are achieved by FOCAL and TruFor, but they are not as high as those achieved by the same methods in other datasets. This was foreseeable, given the level of difficulty that this dataset presents, but it still questions the performance of these methods on in-the-wild forgeries.

Another remarkable result is obtained by Noisesniffer, which consistently obtains third or fourth-best results across all of the weighted metrics despite it being a classical method. Splicebuster also appears to capture some forgeries, as its performance is not far behind that of Noisesniffer. This, paired with TruFor's performance, seems to indicate that noise-based traces are useful on these datasets. Adaptive CFA achieves its best performance across all popular datasets in Korus, competing for third place with Noisesniffer across the three metrics. Noisesniffer wins in MCC_w^{v1}, while Adaptive CFA wins in IoU_w^{v1} and $\mathrm{F1}_w^{v1}$, meaning that Adaptive CFA is overestimating the forged area.

Once again, a curious observation is that ZERO achieves an almost zero but still positive MCC_w^{v1} score in this uncompressed dataset. The explanation for this lies in image `DSC07222`, which is spliced with an image originally JPEG compressed.

7.3.1.6 WebP

In Section 3.2.9, we present our novel WebP datasets, which are compressed versions of Columbia and Korus, to evaluate the impact of this increasingly popular lossy image compression (Section 2.1.4.4). Examples of the impact can be seen in Figures 7.10 and 7.9, which show the outputs obtained on both the original and the WebP compressed image.

Given that some methods already did poorly on the original Korus dataset, we will limit our analysis to the ones that were already proficient. As expected, performance goes down for all methods, especially Adaptive CFA, which obtains

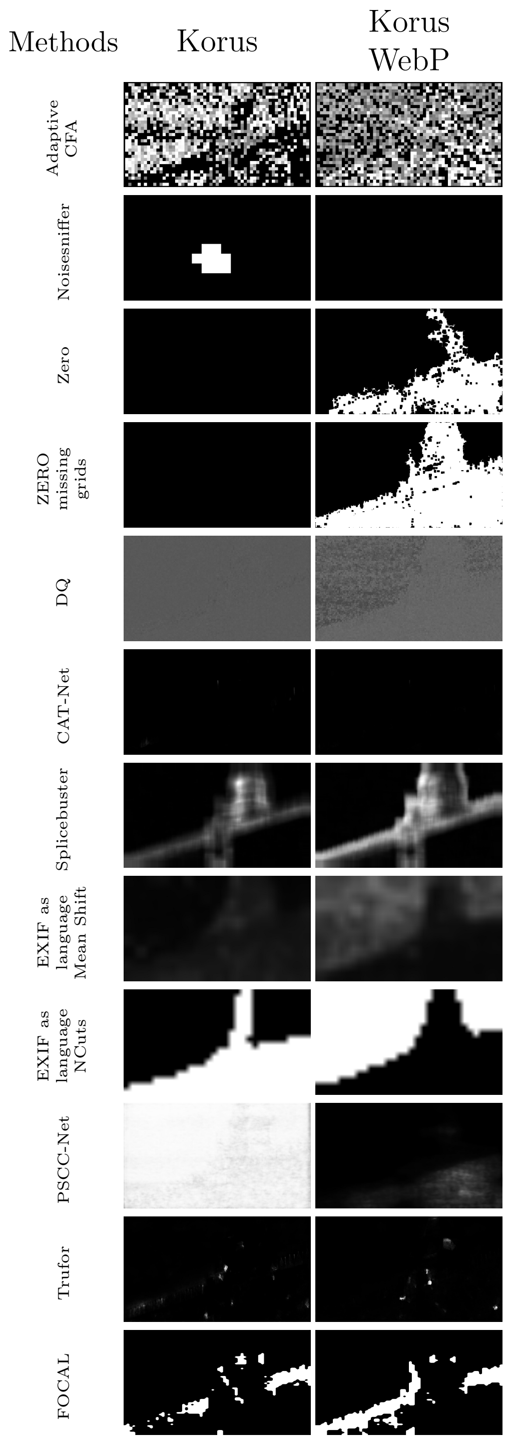


Figure 7.9: Outputs in all Korus variants of all of the evaluated methods. Original image and corresponding mask are shown in Figure 7.4.

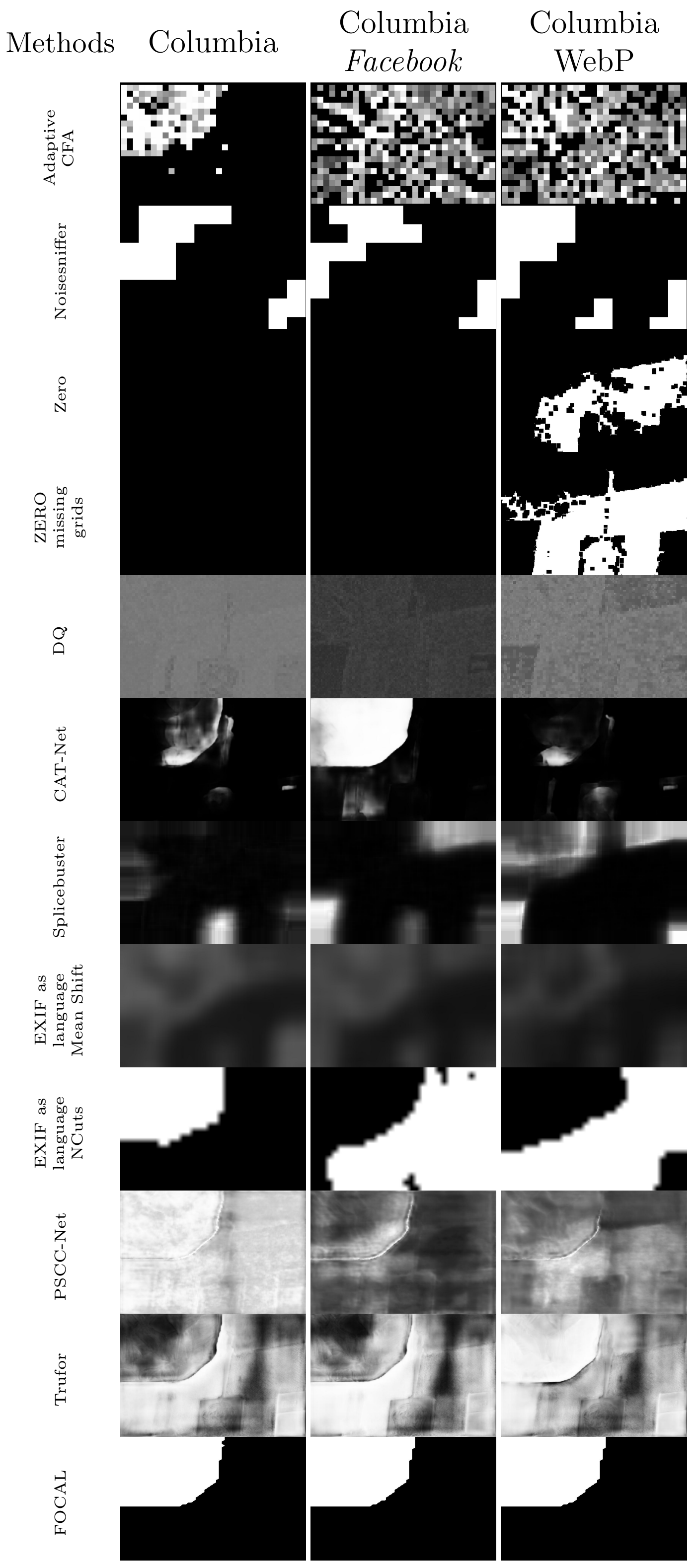


Figure 7.10: Outputs in all Columbia variants of all of the evaluated methods. Original image and corresponding mask are shown in Figure 7.4.

a result ten times worse on $\mathrm{MCC}_{w}^{v_1}$. This degradation comes from the method's sensitivity to JPEG compression, which WebP resembles. TruFor retains its second place but with less than half of the original performance. Splicebuster also halves its previous result. FOCAL is the least affected, which is indicative of FOCAL's robustness to compression. Noisesniffer's relative loss in performance is not as small as FOCAL, but it is still much better than the rest.

The Columbia WebP datasets show a more promising scenario, with a smaller performance loss in most methods. CAT-Net maintains the second place, and TruFor the third. Once again, PSCC-Net, now joined by FOCAL, does better in the WebP compressed images than the original. This might have to do with the method's training, making it perform better on compressed images. ZERO with missing grids improved its performance, although marginally.

The most surprising result of the WebP datasets comes from ZERO, which becomes a less reliable method in the sense that it produces more false detections. This is particularly evident in the tampered and pristine evaluation of Korus, where it goes from being the best method by a margin to second place. It is also the case for ZERO with missing grids, where its performance drops close to zero. On Columbia, the result is less extreme yet noticeable, coming from a method that has been reliable so far. It appears the resemblance between WebP and JPEG compression is enough to trigger false positives in ZERO's algorithm.

7.3.1.7 AutoSplice

AutoSplice is the dataset that contains images inpainted using text-to-image generation models, which is a type of forgery gaining popularity. The dataset is presented in three different compression rates: 100, 90, and 75 quality factor. An example of the outputs obtained for these datasets is presented in Figure 7.11.

We first analyze the results on the uncompressed dataset, AutoSplice100. CAT-Net performs best in all metrics, followed by TruFor and FOCAL, with TruFor leading in $\mathrm{MCC}_{m}^{v_1}$ and FOCAL leading the rest. The performance of CAT-Net is remarkable, especially considering it has not seen this type of forgery during training. PSCC-Net falls a little behind these methods, getting fourth place. One surprising observation is that ZERO with missing grids seems to be locating some forgeries, indicating that the inpainted regions are missing the original grid traces, even after the 100-quality compression. This might explain CAT-Net's exceptional performance, where its DCT stream allows it to exploit missing grid traces. The rest of the methods fail to detect anything valuable. These methods score really low values of MCC but higher ones in IoU and F1, which indicates they are overestimating the forged areas.

When analyzing the results of the compressed datasets, CAT-Net's performance decays heavily on Autosplice75, falling behind FOCAL, which takes the lead. FOCAL is once again the most robust method of compression, but still worsens its performance to almost half on both $\mathrm{MCC}_{w}^{v_1}$ and $\mathrm{IoU}_{w}^{v_1}$ when comparing Autosplice100 and Autosplice75. CAT-Net's behavior is probably due to the fact that such a strong compression hides some of the traces left by the previous compression of the original image, making the compression traces uniform

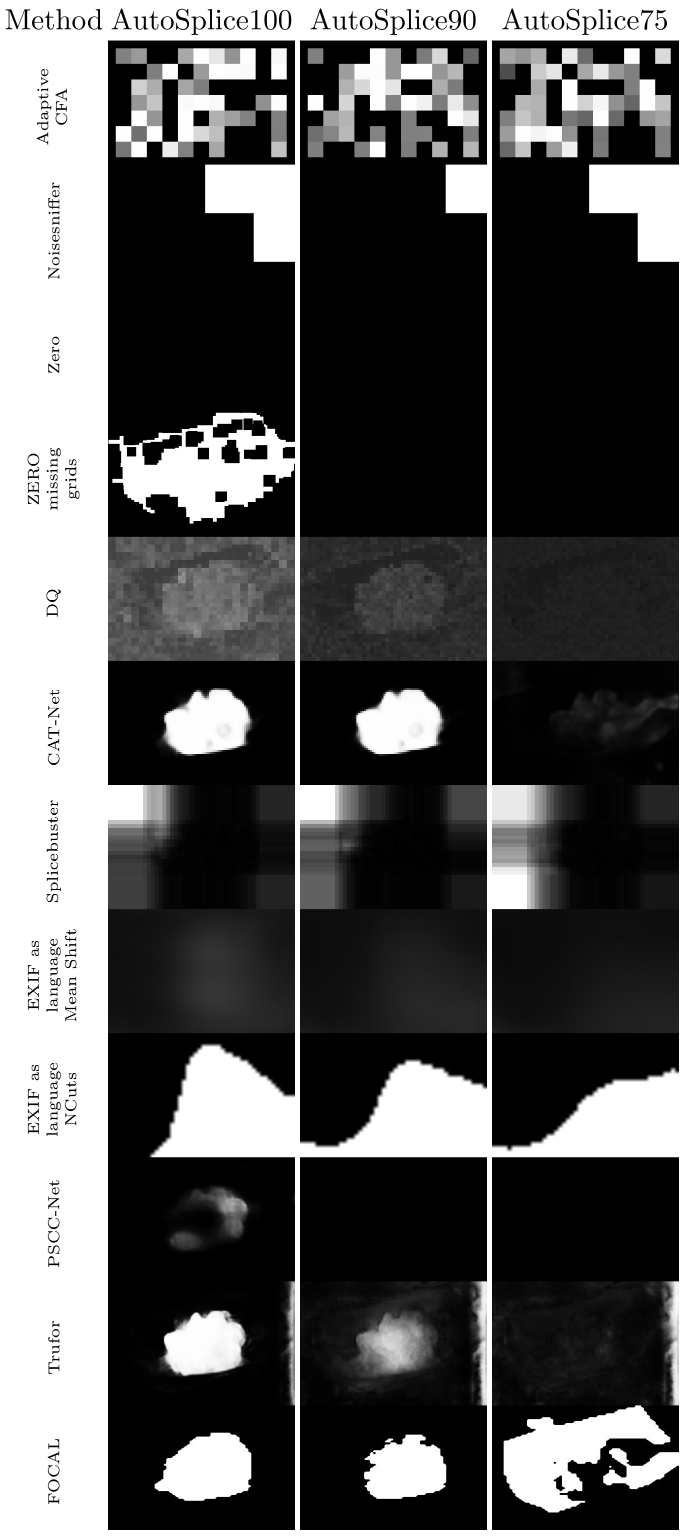


Figure 7.11: Outputs in AutoSplice of all of the evaluated methods. Original image and corresponding mask are shown in Figure 7.4.

throughout the image, including the inpainted regions. The 90 and 75 quality factor compression means ZERO with missing grids is not locating forgeries anymore, given there is a new grid on the whole image.

7.3.1.8 AUROC

The AUROC results of the methods that output a continuous heatmap are shown in Table 7.11. There are two clear winners in this analysis: CAT-Net and TruFor, which score first or second place in almost all datasets. DQ has particularly bad results in the CASIA 1.0 Copy-Move and Korus datasets, performs surprisingly well in the AutoSplice100 and AutoSplice90 datasets, has average results in the Columbia and Columbia WebP datasets, and performs poorly in the rest of the datasets. Once again, the high performance of DQ on the AutoSplice datasets can be due to the method's ability to distinguish between two classes but with a low dynamic range.

Adaptive CFA does not excel in any dataset, achieving results that are barely better than those of a random classifier. EXIF as Language and PSCC-Net have ups and downs across the datasets, performing very well in some and very poorly in others. Splicebuster does a slightly worse job than the last two methods.

7.3.2 Detection

The evaluation results of the detection problem in the popular datasets are presented in MCC_w^{v2}, IoU_w^{v2} and $\mathrm{F1}_w^{v2}$ are presented in Tables 7.12, 7.14, and 7.13 respectively.

As mentioned in Section 7.2.2, our solution for the detection score of EXIF as Language was not optimal, as the method always looks for two classes in the image, and so the image is always classified as tampered. This is why the MCC_w^{v2} is always 0, and the IoU_w^{v2} and $\mathrm{F1}_w^{v2}$ are always 1.

When looking at the results of the other methods, it is clear that TruFor and PSCC-Net are the methods that excel in the detection task, with TruFor being the best one. A little behind these two methods is Noisesniffer, which has medium to high scores in IoU and F1 but a lower score in MCC.

ZERO continues to be a reliable method when it comes to making a positive detection as it does not hallucinate any tampering. The results in the tables show that, despite this, the method gets low scores in IoU, F1, and MCC.

7.4 Summary

The evaluation provided a detailed report of the performance of the ten methods selected. This section serves as a summary, listing the most important findings.

Firstly, the results and analysis presented in this chapter show that *Photo-Holmes* fulfilled its goal of being a library that allows to make a comprehensive analysis of forgery detection methods easily.

Method	Columbia	Columbia WebP	CASIA1.0 SP	CASIA1.0 CM	COVERAGE	DSO-1	Korus	Korus WebP	AutoSplice 100	AutoSplice 90	AutoSplice 75
Adaptive CFA	0.102	0.026	0.019	0.016	0.019	0.042	0.163	0.006	0.058	-	-
	0.099	0.053	0.051	0.045	0.039	0.086	0.153	0.012	0.094	0.088	0.083
	0.002	-	0.042	0.044	-	0.009	-	-	-	-	-
Noisesniffer	0.460	0.437	0.385	0.371	0.389	0.142	0.236	**0.188**	0.302	-	-
	0.083	0.014	0.034	-0.006	0.026	0.217	0.176	0.107	-0.025	0.007	0.017
	0.015	-	0.036	-0.004	-	0.209	-	-	-	-	-
ZERO	**0.504**	**0.467**	**0.651**	**0.632**	**0.528**	**0.568**	**0.503**	0.170	0.377	-	-
	0.000	-0.002	0.054	0.001	0.009	0.114	0.005	0.004	-0.015	0.000	0.000
	0.000	-	0.000	0.000	-	0.000	-	-	-	-	-
ZERO with missing grids	0.466	0.423	0.608	0.593	0.512	0.267	0.501	0.071	**0.482**	-	-
	-0.078	0.031	0.052	0.008	0.008	0.212	0.002	0.009	0.196	0.002	0.005
	-0.007	-	-0.002	0.005	-	0.002	-	-	-	-	-
DQ	0.002	0.005	0.003	0.000	0.000	0.008	0.000	0.000	0.049	-	-
	-0.002	0.010	0.007	0.001	-0.001	0.017	0.000	0.001	0.080	0.057	0.009
	0.003	-	0.002	0.001	-	0.002	-	-	-	-	-
CAT-Net	0.367	0.341	0.266	0.208	0.143	0.228	0.038	0.029	0.455	-	-
	0.739	0.688	0.723	0.567	0.300	0.468	0.076	0.057	**0.741**	**0.673**	0.289
	0.866	-	0.573	0.553	-	0.103	-	-	-	-	-
Splicebuster	0.164	0.015	-0.016	-0.001	-0.024	0.137	0.070	0.031	-0.081	-	-
	0.332	0.031	-0.054	-0.038	-0.050	0.281	0.140	0.063	-0.137	-0.121	-0.115
	0.144	-	-0.059	-0.034	-	0.155	-	-	-	-	-
EXIF as Language Mean Shift	0.142	0.057	0.005	-0.001	0.001	0.127	0.020	0.008	-0.004	-	-
	0.282	0.109	0.015	-0.002	0.002	0.261	0.039	0.017	-0.008	-0.019	-0.022
	0.214	-	0.009	-0.002	-	0.163	-	-	-	-	-
EXIF as Language NCuts	0.146	0.043	0.013	-0.005	-0.039	0.108	0.023	0.006	-0.014	-	-
	0.293	0.086	0.036	-0.013	-0.082	0.221	0.046	0.013	-0.024	-0.026	-0.027
	0.133	-	-0.010	-0.012	-	0.210	-	-	-	-	-
PSCC-Net	0.180	0.205	0.174	0.156	0.135	0.113	0.004	0.006	0.281	-	-
	0.363	0.413	0.472	0.424	0.283	0.252	0.007	0.012	0.459	0.062	0.037
	0.561	-	0.354	0.419	-	0.004	-	-	-	-	-
TruFor	0.323	0.300	0.264	0.184	0.224	0.406	0.170	0.065	0.342	-	-
	0.650	0.605	0.717	0.502	0.368	**0.834**	0.300	0.130	0.556	0.433	0.265
	0.615	-	0.667	0.497	-	0.521	-	-	-	-	-
FOCAL	0.446	0.429	0.283	0.213	0.288	0.255	0.168	0.143	0.311	-	-
	0.900	**0.866**	**0.769**	**0.580**	**0.604**	0.524	**0.336**	**0.287**	0.506	0.470	**0.297**
	0.922	-	**0.718**	**0.572**	-	**0.530**	-	-	-	-	-

Table 7.8: Localization performance in terms of the mean weighted MCC score ($\text{MCC}_w^{v_1}$) in popular datasets, for original dataset with tampered and pristine images, original dataset with only tampered images and only tampered images through *Facebook*. In **bold**, the highest score in each dataset, and underlined, the second highest one.

Secondly, the results confirm that the methods chosen to be included in *Photo-Holmes* are complementary to each other. No method is able to solve the problem at hand perfectly, but each method tries to solve the problem from different perspectives, contributing to the overall goal of the library. This also suggests that there is significant room for improvement in developing new methods and enhancing performance, particularly in real-life scenarios.

Regarding non-semantic evaluation, the following remarks can be made:

- Methods that were created to find inconsistencies in different traces that the camera pipeline leaves in the images are the ones that excel in each dataset: Noisesniffer in noise, ZERO in JPEG, and Adaptive CFA in CFA.
- ZERO is more reliable in its predictions in the original version than in the version with missing grids, as the latter has a higher amount of false alarms.
- TruFor achieves good results with JPEG and Hybrid datasets but completely fails with the CFA ones.

Method	Columbia	Columbia WebP	CASIA1.0 SP	CASIA1.0 CM	COVERAGE	DSO-1	Korus	Korus WebP	AutoSplice 100	AutoSplice 90	AutoSplice 75
Adaptive CFA	0.127 0.257 0.189	0.113 0.227 -	0.038 0.103 0.099	0.026 0.070 0.070	0.055 0.114 -	0.071 0.145 0.115	0.067 0.134 -	0.027 0.054 -	0.155 0.252 -	- 0.243 -	- 0.240 -
Noisesniffer	0.035 0.071 0.031	0.008 0.017 -	0.017 0.047 0.048	0.008 0.022 0.022	0.019 0.040 -	0.103 0.212 0.196	0.066 0.132 -	0.041 0.082 -	0.021 0.033 -	- 0.024 -	- 0.027 -
ZERO	0.000 0.000 0.000	0.004 0.008 -	0.018 0.048 0.000	0.000 0.001 0.000	0.004 0.008 -	0.053 0.108 0.000	0.001 0.003 -	0.011 0.023 -	0.000 0.000 -	- 0.000 -	- 0.000 -
ZERO with missing grids	0.001 0.003 0.001	0.037 0.075 -	0.018 0.049 0.002	0.004 0.011 0.011	0.004 0.008 -	0.111 0.228 0.004	0.002 0.004 -	0.022 0.053 -	0.175 0.284 -	- 0.005 -	- 0.006 -
DQ	0.088 0.178 0.119	0.090 0.182 -	0.021 0.056 0.053	0.013 0.036 0.036	0.047 0.099 -	0.053 0.109 0.069	0.023 0.046 -	0.023 0.047 -	0.156 0.255 -	- 0.157 -	- 0.084 -
CAT-Net	0.344 0.815 0.870	0.312 0.630 -	0.234 0.636 0.484	0.172 0.469 0.456	0.114 0.240 -	0.188 0.386 0.090	0.022 0.043 -	0.018 0.035 -	**0.451** **0.734** -	- **0.654** -	- **0.315** -
Splicebuster	0.142 0.287 0.163	0.055 0.110 -	0.016 0.043 0.040	0.012 0.034 0.035	0.023 0.048 -	0.098 0.200 0.118	0.051 0.101 -	0.029 0.059 -	0.050 0.082 -	- 0.088 -	- 0.089 -
EXIF as Language Mean Shift	0.153 0.309 0.223	0.080 0.161 -	0.018 0.050 0.046	0.010 0.027 0.027	0.034 0.070 -	0.111 0.228 0.156	0.026 0.052 -	0.021 0.043 -	0.071 0.115 -	- 0.082 -	- 0.067 -
EXIF as Language NCuts	0.272 0.549 0.439	0.173 0.350 -	0.045 0.122 0.104	0.022 0.059 0.059	0.037 0.078 -	0.181 0.371 0.322	0.038 0.075 -	0.029 0.058 -	0.173 0.281 -	- 0.274 -	- 0.268 -
PSCC-Net	0.215 0.433 0.541	0.216 0.435 -	0.147 0.398 0.280	0.130 0.353 0.347	0.114 0.240 -	0.113 0.232 0.016	0.028 0.056 -	0.018 0.036 -	0.266 0.433 -	- 0.051 -	- 0.023 -
TruFor	0.334 0.674 0.649	0.307 0.619 -	0.235 0.636 0.579	**0.209** 0.423 0.419	0.190 0.300 -	**0.367** **0.754** 0.447	**0.128** 0.189 -	0.050 0.100 -	0.313 0.509 -	- 0.378 -	- 0.232 -
FOCAL	**0.437** **0.881** **0.934**	**0.421** **0.848** -	**0.263** **0.713** **0.661**	0.191 **0.521** **0.515**	**0.248** **0.521** -	0.218 0.448 **0.453**	0.132 **0.263** -	**0.113** **0.227** -	0.356 0.580 -	- 0.451 -	- 0.303 -

Table 7.9: Localization performance in terms of the mean weighted IoU score ($\mathrm{IoU}_w^{v_1}$) in popular datasets, for original dataset with tampered and pristine images, original dataset with only tampered images and only tampered images through *Facebook*. In **bold**, the highest score in each dataset, and underlined, the second highest one.

- FOCAL's performance proves that the method does not look for inconsistencies in any specific trace but rather performs image detection at a semantic level.
- DQ gives low probabilities in the heatmaps, but the higher probabilities are in the right places, which means that with the correct threshold, or including the SVM, DQ achieves good localization with JPEG inconsistencies.

From the popular dataset evaluation, we can make the following comments:

- All methods, except Noisesniffer and ZERO, tend to hallucinate forgeries even when there are none. This result is alarming as it shows that the methods tend always to flag a forgery. The community should address this issue as it is crucial to use them in real-world scenarios.
- All types of forgeries presented in Section 2.2.5 can be detected by at least one of the methods selected,as visible in Figure 7.12, validating the claim made that the selection of methods was appropiate.

Method	Columbia	Columbia WebP	CASIA1.0 SP	CASIA1.0 CM	COVERAGE	DSO-1	Korus	Korus WebP	AutoSplice 100	AutoSplice 90	AutoSplice 75
Adaptive CFA	0.183	0.181	0.066	0.048	0.096	0.120	0.102	0.049	0.239	-	-
	0.370	0.366	0.178	0.129	0.201	0.246	0.204	0.099	0.389	0.379	0.375
	0.314	-	0.171	0.129	-	0.202	-	-	-	-	-
Noisesniffer	0.050	0.012	0.027	0.013	0.029	0.139	0.099	0.065	0.033	-	-
	0.101	0.023	0.073	0.036	0.062	0.285	0.197	0.130	0.053	0.038	0.041
	0.044	-	0.074	0.036	-	0.273	-	-	-	-	-
ZERO	0.000	0.006	0.020	0.000	0.004	0.057	0.002	0.021	0.000	-	-
	0.000	0.011	0.054	0.001	0.009	0.116	0.004	0.042	0.000	0.000	0.000
	0.000	-	0.000	0.000	-	0.000	-	-	-	-	-
ZERO with missing grids	0.002	0.053	0.021	0.007	0.006	0.158	0.003	0.039	0.236	-	-
	0.002	0.107	0.057	0.019	0.013	0.324	0.006	0.094	0.384	0.008	0.010
	0.001	-	0.003	0.020	-	0.008	-	-	-	-	-
DQ	0.148	0.151	0.038	0.025	0.083	0.094	0.043	0.044	0.241	-	-
	0.299	0.305	0.104	0.068	0.175	0.193	0.087	0.087	0.393	0.268	0.154
	0.211	-	0.099	0.069	-	0.128	-	-	-	-	-
CAT-Net	0.383	0.353	0.271	0.212	0.154	0.231	0.033	0.027	**0.513**	-	-
	0.773	0.713	0.735	0.577	0.322	0.474	0.066	0.053	**0.835**	**0.767**	**0.433**
	0.902	-	0.589	0.563	-	0.125	-	-	-	-	-
Splicebuster	0.203	0.091	0.027	0.022	0.040	0.150	0.084	0.052	0.086	-	-
	0.410	0.183	0.075	0.061	0.085	0.307	0.168	0.103	0.140	0.150	0.152
	0.260	-	0.071	0.063	-	0.196	-	-	-	-	-
EXIF as Language Mean Shift	0.227	0.135	0.034	0.019	0.062	0.167	0.048	0.040	0.124	-	-
	0.458	0.274	0.093	0.052	0.130	0.344	0.096	0.080	0.201	0.150	0.124
	0.356	-	0.087	0.053	-	0.253	-	-	-	-	-
EXIF as Language NCuts	0.302	0.210	0.068	0.037	0.064	0.213	0.062	0.050	0.233	-	-
	0.609	0.425	0.185	0.102	0.133	0.437	0.124	0.099	0.380	0.374	0.371
	0.495	-	0.158	0.103	-	0.387	-	-	-	-	-
PSCC-Net	0.289	0.292	0.176	0.157	0.170	0.172	0.051	0.034	0.354	-	-
	0.580	0.589	0.478	0.428	0.358	0.353	0.101	0.068	0.577	0.085	0.040
	0.681	-	0.359	0.422	-	0.027	-	-	-	-	-
TruFor	0.381	0.361	0.271	0.191	0.245	**0.408**	0.173	0.080	0.396	-	-
	0.769	0.728	0.734	0.520	0.461	**0.838**	0.279	0.159	0.646	0.515	0.349
	0.740	-	0.685	0.516	-	0.574	-	-	-	-	-
FOCAL	**0.459**	**0.448**	**0.289**	**0.219**	**0.298**	0.281	**0.177**	**0.155**	0.428	-	-
	0.927	**0.904**	**0.784**	**0.596**	**0.626**	0.576	**0.355**	**0.311**	0.696	0.585	0.429
	0.950	-	**0.739**	**0.589**	-	**0.581**	-	-	-	-	-

Table 7.10: Localization performance in terms of the mean weighted F1 score ($F1_w^{v_1}$) in popular datasets, for original dataset with tampered and pristine images, original dataset with only tampered images and only tampered images through *Facebook*. In **bold**, the highest score in each dataset, and underlined, the second highest one.

- Although methods aimed at detecting generative inpainting were not included, some methods still achieved good performance in datasets with this type of forgery, such as CAT-Net, TruFor, and FOCAL.
- Noisesniffer competes with deep learning-based methods in Korus, which proved to be the most challenging dataset included, proving that classical methods are still relevant in the state of the art.
- FOCAL seems to be the most robust method when it comes to various types of compression, where most methods are challenged.

Finally, regarding the detection performance, it is important to note that the community has neglected this problem when proposing new methods, as only 5 out of the selected ten methods provide detection scores. In addition, methods like EXIF as Language in their original version provide a score that is difficult to interpret. Methods that output a probability, like PSCC-Net and TruFor, seem to perform worse than in localization, probably because the probability output is

Method	Columbia	Columbia WebP	CASIA1.0 SP	CASIA1.0 CM	COVERAGE	DSO-1	Korus	Korus WebP	AutoSplice 100	AutoSplice 90	AutoSplice 75
Adaptive CFA	0.528	0.561	0.573	0.560	0.537	0.600	0.585	0.519	0.555	-	-
	0.554	0.549	0.565	0.559	0.539	0.570	0.585	0.518	0.562	0.555	0.549
	0.506	-	0.550	0.558	-	0.510	-	-	-	-	-
DQ	0.702	0.731	0.496	0.296	0.576	0.548	0.488	0.491	0.907	-	-
	0.514	0.570	0.551	0.483	0.567	0.555	0.465	0.471	0.811	0.803	0.543
	0.564	-	0.585	0.484	-	0.495	-	-	-	-	-
CAT-Net	0.928	0.913	**0.981**	**0.920**	0.762	0.729	0.535	0.530	0.895	-	-
	0.921	0.913	**0.983**	**0.923**	0.751	0.743	0.535	0.531	**0.935**	**0.930**	**0.777**
	0.979	-	0.918	**0.918**	-	0.539	-	-	-	-	-
Splicebuster	0.779	0.582	0.415	0.446	0.394	0.776	0.670	0.612	0.414	-	-
	0.827	0.554	0.414	0.426	0.395	0.827	0.700	0.628	0.399	0.400	0.415
	0.676	-	0.400	0.433	-	0.724	-	-	-	-	-
EXIF as Language Mean Shift	0.905	0.843	0.629	0.416	0.578	0.950	0.667	0.624	0.694	-	-
	0.861	0.827	0.570	0.498	0.560	0.927	0.666	0.622	0.597	0.510	0.449
	0.915	-	0.536	0.498	-	0.882	-	-	-	-	-
PSCC-Net	0.549	0.871	0.858	0.787	0.812	0.857	0.499	0.562	**0.925**	-	-
	0.811	0.874	0.857	0.779	0.796	0.767	0.503	0.574	0.864	0.608	0.550
	0.922	-	0.815	0.779	-	0.518	-	-	-	-	-
TruFor	**0.937**	**0.927**	0.970	0.906	**0.904**	**0.995**	**0.744**	**0.670**	0.870	-	-
	0.926	**0.924**	0.980	0.906	0.905	**0.998**	**0.745**	**0.665**	0.849	0.849	**0.777**
	0.930	-	**0.971**	0.905	-	**0.919**	-	-	-	-	-

Table 7.11: Localization performance in terms of the AUROC in popular datasets, for original dataset with tampered and pristine images, original dataset with only tampered images and only tampered images through *Facebook*. In **bold**, the highest score in each dataset, and underlined, the second highest one.

Method	Columbia	Columbia WebP	CASIA1.0 SP	CASIA1.0 CM	COVERAGE	DSO-1	Korus	Korus WebP	AutoSplice 100
Noisesniffer	0.142	-0.063	-0.151	0.040	0.066	-0.045	0.206	**0.075**	0.118
ZERO	0.000	0.013	0.198	0.011	0.076	0.307	0.068	0.059	0.143
EXIF as Language	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
PSCC-Net	0.131	0.464	0.460	**0.490**	0.100	0.173	0.012	-0.014	**0.701**
TruFor	**0.805**	**0.686**	**0.608**	0.448	**0.325**	**0.709**	**0.315**	0.000	0.000

Table 7.12: Detection performance in terms of the dataset-level weighted MCC score ($\mathrm{MCC}_w^{v_2}$) for original dataset with tampered and pristine images. In **bold**, the highest score in each dataset, and underlined, the second highest one.

Method	Columbia	Columbia WebP	CASIA1.0 SP	CASIA1.0 CM	COVERAGE	DSO-1	Korus	Korus WebP	AutoSplice 100	AutoSplice 90	AutoSplice 75
Noisesniffer	0.246	0.092	0.232	0.264	0.261	0.457	**0.512**	0.459	0.255	-	-
	0.289	0.106	0.395	0.451	0.341	0.905	0.873	0.795	0.284	0.182	0.170
	0.237	-	0.388	0.438	-	0.958	-	-	-	-	-
ZERO	0.000	0.073	0.065	0.002	0.011	0.168	0.009	0.432	0.052	-	-
	0.000	0.078	0.065	0.002	0.011	0.168	0.009	0.718	0.052	0.000	0.000
	0.000	-	0.000	0.000	-	0.000	-	-	-	-	-
EXIF as Language	0.496	0.496	0.369	0.367	**0.476**	0.487	0.500	**0.500**	0.614	-	-
	1.000	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
	1.000	**1.000**	**1.000**	**1.000**	-	**1.000**	-	-	-	-	-
PSCC-Net	0.505	0.545	0.348	0.380	0.171	0.110	0.407	0.181	0.723	-	-
	0.992	0.654	0.361	0.396	0.194	0.113	0.679	0.223	0.728	0.103	0.071
	0.791	-	0.201	0.397	-	0.018	-	-	-	-	-
TruFor	**0.826**	**0.737**	**0.614**	**0.481**	0.426	**0.751**	0.323	0.269	**0.761**	-	-
	0.965	0.969	0.822	0.644	0.526	0.947	0.352	0.368	0.761	0.596	0.482
	0.973	-	0.768	0.644	-	0.599	-	-	-	-	-

Table 7.13: Detection performance in terms of the dataset-level weighted IoU score ($\mathrm{IoU}_w^{v_2}$) in popular datasets, for original dataset with tampered and pristine images, original dataset with only tampered images and only tampered images through *Facebook*. In **bold**, the highest score in each dataset, and underlined, the second highest one.

Method	Columbia	Columbia WebP	CASIA1.0 SP	CASIA1.0 CM	COVERAGE	DSO-1	Korus	Korus WebP	AutoSplice 100	AutoSplice 90	AutoSplice 75
Noisesniffer	0.395	0.168	0.376	0.418	0.413	0.628	**0.677**	0.629	0.406	-	-
	0.448	0.191	0.566	0.622	0.508	0.950	0.932	0.886	0.442	0.306	0.291
	0.383	-	0.559	0.609	-	0.978	-	-	-	-	-
ZERO	0.000	0.135	0.122	0.004	0.022	0.288	0.018	**0.836**	0.098	-	-
	0.000	0.144	0.122	0.004	0.022	0.288	0.018	0.836	0.098	0.001	0.001
	0.000	-	0.000	0.000	-	0.000	-	-	-	-	-
EXIF as Language	0.663	0.663	0.539	0.537	**0.645**	0.655	0.667	0.667	0.761	-	-
	1.000	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
	1.000	-	**1.000**	**1.000**	-	**1.000**	-	-	-	-	-
PSCC-Net	0.672	0.706	0.516	0.551	0.292	0.199	0.579	0.306	0.839	-	-
	0.996	0.791	0.531	0.567	0.324	0.203	0.809	0.365	0.843	0.187	0.131
	0.883	-	0.334	0.568	-	0.035	-	-	-	-	-
TruFor	**0.905**	**0.849**	**0.761**	**0.649**	0.580	**0.857**	0.488	0.424	**0.865**	-	-
	0.982	0.984	0.902	0.784	0.690	0.973	0.521	0.538	0.865	0.747	0.650
	0.986	-	0.869	0.783	-	0.749	-	-	-	-	-

Table 7.14: Detection performance in terms of the dataset-level weighted F1 score ($F1_w^{v_2}$) in popular datasets, for original dataset with tampered and pristine images, original dataset with only tampered images and only tampered images through *Facebook*. In **bold**, the highest score in each dataset, and underlined, the second highest one.

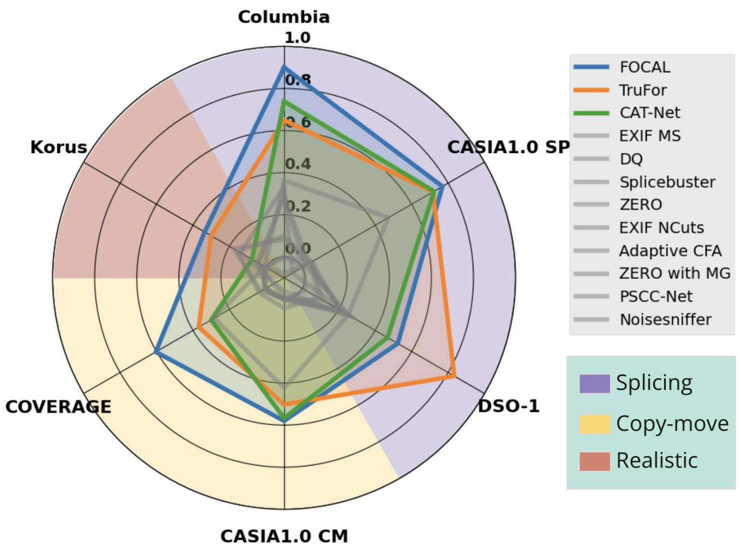


Figure 7.12: A spider diagram illustrating the $MCC_w^{v_1}$ performance for the most notorious methos in the popular datasets, according to Table 7.8. The absolute value of the MCC is plotted on every axis, where each axis corresponds to a different dataset. The top three perforing methods are outlined with a color, while the remaining methods' performance is shown as a gray shade. Different dataset types are also highlighted with different background colors. This figure is used to show the complimentary nature of the methods across these datasets and forgery types.

still low when the image is indeed forged. Lastly, methods like Noisesniffer and ZERO are the most reliable ones in this sense, as they are less likely to hallucinate forgeries. Given that the detection output is taken out of the predicted mask, this remark still holds true for the detection problem.

This chapter made use of the Benchmark tool of PhotoHolmes presented in the previous chapter. We evaluated ten state-of-the-art methods in the selected datasets with different metrics. We analyzed the reports and made interpretations that supported the results.

This was the last chapter that included the content of this thesis. The following chapter will present the conclusions and future work.

This page intentionally left blank.

Chapter 8

Conclusions and future work

"The work is its own reward."

Sherlock Holmes

The problem addressed by this thesis was the detection of digital image forgeries. This is a highly relevant social issue that has been addressed by members of the image processing community in recent years. Our contribution to the matter was introducing *PhotoHolmes*, a novel open-source *Python* library designed to run easily and benchmark forgery detection methods. The library comprises ten state-of-the-art methods, a great variety of datasets and metrics, and modules that simplify the execution and benchmark of a method. Most importantly, *PhotoHolmes* is designed to be extensible, encouraging the community to contribute new methods, datasets, metrics, and tools.

In Chapter 1, we introduced the problem and its history and reported related and previous works.

Chapter 2 consisted of the analysis of the image acquisition pipeline, starting with the noise introduced by the sensors, followed by how it was modified by the subsequent stages of the image processing pipeline: demosaicing, then color correction, and the different formats an image can be compressed and saved into. This analysis permits to understand which parts of the imaging pipeline leave traces in an image, so in case there are inconsistencies in these traces in different parts of the same image, it could indicate the presence of a forgery. This chapter also delved into important definitions, such as what constitutes a forgery, how a forgery can be detected, and the different falsification types. In addition, the chapter explores the main concepts behind machine learning to give context to some of the methods and techniques that follow in the study.

After understanding what constitutes a digital image, a forgery, and how they can be detected, Chapter 3 laid out the basis of how data is used in the field of image forensics. In this chapter, we explained the importance of data in general and described the particularities of data in the subject at hand. The predominant section in this chapter describes in detail the datasets chosen for an extensive evaluation of a method's performance. We included popular datasets, contemplating various forgery types and other properties, and even some versions containing images uploaded to *Facebook* to account for possible transformation an image goes through when it is shared through this popular social network. We also introduced the WebP compressed versions of two popular datasets because we identified a lack of datasets in this increasingly popular format. We also described the inclusion of the Trace dataset to do a non-semantic evaluation, from which we created the miniTrace dataset, a smaller version of the original Trace.

Chapter 4 delved into another important piece of this dissertation: the metrics used to evaluate performance. To do this, we first examine the importance of selecting good metrics. Taking into account that we were evaluating the methods on datasets, we explored two options on how to aggregate metrics to get a final score and how each aggregation affects the result. Within the chapter, we presented 8 different metrics and 3 weighted metrics that allow for comparing the different kinds of outputs methods can have.

Next, Chapter 5 describes the methods studied in this work. From all of the different methods we analyzed, we chose ten, taking into account diversity, performance, and how interesting the presented idea was, obtaining a set of methods that work in a complementary fashion. We explained the mechanics behind each method and gave some insights regarding what we would expect from each method at the time of evaluation.

In Chapter 6, we presented the *PhotoHolmes* library. This chapter includes a detailed explanation of every module included in the library, explaining how they were designed and with usage examples in every case. The two main features presented in this chapter are the Command Line Interface (CLI), which allows users to run a method on an image, and the Benchmark, which allows users to easily evaluate a method on multiple metrics on a certain dataset.

Finally, Chapter 7 presents the evaluation and analysis of the results obtained using the Benchmark tool in *PhotoHolmes*. We evaluated all implemented methods in all of the included datasets in the weighted metrics and the AUROC.

The work done during this thesis opens up a diverse variety of different follow-up works. Firstly, we intend to build a community around *PhotoHolmes* that proposes new methods, datasets, and metrics to be included in the library and that contribute to it. Using *PhotoHolmes* will result in more reproducible research and a lower technical barrier for those interested in researching image forensics. With that objective, the first follow-up work is the maintenance and continuous development of the library.

Secondly, new datasets inside miniTrace could be generated, such as one that includes double compression, the appearance of missing JPEG grids or WebP compression. This will enrich the variety of datasets available for non-semantic evaluation, extending the trace detection analysis that can be applied to a method. This type of analysis is crucial, especially in deep-learning methods, giving us a way to interpret what information the method might use when predicting.

Also, given the thorough analysis of the included methods, we were able to pinpoint many of the method's weaknesses, which gives place to the proposal of new methods or method improvements that could be easily evaluated by using the

PhotoHolmes Benchmark tool. If successful, those methods can be included in newer library releases.

Regarding new methods, this thesis only focused on methods aimed at detecting forgeries on digital images. However, the recent threats caused by deep-fake images and videos raise the need to open the scope of the library and make accessible methods that solely focus on these kinds of manipulations.

Lastly, while the CLI allows for a quick evaluation of a suspicious image, for people with no coding experience, it might still be a complex procedure. On top of that, the user also has to understand which methods to run and what the results mean. Hence, to make this method more accessible, a web interface or a graphic interface that allows a user to upload an image and receive a full report can be designed using the implementations from the *PhotoHolmes* library. In the meantime, we are also available to help individuals and the press analyze suspicious images and provide them with comprehensive reports since the methods may fail or not even present sufficient confidence in the predictions made. Caution should be made regarding how the results are interpreted, and, as we have already seen, none of these methods are infallible, so their predictions should not be taken as truth indiscriminately.

To sum up, with our survey of the state of the art, we presented *PhotoHolmes*, a novel *Python* library for forgery detection in digital images. The proposed library comprises a wide array of methods, datasets, and metrics that, combined through the *Benchmark* module, allows a user to do comparative studies of state-of-the-art methods. Using these tools, we did an extensive report and analysis of the methods selected.

Given the designed principles, we aim to keep extending *PhotoHolmes* by keeping it up to date with the state of the art, with the hope of building a community that will use it and help expand it.

This page intentionally left blank.

Appendix A

Photoholmes library code

"I am a brain, Watson. The rest of me is a mere appendix."

Sherlock Holmes

A.1 Dataset

A.1.1 Base

```python
import logging
import os
from abc import ABC, abstractmethod
from typing import Dict, List, Literal, Optional, Tuple,
   Union

import torch
from torch import Tensor
from torch.utils.data import Dataset

from photoholmes.preprocessing import PreProcessingPipeline
from photoholmes.utils.image import read_image,
   read_jpeg_data

logging.basicConfig(level=logging.INFO, format="%(levelname)s
    - %(message)s")
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)


class AttributeOverrideError(NotImplementedError):
    """
    Exception raised when a subclass fails to override a
   required class attribute.
    """

    def __init__(self, attribute_name: str):
           message = f"Subclasses must override {attribute_name}
      "
           super().__init__(message)


   class BaseDataset(ABC, Dataset):
       """
       Base class for datasets.

       Subclasses must override the IMAGE_EXTENSION and
       MASK_EXTENSION attributes.
       The _get_paths and _get_mask_path methods must be
       implemented as well, and in some
       cases binarize_mask must alos be overriden.
       """

       IMAGE_EXTENSION: Union[str, List[str]]
       MASK_EXTENSION: Union[str, List[str]]

       def __init__(
           self,
           dataset_path: str,
           preprocessing_pipeline: Optional[
       PreProcessingPipeline] = None,
           load: List[Literal["image", "dct_coefficients", "
       qtables",]] = [
               "image",
               "dct_coefficients",
               "qtables",
           ],
           tampered_only: bool = False,
       ):
           """
           Initialize the dataset.

           Args:
               dataset_path (str): Path to the dataset.
               preprocessing_pipeline (Optional[
       PreProcessingPipeline]): Preprocessing
                   pipeline to apply to the images.
               load (List[Literal["image", "dct_coefficients", "
       qtables"]]): List of
                   items to load. Possible values are "image", "
       dct_coefficients" and
                   "qtables". If the preprocessing_pipeline is
       not None, the load
                   attribute will be ignored and the
       preprocessing pipeline inputs will
                   be used instead.
               tampered_only (bool): If True, only load tampered
        images.
```

```
        Raises:
            FileNotFoundError: If the dataset_path does not
exist.
            AttributeOverrideError: If the subclass has not
overridden the
                IMAGE_EXTENSION and MASK_EXTENSION attributes
.
        """
        if not os.path.exists(dataset_path):
            raise FileNotFoundError(f"Directory {dataset_path
} does not exist.")

        self.load = preprocessing_pipeline.inputs if
preprocessing_pipeline else load
        self.load_jpeg_data = "dct_coefficients" in self.load
 or "qtables" in self.load
        self.load_image_data = "image" in self.load

        if self.load_jpeg_data:
            self.check_jpeg_warning()
        self.check_attribute_override()

        self.dataset_path = dataset_path
        self.tampered_only = tampered_only

        self.preprocessing_pipeline = preprocessing_pipeline

        if preprocessing_pipeline:
            if set(load) != set(preprocessing_pipeline.inputs
):
                logger.warning(
                    "The load attribute and the preprocessing
 pipeline inputs do not "
                    f"match. Using the preprocessing pipeline
 inputs: "
                    f"{preprocessing_pipeline.inputs}"
                )

        self.image_paths, self.mask_paths = self._get_paths(
dataset_path, tampered_only)

 @abstractmethod
 def _get_paths(
        self, dataset_path, tampered_only
 ) -> Tuple[List[str], List[str] | List[str | None]]:
        """
        Abstract method that returns an ordered list of image
 and mask paths, mapped
        in the correct order.
        The correct implementation in a child class must
```

```
follow:
      - Make use of the dataset_path and tampered_only
arguments.
      - In the case of pristine images, the corresponding
mask path must be set to 'None'.
      - Mask paths must be obtained by a correspondance of
 the image path,
      using the _get_mask_path method.

     Args:
         dataset_path (str): Path to the dataset.
         tampered_only (bool): Whether to load only the
tampered images.

     Returns:
         Tuple[List[str], List[str] | List[str | None]]:
Tuple with lists of image and mask paths (or None in
pristine images).
     """
     pass

 @abstractmethod
 def _get_mask_path(self, image_path: str) -> str:
     """Abstract method that returns the corresponding
mask path for a given image path."""
     pass

 def __len__(self) -> int:
     """Return the length of the dataset."""
     return len(self.image_paths)

 def __getitem__(self, idx: int) -> Tuple[Dict, Tensor,
str]:
     """Return the item at the given index."""
     x, mask, image_name = self._get_data(idx)
     if self.preprocessing_pipeline is not None:
         x = self.preprocessing_pipeline(**x)
     return x, mask, image_name

 def _get_data(self, idx: int) -> Tuple[Dict, Tensor, str
]:
     """
     Return the data at the given index.

     Args:
         idx (int): Index of the item to return.

     Returns:
         Tuple[Dict, Tensor, str]: A tuple containing the
data, the mask and the
             image name.
```

```
        """
        x = {}

        image_path = self.image_paths[idx]
        image_name = image_path.split("/")[-1].split(".")[0]

        if self.load_image_data:
            image = read_image(image_path)
            x["image"] = image
        if self.load_jpeg_data:
            dct, qtables = read_jpeg_data(image_path,
suppress_not_jpeg_warning=True)
            if "dct_coefficients" in self.load:
                x["dct_coefficients"] = dct
            if "qtables" in self.load:
                x["qtables"] = qtables

        if self.mask_paths[idx] is None:
            mask = torch.zeros(image.shape[-2:], dtype=torch.
bool)
        else:
            mask_im = read_image(self.mask_paths[idx])
            mask = self._binarize_mask(mask_im)

        return x, mask, image_name

    def _binarize_mask(self, mask_image: Tensor) -> Tensor:
        """
        Overridable method to binarize the mask image.
Binarized masks are boolean
        tensors of one channel, regarding any degree of
tampering as True.
        Arguments:
            mask_image (Tensor): Original mask image.
        Outputs:
            Tensor: Binarized mask image.
        """
        assert (mask_image <= 1).all()
        return (mask_image == 1).float()

    def check_attribute_override(self):
        """
        Check that the subclass has overridden
IMAGE_EXTENSION and MASK_EXTENSION.
        Raises an error if not.
        """
        if not hasattr(type(self), "IMAGE_EXTENSION"):
            raise AttributeOverrideError("IMAGE_EXTENSION")
        if not hasattr(type(self), "MASK_EXTENSION"):
            raise AttributeOverrideError("MASK_EXTENSION")
    def check_jpeg_warning(self):
        """
        Check if the images are in JPEG format. If not, a
   warning is issued.
        """
        if not isinstance(self.IMAGE_EXTENSION, list):
            image_ext = [self.IMAGE_EXTENSION]
        else:
            image_ext = self.IMAGE_EXTENSION
        if not all(
            [
                ext in [".jpg", ".jpeg", ".JPG", ".JPEG", "
   jpg", "jpeg", "JPEG", "JPG"]
                for ext in image_ext
            ]
        ):
            logger.warning(
                "Not all images are in JPEG format. When
   needed, an approximation will "
                "be loaded by compressing the image in
   quality 100."
            )
```

A.1.2 Implementation example

```
import glob
import os
from typing import List, Optional, Tuple

from torch import Tensor

from .base import BaseDataset


class CasiaBaseDataset(BaseDataset):
    IMAGES_TAMPERED_DIR: str
    MASK_TAMPERED_DIR: str
    AUTH_DIR: str = "Au"
    IMAGE_EXTENSION: str = ".jpg"
    MASK_EXTENSION: str = ".png"

    def _get_paths(
        self, dataset_path: str, tampered_only: bool
    ) -> Tuple[List[str], List[str] | List[str | None]]:
        """
        Get the paths of the images and masks in the dataset.

        Args:
            dataset_path (str): Path to the dataset.
            tampered_only (bool): Whether to load only the
```

```
tampered images.

     Returns:
         Tuple[List[str], List[str] | List[str | None]]:
Paths of the images and
             masks.
     """
     image_paths = glob.glob(
         os.path.join(
             dataset_path, self.IMAGES_TAMPERED_DIR, f"*{
self.IMAGE_EXTENSION}"
         )
     )
     mask_paths: List[Optional[str]] = [
         os.path.join(dataset_path, self._get_mask_path(
image_path))
         for image_path in image_paths
     ]

     if not tampered_only:
         pris_paths = glob.glob(
             os.path.join(dataset_path, self.AUTH_DIR, f"
*{self.IMAGE_EXTENSION}")
         )

         pris_msk_paths = [None] * len(pris_paths)
         image_paths += pris_paths
         mask_paths += pris_msk_paths

     return image_paths, mask_paths

 def _get_mask_path(self, image_path: str) -> str:
     """
     Get the path of the mask for the given image path.

     Args:
         image_path (str): Path to the image.

     Returns:
         str: Path to the mask.
     """
     image_filename = image_path.split("/")[-1]
     image_name_list = ".".join(image_filename.split(".")
[:-1]).split("_")
     mask_name = "_".join(image_name_list + ["gt"])
     mask_filename = mask_name + self.MASK_EXTENSION
     return os.path.join(self.MASK_TAMPERED_DIR,
mask_filename)

 def _binarize_mask(self, mask_image: Tensor) -> Tensor:
     """
```

```
        Binarize the mask.

        Args:
            mask_image (Tensor): Mask image.

        Returns:
            Tensor: Binarized mask image.
        """
        return mask_image[0, :, :] > 0


class Casia1SplicingDataset(CasiaBaseDataset):
    """
    Class for the CASIA 1.0 Splicing (Sp) subset.
    """

    IMAGES_TAMPERED_DIR = "Tp/Sp"
    MASK_TAMPERED_DIR = "CASIA 1.0 groundtruth/Sp"


class Casia1CopyMoveDataset(CasiaBaseDataset):
    """
    Class for the CASIA 1.0 Copy Move (CM) subset.
    """

    IMAGES_TAMPERED_DIR = "Tp/CM"
    MASK_TAMPERED_DIR = "CASIA 1.0 groundtruth/CM"
```

A.2 Preprocessing

A.2.1 Base

```
from abc import ABC, abstractmethod
from typing import Any, Dict


class BasePreprocessing(ABC):
    @abstractmethod
    def __call__(self, *args, **kwargs) -> Dict[str, Any]:
        pass
```

A.2.2 Pipeline

```
   import logging
from typing import Any, Dict, List, Literal

from photoholmes.preprocessing.base import BasePreprocessing
```

```
logger = logging.getLogger(__name__)


class PreProcessingPipeline:
    """
    A pipeline of preprocessing transforms. In this library,
   the standard way of defining
    the preprocessing of a method is by creating an instance
   of this class with the corresponding
    sequence of transforms.
    """

    inputs: List[Literal["image", "dct_coefficients", "
   qtables"]]
    outputs_keys: List[str]

    def __init__(
        self,
        transforms: List[BasePreprocessing],
        inputs: List[Literal["image", "dct_coefficients", "
   qtables"]],
        outputs_keys: List[str],
    ) -> None:
        """
        Initializes a new preprocessing pipeline.

        Args:
            transforms (List[BasePreprocessing]): A list of
   preprocessing transforms to
                apply to the input.
            inputs (List[str]): the inputs that the pipeline
   will receive.
            outputs_keys (List[str]): the keys of the outputs
    that the pipeline will
                return. These must coincide with the keyword
   arguments of the predict and benchmark methods.
        """
        self.transforms = transforms
        self.inputs = inputs
        self.outputs_keys = outputs_keys

    def __call__(self, **kwargs) -> Dict[str, Any]:
        """
        Applies the preprocessing pipeline to the input.

        Args:
            **kwargs: Keyword arguments representing the
   input to the pipeline.

        Returns:
            Dict[str, Any]: A dictionary with the output of
      the last transform in the
                                  pipeline.
           """
           self._check_inputs(kwargs)

           for t in self.transforms:
               kwargs = t(**kwargs)

           return {k: v for k, v in kwargs.items() if k in self.
      outputs_keys}

       def _check_inputs(self, inputs: Dict[str, Any]) -> None:
           """
           Checks the inputs required are included in the ones
      declared in the pipeline and raises
            a Warning for any input not used in the pipeline.
           """
           for input_ in self.inputs:
               if input_ not in inputs:
                   raise ValueError(f"Missing input {input_} in
      inputs")

           for input_ in inputs.keys():
               if input_ not in self.inputs:
                   logger.warn(f"Input {input_} is not used by
      the pipeline")

       def append(self, transform: BasePreprocessing):
           self.transforms.append(transform)

       def insert(self, index: int, transform: BasePreprocessing
      ):
           self.transforms.insert(index, transform)
```

A.3 Method

A.3.1 Base

```
import logging
from abc import ABC, abstractmethod
from pathlib import Path
from typing import Any, Dict, Optional, TypedDict, TypeVar,
    Union

import torch
from numpy.typing import NDArray
from torch import Tensor
from torch.nn import Module
from typing_extensions import NotRequired
from photoholmes.utils.generic import load_yaml

logging.basicConfig(level=logging.INFO, format="%(levelname)s
    - %(message)s")
log = logging.getLogger(__name__)
log.setLevel(logging.INFO)


T = TypeVar("T", NDArray, Tensor)


class BenchmarkOutput(TypedDict):
    """
    Structure of the output expected from the benchmark
   method.

    Expected keys:
        - heatmap: a probability map predicted by the method.
        - mask: a binary mask predicted by the method.
        - detection: score between 0 and 1, where 1 indicates
    a forged image.

    Extra keys:
        - extra_outputs: any extra outputs that the method
   might have
              and could be useful to save.
    """

    heatmap: Optional[Tensor]
    mask: Optional[Tensor]
    detection: Optional[Tensor]
    extra_outputs: NotRequired[Dict[str, Any]]


class BaseMethod(ABC):
    """
    Abstract class as a base for the methods.
    Every method should inherit from this class (or
   BaseTorchMethod) and implement the
    'predict', 'benchmark' and '__init__' methods to enable
   their utility within the library.
    """

    device: torch.device

    def __init__(self) -> None:
        self.device = torch.device("cpu")

    def predict(self, *args, **kwargs) -> Any:
        """
        Runs method on an image's data, and returns the
    output of the original implementation.
        """
        raise NotImplementedError("Method ‘predict‘ not
    implemented")

    @abstractmethod
    def benchmark(self, *args, **kwargs) -> BenchmarkOutput:
        """
        Runs method on an image's data and returns the output
    in the benchmark
        format (BenchMarkOutput).
        """
        return {"heatmap": None, "mask": None, "detection":
    None}

    @classmethod
    def from_config(cls, config: Optional[str | Dict[str, Any
    ] | Path]):
        """
        Instantiate the model from configuration dictionary
    or yaml.

        Params:
            config: path to the yaml configuration or a
    dictionary with
                    the parameters for the model.
        """
        if isinstance(config, (str, Path)):
            config = load_yaml(config)

        if config is None:
            config = {}

        return cls(**config)

    def to_device(self, device: Union[str, torch.device]):
        """Send the model to the device."""
        logging.warning(
            f"Method {self.__class__} isn't a TorchMethod, so
    it can't be sent "
            "to a device. If the method utilizes a torch
    model as a feature extractor,"
            "override this method to sent it to the device."
        )
        self.device = torch.device("cpu")


class BaseTorchMethod(BaseMethod, Module):
    """
    Abstract class as a base for methods that are end-to-end
Torch modules.
 The child classes must implement the 'predict', '
benchmark' and '__init__' methods
 to enable their utility within the library.
 """

 def __init__(self, *args, **kwargs) -> None:
     Module.__init__(self, *args, **kwargs)
     BaseMethod.__init__(self)

 def load_weights(self, weights: Union[str, Path, dict]):
     """Load weights from a dictionary or a file when
given its path."""
     if isinstance(weights, (str, Path)):
         weights_ = torch.load(weights, map_location=self.
device)
     else:
         weights_ = weights

     if "state_dict" in weights_.keys():
         weights_ = weights_["state_dict"]

     self.load_state_dict(
         weights_, assign=True
     )  # FIXME: asign limits torch version to >=2.1

 def to_device(self, device: Union[str, torch.device]):
     """Send the model to the device."""
     self.to(device)
     self.device = torch.device(device)
```

A.4 Benchmark

```
import json
import logging
import os
import time
from pathlib import Path
from typing import Any, Dict, List, Literal, Optional, Union

import numpy as np
import torch
from torchmetrics import Metric, MetricCollection
from tqdm import tqdm

from photoholmes.datasets.base import BaseDataset
from photoholmes.methods.base import BaseMethod,
    BenchmarkOutput
```

```
logging.basicConfig(format="%(levelname)s - %(message)s")
IO_MESSAGE = 11
logging.addLevelName(IO_MESSAGE, "IO_MESSAGE")


def io_message(self, message, *args, **kws):
    if self.isEnabledFor(IO_MESSAGE):
        self._log(IO_MESSAGE, message, args, **kws)


logging.Logger.io_message = io_message
log = logging.getLogger(__name__)

verbose_dict = {
    0: logging.WARNING,
    1: logging.INFO,
    2: IO_MESSAGE,
}


class Benchmark:
    """
    Benchmark class for evaluating the performance of image
   processing methods.

    Attributes:
        save_method_outputs (bool): Whether to save the
   method outputs.
        save_extra_outputs (bool): Whether to save extra
   outputs.
        save_metrics_flag (bool): Whether to save metrics.
        output_path (Path): Path to the output folder.
        device (torch.device): Device for computation.
        use_existing_output (bool): Whether to use existing
   saved outputs.
        verbose (int): Verbosity level.

    Methods:
        run(method, dataset, metrics):
            Run the benchmark using the specified method,
   dataset, and metrics.
    """

    def __init__(
        self,
        save_method_outputs: bool = True,
        save_extra_outputs: bool = False,
        save_metrics: bool = True,
        output_folder: str = "output/",
        device: str = "cpu",
        use_existing_output: bool = True,
     verbose: Literal[0, 1, 2] = 1,
 ):
     """
     Args:
         save_method_outputs (bool): Whether to save the
method outputs.
             Default is True.
         save_extra_outputs (bool): Whether to save extra
outputs.
             Default is False.
         save_metrics (bool): Whether to save metrics.
Default is True.
         output_folder (str): Folder to save outputs.
Default is "output/".
         device (str): Device for computation (e.g., "cpu"
 or "cuda").
             Default is "cpu".
         use_existing_output (bool): Whether to use
existing saved outputs.
             Default is True.
         verbose (Literal[0, 1, 2]): Verbosity level (0,
1, or 2). Default is 1.
     """
     self.save_method_outputs = save_method_outputs
     self.save_extra_outputs = save_extra_outputs
     self.save_metrics_flag = save_metrics
     self.output_path = Path(output_folder)
     self.use_existing_output = use_existing_output
     self.verbose = verbose

     if self.verbose not in verbose_dict:
         log.warning(
             f"Invalid verbose level '{self.verbose}'. "
             f"Using default verbose level '1'."
         )
         self.verbose = 1

     log.setLevel(verbose_dict[self.verbose])

     if device.startswith("cuda") and not torch.cuda.
is_available():
         log.warning(
             f"Requested device '{device}' is not
available. Falling back to 'cpu'."
         )
         self.device = torch.device("cpu")
     else:
         self.device = torch.device(device)

     self._mask = False
     self._heatmap = False
```

```
        self._detection = False

    def run(
        self,
        method: BaseMethod,
        dataset: BaseDataset,
        metrics: Union[MetricCollection, List[Metric]],
    ):
        """
        Run the benchmark using the specified method, dataset
    , and metrics.

        Args:
            method (BaseMethod): The method to evaluate.
            dataset (BaseDataset): Dataset to run the
    evaluation on.
            metrics (MetricCollection): Collection of metrics
     to compute.

        Returns:
            dict: Computed metrics for the benchmark.
        """
        log.info(f"Using device: {self.device}")
        if method.device != self.device:
            method.to_device(self.device)

        if isinstance(metrics, list):
            metrics = MetricCollection(metrics)

        output_path = (
            self.output_path
            / method.__class__.__name__.lower()
            / dataset.__class__.__name__.lower()
        )
        self._print_setup_message(method, dataset, metrics,
    output_path)

        heatmap_metrics = metrics.clone(prefix="heatmap").to(
            self.device, dtype=torch.float32
        )
        mask_metrics = metrics.clone(prefix="mask").to(self.
    device, dtype=torch.float32)
        detection_metrics = metrics.clone(prefix="detection")
    .to(
            self.device, dtype=torch.float32
        )

        image_count = 0
        for data, mask, image_name in tqdm(dataset, desc="
    Processing Images"):  # type: ignore
            output = None
```

```
         if self.use_existing_output:
             output = self._load_existing_output(
output_path, image_name)

         if output is None:
             data_on_device = self._dict_to_device(data)
             output = method.benchmark(**data_on_device)

             if self.save_method_outputs:
                 self._save_predicted_output(output_path,
image_name, output)

         mask = mask.to(self.device)
         if output["detection"] is not None:
             if output["detection"].ndim == 2:
                 output["detection"] = output["detection"
].squeeze(0)

             detection_gt = (
                 torch.tensor(int(torch.any(mask)))
                 .unsqueeze(0)
                 .to(self.device, dtype=torch.int32)
             )

             detection_metrics.update(output["detection"],
 detection_gt)
             self._detection = True

         if output["mask"] is not None:
             if output["mask"].ndim == 3:
                 output["mask"] = output["mask"].squeeze
(0)

             mask_metrics.update(output["mask"], mask)
             self._mask = True

         if output["heatmap"] is not None:
             if output["heatmap"].ndim == 3:
                 output["heatmap"] = output["heatmap"].
squeeze(0)

             heatmap_metrics.update(output["heatmap"],
mask)
             self._heatmap = True

         image_count += 1

     log.info("-" * 80)
     log.info("-" * 80)

     if self.save_metrics_flag:
            tampered = (
                "tampered_only" if dataset.tampered_only else
    "tampered_and_pristine"
            )
            timestamp = time.strftime("%Y%m%d_%H:%M")

            report_id = f"{timestamp}_{tampered}"
            if self._heatmap:
                log.info("     - Saving heatmap metrics")
                self._save_metrics(
                    output_path=output_path,
                    metrics=heatmap_metrics,
                    report_id=report_id,
                    total_images=image_count,
                )
            else:
                log.info("     - No heatmap metrics to save")

            if self._mask:
                log.info("     - Saving mask metrics")
                self._save_metrics(
                    output_path=output_path,
                    metrics=mask_metrics,
                    report_id=report_id,
                    total_images=image_count,
                )
            else:
                log.info("     - No mask metrics to save")

            if self._detection:
                log.info("     - Saving detection metrics")
                self._save_metrics(
                    output_path=output_path,
                    metrics=detection_metrics,
                    report_id=report_id,
                    total_images=image_count,
                )
            else:
                log.info("     - No detection metrics to save
   ")
        else:
            log.info("     - Not saving metrics")

        log.info("-" * 80)
        log.info("-" * 80)
        log.info("Benchmark finished")
        log.info("-" * 80)
        log.info("-" * 80)

        metrics_return = {}
        if self._heatmap:
         metrics_return["heatmap"] = heatmap_metrics.
compute()
     if self._mask:
         metrics_return["mask"] = mask_metrics.compute()
     if self._detection:
         metrics_return["detection"] = detection_metrics.
compute()

     return metrics_return

 def _print_setup_message(
     self,
     method: BaseMethod,
     dataset: BaseDataset,
     metrics: MetricCollection,
     output_path: Path,
 ):
     """
     Print the benchmark setup message.

     Args:
         method (BaseMethod): The method to evaluate.
         dataset (BaseDataset): Dataset to run the
evaluation on.
         metrics (MetricCollection): Collection of metrics
 to compute.
         output_path (Path): Path to the output folder.
     """
     log.info("-" * 80)
     log.info("-" * 80)
     log.info("Running the benchmark")
     log.info("-" * 80)
     log.info("-" * 80)
     log.info("Benchmark configuration:")
     log.info(f"    Method: {method.__class__.__name__}")
     log.info(f"    Dataset: {dataset.__class__.__name__}"
)
     log.info("    Metrics:")
     for metric in metrics:
         log.info(f"       - {metric}")
     log.info(f"    Output path: {output_path}")
     log.info(f"    Save method outputs: {self.
save_method_outputs}")
     log.info(f"    Save metrics: {self.save_metrics_flag}
")
     log.info(f"    Device: {self.device}")
     log.info(f"    Load existing outputs: {self.
use_existing_output}")
     log.info(f"    Verbosity: {logging._levelToName[
verbose_dict[self.verbose]]}")
     log.info("-" * 80)
```

```
        log.info("-" * 80)

    def _dict_to_device(self, data: Dict[str, Any]) -> Dict[
str, Any]:
        """
        Move dict items to the benchmark's device.

        Args:
            data (Dict[str, Any]): Data to move to the
benchmark's device.

        Returns:
            Dict[str, Any]: Data moved to the benchmark's
device.
        """
        return {
            key: (
                value.to(self.device, dtype=torch.float32)
                if isinstance(value, torch.Tensor)
                else value
            )
            for key, value in data.items()
        }

    def _save_metrics(
        self,
        output_path: Path,
        metrics: MetricCollection,
        report_id: str,
        total_images: int,
    ):
        """
        Save predicted outputs for an image.

        Args:
            output_path (Path): Path to the output folder.
            metrics (MetricCollection): Collection of metrics
 to compute.
            report_id (str): ID for the report.
            total_images (int): Total number of images
processed.

        """
        metrics_path = output_path / "metrics" / report_id
        os.makedirs(metrics_path, exist_ok=True)

        metric_compute = metrics.compute()
        torch.save(metrics.state_dict(), metrics_path / f"{
metrics.prefix}_state.pt")

        metric_report: Dict[str, Any] = {}
        for key, value in metric_compute.items():
            if isinstance(value, torch.Tensor) and value.dim
() == 0:
                metric_report[key] = float(value)
            elif isinstance(value, tuple) and all(
                isinstance(v, torch.Tensor) for v in value
            ):
                metric_report[key] = [v.tolist() for v in
value]
            elif (
                isinstance(value, int)
                or isinstance(value, float)
                or isinstance(value, str)
            ):
                metric_report[key] = value
            else:
                log.warning(f"Skipping metric '{key}' of type
 '{type(value)}'")

        report = {
            "metrics": metric_report,
            "total_images": total_images,
            "type": metrics.prefix,
        }

        with open(metrics_path / f"{metrics.prefix}_report.
json", "w") as f:
            json.dump(report, f)

 def _save_predicted_output(
        self, output_path: Path, image_name: str, output:
BenchmarkOutput
 ):
        """
        Save predicted outputs for an image.

        Args:
            output_path (Path): Path to the output folder.
            image_name (str): Name of the processed image.
            output (BenchmarkOutput): Output to save.
        """
        image_save_path = output_path / "outputs" /
image_name
        os.makedirs(image_save_path, exist_ok=True)

        output_dict = {}
        if output["heatmap"] is not None:
            output_dict["heatmap"] = output["heatmap"].cpu().
numpy()
        if output["mask"] is not None:
            output_dict["mask"] = output["mask"].cpu().numpy
```

```
()
        if output["detection"] is not None:
            output_dict["detection"] = output["detection"].
cpu().numpy()
        np.savez_compressed(image_save_path / "output", **
output_dict)

        if "extra_outputs" in output:
            extra_outputs_arrays = {}
            extra_outputs_other = {}

            for key, value in output["extra_outputs"].items()
:
                if isinstance(value, (torch.Tensor)):
                    extra_outputs_arrays[key] = value.cpu()
                elif isinstance(value, (list, np.ndarray)):
                    extra_outputs_arrays[key] = value
                else:
                    extra_outputs_other[key] = value

            if self.save_extra_outputs:
                np.savez_compressed(
                    image_save_path / "extra_outputs_arrays",
 **extra_outputs_arrays
                )
                with open(image_save_path / "
extra_outputs_other.json", "w") as f:
                    json.dump(extra_outputs_other, f)

        log.io_message(f"Output for image '{image_name}'
saved.")

    def _load_existing_output(
        self, output_path: Path, image_name: str
    ) -> Optional[BenchmarkOutput]:
        """
        Load existing output for a given image.

        Args:
            output_path (Path): Path to the output folder.
            image_name (str): Name of the processed image.

        Returns:
            Optional[BenchmarkOutput]: Loaded output if
available, else None.
        """
        output_path = output_path / "outputs"
        if not os.path.exists(output_path):
            return None

        if os.path.exists(output_path / image_name / "output.
npz"):
         log.io_message(f"Loading existing output for
image '{image_name}'")

         prior_output = np.load(
             output_path / image_name / "output.npz",
             allow_pickle=True,
         )
         output: BenchmarkOutput = {
             "heatmap": (
                 torch.tensor(prior_output["heatmap"],
device=self.device)
                 if "heatmap" in prior_output
                 else None
             ),
             "mask": (
                 torch.tensor(prior_output["mask"], device
=self.device)
                 if "mask" in prior_output
                 else None
             ),
             "detection": (
                 torch.tensor(prior_output["detection"],
device=self.device)
                 if "detection" in prior_output
                 else None
             ),
         }
         return output

     log.io_message(f"No existing output found for image
'{image_name}'.")
     return None
```

This page intentionally left blank.

Bibliography

[1] Adobe Photoshop. https://www.adobe.com/la/products/photoshop.html.

[2] Adobe Photoshop Generative Fill. https://www.adobe.com/products/photoshop/generative-fill.html.

[3] ClipDrop Generative Fill. https://clipdrop.co/generative-fill.

[4] How Stalin's propaganda machine erased people from photographs, 1922-1953 — rarehistoricalphotos.com. https://rarehistoricalphotos.com/stalin-photo-manipulation-1922-1953/. [Accessed 29-07-2023].

[5] Sad Taylor Swift—Photo Of The Pop Star Sat On A Bench Gets A Very Funny Photoshop Battle — voomed.com. https://www.voomed.com/sad-taylor-swift-pop-star-sat-bench-gets-funny-photoshop-battle/. [Accessed 29-07-2023].

[6] The Man Who Wasn't There;: Story of the Tourist Guy Photo — septterror.tripod.com. https://septterror.tripod.com/touristguy.html. [Accessed 01-08-2023].

[7] Paul in scotland, 1970 #throwbackthursday #tbt. https://twitter.com/PaulMcCartney/status/824659293022679042, Jan 2017. [Accessed 04-09-2023].

[8] Hesham A. Alberry, Abdelfatah A. Hegazy, and Gouda I. Salama. A fast sift based method for copy move forgery detection. *Future Computing and Informatics Journal*, 3(2):159–165, 2018.

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

[10] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H. Lin. *Learning From Data, a short course.* AMLbook, 2012.

[11] Adobe. What are tiff files and how do you open them? `https://www.adobe.com/creativecloud/file-types/image/raster/tiff-file.html`. [Accessed 9th April, 2024].

[12] Cecilia Aguerrebere, Julie Delon, Yann Gousseau, and Pablo Musé. Study of the digital camera acquisition process and statistical modeling of the sensor raw data. *hal-00733538v3*, 2013.

[13] Samet Akcay, Dick Ameln, Ashwin Vaidya, Barath Lakshmanan, Nilesh Ahuja, and Utku Genc. Anomalib: A deep learning library for anomaly detection. In *2022 IEEE International Conference on Image Processing (ICIP)*, pages 1706–1710. IEEE, 2022.

[14] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

[15] Quentin Bammey, Tina Nikoukhah, Marina Gardella, Rafael Grompone, Miguel Colom, and Jean-Michel Morel. Non-semantic evaluation of image forensics tools: Methodology and database, 2021.

[16] Quentin Bammey, Rafael Grompone von Gioi, and Jean-Michel Morel. An adaptive neural network for unsupervised mosaic consistency analysis in image forensics. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14182–14192, 2020.

[17] Bryce Edward Bayer. Color imaging array, Jul 1976.

[18] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, Wesam Manassra, Prafulla Dhariwal, Casey Chu, Yunxin Jiao, and Aditya Ramesh. Improving image generation with better captions. *Openai*, 2023.

[19] Mehdi Boroumand, Mo Chen, and Jessica Fridrich. Deep residual network for steganalysis of digital images. *IEEE Transactions on Information Forensics and Security*, 14(5):1181–1193, 2019.

[20] Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, Wansen Feng, Ziwei Liu, Jiarui Xu, Zheng Zhang, Dazhi Cheng, Chenchen Zhu, Tianheng Cheng, Qijie Zhao, Buyu Li, Xin Lu, Rui Zhu, Yue Wu, Jifeng Dai, Jingdong Wang, Jianping Shi, Wanli Ouyang, Chen Change Loy, and Dahua Lin. MMDetection: Open mmlab detection toolbox and benchmark. *arXiv preprint arXiv:1906.07155*, 2019.

[21] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21(1):6, Jan 2020.

[22] D. R. Cok. Signal processing method and apparatus for producing interpolated chrominance values in a sampled color image signal. US Patent, 4642678, 1987.

[23] MMPose Contributors. Openmmlab pose estimation toolbox and benchmark. `https://github.com/open-mmlab/mmpose`, 2020.

[24] MMSegmentation Contributors. MMSegmentation: Openmmlab semantic segmentation toolbox and benchmark. `https://github.com/open-mmlab/mmsegmentation`, 2020.

[25] Aldus Corporation. Tiff revision 6.0, June 3, 1992.

[26] D. Cozzolino and L. Verdoliva. Noiseprint: A cnn-based camera model fingerprint. *IEEE Transactions on Information Forensics and Security*, 15:144–159, 2020.

[27] Davide Cozzolino, Giovanni Poggi, and Luisa Verdoliva. Splicebuster: A new blind image splicing detector. *Conference: IEEE Workshop on Information Forensics and Security*, 2015.

[28] D.-T. Dang-Nguyen, C. Pasquini, V. Conotter, and G. Boato. RAISE – A Raw Images Dataset for Digital Image Forensics. In *ACM Multimedia Systems*, Portland, Oregon, March 18-20 2015.

[29] T.J. de Carvalho, C. Riess, E. Angelopoulou, H. Pedrini, and A. de Rezende Rocha. Exposing digital image forgeries by illumination color classification. *Information Forensics and Security, IEEE Transactions on*, 8(7):1182–1194, 2013.

[30] Agnès Desolneux, Lionel Moisan, and Jean-Michel Morel. *From gestalt theory to Image Analysis: A probabilistic approach.* Springer, 2011.

[31] Nicki Skafte Detlefsen, Jiri Borovec, Justus Schock, Ananya Harsh, Teddy Koker, Luca Di Liello, Daniel Stancl, Changsheng Quan, Maxim Grechkin, and William Falcon. Torchmetrics - measuring reproducibility in pytorch, 2 2022.

[32] Jing Dong, Wei Wang, and Tieniu Tan. CASIA image tampering detection evaluation database. In *2013 IEEE China Summit and International Conference on Signal and Information Processing*. IEEE, July 2013.

[33] Marc Ebner. Color constancy. *Color Constancy*, pages 104–106, 05 2007.

[34] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Knowledge Discovery and Data Mining*, 1996.

[35] Canon Europe. White balance: definition and settings. `https://www.canon-europe.com/pro/infobank/white-balance/`.

[36] Python Software Foundation. Python programming language, 1991. Accessed: April 17, 2024.

[37] W. T. Freeman. Median filter for reconstructing missing color samples. US Patent, 4724395, 1988.

[38] Marina Gardella, Pablo Musé, Miguel Colom, and Jean-Michel Morel. Image Forgery Detection Based on Noise Inspection: Analysis and Refinement of the Noisesniffer Method. *Image Processing On Line*, 14:86–115, 2024. `https://doi.org/10.5201/ipol.2024.462`.

[39] GfG. Artificial neural networks and its applications, Jun 2023.

[40] Google. An image format for the web. `https://developers.google.com/speed/webp`. [Accessed 9th April, 2024].

[41] PNG Development Group. Png (portable network graphics) specification, version 1.2. `http://www.libpng.org/pub/png/spec/1.2/png-1.2.pdf`. [Accessed 9th April, 2024].

[42] Fabrizio Guillaro, Davide Cozzolino, Avneesh Sud, Nicholas Dufour, and Luisa Verdoliva. Trufor: Leveraging all-round clues for trustworthy image forgery detection and localization, 2023.

[43] Aurélien Géron. *Hands-on Machine Learning with Scikit-learn, Keras and TensorFlow*. O'Reilly Media, Inc, 2019.

[44] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[45] Y.-F. Hsu and S.-F. Chang. Detecting image splicing using geometry invariants and camera characteristics consistency. In *International Conference on Multimedia and Expo*, 2006.

[46] Minyoung Huh, Andrew Liu, Andrew Owens, and Alexei A. Efros. Fighting fake news: Image splice detection via learned self-consistency, 2018.

[47] Minyoung Huh, Andrew Liu, Andrew Owens, and Alexei A. Efros. Fighting fake news: Image splice detection via learned self-consistency, 2018.

[48] M. Jansen and N. Petrov. *Stalin's Loyal Executioner: People's Commissar Nikolai Ezhov, 1895-1940*. None. Hoover Institution Press, 2013.

[49] Shan Jia, Mingzhen Huang, Zhou Zhou, Yan Ju, Jialing Cai, and Siwei Lyu. Autosplice: A text-prompt manipulated image dataset for media forensics. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 893–903, 2023.

[50] P. Korus and J. Huang. Evaluation of random field models in multi-modal unsupervised tampering localization. In *Proc. of IEEE Int. Workshop on Inf. Forensics and Security*, 2016.

[51] P. Korus and J. Huang. Multi-scale analysis strategies in prnu-based tampering localization. *IEEE Trans. on Information Forensics and Security*, 2017.

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[53] Zhanghui Kuang, Hongbin Sun, Zhizhong Li, Xiaoyu Yue, Tsui Hin Lin, Jianyong Chen, Huaqiang Wei, Yiqin Zhu, Tong Gao, Wenwei Zhang, Kai Chen, Wayne Zhang, and Dahua Lin. Mmocr: A comprehensive toolbox for text detection, recognition and understanding. *arXiv preprint arXiv:2108.06543*, 2021.

[54] Myung-Joon Kwon, In-Jae Yu, Seung-Hun Nam, and Heung-Kyu Lee. Catnet: Compression artifact tracing network for detection and localization of image splicing. In *2021 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 375–384, 2021.

[55] C. A. Laroche and M. A. Prescott. Apparatus and method for adaptively interpolating a full color image utilizing chrominance gradients. US Patent, 5373322, 1994.

[56] Zhouchen Lin, Junfeng He, Xiaoou Tang, and Chi-Keung Tang. Fast, automatic and fine-grained tampered jpeg image detection via dct coefficient analysis. *Pattern Recognition*, 2009.

[57] Xiaohong Liu, Yaojie Liu, Jun Chen, and Xiaoming Liu. Pscc-net: Progressive spatio-channel correlation network for image manipulation detection and localization, 2022.

[58] H. Maître. *From Photon to Pixel: The Digital Camera Handbook.* ISTE Limited, 2017.

[59] Hannes Mareen, Dante Vanden Bussche, Fabrizio Guillaro, Davide Cozzolino2, Glenn Van Wallendael, Peter Lambert, and Luisa Verdoliva. Comprint: Image forgery detection and localization using compression fingerprints. *Proc. Int. Conf. on Pattern Recognition (ICPR)*, 2022.

[60] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442–451, 1975.

[61] Inc Midjourney. Midjourney. https://www.midjourney.com/home, 2020.

[62] Tina Nikoukhah, J Anger, T Ehret, Miguel Colom, J M Morel, and R Grompone von Gioi. JPEG Grid Detection based on the Number of DCT Zeros and its Application to Automatic and Localized Forgery Detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, Long Beach, United States, June 2019.

[63] Tina Nikoukhah, Jérémy Anger, Miguel Colom, Jean-Michel Morel, and Rafael Grompone von Gioi. ZERO: a Local JPEG Grid Origin Detector Based on the Number of DCT Zeros and its Applications in Image Forensics. *Image Processing On Line*, 11:396–433, 2021. `https://doi.org/10.5201/ipol.2021.390`.

[64] Nam Thanh Pham, Jong-Weon Lee, Goo-Rak Kwon, and Chun-Su Park. Hybrid image-retrieval method for image-splicing validation. *Symmetry*, 11(1):83, 2019.

[65] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. Sdxl: Improving latent diffusion models for high-resolution image synthesis. (arXiv:2307.01952), July 2023. arXiv:2307.01952 [cs].

[66] Alin Popescu and Hany Farid. Statistical tools for digital forensics. volume 3200, 05 2004.

[67] Alin Popescu and Hany Farid. Exposing digital forgeries in color filter array interpolated images. *Signal Processing, IEEE Transactions on*, 53:3948 – 3959, 11 2005.

[68] David M W Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *Journal of Machine Learning Technologies.*, 2011.

[69] Charles A. Poynton. Digital video and hdtv algorithms and interfaces. 2012.

[70] William Puech. *Multimedia Security 1: Authentication and Data Hiding.* Wiley-ISTE, 2022.

[71] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[72] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[73] Average Rob. Instagram instagram.com. `https://www.instagram.com/averagerob/`. [Accessed 29-07-2023].

[74] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[75] Konstantinos Sechidis, Grigorios Tsoumakas, and Ioannis Vlahavas. On the stratification of multi-label data. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part III 22*, pages 145–158. Springer, 2011.

[76] Jianbo Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[77] So S Stevens. Neural events and the psychophysical law: Power functions like those that govern subjective magnitude show themselves in neurelectric effects. *Science*, 170(3962):1043–1050, 1970.

[78] Denis Teyssou, Jean-Michel Leung, Evlampios Apostolidis, Konstantinos Apostolidis, Symeon Papadopoulos, Markos Zampoglou, Olga Papadopoulou, and Vasileios Mezaris. The invid plug-in: web video verification on the browser. In *Proceedings of the first international workshop on multimedia verification*, pages 23–30, 2017.

[79] trent b. Iterative stratification. `https://github.com/trent-b/iterative-stratification`, 2023.

[80] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding, 2019.

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[82] Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao. Deep high-resolution representation learning for visual recognition, 2020.

[83] Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao. Deep high-resolution representation learning for visual recognition, 2020.

[84] Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao. Deep high-resolution representation learning for visual recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(10):3349–3364, 2021.

[85] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[86] Bihan Wen, Ye Zhu, Ramanathan Subramanian, Tian-Tsong Ng, Xuanjing Shen, and Stefan Winkler. Coverage - a novel database for copy-move forgery detection. In *IEEE International Conference on Image processing (ICIP)*, 2016.

[87] Papers with Code. Papers with code : Trends.

[88] Haiwei Wu, Yiming Chen, and Jiantao Zhou. Rethinking image forgery detection via contrastive learning and unsupervised clustering, 2023.

[89] Haiwei Wu, Jiantao Zhou, Jinyu Tian, and Jun Liu. Robust image forgery detection over online social network shared images. pages 13430–13439. IEEE, 6 2022.

[90] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M. Alvarez, and Ping Luo. Segformer: Simple and efficient design for semantic segmentation with transformers, 2021.

[91] yizhe ang. Media forensics / fake detection experiments in pytorch. `https://github.com/yizhe-ang/fake-detection-lab`, 2021.

[92] Markos Zampoglou. Matlab forensics. `https://github.com/caomw/matlab-forensics`, 12 2016.

[93] Markos Zampoglou, Symeon Papadopoulos, and Yiannis Kompatsiaris. Large-scale evaluation of splicing localization algorithms for web images. *Multimedia Tools and Applications*, 76(4):4801–4834, 2017.

[94] Markos Zampoglou, Symeon Papadopoulos, Yiannis Kompatsiaris, Ruben Bouwmeester, and Jochen Spangenberg. Web and social media image forensics for news professionals. In *Social Media In the NewsRoom, SMNews16@CWSM, Tenth International AAAI Conference on Web and Social Media workshops*, 2016.

[95] Jiaming Zhang, Huayao Liu, Kailun Yang, Xinxin Hu, Ruiping Liu, and Rainer Stiefelhagen. Cmx: Cross-modal fusion for rgb-x semantic segmentation with transformers, 2023.

[96] Chenhao Zheng, Ayush Shrivastava, and Andrew Owens. Exif as language: Learning cross-modal associations between images and camera metadata, 2023.

This page intentionally left blank.

List of Tables

List of Figures