FORENSIC SCANNER IDENTIFICATION USING DEEP LEARNING

Aakash Kumar, A.Nikhil, K.Pawan Sai Krishna Reddy, V. Vishal

Mrs. G Kalpana (Guide)

Department of Computer Science & Engineering

BACHELOR OF TECHNOLOGY

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(An Autonomous Institution)

(Approved by AICTE , Accredited by NAAC, NBA & permanently Affiliated to JNTUH,)

Aziz Nagar Gate, C.B. Post, Hy derabad-500075

ABSTRACT

Due to the increasing availability and functionality of image editing tools, many forensic techniques such as digital image authentication, source identification and tamper detection are important for forensic image analysis. In this project, we describe a machine learning based system to address the forensic analysis of scanner devices. The proposed system uses machine-learning to automatically learn the intrinsic features from various scanned images.

Our experimental results show that high accuracy can be achieved for source scanner identification. The proposed system can also generate a reliability map that indicates the manipulated regions in an scanned image.

Keywords:

Reliability map, digital image authentication, deep learning, convolutional neural network.

CHAPTER-1

INTRODUCTION

With powerful image editing tools such as Photoshop and GIMP being easily accessible, image manipulation has become very easy. Hence, developing forensic tools to determine the origin or verify the authenticity of a digital image is important. These tools provide an indication as to whether an image is modified and the region where the modification has occurred. A number of methods have been developed for digital image forensics.

For example, forensic tools have been developed to detect copy-move attacks and splicing attacks. Methods are also able to identify the manipulated region regardless of the manipulation types. Other tools are able to identify the digital image capture device used to acquire the image which can be a first step in many types of image forensics analysis.

The capture of "real" digital images (not computer-generated images) can be roughly divided into two categories: digital cameras and scanners. In this paper, we are interested in forensics analysis of images captured by scanners. Unlike camera images, scanned images usually contain additional features produced in the pre-scanning stage, such as noise patterns or artifacts generated by the devices producing the "hard-copy" image or document.

These scanner independent features increase the difficulty in scanner model identification. Many scanners also use 1D "line" sensors, which are different than the 2D "area" sensors used in cameras. Previous work in scanner classification and scanned image forensics mainly focus on handcrafted feature extraction. They extract features unrelated to image content, such as sensor pattern noise, dust and scratches.

Extract statistical features from images and use principle component analysis (PCA) and support vector machine (SVM) to do scanner model identification. The goal is to classify an image based on scanner model rather than the exact instance of the image. In linear discriminant analysis (LDA) and SVM are used with the features which describe the noise pattern of a scanned image to identify the scanner model.

The increasing availability and functionality of image editing tools, many forensic techniques such as digital image authentication, source identification and tamper detection are important for forensic image analysis. In this project, we describe a machine learning based system to address the forensic analysis of scanner devices. The proposed system uses machine-learning to automatically learn the intrinsic features from various scanned images.

Unlike camera images, scanned images usually contain additional features produced in the pre-scanning stage, such as noise patterns or artifacts generated by the devices producing the “hard-copy” image or document. These scanner independent features increase the difficulty in scanner model identification. Many scanners also use 1D “line” sensors, which are different than the 2D “area” sensors used in cameras. Previous work in scanner classification and scanned image forensics mainly focus on handcrafted feature extraction. They extract features unrelated to image content, such as sensor pattern noise ,dust and scratches.

This method achieves high classification accuracy and is robust under various post-processing (e.g. , contrast stretching and sharpening). In Dirik et al. propose to use the impurities (i.e. , dirt) on the scanner pane to identify the scanning device. Convolutional neural networks (CNNs) such as VGG , ResNet , GoogleNet , and Xception have produced state-of-art results in object classification on ImageNet .

CNNs have large learning capacities to “describe” imaging sensor characterstics by capturing low/median/high-level features of images [8]. For this reason, they have been used for camera model identification , and have achieved state-of-art results. In this paper, we propose a CNN-based system for scanner model identification.

We will investigate the reduction of the network depth and number of parameters to account for small image patches (i.e. , 64 × 64 pixels) while keeping the time for training in a reasonable range. Inspired by we propose a network that is light-weight and also combines the advantages of ResNet and GoogleNet .The proposed system can achieve a good classification accuracy and generate a reliability map (i.e. , a heat map, to indicate the suspected manipulated region)

CHAPTER-2
LITERATURE SURVEY

Digital camera identification from sensor pattern noise

In this project, we propose a new method for the problem of digital camera identification from its images based on the sensor's pattern noise. For each camera under investigation, we first determine its reference pattern noise, which serves as a unique identification fingerprint. This is achieved by averaging the noise obtained from multiple images using a de-noising filter.

To identify the camera from a given image, we consider the reference pattern noise as a spreadspectrum watermark, whose presence in the image is established by using a correlation detector. Experiments on approximately 320 images taken with nine consumer digital cameras are used to estimate false alarm rates and false rejection rates. Additionally, we study how the error rates change with common image processing, such as JPEG compression or gamma correction. As digital images and video continue to replace their analog counterparts, the importance of reliable, inexpensive, and fast identification of digital image origin will only increase.

Reliable identification of the device used to acquire a particular digital image would especially prove useful in the court for establishing the origin of images presented as evidence. In the same manner as bullet scratches allow forensic examiners to match a bullet to a particular barrel with reliability high enough to be accepted in courts, a digital equivalent of bullet scratches should allow reliable matching of a digital image to a sensor.

Source camera identification based on CFA interpolation

In this work, we focus our interest on blind source camera identification problem by extending our results in the direction of M. Kharrazi et al. (2004). The interpolation in the color surface of an image due to the use of a color filter array (CFA) forms the basis of the paper. We propose to identify the source camera of an image based on traces of the proprietary interpolation algorithm deployed by a digital camera. For this purpose, a set of image characteristics are defined and then used in conjunction with a support vector machine based multi-class classifier to determine the originating digital camera. We also provide initial results on identifying source among two and three digital cameras. In contrast to the other two papers, Kirchner and B¨ohme describe a method to circumvent the forgery detection described above by restoring CFA-like correlations. The naıve method would be to simply sample the forged image with a CFA filter, and reinterpolate it. However, this form of sub-sampling the image could introduce aliasing if the forged area's spectrum was too wide. A much better method would be to find the additive tamper error (the difference between tampered and original), low-pass filter it (e.g. Gaussian blur), and add it to the image before sampling the result with a CFA filter. This would work, however it is not guaranteed to give the minimal error from the original.

Camera model identification with the use of deep convolutional neural networks

In this project, we propose a camera model identification method based on deep convolutional neural networks (CNNs). Unlike traditional methods, CNNs can automatically and simultaneously extract features and learn to classify during the learning process. A layer of preprocessing is added to the CNN model, and consists of a high pass filter which is applied to the input image. Before feeding the CNN, we examined the CNN model with two types of residuals.

The convolution and classification are then processed inside the network. The CNN outputs an identification score for each camera model. Experimental comparison with a classical two steps machine learning approach shows that the proposed method can achieve significant detection performance. The well known object recognition CNN models, AlexNet and GoogleNet, are also examined.

CHAPTER-3
FEASIBILITY STUDY

In order to evaluate if the project can be done in the given time frame, we are using the TEL-evaluation methods, where we cover the feasibility of the project from a technological, economical and legal perspective. Those perspectives would help us have a broad vision on the requirements and implications related to the project. We also discuss in this section the methodology used in conducting the project.

3.1 Technological Side

This project would be developed using technologies and libraries pertinent to object detection and tracking.

3.2 Economical Side

This project will be based on Free and Open Source Technologies and Libraries that are readily available to developers and scientists, free of cost. This means that we don’t have to worry about costs related to licensing or reusing source code and that the only costs related to the project are related to the time and the effort spent into developing it.

CHAPTER-4
SYSTEM REQUIREMENTS

4.1 EXISTING SYSTEM:

With powerful image editing tools such as Photoshop and GIMP being easily accessible, image manipulation has become very easy. Hence, developing forensic tools to determine the origin or verify the authenticity of a digital image is difficult. These tools doesn't provide an indication as to whether an image is modified and the region where the modification has occurred. A number of methods have been developed for digital image forensics. For example, forensic tools have been developed to detect copy-move attacks and splicing attacks.

Limitations:

- Does not indicate the area where the tampering has occurred.
- Less Accuracy

4.2 PROPOSED SYSTEM:

The proposed system An input image is first split into smaller sub-images Is of size n ×m pixels. This is done for four reasons: a) to deal with large scanned images at native resolution, b) to take location independence into account, c) to enlarge the dataset, and d) to provide low pre-processing time.

Advantages:

- Indication of tampering is specified.
- More accurate.

4.3 SYSTEM REQUIREMENTS:

4.3.1 SOFTWARE REQUIREMENTS:

• OS	:	Windows 7 and above (with any web browser)
• Software	:	Jupiter
• Libraries	:	OpenCV, Numpy, Pandas, Pillow, Matplotlib, PyLab, sklearn, keras

OpenCV:

The OpenCV full form is Open Source Computer Vision Library .OpenCV is a Python library that allows you to perform image processing and computer vision tasks. It provides a wide range of features, including object detection, face recognition, and tracking.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, track camera movements, track moving objects etc.

From this opencv module we used background subtraction algorithm for detection of vehicles. This algorithm extracts the foreground objects from dynamic videos sequence i.e moving objects. We also used morphology operations which are present in this module used for hole filling and noise removal.

Numpy:

NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

In particular, NumPy arrays provide an efficient way of storing and manipulating data. NumPy also includes a number of functions that make it easy to perform mathematical operations on arrays. This can be really useful for scientific or engineering applications.

In our project we used this NumPy array to detect the vehicle and also to store the information about the vehicle that is being detected for further tracking. Based on this information the vehicle counter is increased efficiently.

Pandas:

Pandas is an open-source library that is made mainly for working with relational or labeled data both easily and intuitively. It provides various data structures and operations for manipulating numerical data and time series.

This library is built on top of the NumPy library. Pandas is fast and it has high performance & productivity for users. It offers a variety of data structures and operations for working with time series and numerical data. This library is developed on top of the NumPy library, which supports multi-dimensional arrays.

Pandas are quick and offer users high performance and productivity. Being one of the most widely used data-wrangling tools, Pandas integrates well with a variety of different data science modules within the Python environment and is frequently available in all Python distributions, including those that come with your operating system and those sold by commercial vendors like ActiveState's ActivePython.

Matplotlib :

Matplotlib is a python library used to create 2D graphs and plots by using python scripts. It has a module named pyplot which makes things easy for plotting by providing feature to control line styles, font properties, formatting axes etc. It supports a very wide variety of graphs and plots namely - histogram, bar charts, power spectra, error charts etc. It is used along with NumPy to provide an environment that is an effective open source alternative for MatLab. It can also be used with graphics toolkits like PyQt and wxPython.

Matplotlib is a cross-platform, data visualization and graphical plotting library for Python and its numerical extension NumPy. As such, it offers a viable open source alternative to MATLAB. Developers can also use matplotlib's APIs (Application Programming Interfaces) to embed plots in GUI applications.

A Python matplotlib script is structured so that a few lines of code are all that is required in most instances to generate a visual data plot. The matplotlib scripting layer overlays two APIs:
The pyplot API is a hierarchy of Python code objects topped by matplotlib. pyplot. An OO (Object-Oriented) API collection of objects that can be assembled with greater flexibility than pyplot. This API provides direct access to Matplotlib's backend layers.

PyLab :

PyLab is a Python package that provides us a namespace in Python programming, which is very similar to MATLAB interface, by importing the functions from Python Numpy and Matplotlib Module. If we talk about these modules' role in the PyLab package, Matplotlib Module provides functions that help us to create visualizations of data, whereas the Numpy Module provides efficient numerical vector calculation functions that are based on underlying C and Fortran binary libraries. We will learn about PyLab Module in this section, and we will then plot some basic graphs and charts using the elements & functions which this module provides to us.

PyLab Module is an associated module with the Matplotlib Module of Python, and it gets installed alongside when we are installing Matplotlib Module in our system. We can also say that PyLab is a procedural interface of the Matplotlib Module, an object-oriented plotting library of Python. PyLab in itself is a convincing module for us because its bulky import the NumPy Module's functions and matplotlib.pyplot package in a single namespace to provide us a MATLAB-like namespace.

While performing some tasks, we have to use graphs like line charts, bar graphs, etc., for many reasons like to make the task more interactive, to pass the information in a very interesting way, graphs are easy and self-explanatory, etc. That's why plotting a graph or chart is a very important and integrated part of many functions. Graphs and charts play a very important role in the field of programming, and developers are always recommended for using graphs in their programs.

Therefore, it becomes very important that we should be aware of how we can plot graphs from a program. MATLAB is considered the best to plot graphs and charts, but it is not possible for everyone to use MATLAB for plotting graphs & charts. We have many interactive modules present in Python that allow us to plot graphs and charts in the output, but here we will talk about the module, which provides us a MATLAB-like namespace by importing functions.

Keras :

Keras is an open-source high-level Neural Network library, which is written in Python is capable enough to run on Theano, TensorFlow, or CNTK. It was developed by one of the Google engineers, Francois Chollet.

It is made user-friendly, extensible, and modular for facilitating faster experimentation with deep neural networks. It not only supports Convolutional Networks and Recurrent Networks individually but also their combination.
It cannot handle low-level computations, so it makes use of the Backend library to resolve it. The backend library act as a high-level API wrapper for the low-level API, which lets it run on TensorFlow, CNTK, or Theano.

Initially, it had over 4800 contributors during its launch, which now has gone up to 250,000 developers. It has a 2X growth ever since every year it has grown. Big companies like Microsoft, Google, NVIDIA, and Amazon have actively contributed to the development of Keras. It has an amazing industry interaction, and it is used in the development of popular firms likes Netflix, Uber, Google, Expedia, etc.

Keras being a model-level library helps in developing deep learning models by offering high-level building blocks. All the low-level computations such as products of Tensor, convolutions, etc. are not handled by Keras itself, rather they depend on a specialized tensor manipulation library that is well optimized to serve as a backend engine. Keras has managed it so perfectly that instead of incorporating one single library of tensor and performing operations related to that particular library, it offers plugging of different backend engines into Keras.

Keras consist of three backend engines, which are as follows:

TensorFlow

TensorFlow is a Google product, which is one of the most famous deep learning tools widely used in the research area of machine learning and deep neural network. It came into the market on 9th November 2015 under the Apache License 2.0. It is built in such a way that it can easily run on multiple CPUs and GPUs as well as on mobile operating systems. It consists of various wrappers in distinct languages such as Java, C++, or Python.

Theano

Theano was developed at the University of Montreal, Quebec, Canada, by the MILA group. It is an open-source python library that is widely used for performing mathematical operations on multi-dimensional arrays by incorporating scipy and numpy. It utilizes GPUs for faster computation and efficiently computes the gradients by building symbolic graphs automatically. It has come out to be very suitable for unstable expressions, as it first observes them numerically and then computes them with more stable algorithms.

CNTK

Microsoft Cognitive Toolkit is deep learning's open-source framework. It consists of all the basic building blocks, which are required to form a neural network. The models are trained using C++ or Python, but it incorporates C# or Java to load the model for making predictions**.**

SkLearn :

Scikit-learn is an open source data analysis library, and the gold standard for Machine Learning (ML) in the Python ecosystem. Key concepts and features include:

Algorithmic decision-making methods, including:

Classification: identifying and categorizing data based on patterns.

Regression: predicting or projecting data values based on the average mean of existing and planned data.

Clustering: automatic grouping of similar data into datasets.

Algorithms that support predictive analysis ranging from simple linear regression to neural network pattern recognition.

Interoperability with NumPy, pandas, and matplotlib libraries.
ML is a technology that enables computers to learn from input data and to build/train a predictive model without explicit programming. ML is a subset of Artificial Intelligence (AI).
Scikit-learn (Sklearn) is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python. This library, which is largely written in Python, is built upon NumPy, SciPy and Matplotlib.

Pillow:
The Python Pillow library is a fork of an older library called PIL. PIL stands for Python Imaging Library, and it's the original library that enabled Python to deal with images. PIL was discontinued in 2011 and only supports Python 2.

To use its developers' own description, Pillow is the friendly PIL fork that kept the library alive and includes support for Python 3
There's more than one module in Python to deal with images and perform image processing. If you want to deal with images directly by manipulating their pixels, then you can use NumPy and SciPy. Other popular libraries for image processing are OpenCV, scikit-image, and Mahotas. Some of these libraries are faster and more powerful than Pillow.

However, Pillow remains an important tool for dealing with images. It provides image processing features that are similar to ones found in image processing software such as Photoshop. Pillow is often the preferred option for high-level image processing tasks that don't require more advanced image processing expertise. It's also often used for exploratory work when dealing with images. Pillow also has the advantage of being widely used by the Python community, and it doesn't have the same steep learning curve as some of the other image processing libraries.

4.3.2 HARDWARE REQUIREMENTS:

- Processor : Pentium IV onwards or Intel I5+
- RAM : 2 GB or higher
- Hard Disk Space : 20 GB or higher
- Input : Image

4.4 REQUIREMENTS :

After the severe continuous analysis of the problems that rose in the existing system, we are now familiar with the requirement that is required by the current system. The requirements that the system needs is categorized into the functional and non-functional requirements. These requirements are listed below:

4.4.1 FUNCTIONAL REQUIREMENTS

Functional requirement define which functions or features that are to be incorporated in any system to full fill the business requirements and to be acknowledged by the clients. On the premise, the functional requirements specify relationship between the inputs and outputs. All the operations to be performed on the input data to obtain output are to be specified. This includes specifying the validity checks on the input and output data, parameters affected by the operations and the other operations, which must be used to transform the inputs into outputs. Functional requirements specify the behavior of the system for valid input and outputs.

4.4.2 NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements provide a description of features, characteristics and capacity of the system and furthermore it may constraints the boundaries of the proposed system.
The following are the non-functional requirements that are essential depending on the performance, cost, control and gives security efficiency and services.
Based on the above explained non-functional pre-requisites are as follows:

- User friendly
- System should provide better accuracy
- To perform efficiently with better throughput and response time

CHAPTER-5

HIGH LEVEL DESIGN

5.1 Design consideration

In the design consideration the high level design of our project it includes four modules, for this architecture pattern is as follows. Software design is a process of envisioning and defining software solutions to one or more set of solutions one of the main component software design is the software requirement analysis.

5.2 Architecture Design

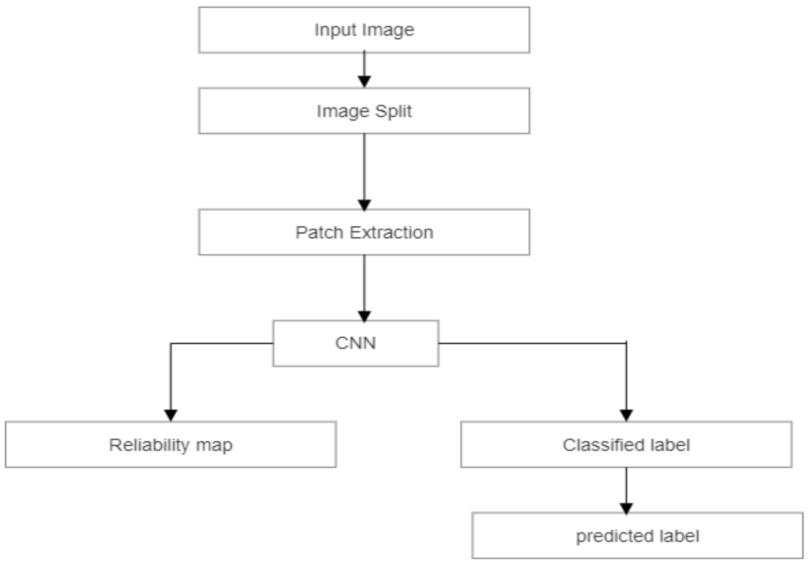


Figure: 5.1 Architecture

5.2.1 Image Split

The input image I is split into sub-images Is (n × m pixels) in zig-zag form. The values of n and m should be no smaller than 64.The basic idea of region splitting is to break the image into a set of disjoint regions which are coherent within themselves:

- Initially take the image as a whole to be the area of interest.
- Look at the area of interest and decide if all pixels contained in the region satisfy some similarity constraint.
- If **TRUE** then the area of interest corresponds to a region in the image.
- If **FALSE** split the area of interest (usually into four equal sub-areas) and consider each of the sub-areas as the area of interest in turn.

- This process continues until no further splitting occurs. In the worst case this happens when the areas are just one pixel in size.

If only a splitting schedule is used then the final segmentation would probably contain many neighbouring regions that have identical or similar properties.

5.2.2 Patch Extraction

From each Is, a patch of size 64 × 64 is extracted from a random location. We denote this extracted patch as Ip. These extracted patches Ip along with their corresponding labels S are inputs into the network. This pre-processing enables the proposed system to work with small-size images and use a smaller network architecture to save training time and memory usage. Designing a suitable network architecture is an important part in the scanner model identification system. There are several factors that need to be considered to build the network: a) the kernel size, b) the utilization of pooling layers, c) the depth of the network, and d) the implementation of the network modules.

A patch is small (generally rectangular) piece of an image. For example, an 8x8 patch is a square patch containing 64 pixels of a larger image (of size say, 256x256 pixes). Due to the smaller size, some of the image processing algorithms such as denoising/super resolution etc. are easier to operate on patches rather than operating on the entire image itself. These algorithms split an image into several smaller sized patches (of size say, 8x8), operate individually on each of these patches, and finally tile all these patches at their respective locations.

Image patch is a container of pixels in larger form. For example, let’s say you have a image of 100px by 100px. If you divide this images into 10x10 patches then you will have an image with 100 patches (that is 100px in each patch). If you have developed an algorithm that will operate on 10px by 10px, that 10px by 10px is the patch size. For example, pooling layer of CNN takes larger patches and turns them into one pixel. You may think of it as window in signal processing.

In image processing patch and window is interchangeable most of the time, but patch is usually used in context when your algorithm mainly focused on the fact that bunch of pixels share similar property. For instance, patch is used in context of sparse representation or image compression, while window is used in edge detection or image enhancement.

5.2.3 CNN

CNN is a type of deep learning model for processing data that has a grid pattern, such as images, which is inspired by the organization of animal visual cortex and designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns. CNN is a mathematical construct that is typically composed of three types of layers (or building blocks): convolution, pooling, and fully connected layers. The first two, convolution and pooling layers, perform feature extraction, whereas the third, a fully connected layer, maps the extracted features into final output, such as classification.

A convolution layer plays a key role in CNN, which is composed of a stack of mathematical operations, such as convolution, a specialized type of linear operation. In digital images, pixel values are stored in a two-dimensional (2D) grid, i.e., an array of and a small grid of parameters called kernel, an optimizable feature extractor, is applied at each image position, which makes CNNs highly efficient for image processing,

since a feature may occur anywhere in the image. As one layer feeds its output into the next layer, extracted features can hierarchically and progressively become more complex. The process of optimizing parameters such as kernels is called training, which is performed so as to minimize the difference between outputs and ground truth labels through an optimization algorithm called backpropagation and gradient descent, among others.

In any Neural Network, first layer will be input layer and last will be the output layer. Input layer contains all the inputs, here images is inputs. These images are given as input to the first convolutional layer. The output of 1st layer will be given as input to the 2nd layer, so on & so forth. This process will continue till the last layer.

While defining Neural Network, first convolutional layer requires the shape of image that is passed to it as input. After passing the image, through all convolutional layers and pooling layers, output will be passed to dense layer.

We can not pass output of convolutional layer directly to the dense layer because output of convolutional layer is in multi-dimensional shape and dense layer requires input in single-dimensional shape i.e. 1-D array.

So we will use Flatten() method in between convolutional and dense layer. Flatten() method converts multi-dimensional matrix to single dimensional matrix. In Neural Network, non-linear function is used as activation function.

Pooling layer

A pooling layer provides a typical down sampling operation which reduces the in-plane dimensionality of the feature maps in order to introduce a translation invariance to small shifts and distortions, and decrease the number of subsequent learnable parameters. It is of note that there is no learnable parameter in any of the pooling layers, whereas filter size, stride, and padding are hyperparameters in pooling operations, similar to convolution operations.

Max pooling

The most popular form of pooling operation is max pooling, which extracts patches from the input feature maps, outputs the maximum value in each patch, and discards all the other values A max pooling with a filter of size 2 × 2 with a stride of 2 is commonly used in practice. This down samples the in-plane dimension of feature maps by a factor of 2. Unlike height and width, the depth dimension of feature maps remains unchanged.

Convolution layer

A convolution layer is a fundamental component of the CNN architecture that performs feature extraction, which typically consists of a combination of linear and nonlinear operations, i.e., convolution operation and activation function.

Convolution

Convolution is a specialized type of linear operation used for feature extraction, where a small array of numbers, called a kernel, is applied across the input, which is an array of numbers, called a tensor. An element-wise product between each element of the kernel and the input tensor is calculated at each location of the tensor and summed to obtain the output value in the corresponding position of the output tensor, called a feature map This procedure is repeated applying multiple kernels to form an arbitrary number of feature maps, which represent different characteristics of the input tensors; different kernels can, thus, be considered as different feature extractors Two key hyperparameters that define the convolution operation are size and number of kernels. The former is typically 3 × 3, but sometimes 5 × 5 or 7 × 7. The latter is arbitrary, and determines the depth of output feature maps.

The Dropout Layer

Another typical characteristic of CNNs is a Dropout layer. The Dropout layer is a mask that nullifies the contribution of some neurons towards the next layer and leaves unmodified all others. We can apply a Dropout layer to the input vector, in which case it nullifies some of its features; but we can also apply it to a hidden layer, in which case it nullifies some hidden neurons.

Dropout layers are important in training CNNs because they prevent overfitting on the training data. If they aren't present, the first batch of training samples influences the learning in a disproportionately high manner. This, in turn, would prevent the learning of features that appear only in later samples or batches.

Flatten operation

Intuition behind flattening layer is to converts data into 1-dimentional array for feeding next layer. we flatted output of convolutional layer into single long feature vector. which is connected to final classification model, called fully connected layer. let's suppose we've [5,5,5] pooled feature map are flattened into 1x125 single vector. So, flatten layers converts multidimensional array to single dimensional vector.

Dense Layer

it is simple layer of neurons in which each neuron receives input from all the neurons of previous layer, thus called as dense. Dense Layer is used to classify image based on output from convolutional layers. A dense layer is connected deeply with preceding layers in any neural network. Each neuron in the dense layer is connected to every neuron of its preceding layer. Dense layers are the most commonly used layers in Artificial Neural Networks models. The neurons in the dense layers in a model receive an outcome from every neuron of the preceding layer. That's where neurons of the dense layer perform matrix-vector multiplication. So in the background, the dense layer performs a matrix-vector multiplication. It is a procedure where the row vector of the outcome from its previous layers equals the column vector of the dense layer.

5.2.4 Reliability Map

Since our system is aimed at extracting intrinsic features of scanner models, it should also be able to identify manipulated region irrespective of image content. In this task, we investigate to generate a reliability map (i.e. a heat map) that can indicate suspicious forged areas in the images. The reliability map is generated based on the predicted label obtained by majority vote.

In the reliability map, the color of the pixel represents the probability that it is generated by the predicted scanner model. Color “dark red” indicates a probability value equal to 1.0, and color “dark blue” indicates a probability value equal to 0.0. Then we use the original image to generate manipulated images in Photoshop. The forged images are shown in the first column. The top one is generated by self-image copy-move with translation operations. The bottom one is generated by copy-pasting regions in an other image source from different scanner model. The reliability maps generated with different stride size for these two forged images. These results indicate the effectiveness of using our reliability map to indicate the suspicious forgery.

Reliability maps are a tool that many industries use to plot the level of reliability of a product, service, or system for a particular time frame. A reliability map can be used to make informed decisions on how to improve the reliability of a particular product, service, or system. Reliability maps can plot the likelihood of failure of a product or system, which can help managers determine how much time, effort, and resources they need to invest in improving the system. Furthermore, reliability maps can also plot the failure density on a particular system or product, allowing engineers to focus their efforts on specific areas that might require more maintenance or replacement. Reliability maps are a valuable tool for predictive maintenance, contributing to the optimization of the life cycle cost of equipment. With advanced IoT sensors and analytics, reliability maps provide real-time insights into product performance, indicating when a system or component is approaching its failure threshold.

5.3 Modules Specification

Vehicle detection and counting method mainly consist of 3 different modules.

5.3.1 PreProcessing

Pre-processing is a common name for operations with images at the lowest level of abstraction - both input and output are intensity images. These iconic images are of the same kind as the original data captured by the sensor, with an intensity image usually represented by a matrix of image function values (brightnesses).

The aim of pre-processing is an improvement of the image data that suppresses unwilling distortions or enhances some image features important for further processing, although geometric transformations of images (e.g. rotation, scaling, translation) are classified among pre-processing methods here since similar techniques are used. Image pre-processing methods are classified into four categories according to the size of the pixel neighborhood that is used for the calculation of a new pixel brightness. In this module we try to improve the performance of forensic scanner.

- It is also helpful in reducing the training period of the model.
- Here the given image as a input is splitted into number of sub-images randomly.
- Patches are extracted of size 64x64 from random location for each sub image.
- This Preprocessing enable the proposal system to work with small size images and use a small network architecture to save training and memory usage.

5.3.2 Forensic Scanner

The capture of "real" digital images (not computer-generated images) can be roughly divided into two categories: digital cameras and scanners. In this paper, we are interested in forensics analysis of images captured by scanners. Unlike camera images, scanned images usually contain additional features produced in the pre-scanning stage, such as noise patterns or artifacts generated by the devices producing the "hard-copy" image or document.

These scanner independent features increase the difficulty in scanner model identification. Many scanners also use 1D "line" sensors, which are different than the 2D "area" sensors used in cameras. Previous work in scanner classification and scanned image forensics mainly focus on handcrafted feature extraction. They extract features unrelated to image content, such as sensor pattern noise, dust and scratches.

- In This module we try to get the tampered image as an input from user.
- The execution of the project begins with this module.
- This module perform an important task of training the model for detecting tampered area in the image.
- It also classifies and sets the metrics for CNN algorithm.

5.3.3 Tampered Area Detection:

The tamper detection design can be implemented to sense different types, techniques, and sophistication of tampering, depending on the perceived threats and risks. The methods used for tamper detection are typically designed as a suite of sensors each specialized on a single threat type, some of which may be physical penetration, hot or cold temperature extremes, input voltage variations, input frequency variations, x-rays, and gamma rays. Examples of techniques used to detect tampering may include any or all of the following: switches to detect the opening of doors or access covers, sensors to detect changes in light or pressure.

- In this module we try to detect the area image which have been tampered.
- The image is passed to CNN algorithm which is trained in Forensic Scanner module.
- It processes the image and generate a reliability map as a result.
- Reliability map indicates which portion of the image contain reliable camera traces.

CHAPTER-6

UML Diagrams

UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools include. In its simplest form, a use case can be described as a specific way of using the system from a User's (actor's) perspective. A more detailed description might characterize a use case as:

- a pattern of behavior the system exhibits
- a sequence of related transactions performed by an actor and the system
- delivering something of value to the actor

Use cases provide a means to:

- capture system requirements
- communicate with the end users and domain experts
- Test the system

Use cases are best discovered by examining the actors and defining what the actor will be able to do with the system. Since all the needs of a system typically cannot be covered in one use case, it is usual to have a collection of use cases. Together this use case collection specifies all the ways of using the system.

A UML system is represented using five different views that describe the system from distinctly different perspective. Each view is defined by a set of diagrams, which is as follows.

User Model View

- This view represents the system from the user's perspective.
- The analysis representation describes a usage scenario from the end-user's perspective.

Structural model view

- In this model the data and functionality are arrived from inside the system.
- This model view models the static structures.

Behavioral Model View

- It represents the dynamic of behavioral as parts of the system, depicting the interactions of collection between various structural elements described in the user model and structural model view.

Implementation Model View

- In this the structural and behavioral as parts of the system are represented as they are to be built.

Environmental Model View

- In this the structural and behavioral aspect of the environment in which the system is to be implemented are represented.

UML is specifically constructed through two different domains they are:

- UML Analysis modeling, this focuses on the user model and structural model views of the system.
- UML design modeling, which focuses on the behavioral

6.1 CLASS DIAGRAM:

Class diagrams are the blueprints of your system or subsystem. You can use class diagrams to model the objects that make up the system, to display the relationships between the objects, and to describe what those objects do and the services that they provide. Class diagrams are useful in many stages of system design.

The class diagram is used to represent a static view of the system. It plays an essential role in the establishment of the component and deployment diagrams. It helps to construct an executable code to perform forward and backward engineering for any system, or we can say it is mainly used for construction. It represents the mapping with object-oriented languages that are C++, Java, etc. Class diagrams can be used for the following purposes:

1. To describe the static view of a system.
2. To show the collaboration among every instance in the static view.
3. To describe the functionalities performed by the system.
4. To construct the software application using object-oriented languages.

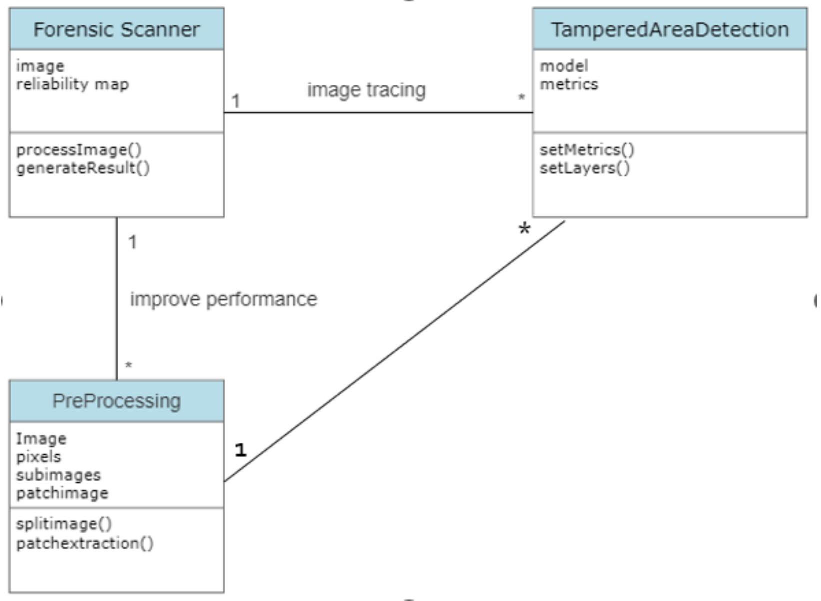


Fig. 6.1 class diagram for forensic scanner identification using deep learning

6.2 ACTIVITY DIAGRAM:

An activity diagram is used to model a large activity's sequential work flow by focusing on action sequences and respective action initiating conditions. The state of an activity relates to the performance of each workflow step. An activity diagram is represented by shapes that are connected by arrows.

An activity partition or a swimlane is a high-level grouping of a set of related actions. A single partition can refer to many things, such as classes, use cases, components, or interfaces. If a partition cannot be shown clearly, then the name of a partition is written on top of the name of an activity. Using a fork and join nodes, concurrent flows within an activity can be generated. A fork node has one incoming edge and numerous outgoing edges.

It is similar to one too many decision parameters. When data arrives at an incoming edge, it is duplicated and split across numerous outgoing edges simultaneously. A single incoming flow is divided into multiple parallel flows.

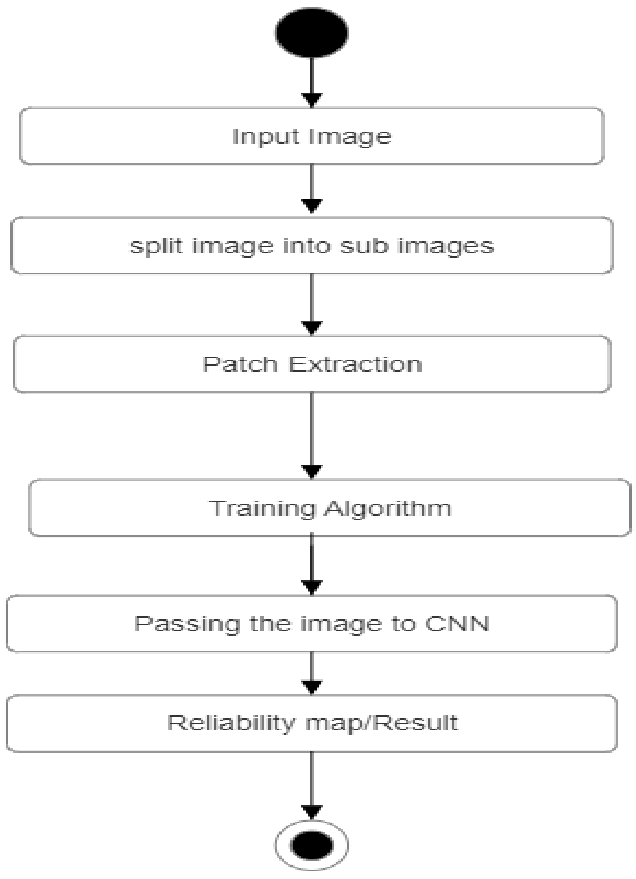


Fig .6.2 Activity diagram for forensic scanner identification using deep learning

6.3 SEQUENCE DIAGRAM

A sequence diagram is a Unified Modeling Language (UML) diagram that illustrates the sequence of messages between objects in an interaction. A sequence diagram consists of a group of objects that are represented by lifelines, and the messages that they exchange over time during the interaction.

A role played by an entity that interacts with the subject is called as an actor. It is out of the scope of the system. It represents the role, which involves human users and external hardware or subjects. An actor may or may not represent a physical entity, but it purely depicts the role of an entity. Several distinct roles can be played by an actor or vice versa. It is represented by a thin rectangle on the lifeline.

It describes that time period in which an operation is performed by an element, such that the top and the bottom of the rectangle is associated with the initiation and the completion time, each respectively. The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

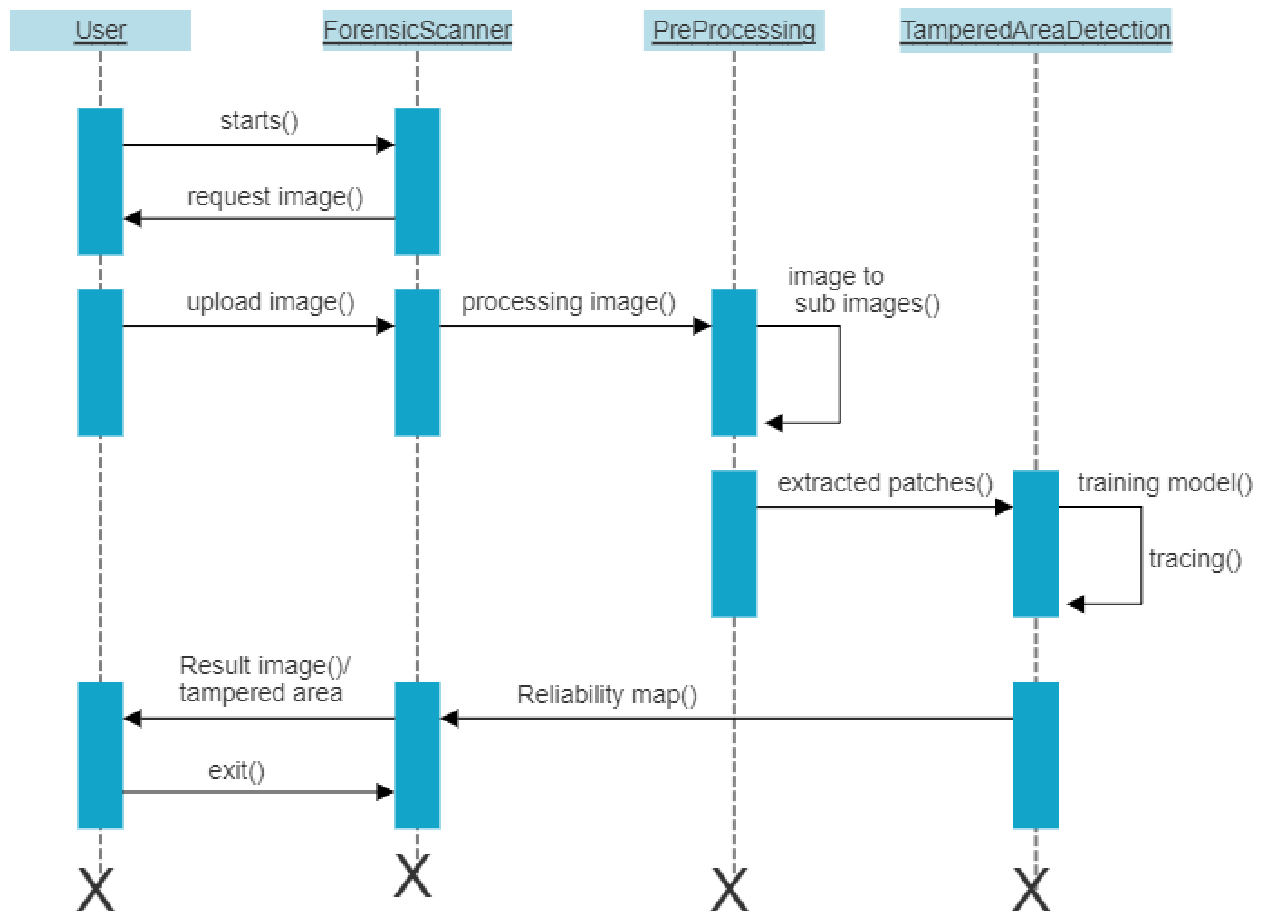


Fig. 6.3 sequence diagram for forensic scanner identification using deep learning

6.4 USECASE DIAGRAM

Use-case diagrams describe the high-level functions and scope of a system. These diagrams also identify the interactions between the system and its actors. The use cases and actors in use-case diagrams describe what the system does and how the actors use it, but not how the system operates internally.

Use-case diagrams illustrate and define the context and requirements of either an entire system or the important parts of the system. You can model a complex system with a single use-case diagram, or create many use-case diagrams to model the components of the system. You would typically develop use-case diagrams in the early phases of a project and refer to them throughout the development process.

Use-case diagrams are helpful in the following situations:

- Before starting a project, you can create use-case diagrams to model a business so that all participants in the project share an understanding of the workers, customers, and activities of the business.

- While gathering requirements, you can create use-case diagrams to capture the system requirements and to present to others what the system should do.
- During the analysis and design phases, you can use the use cases and actors from your use-case diagrams to identify the classes that the system requires.
- During the testing phase, you can use use-case diagrams to identify tests for the system.
- The following topics describe model elements in use-case diagrams:

Use cases

A use case describes a function that a system performs to achieve the user's goal. A use case must yield an observable result that is of value to the user of the system.

Actors

An actor represents a role of a user that interacts with the system that you are modeling. The user can be a human user, an organization, a machine, or another external system.

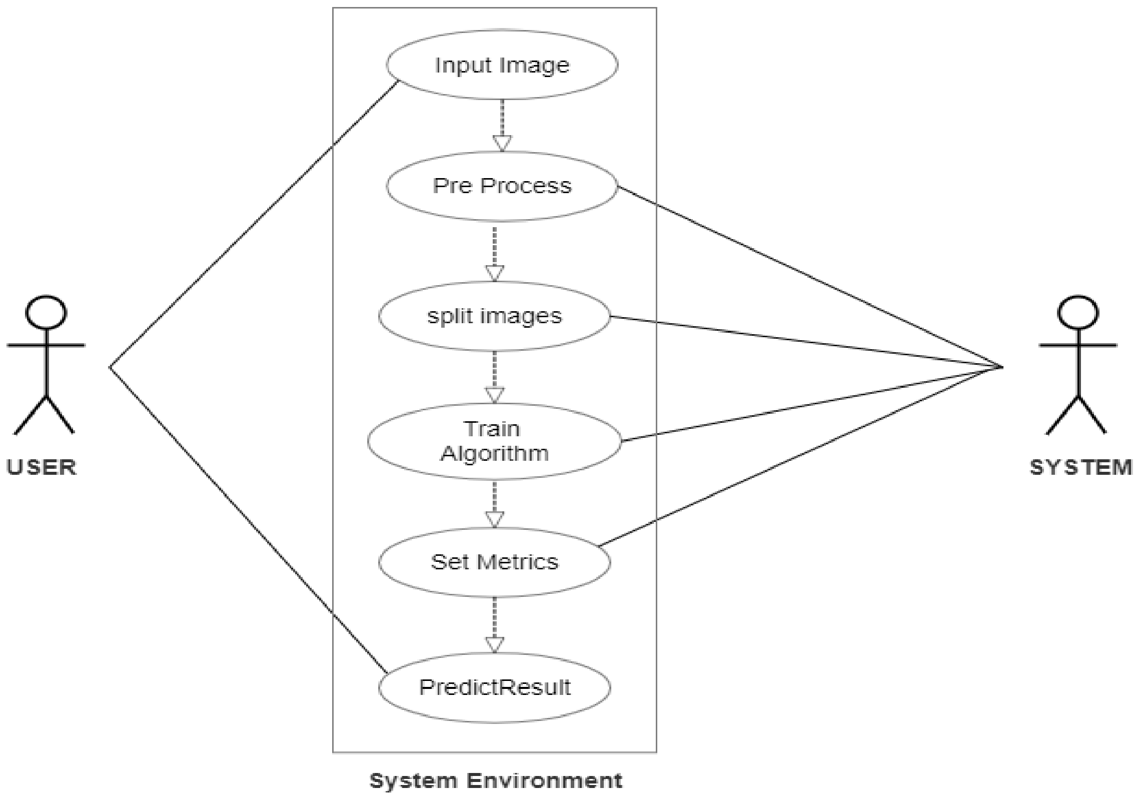


Fig. 6.4 usecase diagram for forensic scanner identification using deep learning

CHAPTER-7

IMPLEMENTATION

In detail design the algorithms of each modules which is used in this project and the detail description of each module is explained.

7.1 Algorithm for Image Splitting

Step 1: Import the necessary modules import PIL from PIL import Image

Step 2: Enter the image file path by using input() function to take the input from the user and save it in a variable.

Step 3: Create an instance of the Image class and open the image using the open() method on this instance.

Step 4: Define the size (height and width) of the sub-images that you would like to split your original image into.

Step 5: Loop through the original image and slice it into the sub-images. You can use the crop() method to extract each sub-image from the original image.

For example, assuming that the sub-image width and height are 100 pixels each:

```
for i in range(0, img.width, 100):
for j in range(0, img.height, 100):
box = (i, j, i+100, j+100)
sub_img = img.crop(box)
sub_img.show()
```

Step 6: Save each sub-image to a separate file. You can use the save() method on each sub-image object to save it to a file.

Note: You can modify the Step 5 as you desire according to your requirements in which equal-sized rectangular sub-images or arbitrary sub-images having the same dimensions have to be produced from the input image.

The Pil module is a python imaging library that provides extensive support for opening, manipulating, and saving different image file formats. One of the most common use-cases of pil module is splitting an image into multiple smaller sub-images, which can then be used for various purposes like creating image galleries, image classification, image segmentation, etc. In this article, we will discuss how to split an image into sub-images using pil module.

Loading the Image The first step in splitting an image is to load the image using the pil module. Suppose that we have an image file "example.jpg" which we want to split into small sub-images. To load this image, we can use the Image.open() method of pil module as shown below: from PIL import Image

```
img = Image.open("example.jpg")
```

Specifying the Size of Sub-images The next step is to specify the size of sub-images. It means that we need to decide how many sub-images we want to create and what should be the size of each sub-image. For example, if we want to create 4 sub-images of an image with dimensions 400x400, then each sub-image should have dimensions 200x200.

Creating Sub-images After specifying the size of sub-images, we can create the sub-images themselves. We can use the crop() method of pil module to extract sub-images from the original image. The crop() method takes a tuple of four values (left, upper, right, lower) that specify the dimensions of the sub-image. Here, left specifies the x-coordinate of the top-left corner of the sub-image, upper specifies the y-coordinate of the top-left corner, right specifies the x-coordinate of the bottom-right corner, and lower specifies the y-coordinate of the bottom-right corner.

The following code snippet shows how to create sub-images from the original image:

```
getting the dimensions of the original image
width, height = img.size
specifying the size of sub-images
sub_image_size = (200, 200)
calculating the number of rows and columns of sub-images
 num_cols = width // sub_image_size[0]
num_rows = height // sub_image_size[1]
creating a list to store sub-images
sub_images = []
iterating over each sub-image and creating it
for i in range(num_rows):
for j in range(num_cols):
left = j * sub_image_size[0]
 upper = i * sub_image_size[1]
right=left+sub_image_size[0]
lower = upper + sub_image_size[1]
sub_images.append(img.crop((left, upper, right, lower)))
```

The above code snippet creates sub-images of size 200x200 from the original image. We first calculate the number of rows and columns of sub-images based on the size we specified. Then we iterate over each sub-image and extract it using the crop() method.

Saving Sub-images

Finally, we can save the sub-images to disk using the save() method of Pil module. The following code snippet shows how to save each sub-image with a unique name:

```
saving each sub-image with a unique name
for i, sub_image in enumerate(sub_images):
sub_image.save(f"sub_image_{i}.jpg")
```

The above code snippet saves each sub-image with a unique name using the enumerate() function. In conclusion, splitting an image into sub-images using Pil module is a straightforward process that involves loading the image, specifying the size of sub-images, creating sub-images using the crop() method, and saving the sub-images to disk using the save() method.

This process can be used for various purposes like creating image galleries, image classification, image segmentation, etc.

7.2 Algorithm for patch extraction

For different reasons, someone might need to split an image into patches of the same size. Once, I had to do it because my Machine Learning model couldn't process high-resolution images, thus, I divided them into multiple parts. In the beginning, I myself wrote the code for splitting, but then I discovered Patchify, which is a great library made for this purpose. It provides two functions: *patchify* and *unpatchify***.** The former is used to split an image into patches and the latter to merge them.

To install the latest version of Patchify from PyPI use:

```
pip install patchify
```

This function splits an image into multiple patches of the same size.

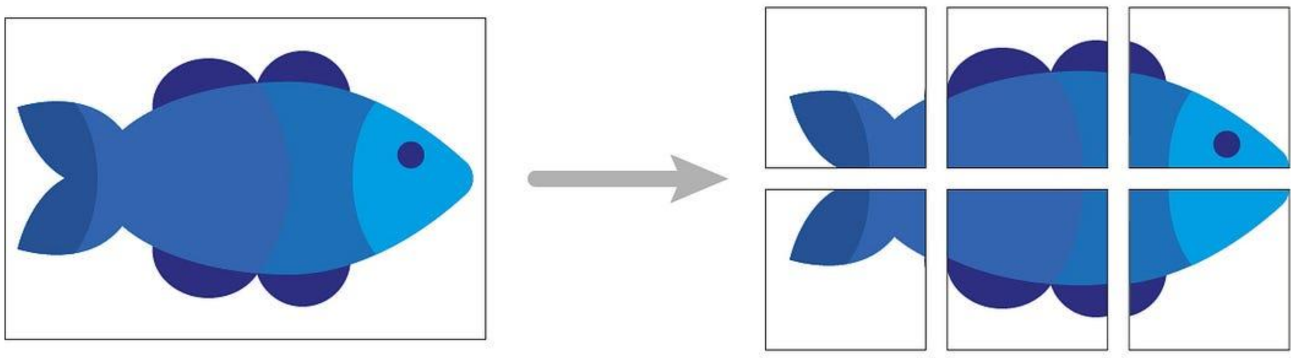


Fig. 7.1 image splitting into patches

To call it use:

```
patchify(image, patch_shape, step)
```

Arguments:

- image is a NumPy array with shape *(image_height, image_width)* for grayscale images or *(image_height, image_width, N)* for N-channels images (*3* if RGB).
- patch_shape is the shape of each patch, *(patch_height, patch_width)* or *(patch_height, patch_width, N)*. It's not required to define a square patch, even a rectangular patch can be defined.
- step defines the distance between one patch and the next one (vertically and horizontally). If *step* ≥ *patch_height* there is no overlap between patches in the same row. If *step* ≥ *patch_width* there is no overlap between patches in the same column.

Return:

If image is N-channels, the function returns a NumPy array with shape *(n_rows, n_cols, 1, H, W, N)*, where *n_rows* is the number of patches for each column and *n_cols* is the number of patches for each row.

Otherwise, if image is grayscale, the function returns a NumPy array with shape *(n_rows, n_cols, 1, H, W)*.The code below splits an RGB image and saves each patch in a new file using an incrementing filename.

```
import
numpy
as np
from patchify import patchify
from PIL import Image

image = Image.open("image.jpg")  # for example (3456, 5184, 3)
image = np.asarray(image)
patches = patchify(image, (512, 512, 3), step=512)
print(patches.shape)  # (6, 10, 1, 512, 512, 3)

for i in range(patches.shape[0]):
    for j in range(patches.shape[1]):
        patch = patches[i, j, 0]
        patch = Image.fromarray(patch)
        num = i * patches.shape[1] + j
        patch.save(f"patch_{num}.jpg")
```

7.3 Algorithm for Error Level Analysis

Step 1:Load the image into the program.

Step 2:Divide the image into blocks.

step 3:For each block, compress it to a predetermined quality level using a standard compression algorithm.

Step 4:Restore the compressed image back to its original quality level.

Step 5:Subtract the restored image from the original image.

Step 6:Calculate the pixel intensity differences between the original and compressed images in each block.

Step 7:Repeat steps 3 to 6 while varying the quality level.

Step 8:Create an error level grid by recording the intensity differences for each block at each quality level.

Step 9:Analyze the error level grid for discrepancies indicating potential areas of manipulation or forgery.

Step 10:Display the results of the analysis.

Note: Error level analysis is a technique used to detect alterations in digital images. The algorithm provided above is a simplified explanation of the process and may be modified for more advanced analysis.

Error Level Analysis (ELA) is a technique used for forensic purposes to detect digital image tampering or manipulation in images. It is based on the fact that every time an image is saved or compressed, it undergoes some changes in the binary data, which creates an error level difference in that image. The method relies on the principle that the regions of compressed or modified images tend to have a significantly different pixel error level than other regions.

According to this principle, ELA method detects the inconsistencies in the pixel brightness levels of an image, which result from different compression levels in the different areas of that image. In other words, ELA is a process to detect discrepancies and variations in the block of pixels within an image. To perform an ELA on an image, the first step is to make a copy of the image and save it using a particular degree of compression. Afterward, the original image and the compressed image are compared by subtracting the pixel values of one image from another at each corresponding block of pixels.

The resulting values give the error level in that block, which helps to identify if the image has been manipulated or not. The areas of the image that are most likely to have been edited or compressed will appear with higher error level values.

The accuracy of the ELA technique is affected by the level and type of compression applied to the image, as well as the amount of noise in the original image. A higher compression level may result in the loss of a considerable amount of data, which can make it harder to detect modifications in the block of pixels. Moreover, images with a low signal-to-noise ratio may also create noise artifacts that may be difficult to differentiate from errors that appear due to image editing.
The versatility of the ELA method can support many forensic applications. For instance, this technique can help identify the original images used in a manipulated image. ELA can also help detect the areas and degree of alteration in an image, which can be useful to understand the extent of image manipulation or tampering. Moreover, ELA can help verify the authenticity of the digital images used as evidence in criminal investigations.

In summary, ELA method is a useful technique to detect digital image manipulations, especially in cases where the images have been compressed or altered. The results obtained from the ELA process provide valuable insights into the method and extent of image tampering, which can be used to support forensic investigations. The limitations of ELA are mainly related to the level and type of compression applied to an image, which may affect the accuracy of the analysis.

7.4 Algorithm for CNN

Step 1:Collect a large dataset of images produced by different forensic scanners. Step 2:Preprocess the images to standardize the color, size, and format for use in the CNN algorithm.
Step 3:Develop a neural network architecture that is optimized for image recognition tasks, such as a deep convolutional neural network.
Step 4:Train the neural network on the preprocessed dataset, using backpropagation to adjust the weights and biases of the network to improve its accuracy.
Step 5:Validate the trained network on a separate test dataset to evaluate its performance and refine the model if necessary.
Step 6:Use the trained CNN model to analyze new images and identify their source by matching their distinctive patterns and features to those learned during the training process.

A convolutional neural network is a feed-forward neural network that is generally used to analyze visual images by processing data with grid-like topology. It’s also known as a ConvNet. A convolutional neural network is used to detect and classify objects in an image.

In CNN, every image is represented in the form of an array of pixel values.
CNN is a type of deep learning model for processing data that has a grid pattern, such as images, which is inspired by the organization of animal visual cortex and designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns. CNN is a mathematical construct that is typically composed of three types of layers (or building blocks): convolution, pooling, and fully connected layers.

The first two, convolution and pooling layers, perform feature extraction, whereas the third, a fully connected layer, maps the extracted features into final output, such as classification.

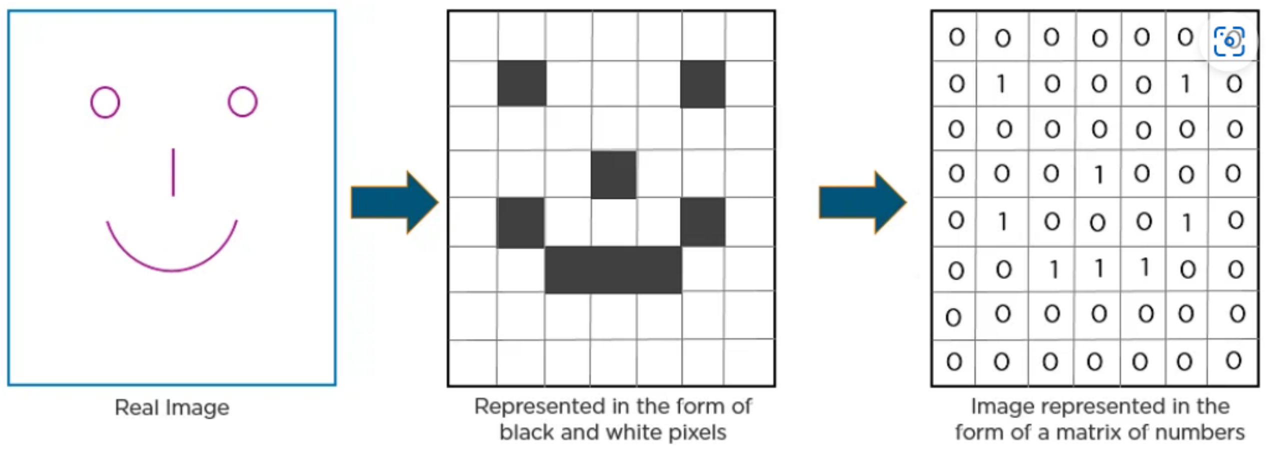
A convolution layer plays a key role in CNN, which is composed of a stack of mathematical operations, such as convolution, a specialized type of linear operation. In digital images, pixel values are stored in a two-dimensional (2D) grid, i.e., an array of and a small grid of parameters called kernel, an optimizable feature extractor, is applied at each image position, which makes CNNs highly efficient for image processing, since a feature may occur anywhere in the image.

As one layer feeds its output into the next layer, extracted features can hierarchically and progressively become more complex. The process of optimizing parameters such as kernels is called training, which is performed so as to minimize the difference between outputs and ground truth labels through an optimization algorithm called backpropagation and gradient descent, among others.

In any Neural Network, first layer will be input layer and last will be the output layer. Input layer contains all the inputs, here images is inputs. These images are given as input to the first convolutional layer. The output of 1st layer will be given as input to the 2nd layer, so on & so forth. This process will continue till the last layer.

While defining Neural Network, first convolutional layer requires the shape of image that is passed to it as input. After passing the image, through all convolutional layers and pooling layers, output will be passed to dense layer.

Here is another example to depict how CNN recognizes an image:


Real Image
Represented in the form of black and white pixels
Image represented in the form of a matrix of numbers

0	0	0	0	0	0	
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

As you can see from the above diagram, only those values are lit that have a value of 1.

Fig. 7.2 CNN recognizes an image the form of a matrix

Layers in a Convolutional Neural Network

Convolution Layer

This is the first step in the process of extracting valuable features from an image. A convolution layer has several filters that perform the convolution operation. Every image is considered as a matrix of pixel values.

Consider the following 5x5 image whose pixel values are either 0 or 1. There's also a filter matrix with a dimension of 3x3. Slide the filter matrix over the image and compute the dot product to get the convolved feature matrix.

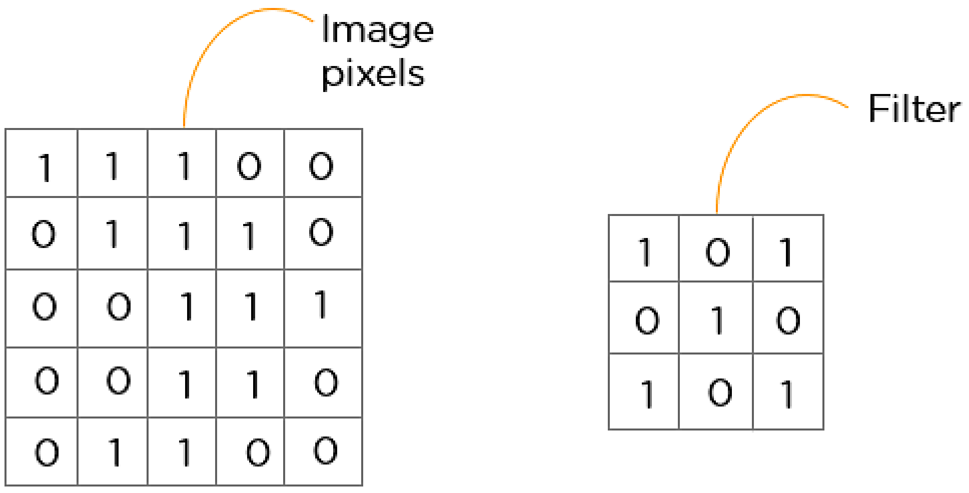


Fig. 7.3 convolved feature matrix

Pooling Layer

Pooling is a down-sampling operation that reduces the dimensionality of the feature map. The rectified feature map now goes through a pooling layer to generate a pooled feature map.

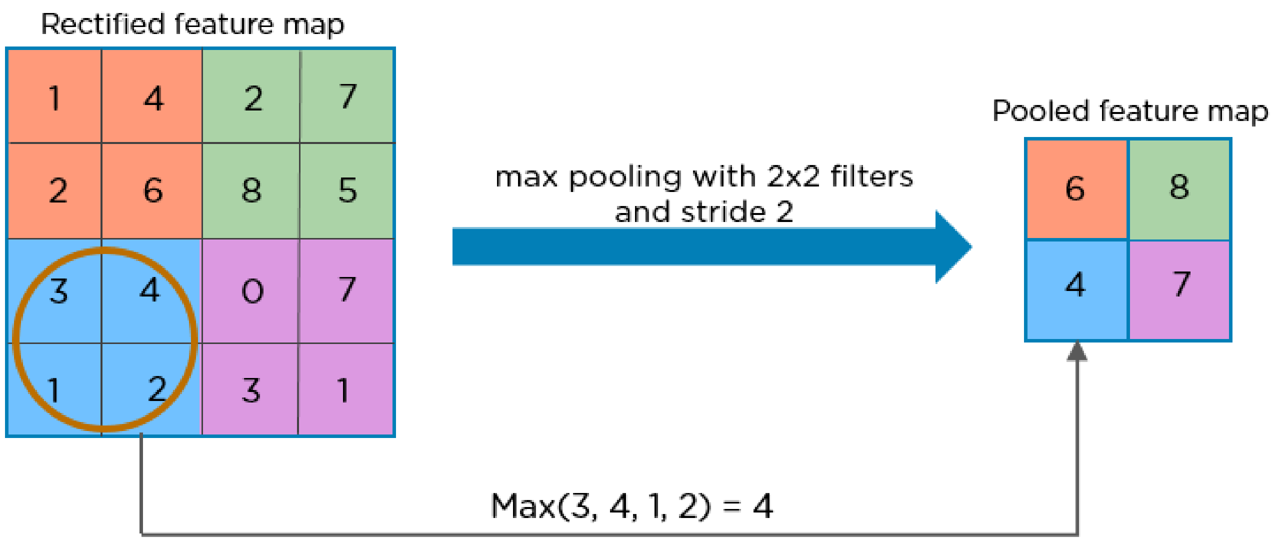


Fig. 7.4 rectified feature map to pooled feature map

The pooling layer uses various filters to identify different parts of the image like edges, corners, body, feathers, eyes, and beak.

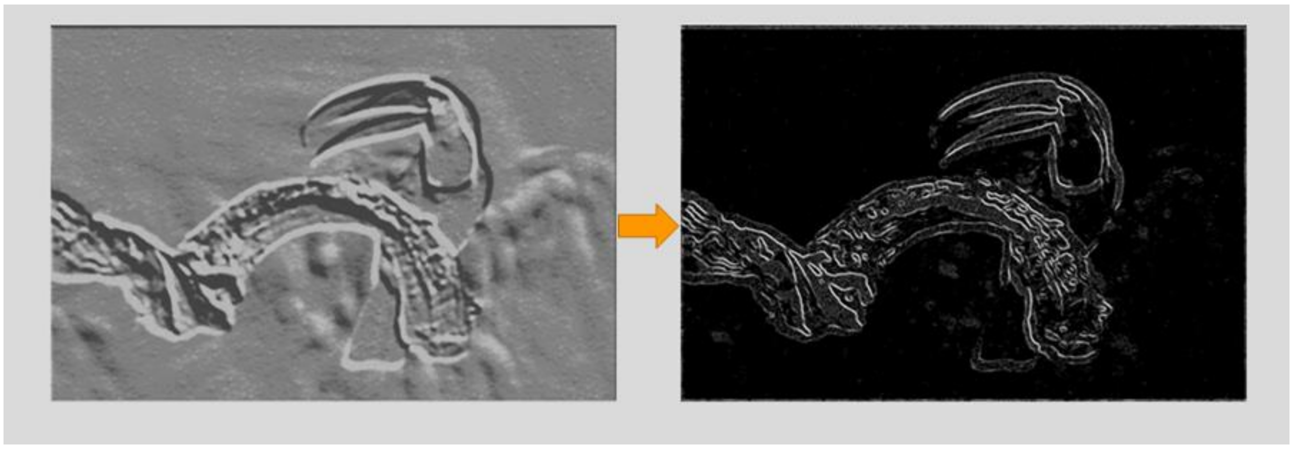


Fig. 7.5 pooling layer using various filters to identify parts of the image

Here's how the structure of the convolution neural network looks so far.

Flattening Layer:

The next step in the process is called flattening. Flattening is used to convert all the resultant 2-Dimensional arrays from pooled feature maps into a single long continuous linear vector.

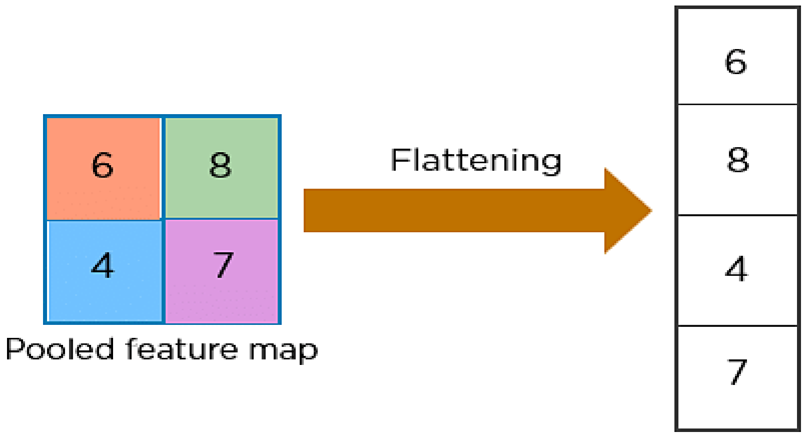


Fig.7.6 pooled feature map to Flattening

The flattened matrix is fed as input to the fully connected layer to classify the image.

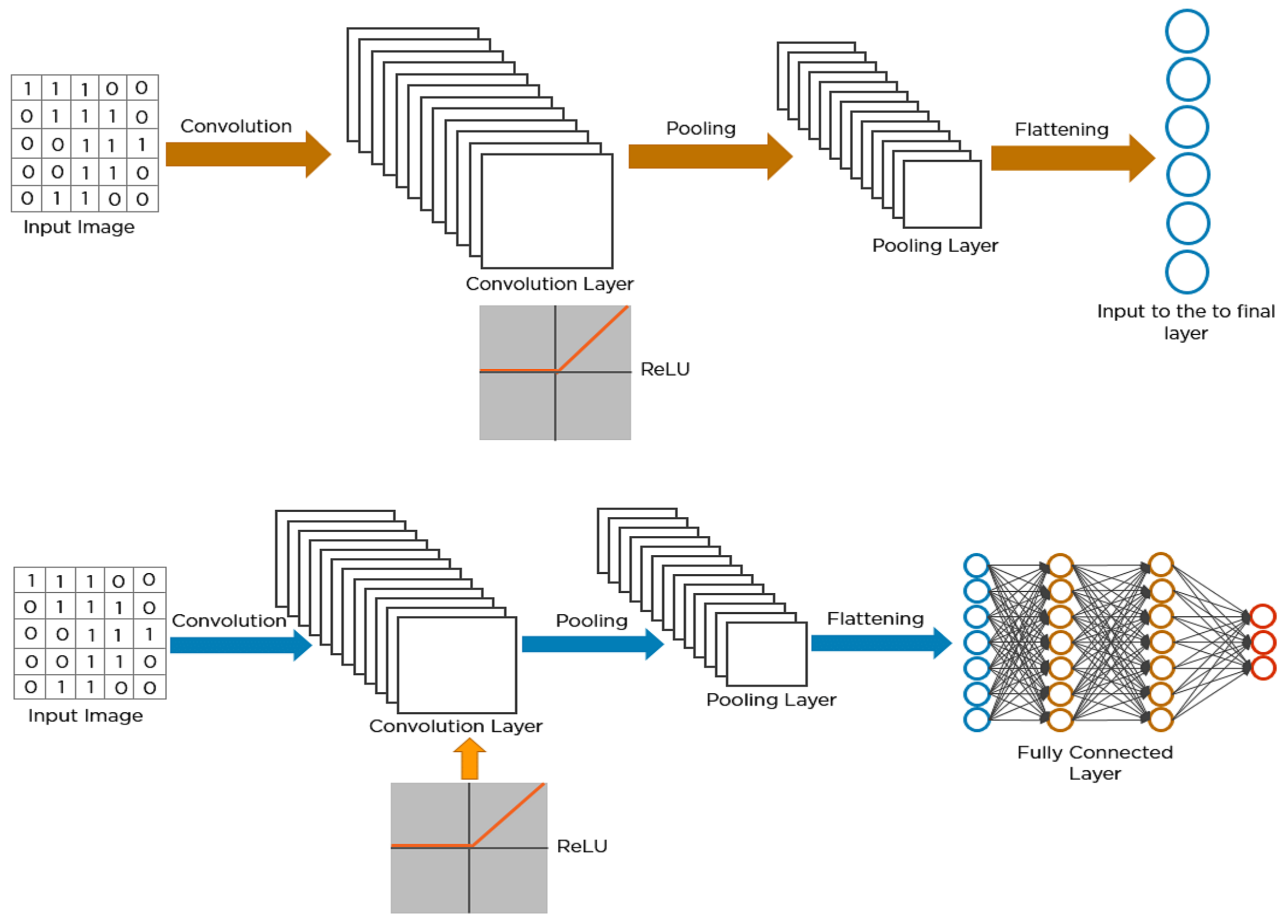
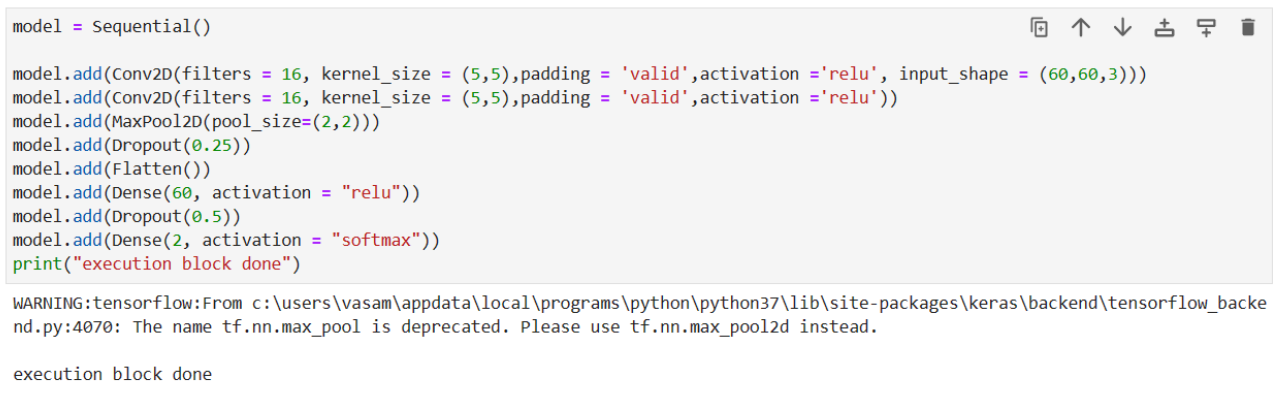


Fig. 7.7 fully connected layer to classify the image.

Deploying the CNN

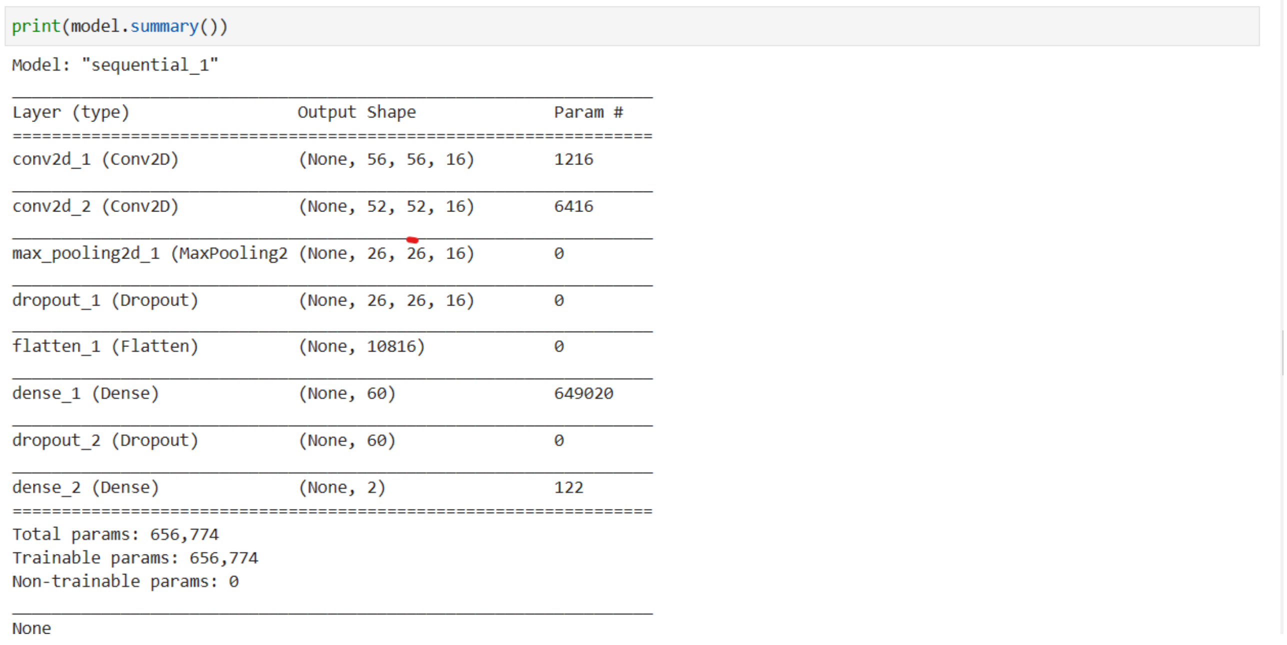


```
model = Sequential()

model.add(Conv2D(filters = 16, kernel_size = (5,5),padding = 'valid',activation ='relu', input_shape = (60,60,3)))
model.add(Conv2D(filters = 16, kernel_size = (5,5),padding = 'valid',activation ='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(60, activation = "relu"))
model.add(Dropout(0.5))
model.add(Dense(2, activation = "softmax"))
print("execution block done")
```

```
WARNING:tensorflow:From c:\users\vasam\appdata\local\programs\python\python37\lib\site-packages\keras\backend\tensorflow_backe
nd.py:4070: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d instead.

execution block done
```



```
print(model.summary())
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 56, 56, 16)        1216
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 52, 52, 16)        6416
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 26, 26, 16)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 26, 26, 16)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 10816)             0
_________________________________________________________________
dense_1 (Dense)              (None, 60)                649020
_________________________________________________________________
dropout_2 (Dropout)          (None, 60)                0
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 122
=================================================================
Total params: 656,774
Trainable params: 656,774
Non-trainable params: 0
_________________________________________________________________
None
```

7.5 Confusion Matrix for CNN Model

A confusion matrix, as the name suggests, is a matrix of numbers that tell us where a model gets confused. It is a class-wise distribution of the predictive performance of a classification model—that is, the confusion matrix is an organized way of mapping the predictions to the original classes to which the data belong.

This also implies that confusion matrices can only be used when the output distribution is known, i.e., in supervised learning frameworks.

The confusion matrix not only allows the calculation of the accuracy of a classifier, be it the global or the class-wise accuracy, but also helps compute other important metrics that developers often use to evaluate their models.

A confusion matrix computed for the same test set of a dataset, but using different classifiers, can also help compare their relative strengths and weaknesses and draw an inference about how they can be combined (ensemble learning) to obtain the optimal performance.

Although the concepts for confusion matrices are similar regardless of the number of classes in the dataset, it is helpful to first understand the confusion matrix for a binary class dataset and then interpolate those ideas to datasets with three or more classes. Let us dive into that next.

Confusion Matrix for Binary Classes

A binary class dataset is one that consists of just two distinct categories of data. These two categories can be named the “positive” and “negative” for the sake of simplicity. Suppose we have a binary class imbalanced dataset consisting of 60 samples in the positive class and 40 samples in the negative class of the test set, which we use to evaluate a machine learning model.

Now, to fully understand the confusion matrix for this binary class classification problem, we first need to get familiar with the following terms:

- *True Positive (TP)* refers to a sample belonging to the positive class being classified correctly.
- *True Negative (TN)* refers to a sample belonging to the negative class being classified correctly.
- *False Positive (FP)* refers to a sample belonging to the negative class but being classified wrongly as belonging to the positive class.
- *False Negative (FN)* refers to a sample belonging to the positive class but being classified wrongly as belonging to the negative class.

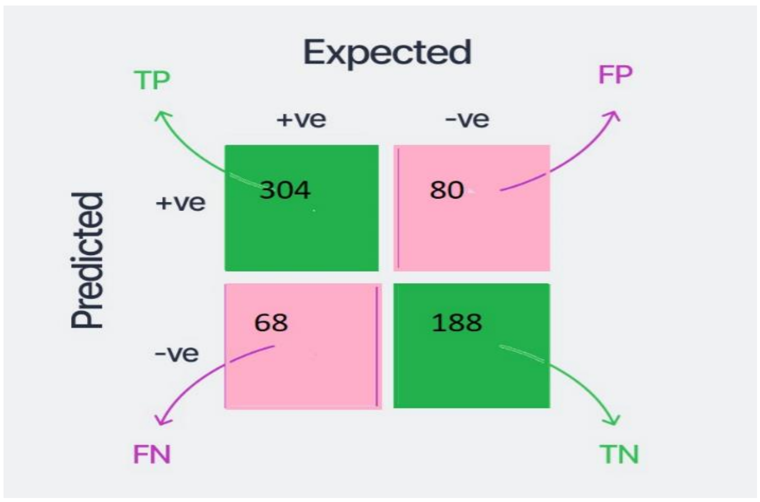


Fig. 7.8 Confusion Matrix

the confusion matrix we may obtain with the trained model is shown above for this dataset. This gives us a lot more information than just the accuracy of the model.

Adding the numbers in the first column, we see that the total samples in the positive class are 304+68=372. Similarly, adding the numbers in the second column gives us the number of samples in the negative class, which is 268 in this case.

The sum of the numbers in all the boxes gives the total number of samples evaluated. Further, the correct classifications are the diagonal elements of the matrix—344 for the positive class and 188 for the negative class.

Now, 68 samples (bottom-left box) that were expected to be of the positive class were classified as the negative class by the model. So it is called "False Negatives" because the model predicted "negative," which was wrong. Similarly, 80 samples (top-right box) were expected to be of negative class but were classified as "positive" by the model.

They are thus called "False Positives." We can evaluate the model more closely using these four different numbers from the matrix.

In general, we can get the following quantitative evaluation metrics from this binary class confusion matrix:

Precision (for the positive class). The number of samples actually belonging to the positive class out of all the samples that were *predicted* to be of the positive class by the model, for the above model the precision is 0.7709195955705345

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall (for the positive class). The number of samples predicted correctly to be belonging to the positive class out of all the samples that *actually belong* to the positive class. for the above model the recall is

0.76875

$$\text{Recall} = \frac{TP}{TP + FN}$$

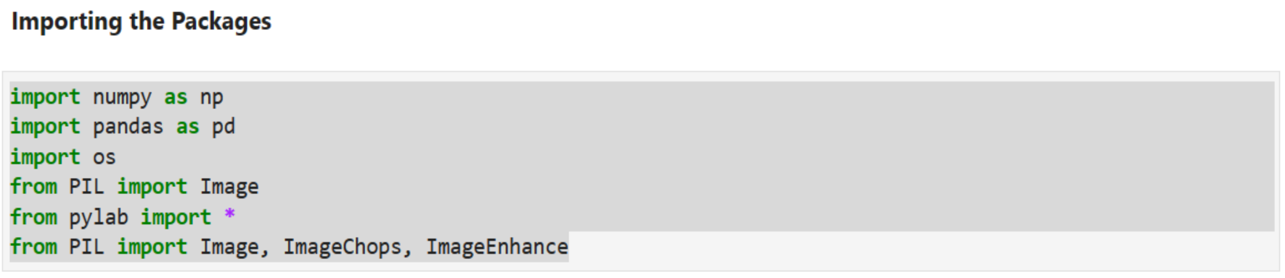
F1-Score (for the positive class). The harmonic mean of the precision and recall scores obtained for the positive class. for the above model the f1 score is 0.7695625833030414

$$\text{F1-Score} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Accuracy. The number of samples correctly classified out of all the samples present in the test set. for the above model the accuracy is 76.8%

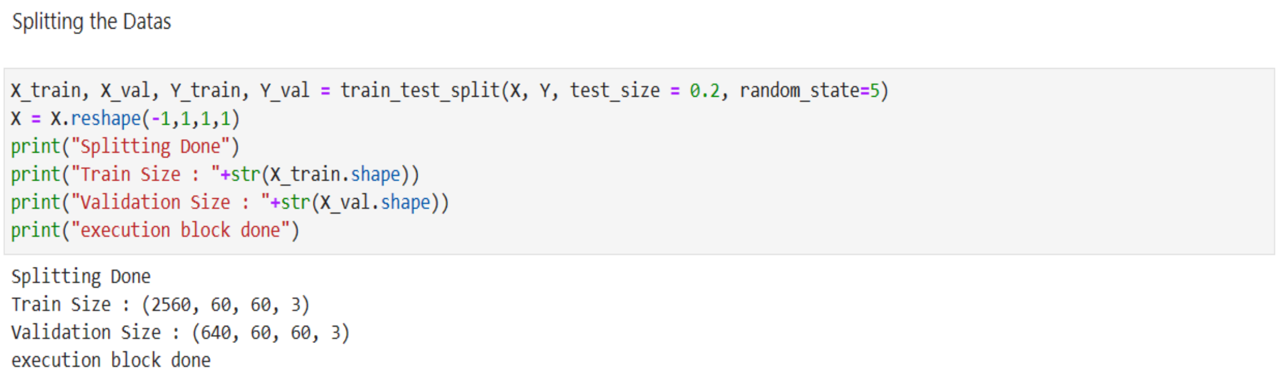
$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

7.6 Sample Code



Importing the Packages

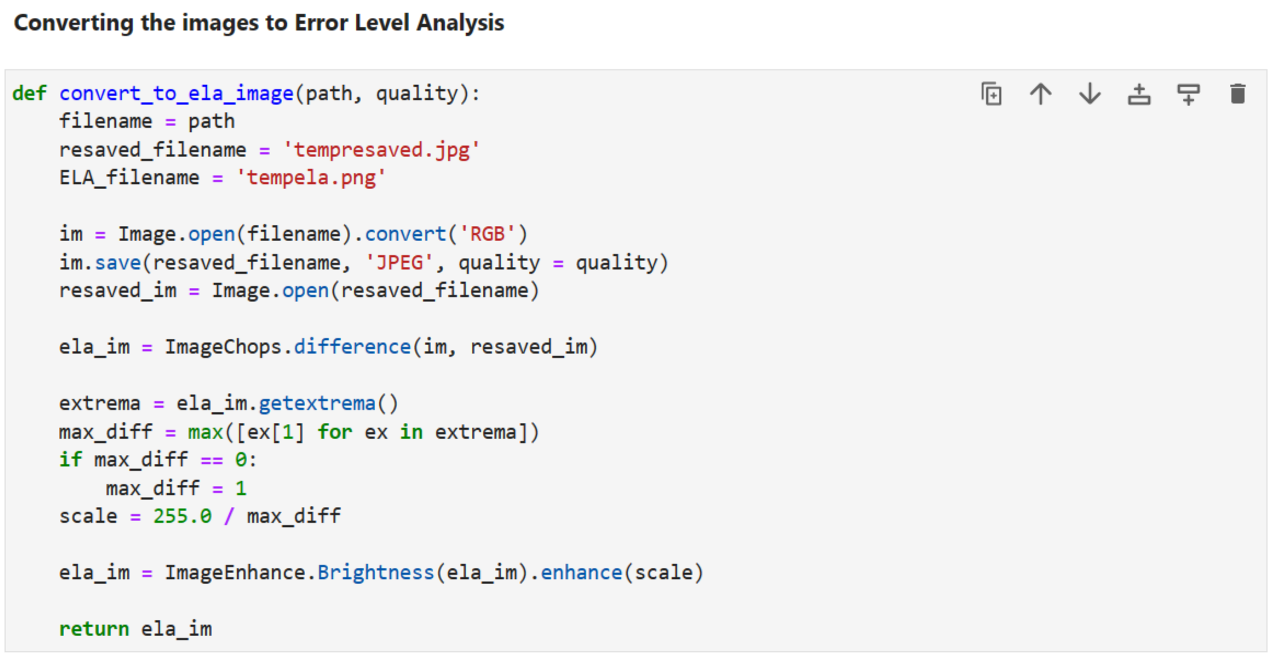
```
import numpy as np
import pandas as pd
import os
from PIL import Image
from pylab import *
from PIL import Image, ImageChops, ImageEnhance
```



Splitting the Datas

```
X_train, X_val, Y_train, Y_val = train_test_split(X, Y, test_size = 0.2, random_state=5)
X = X.reshape(-1,1,1,1)
print("Splitting Done")
print("Train Size : "+str(X_train.shape))
print("Validation Size : "+str(X_val.shape))
print("execution block done")
```

```
Splitting Done
Train Size : (2560, 60, 60, 3)
Validation Size : (640, 60, 60, 3)
execution block done
```



Converting the images to Error Level Analysis

```python
def convert_to_ela_image(path, quality):
    filename = path
    resaved_filename = 'tempresaved.jpg'
    ELA_filename = 'tempela.png'

    im = Image.open(filename).convert('RGB')
    im.save(resaved_filename, 'JPEG', quality = quality)
    resaved_im = Image.open(resaved_filename)

    ela_im = ImageChops.difference(im, resaved_im)

    extrema = ela_im.getextrema()
    max_diff = max([ex[1] for ex in extrema])
    if max_diff == 0:
        max_diff = 1
    scale = 255.0 / max_diff

    ela_im = ImageEnhance.Brightness(ela_im).enhance(scale)

    return ela_im
```

CHAPTER-8
TESTING

The actual purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It is the process of exercising software with the intent of ensuring that the Software system meets its requirements and user expectations and does not fail in an unacceptable manner.

8.1 TYPES OF TESTING

There are many types of testing methods are available in that mainly used testing methods are as follows

8.1.1 Unit Testing

Unit testing involves the design of test cases that validate that the internal program logic is functioning properly, and that program produces valid outputs. All decision branches and internal code flow should be validated. It is the testing of individual software units of the application. It is done after the completion of individual unit before integration.

8.1.2 Integration Testing

Software integration testing is the incremental integration testing of two or more integrated software components on a single platform to produce failures caused by interface defects. The task of the integration test is to check that components or software applications e.g components in the software system or one step up software applications at the company level interact without error.

8.1.3 Regression Testing

Every time a new module is added leads to changes in the program. This type of testing makes sure that the whole component works properly even after adding components to the complete program.

8.1.4 Smoke Testing

This test is done to make sure that the software under testing is ready or stable for further testing
It is called a smoke test as the testing of an initial pass is done to check if it did not catch the fire or smoke in the initial switch on.

8.2 Error Level Analysis of Dataset

By using it, it's possible to rapidly discover image manipulation. The web tool is based on the Python Image Library and the libjpeg library (v6.2.0-822.2). The verification process consists of successive resaves of the image at a predefined quality. The resulting picture is compared with the original one.
If an image hasn't been manipulated, all its parts have been saved the same number of times, images are composed by a portion of other sources, or have been simply been manipulated, will show different level of errors visible in the ELA representation with different colors.

With the ELA method, it's possible to discover image modification by establishing a chronological order of changes of various parts of the image. The lighter parts have been edited most recently, the most opaque have been saved several times.

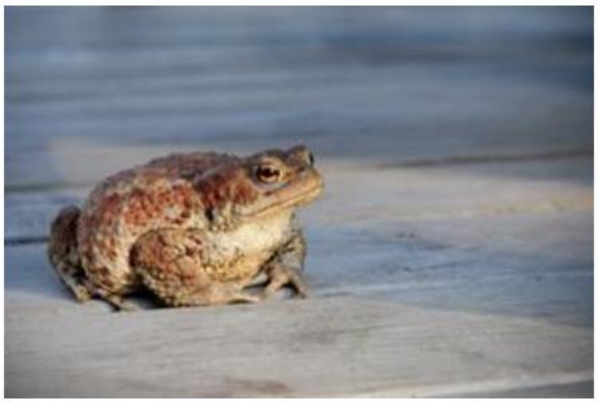


Fig. 8.1 Original image

Then modify it by introducing a stack of coins and changing the aspect of the toad:

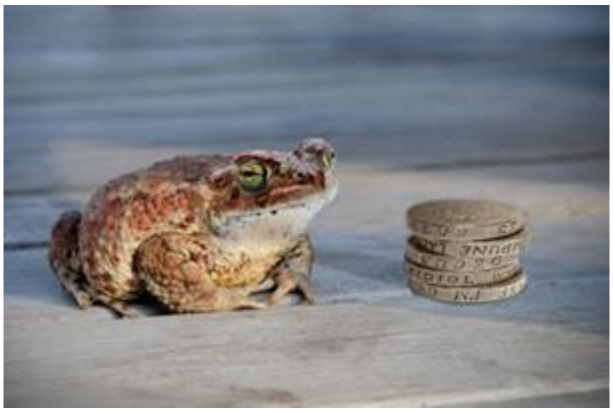


Fig. 8.2 Tampered image

At this point, let’s upload the image to generate the following ELA representation.

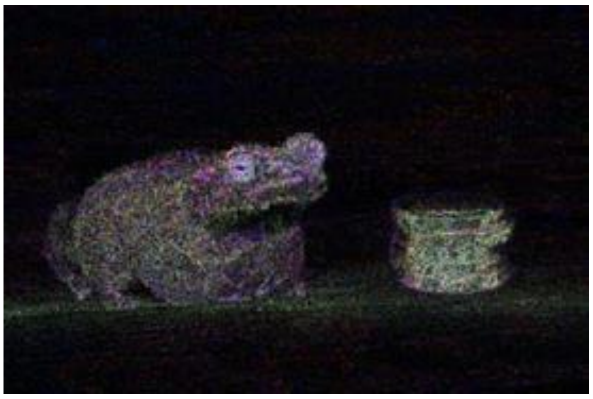


Fig. 8.3 ELA representation

The sections that are black correspond to the parts that usually aren’t manipulated. Solid white blocks usually represent the same. Solid colors present a good level of compression with minimal error levels, displayed as darker areas in the image. ELA highlights the altered portions of the image that represent higher ELA values, and a bright white color. Note that in the outline of objects in high frequency areas, they usually have higher ELA values than the rest of the image. In the following image, the text of the books stands out because the contrast creates a high frequency edge.

After the ela process the oupt shows the only part which was tampered.

8.3 Test Cases

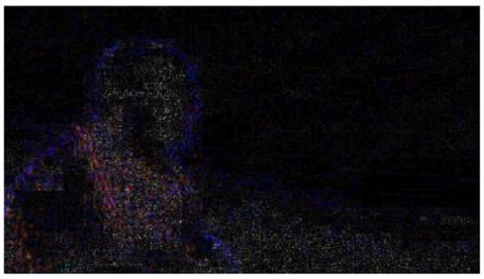
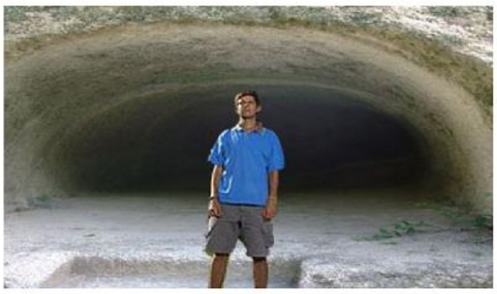
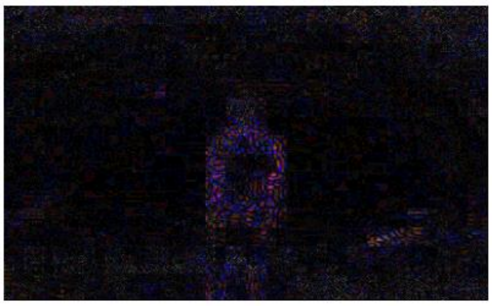
input	output
	
	

Fig. 8.4 set of tampered image and its detected image as output

CHAPTER-9

RESULT ANALYSIS

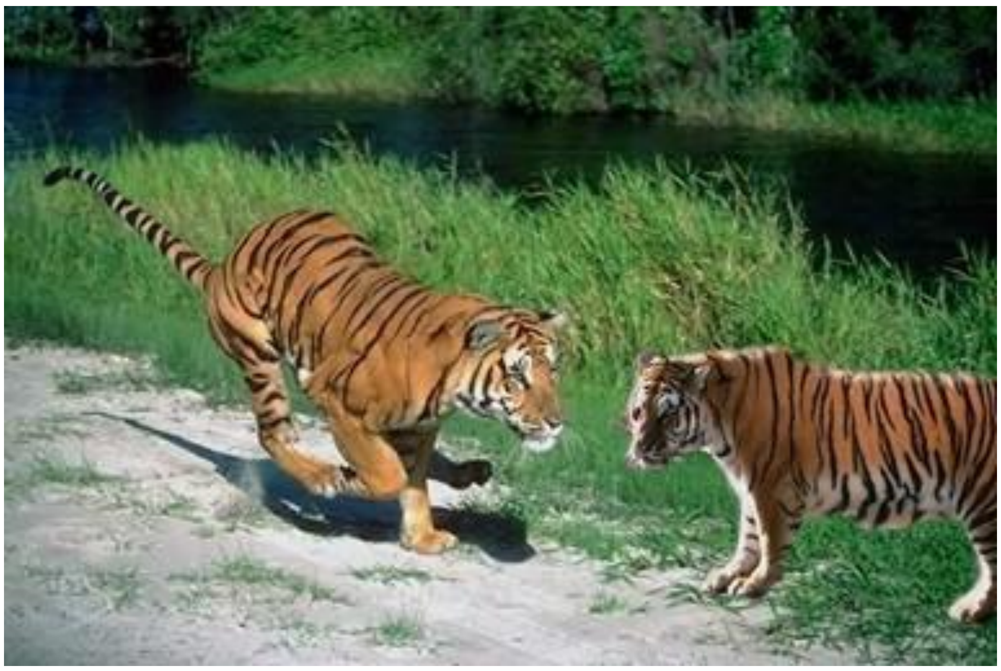


Figure 9.1: Tampered Image

The first step is to input the tampered image to the forensic scanner identification system and there several pre processing techniques will be performed ,at very first the input image is splitted into sub images which is of nXn pixels, the value of each n should not be less than 64 pixels. From each sub-image, a patch of size 64 × 64 is extracted from a random location. We denote this extracted patches Ip. These extracted patches Ip along with their corresponding labels S are inputs into the network. This pre-processing enables the proposed system to work with small-size images and use a smaller network architecture to save training time and memory usage.

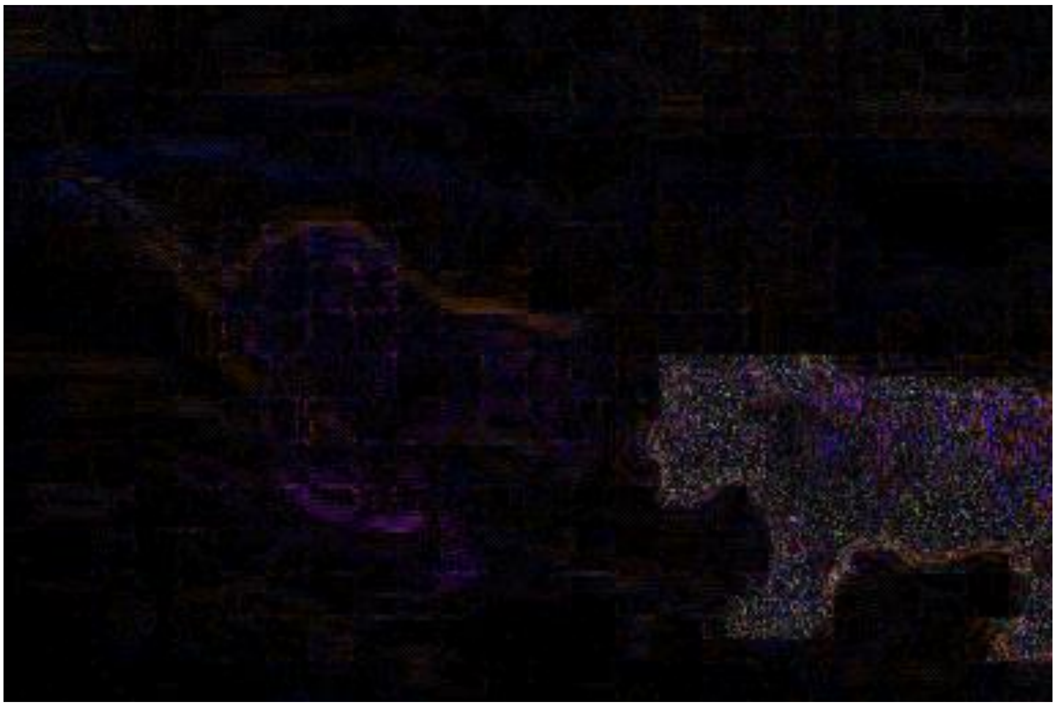


Fig:9.2 Detection of tampered image as output

The final output of the forensic scanner identification system is a reliability map which indicates the tampered image. Since our system is aimed at extracting intrinsic features of scanner models, it should also be able to identify manipulated region irrespective of image content. In this task, we investigate to generate a reliability map (i.e. a heat map) that can indicate suspicious forged areas in the images. The reliability map is generated based on the predicted label obtained by majority vote.

Figure 9.2 shows an example of the reliability map. In the reliability map, the colour of the pixel represents the probability that it is generated by the predicted scanner model.

CHAPTER-10
CONCLUSION AND FUTURE SCOPE

In this project we investigate the use of deep-learning methods to address scanner model classification and localization. Compared with classical methods, our proposed system can: a) learn intrinsic scanner features automatically; b) have no restrictions on data collection; c) associate small image patches (64 × 64 pixels) to scanner models with high accuracy; and d) detect image forgery and localization on small image size. the proposed system can automatically learn the inherit features to differentiate scanner models and is robust to JPEG compression. the ability of the proposed system to identify suspected forged regions in scanned images. These experimental results indicate that our reliability map provides a way to detect forgeries in scanned images. Further work will be devoted to: a) improve the neural network architecture in the proposed system, and b) evaluate the performance of the proposed system on scanned documents.

CHAPTER-11
REFERENCES

- J. Lukas, J. Fridrich, and M. Goljan, “Digital camera identification from sensor pattern noise,” IEEE Transactions on Information Forensics and Security, vol. 1, no. 2, pp. 205–214, June 2006.
- C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9, June 2015, Boston, MA..
- F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1800–1807, July 2017, Honolulu, HI.
- B. Bayar and M. C. Stamm, “A deep learning approach to universal image manipulation detection using a new convolutional layer,” Proceedings of the 4th ACM Workshop on Information Hiding and Multimedia Security, pp. 5– 10, June 2016, Vigo, Galicia, Spain. [Online]. Available: http://dx.doi.org/10.1145/2909827.2930786
- J. Lukas, J. Fridrich, and M. Goljan, “Digital camera identification from sensor pattern noise,” IEEE Transactions on Information Forensics and Security, vol. 1, no. 2, pp. 205–214, June 2006. [Online]. Available: http://dx.doi.org/10.1109/TIFS.2006.873602
- S. Bayram, H. Sencar, N. Memon, and I. Avcibas, “Source camera identification based on cfa interpolation,” Proceedings of the IEEE International Conference on Image Processing, pp. 69–72, September 2005, Genova, Italy. [Online]. Available: http://dx.doi.org/10.1109/ICIP.2005.1530330
- A. Tuama, F. Comb, and M. Chaumont, “Camera model identification with the use of deep convolutional neural networks,” Proceedings of the IEEE International Workshop on Information Forensics and Security (WIFS), pp. 1–6, December 2016, Abu Dhabi, United Arab Emirates. [Online]. Available: http://dx.doi.org/10.1109/WIFS.2016.7823908
- N. Khanna, A. K. Mikkilineni, and E. J. Delp, “Scanner identification using feature-based processing and analysis,” IEEE Transactions on Information Forensics and Security, vol. 4, no. 1, pp. 123–139, March 2009. [Online]. Available: http://dx.doi.org/10.1109/TIFS.2008.2009604