An automatic under-sea feature detection and classification system using computer vision and neural networks

Roussos Ioannis Dimitrios

Diploma Thesis

National Technical
University of Athens

Thesis Supervisor: Assistant Professor G. Papalambrou /SNAME

Committee Member: Assistant Professor Ch. Papadopoulos /SNAME

Committee Member: Associate Professor Alex. Ginis /SNAME

June 2018

Acknowledgements

This work has been carried out at the Laboratory of Marine Engineering (LME) at the School of Naval Architecture and Marine Engineering of the National Technical University of Athens, under the supervision of Assistant Professor George Papalambrou.

I would like to thank my supervisor Assistant Professor George Papalambrou for giving me the chance to work on this thesis. In addition to his constant guidance, valuable comments and fruitful discussions, I have to thank him for his unceasing encouragement and support through this thesis.

I am also grateful to my family for their constant encouragement and support throughout my thesis.

Abstract

The aim of this thesis is to the design of an application for underwater surveillance focused on sites of archaeological interest using image processing and computer vision techniques. The project includes a camera placed placed underwater that watches a designated area and the software that performs recognition of any possible intruder and alarms. The application must be autonomous, run perpetually with minimum need of maintenance or any kind of human intervention.

Taking into account everything above, this thesis designs an algorithm able to run day and night and detecting possible threats. There are two modes of the program. One running during the day when there is plenty of sunlight and we can perform higher level image processing and one during the night where the brightness is below a certain standard.

During the night we assume that a possible intruder, a scuba diver or a ROV, will need some kind of light source in order to guide in situations of minimum light or even absolute dark. In this case our program detects light sources inside our area or reflections on the seabed caused from light sources outside our camera's watching field.

During higher level of light (day) we use a pre-trained neural network that is trained to detect divers or ROVs. Every frame taken from the camera is classified as diver, ROV or seabed. In the case where a diver or a ROV are detected, alarm mode is activated and a signal is sent. Furthermore a registration is created with the exact time of the event and the type of the intruder. Since we did not have access to the ideal resources or actual site our model is trained with images found on the web or extracted from scuba diving videos.

The results in the light detection part are very satisfying. With proper light measurements in tests from the actual environment a very high level of accuracy can be

achieved. In the diver or ROV detection part a high accuracy is achieved but not as high as completely autonomous system would require. Possible modifications for accuracy improvement include a larger and more suitable data-set, a more complex CNN architecture and higher resolution images. All these would require the use of more powerful hardware as well.

Περίληψη

Σκοπός της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός μιας εφαρμογής υποβρύχιας επιτήρησης επικεντρωμένης σε χώρους αρχαιολογικού ενδιαφέροντος με χρήση τεχνικών επεξεργασίας εικόνας και computer vision. Το έργο περιλαμβάνει μια κάμερα τοποθετημένη υποβρύχια που παρακολουθεί μια καθορισμένη περιοχή και το λογισμικό που εκτελεί αναγνώριση οποιουδήποτε πιθανού εισβολέα και ενεργοποιεί κάποιο σήμα συναγερμού. Η εφαρμογή πρέπει να είναι αυτόνομη, να εκτελείται συνεχώς με ελάχιστη ανάγκη για συντήρηση ή κάθε είδους ανθρώπινης παρέμβασης.

Λαμβάνοντας υπόψη όλα τα παραπάνω, αυτή η εργασία σχεδιάζει έναν αλγόριθμο ικανό να τρέχει μέρα και νύχτα και να ανιχνεύει πιθανές απειλές. Υπάρχουν δύο καταστάσεις λειτουργίας του προγράμματος. Μία που τρέχει κατά τη διάρκεια της ημέρας όταν υπάρχει άφθονο ηλιακό φως και μπορούμε να εκτελέσουμε επεξεργασία εικόνας υψηλότερου επιπέδου και μία άλλη κατά τη διάρκεια της νύχτας όπου η φωτεινότητα είναι κάτω από ένα συγκεκριμένο επίπεδο.

Κατά τη διάρκεια της νύχτας υποθέτουμε ότι ένας πιθανός εισβολέας, δύτης ή ROV, θα χρειαστεί κάποια φωτεινή πηγή για να καθοδηγηθεί σε καταστάσεις ελάχιστου φωτισμού ή και απόλυτου σκοταδιού. Σε αυτή την περίπτωση το πρόγραμμά μας ανιχνεύει πηγές φωτός μέσα στην περιοχή επιτήρησης ή αντανακλάσεις στον πυθμένα που προέρχονται από πηγές φωτός έξω από το πεδίο παρακολούθησης της κάμερας.

Κατά τη διάρκεια υψηλότερου φωτισμού (ημέρα) χρησιμοποιούμε ένα νευρωνικό δίκτυο που έχει εκπαιδευτεί εκ των προτέρων για να ανιχνεύει δύτες ή ROVs. Κάθε εικόνα που λαμβάνεται από την κάμερα ταξινομείται ως δύτης, ROV ή βυθός. Στην περίπτωση που εντοπιστεί δύτης ή ROV, ενεργοποιείται η λειτουργία συναγερμού και αποστέλλεται σήμα. Επιπλέον δημιουργείται εγγραφή με την ακριβή ώρα του συμβάντος και τον τύπο του εισβολέα. Δεδομένου ότι δεν είχαμε πρόσβαση στους ιδανικούς πόρους ή στον πραγματικό

τόπο, το μοντέλο μας εκπαιδεύεται με εικόνες που βρίσκονται στο διαδίκτυο ή εξάγονται από βίντεο κατάδυσης.

Τα αποτελέσματα στο τμήμα ανίχνευσης φωτός είναι πολύ ικανοποιητικά. Με τις κατάλληλες μετρήσεις του επιπέδου φωτός από το πραγματικό περιβάλλον και έπειτα από σειρά δοκιμών, μπορεί να επιτευχθεί πολύ υψηλό επίπεδο ακρίβειας. Στον τμήμα ανίχνευσης δύτη ή ROV επιτυγχάνεται υψηλή ακρίβεια, αλλά όχι τόσο υψηλή όσο χρειάζεται σε ένα εντελώς αυτόνομο σύστημα. Πιθανές τροποποιήσεις για τη βελτίωση της ακρίβειας περιλαμβάνουν μία μεγαλύτερη βάση δεδομένων με κατάλληλες εικόνες για την εκπάιδευση του μοντέλου, ένα πιο πολύπλοκο νευρωνικό δίκτυο, εικόνες μεγαλύτερης ανάλυσης και όλα αυτά με την προϋπόθεση ενός επαρκώς ισχυρού hardware.

Contents

List of Figures

Listings

Chapter 1

Introduction

The main purpose of this thesis is the design of an application for underwater surveillance focused on sites of archaeological interest using image processing and computer vision techniques. The project includes a camera placed placed underwater that watches a designated area and the software that performs recognition of any possible intruder and alarms. The application must be autonomous, run perpetually with minimum need of maintenance or any kind of human intervention.

The algorithm divides the problem in two situations, daytime where the brightness is higher and night where brightness is lower or even zero. During the night we assume that a possible intruder, a scuba diver or a ROV, will need some kind of light source in order to guide in situations of minimum light or even absolute dark. In this case our program using image processing tools detects light sources inside our area or reflections on the seabed caused from light sources outside our camera's watching field. During higher level of light (day) we use a pre-trained neural network that is trained to detect divers or ROVs. Every frame taken from the camera is classified as diver, ROV or seabed. In the case where a diver or a ROV are detected, alarm mode is activated and a signal is sent. Furthermore a registration is created with the exact time of the event and the type of the intruder.

1.1 Literature Search

Several studies have been performed in the field of underwater computer vision for exploration and mapping purposes [19]. scientists are now turning to the oceans to dis-

cover new possibilities for telecommunications, biological and geological resources and energy sources. Underwater vehicles play an important role in this exploration as the deep ocean is a harsh and unforgiving environment for human discovery. Unmanned underwater vehicles (UUV) are utilized for many different scientific, military and commercial applications such as high resolution seabed surveying, mine countermeasures, inspection and repair of underwater man-made structures and wreck discovery and.

One of the most challenging problems in underwater robotics is the processing of underwater images. Besides the well known problems to automatically interpret an image in order to interact with the environment, underwater robotics needs to deal with additional problems caused by the degradation of the image due to the light transmission in water. A real time deep learning solution for image dehazing is proposed and compared with other state of the art alternatives. in [18].

When working on underwater computer vision applications the need to to detect certain objects or patterns and classify them emerges. In [17] supervised machine learning methods to automatically detect and recognize coral reef fishes in underwater HD videos are presented. The first method relies on a traditional two-step approach: extraction of HOG features and use of a SVM classifier, while the second method is based on Deep Learning [6].

Another study on underwater image classification is presented in [20].Underwater surveillance cameras collect data which then are used by a video processing software to detect and recognize fish species. This footage is processed on supercomputers, which allow marine biologists to request automatic processing on these videos and afterwards analyze the results using a web-interface that allows them to display counts of fish species in the camera footage.

Computer vision has many applications in other fields as well like autonomous vehicle and face recognition.

KITTI Vision Benchmark [21] takes advantage of their autonomous driving platform to develop novel challenging benchmarks for the tasks of stereo, optical flow, visual odometry / SLAM and 3D object detection. The recording platform is equipped with four high resolution video cameras, a Velodyne laser scanner and a state-of-the-art localization system. The cameras, laser scanner and localization system are calibrated and synchronized, providing accurate ground truth.

In the field of face recognition, apple started using deep learning for face detection in iOS 10. With the release of the Vision framework, developers can now use this technology and many other computer vision algorithms in their applications. [22] discusses the algorithmic approach to deep-learning-based face detection, and how it successfully meets the challenges to achieve state-of-the-art accuracy.

Chapter 2

Image Processing

In this chapter we will discuss the fundamentals about image processing. We will focus on the features of image processing used in this paper, as well as, image processing from the hardware perspective. Special attention will be given to convolution and masking.

2.1 Introduction to image processing

An image is considered to be a function of two real variables, for example, **f(x,y)** with f as the amplitude (e.g. brightness) of the image at the real coordinate position *(x,y)*.

Image processing is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. It is a type of signal processing in which input is an image and output may be image or characteristics/features associated with that image. Nowadays, image processing is among rapidly growing technologies. It forms core research area within engineering and computer science disciplines too.

2.2 Hardware (CCD/CMOS)

2.2.1 Introduction to image sensors

When an image is being captured by a network camera, light passes through the lens and falls on the image sensor. The image sensor consists of picture elements, also called pixels, that register the amount of light that falls on them. They convert the received amount of light into a corresponding number of electrons. The stronger the

light, the more electrons are generated. The electrons are converted into voltage and then transformed into numbers by means of an A/D-converter. The signal constituted by the numbers is processed by electronic circuits inside the camera.

Presently, there are two main technologies that can be used for the image sensor in a camera, CCD (Charge-coupled Device)shown in figure 2.1 and CMOS (Complementary Metal-oxide Semiconductor)shown in figure 2.2.

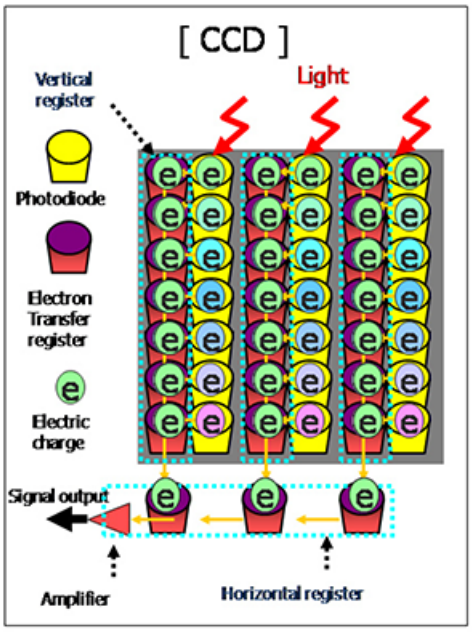

Figure 2.1: CCD

2.2.2 Color Filtering

Image sensors register the amount of light from bright to dark with no color information. Since CMOS and CCD image sensors are /' color blind/', a filter in front of the sensor allows the sensor to assign color tones to each pixel. Two common color registration methods are RGB (Red, Green, and Blue) and CMYG (Cyan, Magenta, Yellow, and Green). Red, green, and blue are the primary colors that, mixed in different combinations, can produce most of the colors visible to the human eye.

Another way to filter or register color is to use the complementary colors, cyan, magenta, and yellow. Complementary color filters on sensors are often combined with green filters to form a CMYG color array, see Figure 2 (right). The CMYG system

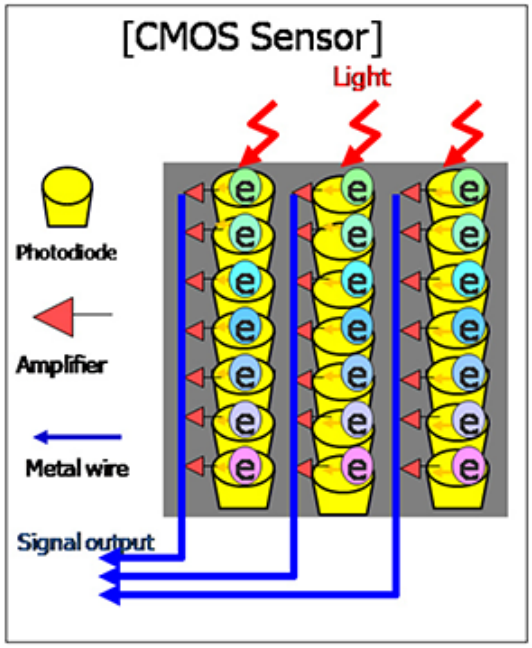

Figure 2.2: CMOS

generally offers higher pixel signals due to its broader spectral band pass. However, the signals must then be converted to RGB since this is used in the final image, and the conversion implies more processing and added noise. The result is that the initial gain in signal-to-noise is reduced, and the CMYG system is often not as good at presenting colors accurately. The CMYG color array is often used in interlaced CCD image sensors, whereas the RGB system primarily is used in progressive scan image sensors. For more information about interlaced CCD image sensors and progressive scan image sensors.

2.2.3 CCD

In a CCD sensor, the light (charge) that falls on the pixels of the sensor is transferred from the chip through one output node, or only a few output nodes. The charges are converted to voltage levels, buffered, and sent out as an analog signal. This signal is then amplified and converted to numbers using an A/D-converter outside the sensor.

The CCD technology was developed specifically to be used in cameras, and CCD sensors have been used for more than 30 years. Traditionally, CCD sensors have had some advantages compared to CMOS sensors, such as better light sensitivity and less

noise. In recent years, however, these differences have disappeared.

The disadvantages of CCD sensors are that they are analog components that require more electronic circuitry outside the sensor, they are more expensive to produce, and can consume up to 100 times more power than CMOS sensors. The increased power consumption can lead to heat issues in the camera, which not only impacts image quality negatively, but also increases the cost and environmental impact of the product.

CCD sensors also require a higher data rate, since everything has to go through just one output amplifier, or a few output amplifiers.

2.2.4 CMOS

Early on, ordinary CMOS chips were used for imaging purposes, but the image quality was poor due to their inferior light sensitivity. Modern CMOS sensors use a more specialized technology and the quality and light sensitivity of the sensors have rapidly increased in recent years.

CMOS chips have several advantages. Unlike the CCD sensor, the CMOS chip incorporates amplifiers and A/D-converters, which lowers the cost for cameras since it contains all the logics needed to produce an image. Every CMOS pixel contains conversion electronics. Compared to CCD sensors, CMOS sensors have better integration possibilities and more functions. However, this addition of circuitry inside the chip can lead to a risk of more structured noise, such as stripes and other patterns. CMOS sensors also have a faster readout, lower power consumption, higher noise immunity, and a smaller system size.

Calibrating a CMOS sensor in production, if needed, can be more difficult than calibrating a CCD sensor. But technology development has made CMOS sensors easier to calibrate, and some are nowadays even self-calibrating.

It is possible to read individual pixels from a CMOS sensor, which allows windowing, which implies that parts of the sensor area can be read out, instead of the entire sensor area at once. This way a higher frame rate can be delivered from a limited part of the sensor, and digital PTZ (pan/tilt/zoom) functions can be used. It is also possible to achieve multi-view streaming, which allows several cropped view areas to be streamed simultaneously from the sensor, simulating several virtual cameras.

2.3 Software (OpenCV)

The program is written in Python [3], [4] using OpenCV (Open Source Computer Vision) [1], [2], [5] which is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel, it was later supported by Willow Garage then Itseez (which was later acquired by Intel). The library is cross-platform and free for use under the open-source BSD license. It is written in C++ and its primary interface is in C++, but there are bindings in Python, Java and MATLAB/OCTAVE.

OpenCV's application areas include:

- 2D and 3D feature toolkits
- Facial recognition system
- Gesture recognition
- Human–computer interaction (HCI)
- Segmentation and recognition
- Stereopsis
- Motion tracking
- Augmented reality

OpenCV also includes a statistical machine learning library that contains:

- Boosting
- Decision tree learning
- Expectation-maximization algorithm
- k-nearest neighbor algorithm
- Naive Bayes classifier
- Artificial neural networks
- Support vector machine (SVM)
- Deep neural networks (DNN)

2.4 Basic Features of Images

Pixels

Pixel (PEL) is the smallest element of an image. Each pixel correspond to any one value. In an 8-bit gray scale image, the value of the pixel between 0 and 255. The value of a pixel at any point correspond to the intensity of the light photons striking at that point. Each pixel store a value proportional to the light intensity at that particular location. The number of PEL would be equal to the number of rows multiply with number of columns.This can be mathematically represented as below:

$$Total\ Number\ Of\ Pixels = (Number\ Of\ Rows) \cdot (Number\ Of\ Columns) \quad (2.1)$$

Or we can say that the number of *(x,y)* coordinate pairs make up the total number of pixels.

The value of the pixel at any point denotes the intensity of image at that location, and that is also known as gray level.

Colors

Bpp or bits per pixel denotes the number of bits per pixel. The number of different colors in an image is depends on the depth of color or bits per pixel.

If we devise a formula for the calculation of total number of combinations that can be made from bit, it would be like this:

$$Total\ Number\ Of\ Colors = 2^{Bpp} \quad (2.2)$$

Therefor 8 Bpp would correspond to 256 colors

0 pixel will always be the black color as it is the lowest number. On the other hand the white does not have a fixed value. The value that denotes white color can be calculated as:

$$White\ Color = 2^{Bpp-1} \quad (2.3)$$

Image size

The size of an image depends upon three things.

- Number of rows
- Number of columns
- Number of bits per pixel

The formula for calculating the size is given below:

$$Size = Rows \cdot Cols \cdot Bpp \tag{2.4}$$

Types of Images

There are many type of images, and we will look in detail about different types of images, and the color distribution in them.

Binary Image The binary image as it name states, contain only two pixel values, 0 and 1. The resulting image that is formed hence consist of only black and white color and thus can also be called as Black and White image. Binary images have a format of PBM (Portable bit map).

2, 3, 4, 5, 6 bit color format The images with a color format of 2, 3, 4, 5 and 6 bit are not widely used today. They were used in old times for old TV displays, or monitor displays. Each of these colors have more then two gray levels, and hence has gray color unlike the binary image. In a 2 bit 4, in a 3 bit 8, in a 4 bit 16, in a 5 bit 32, in a 6 bit 64 different colors are present.

8 bit color format 8 bit color format is one of the most famous image format. It has 256 different shades of colors in it. It is commonly known as Grayscale image. The range of the colors in 8 bit vary from 0-255. Where 0 stands for black, and 255 stands for white, and 127 stands for gray color. This format was used initially by early models of the operating systems UNIX and the early color Macintoshes. The format of these images are PGM

Binary Image16 bit color format It is a color image format. It has 65,536 different colors in it. It has been used by Microsoft in their systems that support more then 8 bit color format. Now in this 16 bit format and the next format we are going to discuss which is a 24 bit format are both color format. The distribution of color in a color image is not as simple as it was in grayscale image. A 16 bit format is actually divided into three further formats which are Red, Green and Blue, also known as channels. The famous (RGB) format.

The most usual distribution of 16 bit is done like this:

5 bits for R, 6 bits for G, 5 bits for B.

The additional is added into the green bit. Because green is the color which is most soothing to eyes in all of these three colors. Note this is distribution is not followed by all the systems. Some have introduced an alpha channel in the 16 bit.

24 bit color format 24 bit color format also known as true color format. Like 16 bit color format, in a 24 bit color format, the 24 bits are again distributed in three different formats of Red, Green and Blue. Since 24 is equally divided on 8, so it has been distributed equally between three different color channels. Unlike a 8 bit gray scale image, which has one matrix behind it, a 24 bit image has three different matrices of R, G, B.

It is the most common used format. Its format is PPM (Portable pixMap) which is supported by Linux operating system. Windows has its own format for it which is BMP (Bitmap).

Grayscale to RGB Conversion

Very often in image processing we tend to work on grayscale format instead of the RGB. This happens because working on one channel is much faster than working on three different channels. In order to do that we need to convert the image from the RGB format to grayscale, process it and then convert back to RGB. There are two methods to convert it. Both has their own merits and demerits. The methods are:

- Average method

- Weighted method or luminosity method

Average method is the most simple one. You just have to take the average of three colors. Since its an RGB image, so it means that you have add r with g with b and then divide it by 3 to get your desired grayscale image. The formula for this method is the following.

$$Grayscale = (R + G + B)/3 \tag{2.5}$$

We can see the results of applying this equation to an image in figure 2.3

Figure 2.3: Gray-scale Transformation

As we can see the outcome is not as expected. We wanted to convert the image into a gray-scale, but this turned out to be a rather black image.This problem arise due to the fact, that we take average of the three colors. Since the three different colors have three different wavelength and have their own contribution in the formation of image, so we have to take average according to their contribution, not done it averagely using average method. Right now what we are doing is this, 1/3 of Red, 1/3 of Green and 1/3 of Blue. That means, each of the portion has same contribution in the image. But in reality that is not the case. The solution to this has been given by luminosity method.

Since red color has more wavelength of all the three colors, and green is the color that has not only less wavelength then red color but also green is the color that gives more soothing effect to the eyes. It means that we have to decrease the contribution of red color, and increase the contribution of the green color, and put blue color contribution in between these two. So the new equation that form is:

$$Grayscale = (0.3 \cdot R) + (0.59 \cdot G) + (0.11 \cdot B) \tag{2.6}$$

According to this equation, Red has contribute 30 per cent, Green has contributed 59 per cent which is greater in all three colors and Blue has contributed 11 per cent. We can see the results of applying this equation to an image in figure 2.4

Figure 2.4: Weighted Gray-scale Transformation

As we can see here, the image has now been properly converted to gray-scale using weighted method. As compare to the result of average method, this image is more brighter.

Resolution

Resolution can be defined in many ways. Such as pixel resolution, spatial resolution, temporal resolution, spectral resolution. Out of which we are going to discuss pixel and spatial resolutions.

In pixel resolution, the term resolution refers to the total number of count of pixels in an digital image. For example. If an image has M rows and N columns, then its resolution can be defined as M X N.

$$Size = Pixel\ Resolution \cdot bpp \tag{2.7}$$

If we define resolution as the total number of pixels, then pixel resolution can be defined with set of two numbers. The first number the width of the picture, or the pixels across columns, and the second number is height of the picture, or the pixels across its width. We can say that the higher is the pixel resolution, the higher is the quality of the image.

The size of an image can be defined by its pixel resolution. Lets say we have an image of dimension: 2500 X 3192. Its pixel resolution = 2500 * 3192 = 7982350 bytes.

Dividing it by 1 million = 7.9 = 8 mega pixel (approximately).

Another important concept with the pixel resolution is aspect ratio. Aspect ratio is the ratio between width of an image and the height of an image. It is commonly explained as two numbers separated by a colon (8:9). This ratio differs in different images, and in different screens. The common aspect ratios are: 1.33:1, 1.37:1, 1.43:1, 1.50:1, 1.56:1, 1.66:1, 1.75:1, 1.78:1, 1.85:1, 2.00:1, e.t.c. Aspect ratio maintains a balance between the appearance of an image on the screen, means it maintains a ratio between horizontal and vertical pixels. It does not let the image to get distorted when aspect ratio is increased.

Spatial resolution states that the clarity of an image cannot be determined by the pixel resolution. The number of pixels in an image does not matter. Spatial resolution can be defined as the smallest discernible detail in an image. Or in other way we can define spatial resolution as the number of independent pixels values per inch.

In short what spatial resolution refers to is that we cannot compare two different types of images to see that which one is clear or which one is not. If we have to compare the two images, to see which one is more clear or which has more spatial resolution, we have to compare two images of the same size.

Since the spatial resolution refers to clarity, so for different devices, different measure has been made to measure it:

- Dots per inch (DPI) usually used in monitors
- Lines per inch (LPI) usually used in laser printers
- Pixels per inch (PPI) usually used for different devices such as tablets , Mobile phones e.t.c.

Zooming

Zooming simply means enlarging a picture in a sense that the details in the image became more visible and clear. We can zoom something at two different steps. The first step includes zooming before taking an particular image. This is known as pre processing zoom. This zoom involves hardware and mechanical movement. The second step is to zoom once an image has been captured. It is done through many different

algorithms in which we manipulate pixels to zoom in the required portion. There are two types of zoom. Optical zoom and digital zoom

Optical Zoom: The optical zoom is achieved using the movement of the lens of your camera. An optical zoom is actually a true zoom. The result of the optical zoom is far better then that of digital zoom. In optical zoom, an image is magnified by the lens in such a way that the objects in the image appear to be closer to the camera. In optical zoom the lens is physically extend to zoom or magnify an object.

Digital Zoom: Digital zoom is basically image processing within a camera. During a digital zoom, the center of the image is magnified and the edges of the picture got crop out. Due to magnified center, it looks like that the object is closer to you. During a digital zoom, the pixels got expand , due to which the quality of the image is compromised. The same effect of digital zoom can be seen after the image is taken through your computer by using an image processing toolbox / software, such as Photoshop.

Brightness and Contrast

Brightness is a relative term. It depends on your visual perception. Since brightness is a relative term, so brightness can be defined as the amount of energy output by a source of light relative to the source we are comparing it to. In some cases we can easily say that the image is bright, and in some cases, its not easy to perceive. Brightness can be simply increased or decreased by simple addition or subtraction, to the image matrix.

Contrast can be simply explained as the difference between maximum and minimum pixel intensity in an image.

$$Contrast = Maximum\ Pixel\ Intensity - Minimum\ Pixel\ Intensity \tag{2.8}$$

2.4.1 Image Transformation

Consider this equation:

$$G(x,y) = Tf(x,y) \tag{2.9}$$

In this equation,

F(x,y) = input image on which transformation function has to be applied.

G(x,y) = the output image or processed image.

T is the transformation function.

This relation between input image and the processed output image can also be represented as:

$$s = T(r) \tag{2.10}$$

where r is actually the pixel value or gray level intensity of f(x,y) at any point. And s is the pixel value or gray level intensity of g(x,y) at any point.

2.5 Histograms

2.5.1 Introduction to Histograms

A histogram is a graph. A graph that shows frequency of anything. Usually histogram have bars that represent frequency of occurring of data in the whole data set. A Histogram has two axis the x axis and the y axis. The x axis contains event whose frequency you have to count. The y axis contains frequency.

Histogram of an image, like other histograms also shows frequency. But an image histogram, shows frequency of pixels intensity values. In an image histogram, the x axis shows the gray level intensities and the y axis shows the frequency of these intensities.

For example we have figure 2.5

The histogram of this image would be something like figure 2.6

The x axis of the histogram shows the range of pixel values. Since its an 8 bpp image, that means it has 256 levels of gray or shades of gray in it. That is why the range of x axis starts from 0 and end at 255 with a gap of 50. Whereas on the y axis, is the count of these intensities.

As we can see from the graph, that most of the bars that have high frequency lies in the first half portion which is the darker portion. That means that the image we have got is darker. And this can be proved from the image too.

Histograms has many uses in image processing. The first use as it has also been discussed above is the analysis of the image. We can predict about an image by just

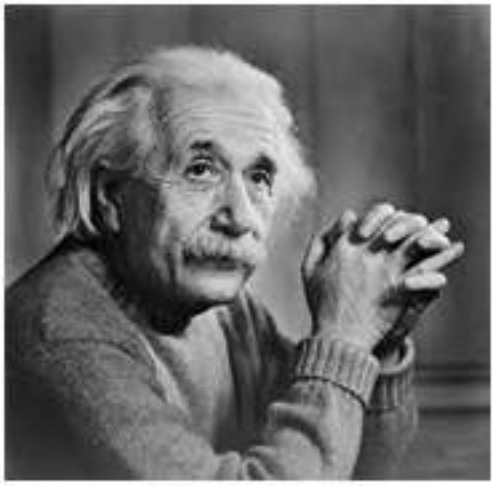

Figure 2.5: Gray-scale Einstein

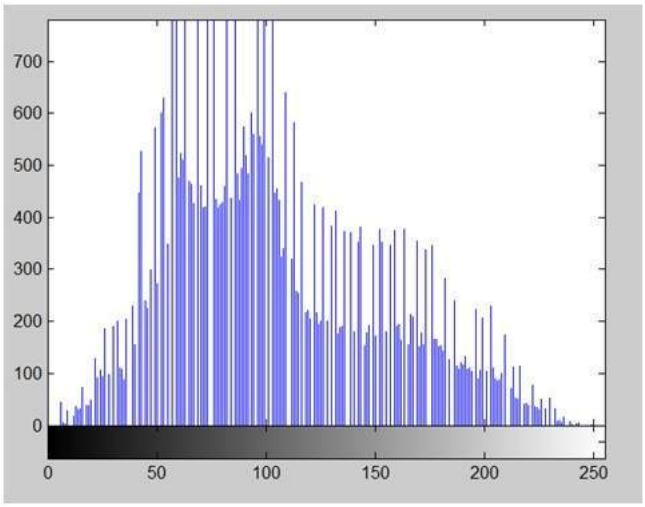

Figure 2.6: Einstein Histogram

looking at its histogram. Its like looking an x ray of a bone of a body. The second use of histogram is for brightness purposes. The histograms has wide application in image brightness. Not only in brightness, but histograms are also used in adjusting contrast of an image. Another important use of histogram is to equalize an image. And last but not the least, histogram has wide use in thresholding. This is mostly used in computer vision.

2.5.2 Histogram Sliding

In histogram sliding, we just simply shift a complete histogram rightwards or leftwards. Shifting an histogram to the right will result in a brighter image (figure 2.7). In order to do that we need to add a fixed value to the image.

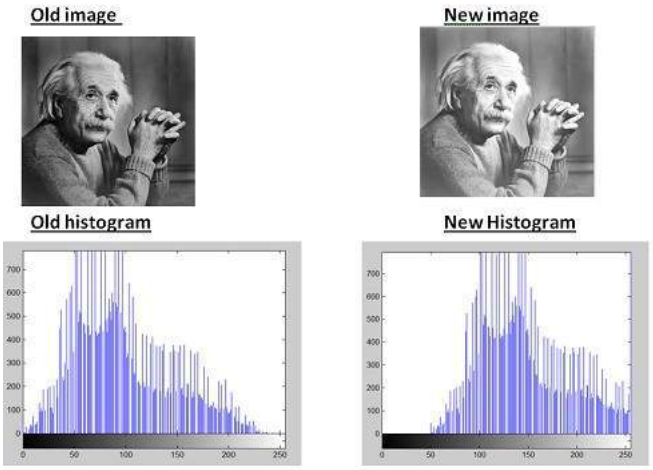

Figure 2.7: Shifting Histogram Rightwards

Respectively Shifting an histogram to the left (figure 2.8) can be done by subtracting a fixed value from them image and it will result in a decrease of brightness.

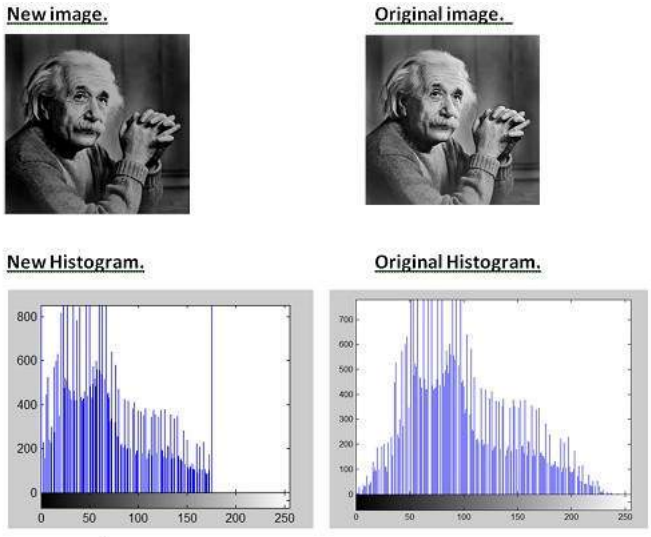

Figure 2.8: Shifting Histogram Leftwards

2.5.3 Histogram stretching

There are two methods of enhancing contrast. The first one is called Histogram stretching. The formula for stretching the histogram of the image to increase the contrast is the following:

$$g(x,y) = \frac{f(x,y) - fmin}{fmax - fmin} \cdot 2^{Bpp} \tag{2.11}$$

The formula requires finding the minimum and maximum pixel intensity multiply by levels of gray. In our case the image is 8bpp, so levels of gray are 256. The minimum value is 0 and the maximum value is 225. So the formula in our case is

$$g(x,y) = \frac{f(x,y) - 0}{255 - 0} \cdot 255 \tag{2.12}$$

where f(x,y) denotes the value of each pixel intensity. For each f(x,y) in an image , we will calculate this formula. After doing this, we will be able to enhance our contrast. The figure 2.9 is created after applying histogram stretching.

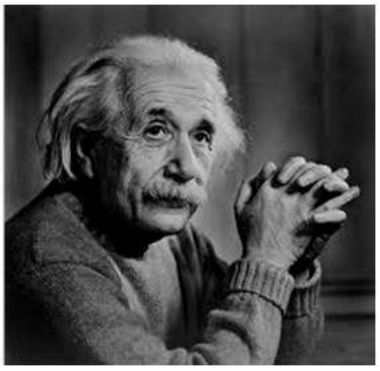

Figure 2.9: Einstein after Histogram Stretching

The stretched histogram of this image can be shown in figure (2.10).

We can notice that the histogram is now stretched or in other means expand.

2.5.4 Histogram Equalization

Another method of enhancing the contrast of an image is Histogram Equalization. It is not necessary that contrast will always be increase in this. There may be some cases were histogram equalization can be worse. In that cases the contrast is decreased.

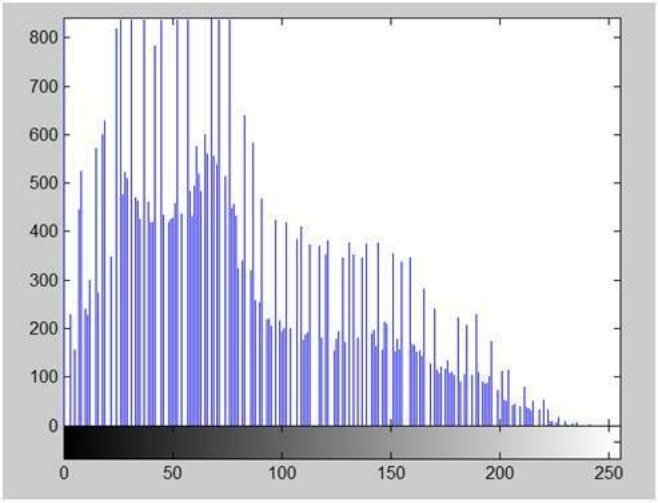

Figure 2.10: Histogram Stretching

This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values.

The method is useful in images with backgrounds and foregrounds that are both bright or both dark. In particular, the method can lead to better views of bone structure in x-ray images, and to better detail in photographs that are over or under-exposed. A key advantage of the method is that it is a fairly straightforward technique and an invertible operator. So in theory, if the histogram equalization function is known, then the original histogram can be recovered. The calculation is not computationally intensive. A disadvantage of the method is that it is indiscriminate. It may increase the contrast of background noise, while decreasing the usable signal.

The results after histogram equalization can be shown in figure 2.11

As we can clearly see from the images that the new image contrast has been enhanced and its histogram has been equalized. There is also one important thing to be note here that during histogram equalization the overall shape of the histogram changes, where as in histogram stretching the overall shape of histogram remains same

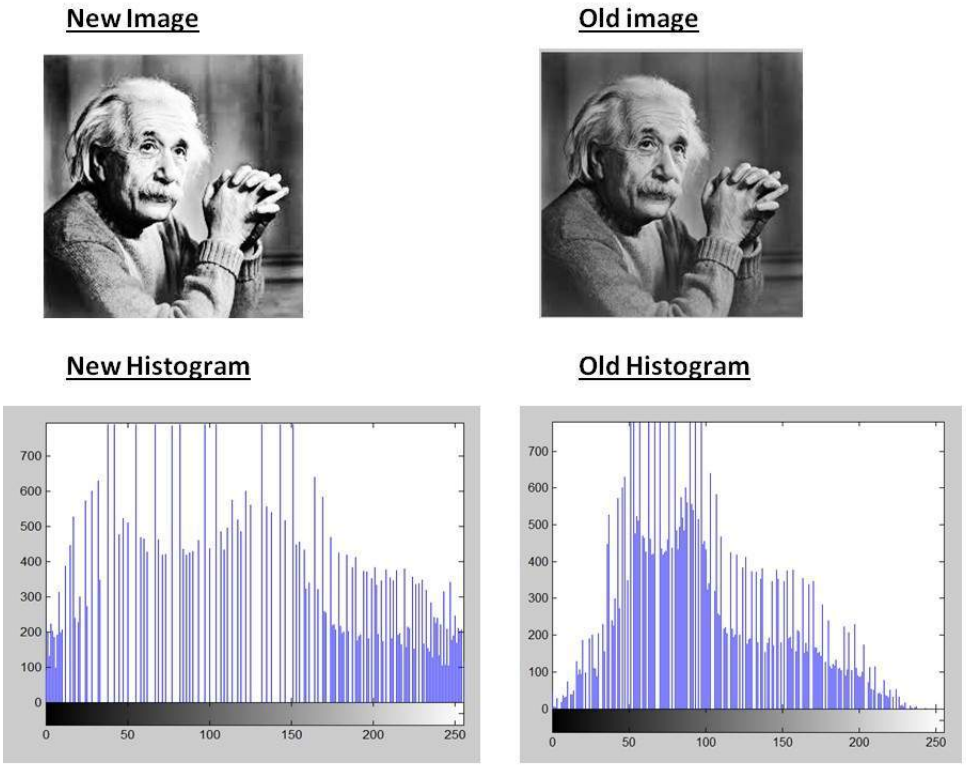

Figure 2.11: Histogram Equalization

2.6 Convolution and Masks

In image processing, a kernel, convolution matrix, or mask is a small matrix. It is used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between a kernel and an image. The mask is usually of the order of 1x1, 3x3, 5x5, 7x7 . A mask should always be in odd number, because other wise you cannot find the mid of the mask.

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. This is related to a form of mathematical convolution. It should be noted that the matrix operation being performed - convolution - is not traditional matrix multiplication, despite being similarly denoted by *.

Convolving mask over image is done in this way. We place the center of the mask at each element of an image, multiply the corresponding elements and then add them , and paste the result onto the element of the image on which you place the center of mask.

For example, if we have two three-by-three matrices, the first a kernel, and the second an image piece, convolution is the process of flipping both the rows and columns of the kernel and then multiplying locally similar entries and summing. The element at coordinates [2, 2] (that is, the central element) of the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel:

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right)[2,2] = (i{\cdot}1){+}(h{\cdot}2){+}(g{\cdot}3){+}(f{\cdot}4){+}(e{\cdot}5){+}(d{\cdot}6){+}(c{\cdot}7){+}(b{\cdot}8){+}(a{\cdot}9) \tag{2.13}$$

A mask is a filter. Concept of masking is also known as spatial filtering. The process of filtering is also known as convolving a mask with an image. As this process is same of convolution so filter masks are also known as convolution masks. The general process of filtering and applying masks is consists of moving the filter mask from point to point in an image. At each point (x,y) of the original image, the response of a filter is calculated by a pre defined relationship. All the filters values are pre defined and are a standard.

Generally there are two types of filters. One is called as linear filters or smoothing filters and others are called as frequency domain filters.Filters are applied on image for multiple purposes. The two most common uses are as following:

- Blurring and noise reduction
- Edge detection and sharpness

2.6.1 Blurring and Noise reduction

Filters are most commonly used for blurring and for noise reduction. Blurring is used in pre-processing steps, such as removal of small details from an image prior to large object extraction. In the process of blurring we reduce the edge content in an image and try to make the transitions between different pixel intensities as smooth as possible. Noise reduction is also possible with the help of blurring.

Blurring can be achieved by many ways. The common type of filters that are used to perform blurring are

- Mean filter
- Weighted average filter
- Gaussian filter

In this paper we focus on Gaussian blur since it is the method used in our project. Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination. Gaussian smoothing is also used as a pre-processing stage in computer vision algorithms in order to enhance image structures at different scales—see scale space representation and scale space implementation.

Mathematically, applying a Gaussian blur to an image is the same as convolving the image with a Gaussian function. This is also known as a two-dimensional Weierstrass transform. By contrast, convolving by a circle (i.e., a circular box blur) would more accurately reproduce the bokeh effect. Since the Fourier transform of a Gaussian is another Gaussian, applying a Gaussian blur has the effect of reducing the image's high-frequency components; a Gaussian blur is thus a low pass filter.

An example of blurring is shown in figure 2.12

2.6.2 Edge Detection and Sharpness

Masks or filters can also be used for edge detection in an image and to increase sharpness of an image. Edge detection includes a variety of mathematical methods that aim at identifying points in a digital image at which the image brightness changes sharply or, more formally, has discontinuities. The points at which image brightness changes sharply are typically organized into a set of curved line segments termed edges. The same problem of finding discontinuities in one-dimensional signals is known as step detection and the problem of finding signal discontinuities over time is known as change

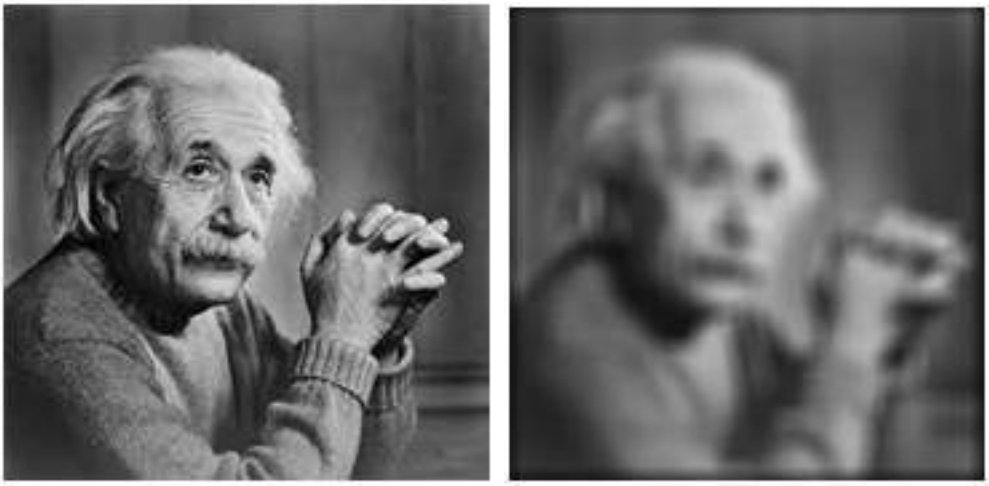

Figure 2.12: Image Blurring

detection. Edge detection is a fundamental tool in image processing, machine vision and computer vision, particularly in the areas of feature detection and feature extraction.

The purpose of detecting sharp changes in image brightness is to capture important events and changes in properties of the world. It can be shown that under rather general assumptions for an image formation model, discontinuities in image brightness are likely to correspond to:

- discontinuities in depth
- discontinuities in surface orientation
- changes in material properties
- variations in scene illumination

Applying an edge detection algorithm to an image may significantly reduce the amount of data to be processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image. If the edge detection step is successful, the subsequent task of interpreting the information contents in the original image may therefore be substantially simplified. However, it is not always possible to obtain such ideal edges from real life images of moderate complexity.

An example of edge detection is shown in figure 2.13

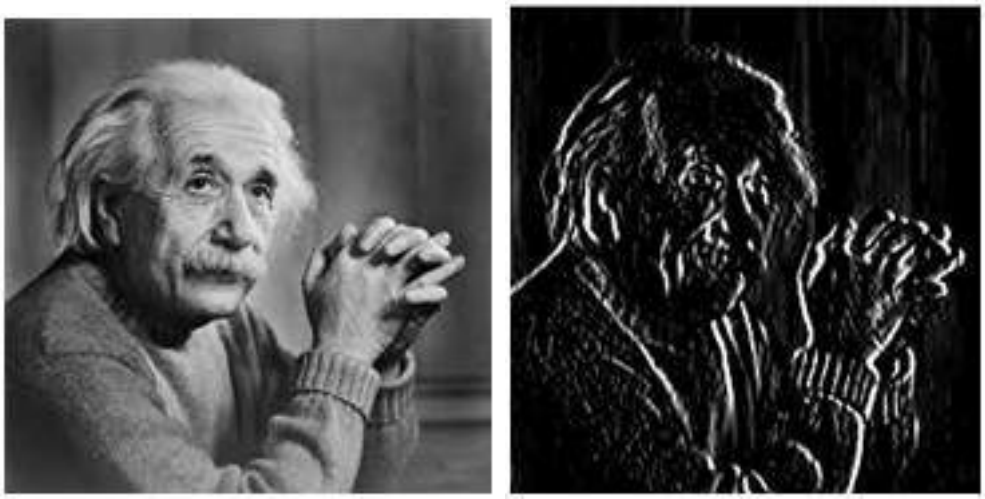

Figure 2.13: Edge Detection

Chapter 3

Neural Networks

In this chapter we will make an introduction to neural networks [16]. We will present different types of neural networks' architecture, the learning process and finally focus on deep learning [14], [15] and convolutional neural networks (CNNs), which is the type of network used for this project.

3.1 Introduction to Neural Networks

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons.

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include:

- Adaptive learning: An ability to learn how to do tasks based on the data given

for training or initial experience.

- Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
- Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
- Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements (neurons) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmic approach like

arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

3.2 Architecture of neural networks

3.2.1 Feed-forward networks

Feed-forward ANNs allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organization is also referred to as bottom-up or top-down. The structure of such a network can be shown in the figure 3.1.

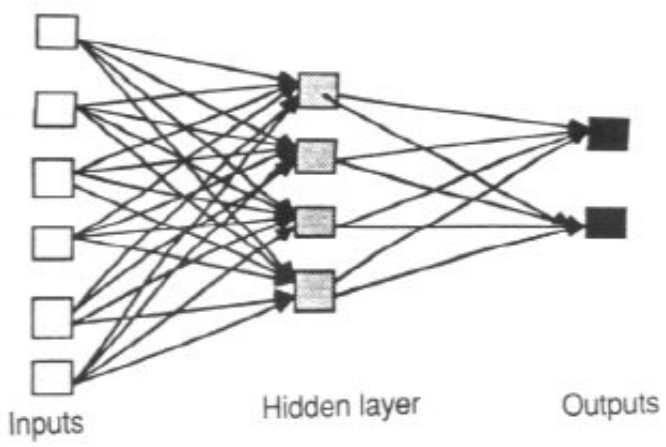

Figure 3.1: Feed-forward ANN

3.2.2 Feedback networks

Feedback networks can have signals traveling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes

and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organizations. The structure of such a network can be shown in the figure 3.2.

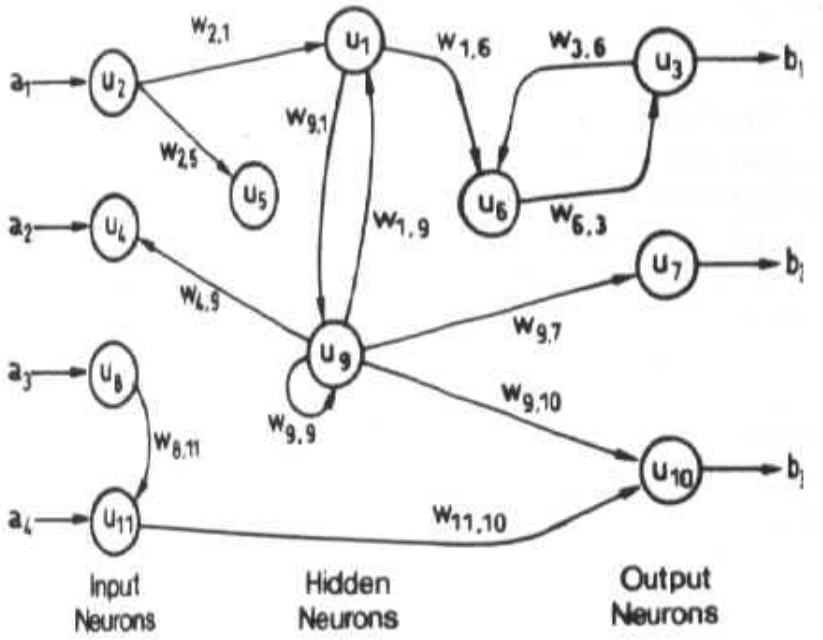

Figure 3.2: Feedback ANN

3.2.3 Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units: a layer of "input" units is connected to a layer of "hidden" units, which is connected to a layer of "output" units.

- The activity of the input units represents the raw information that is fed into the network.
- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behavior of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish single-layer and multi-layer architectures. The single-layer organization, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organizations. In multilayer networks, units are often numbered by layer, instead of following a global numbering.

3.3 Learning Process

The memorization of patterns and the subsequent response of the network can be categorized into two general paradigms:

Associative mapping in which the network learns to produce a particular pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associative mapping can generally be broken down into two mechanisms:

- auto-association: an input pattern is associated with itself and the states of input and output units coincide. This is used to provide pattern completion, i.e. to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network actually stores pairs of patterns building an association between two sets of patterns.

- hetero-association which is related to two recall mechanisms: nearest-neighbor recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and interpolative recall, where the output pattern is a similarity dependent interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, i.e. when there is a fixed set of categories into which the input patterns are to be classified.

Regularity detection in which units learn to respond to particular properties of the input patterns. Whereas in associative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network possesses knowledge which is contained in the values of the connections weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights (figure 3.3).

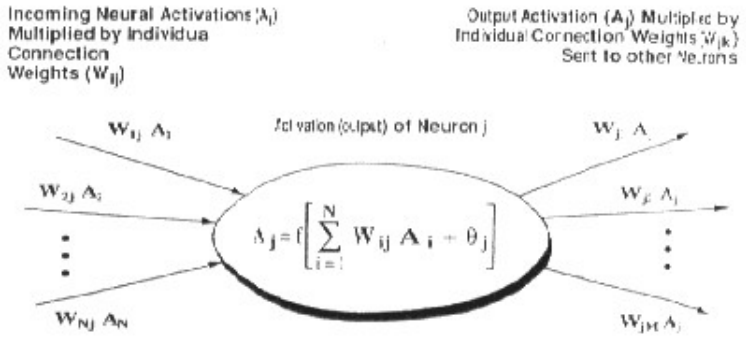

Figure 3.3: ANN's Weight Learning

Information is stored in the weight matrix W of a neural network. Learning is the determination of the weights. Following the way learning is performed, we can distinguish two major categories of neural networks:

- fixed networks in which the weights cannot be changed, i.e. dW/dt=0. In such networks, the weights are fixed in advance according to the problem to solve.
- Adaptive networks which are able to change their weights, i.e. dW/dt not= 0.

All learning methods used for adaptive neural networks can be classified into two major categories:

Supervised learning which incorporates an external teacher, so that each output unit is told what its desired response to input signals ought to be. During the learning process global information may be required. Paradigms of supervised learning include

error-correction learning, reinforcement learning and stochastic learning. An important issue concerning supervised learning is the problem of error convergence, i.e. the minimization of error between the desired and computed unit values. The aim is to determine a set of weights which minimizes the error. One well-known method, which is common to many learning paradigms is the least mean square (LMS) convergence.

Unsupervised learning uses no external teacher and is based upon only local information. It is also referred to as self-organization, in the sense that it self-organizes data presented to the network and detects their emergent collective properties. Paradigms of unsupervised learning are Hebbian learning and competitive learning.

A neural network learns off-line if the learning phase and the operation phase are distinct. A neural network learns on-line if it learns and operates at the same time. Usually, supervised learning is performed off-line, whereas unsupervised learning is performed on-line.

The behavior of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- **linear**, (or ramp) where the output activity is proportional to the total weighted output
- **threshold**, where the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.
- **sigmoid**, where the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurons than do linear or threshold units, but all three must be considered rough approximations

To make a neural network that performs some specific task, we must choose how the units are connected to one another and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

In order to train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and the actual

output is reduced. This process requires that the neural network compute the error derivative of the weights (EW). In other words, it must calculate how the error changes as each weight is increased or decreased slightly. The back propagation algorithm is the most widely used method for determining the EW.

3.4 Deep Learning and Convolutional Neural Networks (CNNs)

3.4.1 Deep Learning

To understand what deep learning is, we first need to understand the relationship deep learning has with machine learning, neural networks, and artificial intelligence. The best way to think of this relationship is to visualize them as concentric circles (figure 3.4):

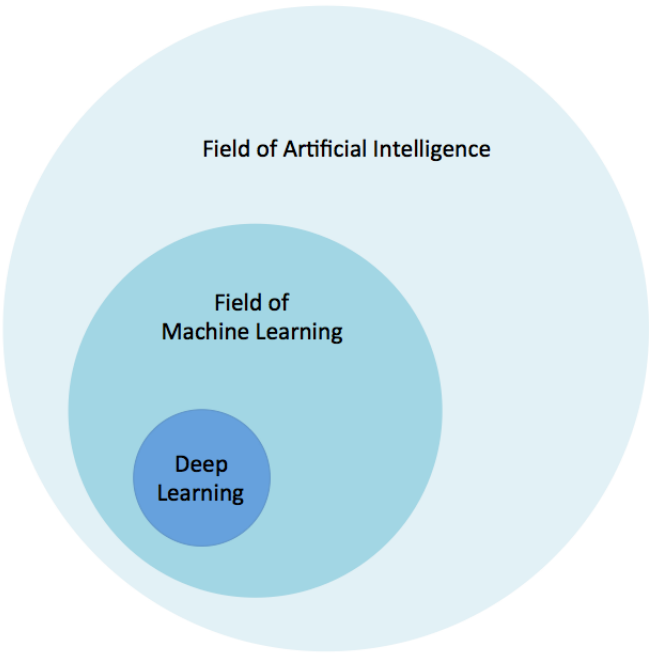

Figure 3.4: AI, Machine Learning and Deep Learning Relationship

Deep learning is a specific subset of Machine Learning, which is a specific subset of Artificial Intelligence. For individual definitions:

- Artificial Intelligence is the broad mandate of creating machines that can think

intelligently

- Machine Learning is one way of doing that, by using algorithms to glean insights from data

- Deep Learning is one way of doing that, using a specific algorithm called a Neural Network

Deep Learning is just a type of algorithm that seems to work really well for predicting things. Deep Learning and Neural Nets, for most purposes, are effectively synonymous. Computer vision is a great example of a task that Deep Learning has transformed into something realistic for business applications. Using Deep Learning to classify and label images is not only better than any other traditional algorithms but it is starting to be better than actual humans.

3.4.2 Convolution Neural Networks (CNNs)

Deep CNNs work by consecutively modeling small pieces of information and combining them deeper in network. One way to understand them is that the first layer will try to detect edges and form templates for edge detection. Then subsequent layers will try to combine them into simpler shapes and eventually into templates of different object positions, illumination, scales, etc. The final layers will match an input image with all the templates and the final prediction is like a weighted sum of all of them. So, deep CNNs are able to model complex variations and behavior giving highly accurate predictions.

A CNN typically consists of 3 types of layers:

- Convolution Layer

- Pooling Layer

- Fully Connected Layer

A CNN used to identify handwritten numbers is shown in figure 3.5.

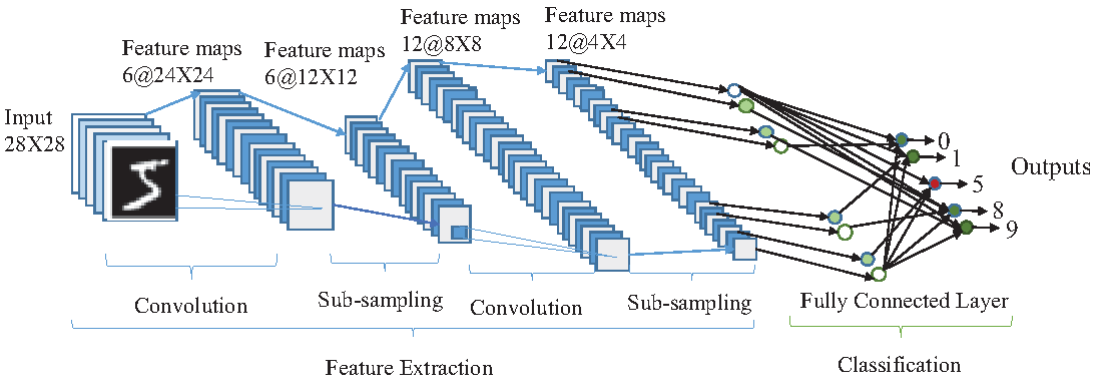

Figure 3.5: CNN Block Diagram

Convolutional Layer

Since convolution layers form the crux of the network, we will consider them first. Each layer can be visualized in the form of a block or a cuboid. In order to explain it better we will use the example of CIFAR-10 a data-set which has 60,000 images with 10 labels and 6,000 images of each type. Each image is colored and 32×32 in size.So for this example the input layer would have the following form (figure 3.6):

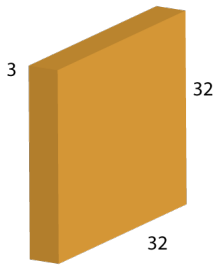

Figure 3.6: Input Layer

Here you can see, this is the original image which is 32×32 in height and width. The depth here is 3 which corresponds to the Red, Green and Blue colors, which form the basis of colored images. Now a convolution layer is formed by running a filter over it. A filter is another block or cuboid of smaller height and width but same depth which is swept over this base block. Lets consider a filter of size 5x5x3 (figure 3.7).

We start this filter from the top left corner and sweep it till the bottom left corner. This filter is nothing but a set of eights, i.e. 5x5x3=75 + 1 bias = 76 weights. At each

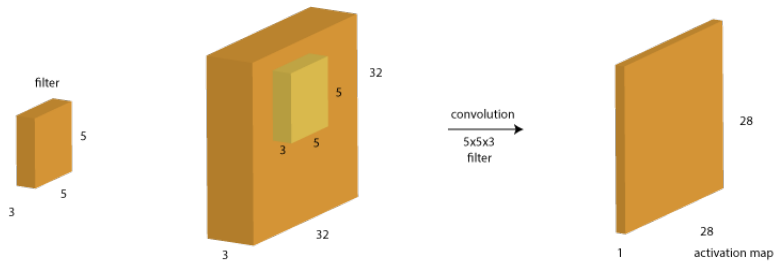

Figure 3.7: Convolution with a 5x5x3 Filter

position, the weighted sum of the pixels is calculated as WTX + b and a new value is obtained. A single filter will result in a volume of size 28x28x1 as shown above.

Multiple filters are generally run at each step. Therefore, if 10 filters are used, the output would look like figure 3.8.

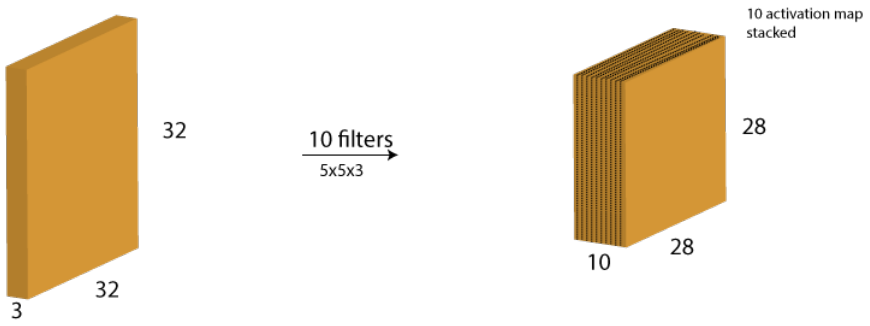

Figure 3.8: Convolution with Multiple 5x5x3 Filters

The filter weights are parameters which are learned during the back-propagation step. We notice that we got a 28×28 block as output when the input was 32×32. The following example will help us understand why.

Suppose the initial image had size 6x6x3 and the filter has size 3x3x3. Since depth is same it is also irrelevant to the outcome. A front view of how filter would work is shown in figure 3.9.

Here we can see that the result would be 4x4x1 volume block. We notice there is a single output for entire depth of the each location of filter.

A generic formula for the size of the output image will be:

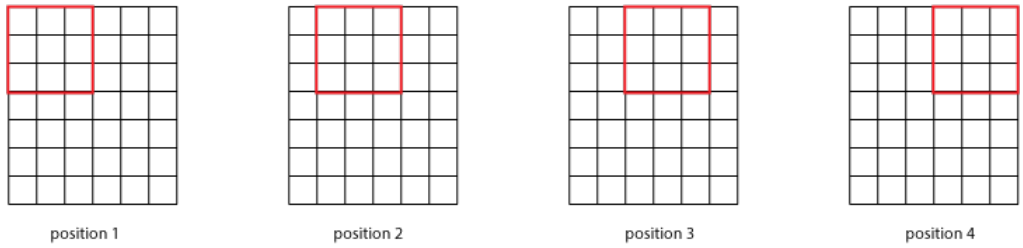

Figure 3.9: Convolution of an 6x6x3 Image with a 3x3x3 Filter

$$OutputSize = \frac{N-F}{S} + 1 \tag{3.1}$$

where:

- N: dimension of our original image
- F: dimension of our filter
- S: stride, the number of cells to move in each step

If we use this formula to our original example with N=32, F=5, S=1 we can see it validates.

We notice that the size of the images is getting shrunk consecutively. This will be undesirable in case of deep networks where the size would become very small too early. Also, it would restrict the use of large size filters as they would result in faster size reduction. To prevent this, we generally use a stride of 1 along with zero-padding of size (F-1)/2. Zero-padding is nothing but adding additional zero-value pixels towards the border of the image.

Pooling Layer

When we use padding in convolution layer, the image size remains same. So, pooling layers are used to reduce the size of image. They work by sampling in each layer using filters. Consider the following 4×4 layer. So if we use a 2×2 filter with a stride value of 2 and max-pooling, we get the response shown in figure 3.10.

Here we can see that 4 2×2 matrix are combined into 1 and their maximum value is taken. Generally, max-pooling is used but other options like average pooling can be considered.

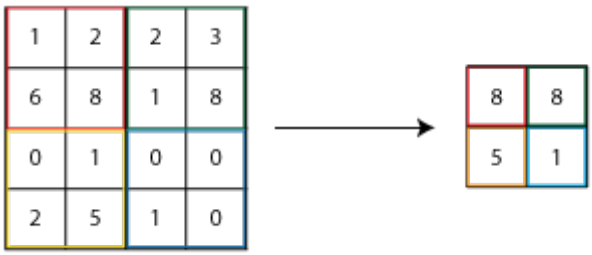

Figure 3.10: Convolution of an 4x4 Image with a 2x2 Filter and Max-pooling

Fully Connected Layer

At the end of convolution and pooling layers, networks generally use fully-connected layers in which each pixel is considered as a separate neuron just like a regular neural network. The last fully-connected layer will contain as many neurons as the number of classes to be predicted. For instance, in CIFAR-10 case, the last fully-connected layer will have 10 neurons.

3.4.3 Software and Frameworks

Many of the advances in practical applications of Deep Learning have been led by the widespread availability of robust open-source software packages. These let developers on-board easily and efficiently, which expands the number of people actively pushing development forward. As Data Science in general has been moving more towards Python lately, most of these packages are most developed for that language.

TensorFlow [10], [11] is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

Caffe aims to provide an easy and straightforward way for you to experiment with deep learning and leverage community contributions of new models and algorithms. You can bring your creations to scale using the power of GPUs in the cloud or to the masses on mobile with Caffe2's cross-platform libraries

Torch is a scientific computing framework with wide support for machine learning algorithms that puts GPUs first. It is easy to use and efficient, thanks to an easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation.

Theano is a Python library that lets you to define, optimize, and evaluate mathematical expressions, especially ones with multi-dimensional arrays (numpy.ndarray). Using Theano it is possible to attain speeds rivaling hand-crafted C implementations for problems involving large amounts of data. It can also surpass C on a CPU by many orders of magnitude by taking advantage of recent GPUs

ConvNetJS is a Javascript library for training Deep Learning models (Neural Networks) entirely in your browser. No specific software, no compilers, no installations, no GPUs requirements.

3.4.4 Keras

Keras [7] [12] is a high-level API, written in Python and capable of running on top of TensorFlow, Theano, etc. The above deep learning libraries are written in a general way with a lot of functionalities. This can be overwhelming for a beginner who has limited knowledge in deep learning. Keras provides a simple and modular API to create and train Neural Networks, hiding most of the complicated details under the hood.

Keras Work-flow

Keras [8], [9] provides a very simple work-flow for training and evaluating the models. Basically, we are creating the model and training it using the training data. Once the model is trained, we take the model to perform inference on test data. Keras' work-flow is described in figure 2.11:

Keras Layers

Layers can be thought of as the building blocks of a Neural Network. They process the input data and produce different outputs, depending on the type of layer, which are then used by the layers which are connected to them. Keras provides a number of core layers which include:

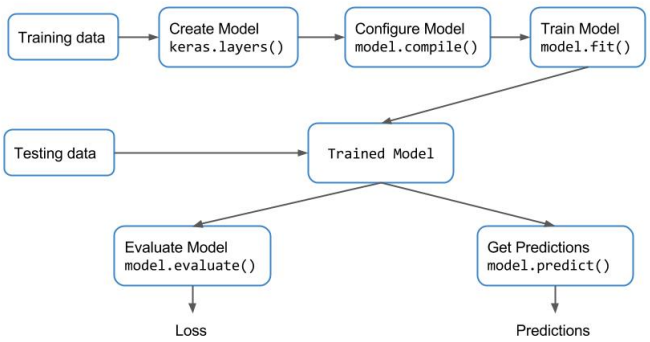

Figure 3.11: Keras' Work-flow

- Dense layers, also called fully connected layer, since, each node in the input is connected to every node in the output,
- Activation layer which includes activation functions like ReLU, tanh, sigmoid among others,
- Dropout layer – used for regularization during training,
- Flatten, Reshape, etc,
- Convolution layers – used for performing convolution,
- Pooling layers – used for down sampling,
- Recurrent layers,
- Locally-connected, normalization, etc.

Keras Models

Keras provides two ways to define a model:

- Sequential, used for stacking up layers – Most commonly used,
- Functional API, used for designing complex model architectures like models with multiple-outputs and shared layers

Training the Model

Once the model is ready, we need to configure the learning process. This means:

- Specify an Optimizer which determines how the network weights are updated
- Specify the type of cost function or loss function
- Specify the metrics we want to evaluate during training and testing
- Create the model graph using the back-end
- Any other advanced configuration

This is done in Keras using the model.compile() function.

Listing 3.1: Model Compile

```
model.compile(optimizer, loss, metrics)
```

Optimizers: Keras provides a lot of optimizers to choose from, which include

- Stochastic Gradient Descent (SGD),
- Adam,
- RMSprop,
- AdaGrad,
- AdaDelta, etc.

Loss functions: In a supervised learning problem, we have to find the error between the actual values and the predicted value. There can be different metrics which can be used to evaluate this error. This metric is often called loss function or cost function or objective function. There can be more than one loss function depending on what you are doing with the error. In general, we use

- binary-cross-entropy for a binary classification problem,
- categorical-cross-entropy for a multi-class classification problem,
- mean-squared-error for a regression problem and so on.

Training: Once the model is configured, we can start the training process. This can be done using the model.fit() function in Keras.

Listing 3.2: Initiation of Training Process

```
model.fit(trainFeatures, trainLabels, batch_size, epochs)
```

We just need to specify the training data, batch size and number of epochs. Keras automatically figures out how to pass the data iteratively to the optimizer for the number of epochs specified. The rest of the information was already given to the optimizer in the previous step.

Model evaluation: Once the model is trained, we need to check the accuracy on unseen test data. This can be done in two ways in Keras.

- model.evaluate() – It finds the loss and metrics specified in the model.compile() step. It takes both the test data and labels as input and gives a quantitative measure of the accuracy. It can also be used to perform cross-validation and further fine-tune the parameters to get the best model.
- model.predict() – It finds the output for the given test data. It is useful for checking the outputs qualitatively.

Chapter 4

Algorithm Design

In this chapter we design the algorithm that will let us detect any scuba divers or ROVs entering our surveillance area. We approach the problem in two different ways. During the time of the day where the brightness is higher (daytime mode) and when the brightness is lower or close to absolute dark (nighttime mode) which happens during the night or on cloudy weather.

A flow-chart that describes the algorithm is shown in figure 4.1.

To begin with, program has to decide in which mode it will run (daytime/nightime). In order to make this decision we first transform our frame to gray-scale and then calculate the average pixel value. If this value is lower than our threshold value then the program will run in nighttime mode while if it is higher then it will run on daytime mode. The threshold value is custom and requires measurements from the specific area.

The code that responsible for that is the following:

Listing 4.1: BGR to Grayscale Conversion and Calculation of Average Pixel Value

```
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
average = np.mean(gray)
```

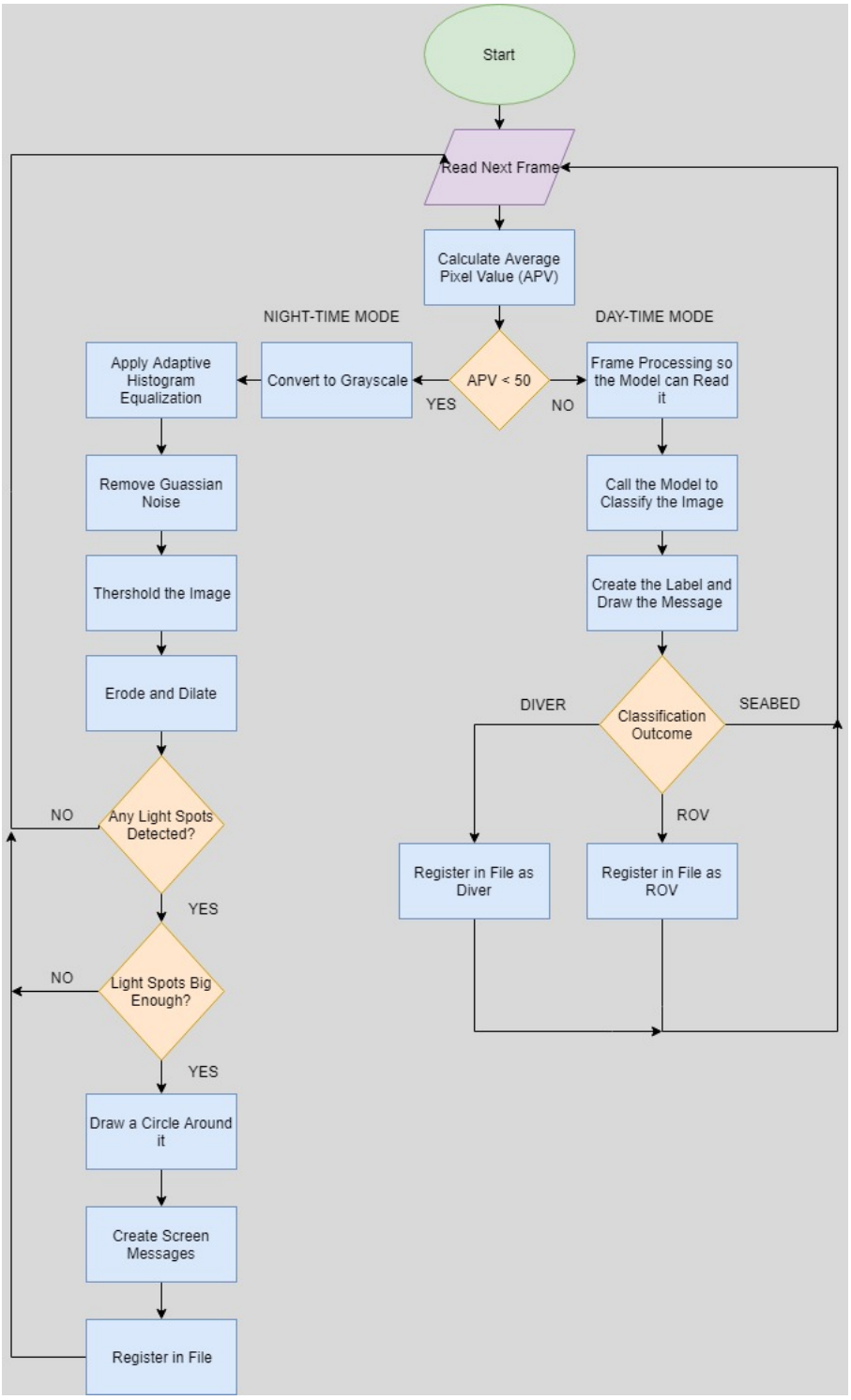

Figure 4.1: Flow-chart

4.1 Nighttime Mode

During times of low brightness we assume that the diver or the ROV will have some kind of light source in order to move around and detect their target. The light source can be something from a typical flashlight to a big searchlight on a boat. In this case using image processing we try to detect in our frame a light spot which can be the actual light source or just a reflection of light if the light source is located outside our surveillance area. In order to be able to detect the light source we first need to do a series of processes to our image.

First we apply adaptive histogram equalization and Gaussian blur to make our frame more suitable for processing and also remove the Gaussian noise. The lines of code responsible for this are the following:

Listing 4.2: Histogram Equalization and Gaussian Blur

```
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
gray = clahe.apply(gray)
blurred = cv2.GaussianBlur(gray, (11, 11), 0)
```

Now that our image is ready for processing we detect the light regions by applying binary thresholding.

Listing 4.3: Thresholding

```
custom_thresh, average, maximum = custom_threshold(gray)
thresh = cv2.threshold(blurred, custom_thresh, 255,
                       cv2.THRESH_BINARY )[1]
```

If the value of a pixel is greater than a fixed value (thresh) then it is replaced with 255 (white), while in the opposite case it is replaced with 0 (black).

There are different types of thresholding like binary, truncy, and tozero and also their inversions (figure 4.2).

As we can see from the code we use the binary thresholding. Light regions are shown in white while everything else is black. We notice from the lines of code above that instead of a fixed threshold value we use *customthresh* which is a function that calculates the threshold for each frame separately. The function looks like this:

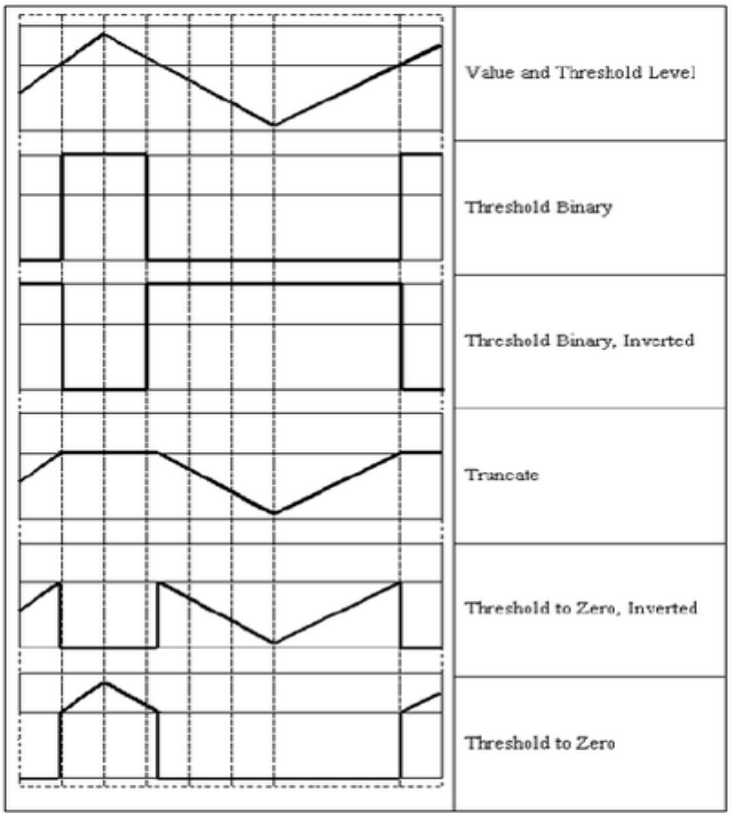

Figure 4.2: Types of Thresholding

Listing 4.4: Custom Thresholding Function

```
def custom_threshold(image):

        # compute the max pixel value of the image
        maximum = np.max(image)

        # compute the threshold
        if average < 20:
            thresh = average*0.3 + maximum*0.7
        elif  (average >= 20) and (average < 30):
            thresh = average*0.2 + maximum*0.8
        elif  (average >= 30) and (average < 40):
            thresh = average*0.1 + maximum*0.9
```

```
        else :
            thresh = average*0.03 + maximum*0.95

        # return the image threshold
        return thresh, maximum
```

The values in this function are chosen after testing and should be different for different surveillance areas.

Although we detected some light regions we are not done yet. We perform a series of erosions and dilations to remove any small blobs of noise from the thresholded image. Changing iterations value makes our detection more or less sensitive.

Listing 4.5: Erode and Dilate

```
thresh = cv2.erode(thresh, None, iterations=2)
thresh = cv2.dilate(thresh, None, iterations=8)
```

Erode consists of convoluting an image with some kernel, which is usually a 3x3, 5x5, 7x7, etc matrix. A pixel of the image (0 or 1) will become 1 only if all pixels under kernel are also 1 else it will become 0. This results to all the pixels on the borders of the image to become 0. This method is used in order to remove small white noises or separate objects.

Dilate is the opposite process of erode. We again perform convolution with our image and a kernel but in this case a pixel of the image will become 1 if atleast one of the pixels under the kernel is 1. It is usually used after erode because erode while it removes the noise it also shrinks our object. We use dilate to restore our object or it can be used to link separated objects.

Afterwards, we perform a connected component analysis on the thresholded image and initialize a mask to store only the sufficiently large components.

Listing 4.6: Thresholded Image Analysis

```
labels = measure.label(thresh, neighbors=8, background=0)
```

```
mask = np.zeros(thresh.shape, dtype="uint8")
# loop over the unique components
  for label in np.unique(labels):
    # if this is the background label, ignore it
        if label == 0:
          continue

        # otherwise, construct the label mask and count the
        # number of pixels
        labelMask = np.zeros(thresh.shape, dtype="uint8")
        labelMask[labels == label] = 255
        numPixels = cv2.countNonZero(labelMask)

        # if the number of pixels in the component is
        # sufficiently large, then add it to our mask
        # of "large blobs" and flag it as alert mode
        if numPixels > 200:
          mask = cv2.add(mask, labelMask)
          # setting alert mode on
          alert = True
```

For testing and demonstrating purposes we also mark the bright spots on the original image

Listing 4.7: White Spots Marking

```
# find the contours in the mask
  cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
  cv2.CHAIN_APPROX_SIMPLE)
  cnts = cnts[0] if imutils.is_cv2() else cnts[1]

  # loop over the contours
  for (i, c) in enumerate(cnts):
    # draw the bright spot on the image
```

```
        (x, y, w, h) = cv2.boundingRect(c)
        ((cX, cY), radius) = cv2.minEnclosingCircle(c)
        cv2.circle(frame, (int(cX), int(cY)), int(radius),
                (0, 0, 255), 3)
```

Finally, we create some screen messages and an output file where we store the time and type of alerts.

Listing 4.8: Messages' Creation

```
#get current date and time
date = datetime.now().strftime('%d-%m-%Y %H:%M:%S')

#checking if light is detected
if alert == True:
  #show values and message on screen
  print 'Light detected!'
  frame = mesage_on_screen(frame, alert, av_str, max_str,
        thresh_str, date)

  # create output text string
  date_text = 'Time of event: ' + date
  text = "Possibly diver or ROV detected! " + date_text

  #append the output string in a txt file
  text_file = open("log.txt", "a")
  text_file.write(text + "\n")
  text_file.close()

else :
  #show values and message on screen
  print 'No light detected'
  text, frame = message_on_screen(frame, alert, av_str,
                                  max_str, thresh_str, date)
```

where the *message_on_screen* function is:

Listing 4.9: The Message Function

```
def message_on_screen(image, alert, av_str, max_str,
        thresh_str, date):

  #create text to be shown on screen
  font = cv2.FONT_HERSHEY_SIMPLEX
  average_pixel_value = 'Average pixel value: ' + av_str
  maximum_pixel_value = 'Maximum pixel value: ' + max_str
  threshold_value = 'Threshold: ' + thresh_str
  date_text = 'Time of event: ' + date
  cv2.putText(image, average_pixel_value, (10,50), font, 0.3,
      (255,255,255),1,cv2.LINE_AA, False)
  cv2.putText(image, maximum_pixel_value, (10,60), font, 0.3,
      (255,255,255),1,cv2.LINE_AA, False)
  cv2.putText(image, threshold_value, (10,70), font, 0.3,
      (255,255,255),1,cv2.LINE_AA, False)
  cv2.putText(image, date_text, (10,80), font, 0.3,
      (255,255,255),1,cv2.LINE_AA, False)

  if alert == True :
      light_detection = 'Light detected!!!'
      cv2.putText(image, light_detection, (200,30), font,
          0.8, (0,0,255),2,cv2.LINE_AA, False)

  else :
      light_detection = 'No light detected'
      cv2.putText(image, light_detection, (200,30), font,
          0.8, (0,255,0),2,cv2.LINE_AA, False)

return image
```

A record in the log file looks like this:

$Possibly diver or ROV detected! Time of event : 28-06-201812:01:53$

The following two figures (4.3 and 4.4) show how our program responds during the night when we give it as input two photos showing divers and a ROV.

Figure 4.3: Light Detection in Image with Divers

More information about what we see will follow in the result chapter.

Figure 4.4: Light detection in Image with ROV

4.2 Daytime Mode

During the daytime mode we use a pre-trained neural network, which we trained ourselves from a dataset of 2500 images, to detect if a diver or a ROV has entered our surveillance area. What our program actually does is classify each frame from the camera to one of the following three categories.

- Diver
- ROV
- Seabed

If the frame is classified as Diver or ROV the alert mode is triggered while if it is classified as Seabed the program keeps running without any actions taken.

4.2.1 Model Training

Model Architecture

The model is trained from a data-set of a total of 2225 images labeled as one of the three categories. The architecture of the model we chose for this purpose is quite simple.

Listing 4.10: Model Architecture

```
def cnn_model(width, height, depth, classes):

  model = Sequential()
  inputshape = (height, width, depth)

  # The first two layers with 32 filters of window size 3x3
  model.add(Conv2D(32, (3, 3), padding='same',
                   activation='relu', input_shape=inputshape))
  model.add(Conv2D(32, (3, 3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))

  # Next two layers with 64 filters of window size 3x3
  model.add(Conv2D(64, (3, 3), padding='same',
                           activation='relu'))
  model.add(Conv2D(64, (3, 3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))

  # Next two layers with 64 filters of window size 3x3
  model.add(Conv2D(64, (3, 3), padding='same',
                           activation='relu'))
  model.add(Conv2D(64, (3, 3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))

  # Dense layer which performs the classification
  # using softmax layer
  model.add(Flatten())
  model.add(Dense(512, activation='relu'))
  model.add(Dropout(0.5))
```

```
model.add(Dense(classes, activation='softmax'))

return model
```

We notice that we have added dropout layers and a softmax layer as the final layer to our model.

Dropout is a regularization technique for reducing over-fitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks.

The softmax layer is often used as the final layer of a neural network-based classifier. Such networks are commonly trained under a log loss (or cross-entropy) regime, giving a non-linear variant of multi-nomial logistic regression.. In mathematics, the softmax function, or normalized exponential function, is a generalization of the logistic function that "squashes" a K-dimensional vector $\mathbf{z}$ of arbitrary real values to a K-dimensional vector $\sigma(\mathbf{z})$ of real values, where each entry is in the range (0, 1), and all the entries adds up to 1. The function is given by

$$\sigma : \mathbb{R}^K \to \left\{ z \in \mathbb{R}^K | z_i > 0, \sum_{i=1}^{K} z_i = 1 \right\} \tag{4.1}$$

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \ j = 1, \ldots, K. \tag{4.2}$$

In probability theory, the output of the softmax function can be used to represent a categorical distribution – that is, a probability distribution over K different possible outcomes. In fact, it is the gradient-log-normalizer of the categorical probability distribution. The softmax function is also the gradient of the LogSumExp (LSE) function.

Training Procedure

Now that we have set up out model's architecture we continue to the training procedure. First we need to set some arguments like the data-set that we will use for our model's training, the name of the model we will create and a plot that contains useful information about our model's training like the accuracy and loss.

Listing 4.11: Arguments

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
        help="path to input dataset")
ap.add_argument("-m", "--model", required=True,
        help="path to output model")
ap.add_argument("-p", "--plot", type=str, default="plot.png",
        help="path to output loss/accuracy plot")
args = vars(ap.parse_args())
```

We also set the number of epochs to train for, the initial learning rate (we do not use a stable learning rate) and the batch size.

Listing 4.12: Epochs Initial Learning Rate and Batch Size

```
# initialize the number of epochs to train for, initial
# learning rate and batch size
EPOCHS = 100
INIT_LR = 0.001
BS = 42
```

The number of epochs is the number of times our data-set is passed forward and backward through our neural network. We are using a limited data-set and to optimize the learning and the graph we are using Gradient Descent which is an iterative process. So, updating the weights with single pass or one epoch is not enough. As the number of epochs increases, more number of times the weight are changed in the neural network and the curve goes from under-fitting to optimal to over-fitting curve (figure 4.5).

Since passing the entire data-set at the same time through our model can be very time consuming we divide it in smaller parts called batches. Batch size is the number of images that pass through our network simultaneously.

Finally, the learning rate as its name suggests is an indicator of how fast our network learns. We use a decreasing learning rate because the later stage of the training the slower the network can learn.

Then we load the data-set, grab the image paths and shuffle them randomly.

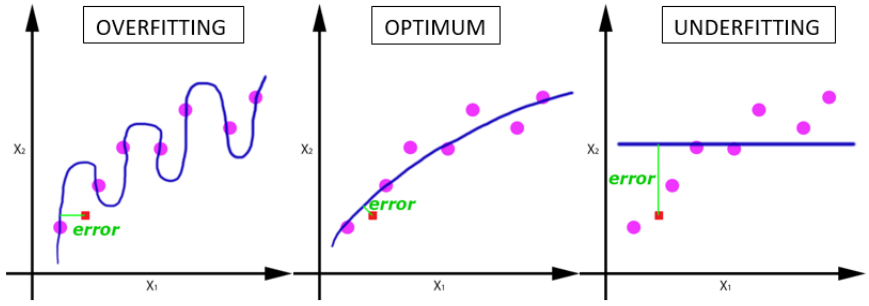

Figure 4.5: Over-fitting and Under-fitting

Listing 4.13: Data-set Loading and Shuffling

```
# initialize the data and labels
print("[INFO] loading images...")
data = []
labels = []

# grab the image paths and randomly shuffle them
imagePaths = sorted(list(paths.list_images(args["dataset"])))
random.seed(42)
random.shuffle(imagePaths)
```

Notice that since we shuffle the data-set, re-training the model with the exact same parameters and data-set will result in a different model.

Now we proceed to the image reading and after some processing (re-sizing and conversion to array) we extract the labels and update the label list.

Listing 4.14: Image Reading and Labeling

```
for imagePath in imagePaths:

        # load the image, pre-process it, and store it in the
        # data list
        image = cv2.imread(imagePath)
        image = cv2.resize(image, (32, 32))
        image = img_to_array(image)
```

```
        data.append(image)

        # extract the class label from the image path and
        # update the labels list
        label = imagePath.split(os.path.sep)[-2]
        if label == "divers":
                label = 1
        if label == "rov":
                label = 2
        if label == "seabed":
                label = 0
        labels.append(label)
```

We scale the pixels to values from 0 to 1 then we split the data into two categories (training and testing. We use 75 per cent of the data as training data and the remaining 25 per cent for testing. Finally we convert the labels from integers to vectors.

Listing 4.15: Training and Testing Data

```
# scale the raw pixel intensities to the range [0, 1]
data = np.array(data, dtype="float64") / 255.0
labels = np.array(labels)

# partition the data into training and testing splits using 75%
# of the data for training and the remaining 25% for testing
(trainX, testX, trainY, testY) = train_test_split(data,
        labels, test_size=0.25, random_state=42)

# convert the labels from integers to vectors
trainY = to_categorical(trainY, num_classes=3)
testY = to_categorical(testY, num_classes=3)
```

Before we continue with the initiation of the training procedure we use a data augmentation technique to expand our data-set. In spite of all the data availability,

fetching the right type of data which matches the exact use-case of our experiment is a daunting task. Moreover, the data has to have good diversity as the object of interest needs to be present in varying sizes, lighting conditions and poses if we desire that our network generalizes well during the testing (or deployment) phase. To overcome this problem of limited quantity and limited diversity of data, we generate(manufacture) our own data with the existing data which we have. This methodology of generating our own data is known as data augmentation.

Listing 4.16: Data Augmentation

```
# construct the image generator for data augmentation
aug = ImageDataGenerator(rotation_range=30,
        width_shift_range=0.1, height_shift_range=0.1,
        shear_range=0.2, zoom_range=0.2, horizontal_flip=True,
        vertical_flip=True, fill_mode="nearest")
```

As we can see, in order to achieve data augmentation we use rotation, scaling, zooming and flipping.

Now that everything is set we continue to our model training.

Listing 4.17: Model Training

```
# initialize the model
print("[INFO] compiling model...")
model = cnn_model(width=32, height=32, depth=3, classes=3)
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
model.compile(loss="categorical_crossentropy", optimizer=opt,
        metrics=["accuracy"])

# train the network
print("[INFO] training network...")
H = model.fit_generator(aug.flow(trainX, trainY,
        batch_size=BS), validation_data=(testX, testY),
        steps_per_epoch=len(trainX) // BS,
        epochs=EPOCHS, verbose=1)
```

```
# save the model to disk
print("[INFO] serializing network...")
model.save(args["model"])
```

We notice that as an optimizer we chose the Adam [13] optimization algorithm. The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.

Adam is different to classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. A learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

Adam combines the advantages of two other extensions of stochastic gradient descent. Specifically:

Adaptive Gradient Algorithm (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).

Adaptive Root Mean Square Propagation (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance).

Here is a graph that compares Adam to other optimization algorithms taken from *"Adam: A Method for Stochastic Optimization, 2015."*

We also notice the decay parameter which is the current learning rate for each epoch and as we see it gets lower after every epoch.

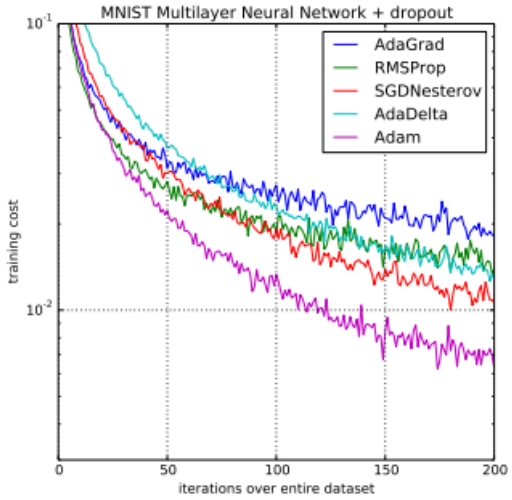

Figure 4.6: Comparison of Optimization Algorithms

Finally, we create a plot to observe our model's training and accuracy loss through the entire training.

Listing 4.18: Generating Training Process Plot

```
# plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure()
N = EPOCHS
plt.plot(np.arange(0, N), H.history["loss"],
        label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"],
        label="val_loss")
plt.plot(np.arange(0, N), H.history["acc"],
        label="train_acc")
plt.plot(np.arange(0, N), H.history["val_acc"],
        label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
```

```
plt.savefig(args["plot"])
```

The plot in figure 4.7 is the actual plot of our model and it will be further explained in the results chapter.

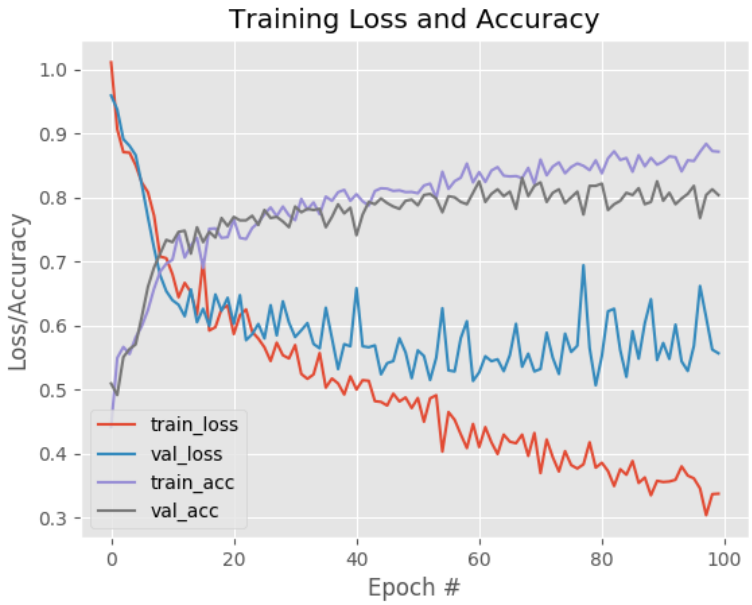

Figure 4.7: Result Plot

4.2.2 Run-time

With our model trained and ready for use we return to our main program. In order to chose the model we want for our project we pass its path as an argument.

Listing 4.19: Arguments

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
        help="path to trained model model")
args = vars(ap.parse_args())
```

For each frame read we perform some image processing to adjust our image to our model's requirements so it can process it.

Listing 4.20: Image Processing

```
# pre-process the image for classification
frame = cv2.resize(frame, (32, 32))
frame = frame.astype("float") / 255.0
frame = img_to_array(frame)
frame = np.expand_dims(frame, axis=0)
```

Then we call our model to perform the classification.

Listing 4.21: Perform Classification

```
# classify the input image
(seabed, divers, rov) = model.predict(frame)[0]
```

We create the labels and draw them on our image. This is done mostly for testing and demonstration purposes and it is not really needed.

Listing 4.22: Label Drawing

```
# build the label
label1 = "{}: {:.2f}%".format("Divers", divers * 100)
label2 = "{}: {:.2f}%".format("Rov", rov * 100)
label3 = "{}: {:.2f}%".format("Seabed", seabed * 100)

# draw the label on the image
output = imutils.resize(orig, width=400)
cv2.putText(output, label1, (10, 25),
        cv2.FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1)
cv2.putText(output, label2, (10, 45),
        cv2.FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1)
cv2.putText(output, label3, (10, 65),
        cv2.FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1)
```

Finally, we set the alert mode on or off depending on whether it has detected a diver, or a ROV, or nothing. Also in case of an alert we make a registration to our log file.

Listing 4.23: Setting Alert Mode

```
# set alert mode on/off, define alert type and append message
# to log.txt file
if divers > rov and divers > seabed:
  alert = True
  type_of_alert = "Diver"
  date_text = 'Time of event: ' + date
  text = "Diver detected! " + date_text
  message = "Diver detected!"
  cv2.putText(output, message, (100,20),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.8,(0,0,255),2,cv2.LINE_AA, False)

  #append the output string in a txt file
  text_file = open("log.txt", "a")
  text_file.write(text + "\n")
  text_file.close()

elif rov > divers and rov > seabed:
  alert = True
  type_of_alert = "ROV"
  date_text = 'Time of event: ' + date
  text = "ROV detected! " + date_text
  message = "ROV detected!"
  cv2.putText(output, message, (100,20),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.8,(0,0,255),2,cv2.LINE_AA, False)

  #append the output string in a txt file
  text_file = open("log.txt", "a")
  text_file.write(text + "\n")
  text_file.close()
```

```
else :
  alert = False
  type_of_alert = None
  message = "Seabed"
  cv2.putText(output, message, (150,20),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.8,(0,255,0),2,cv2.LINE_AA, False)
```

The following three figures (4.8, 4.9 and 4.10) show how our program responds when we give it as input three different photos, one from each category.

Figure 4.8: Diver Detection

More information about what we see will follow in the result chapter.

Figure 4.9: ROV Detection

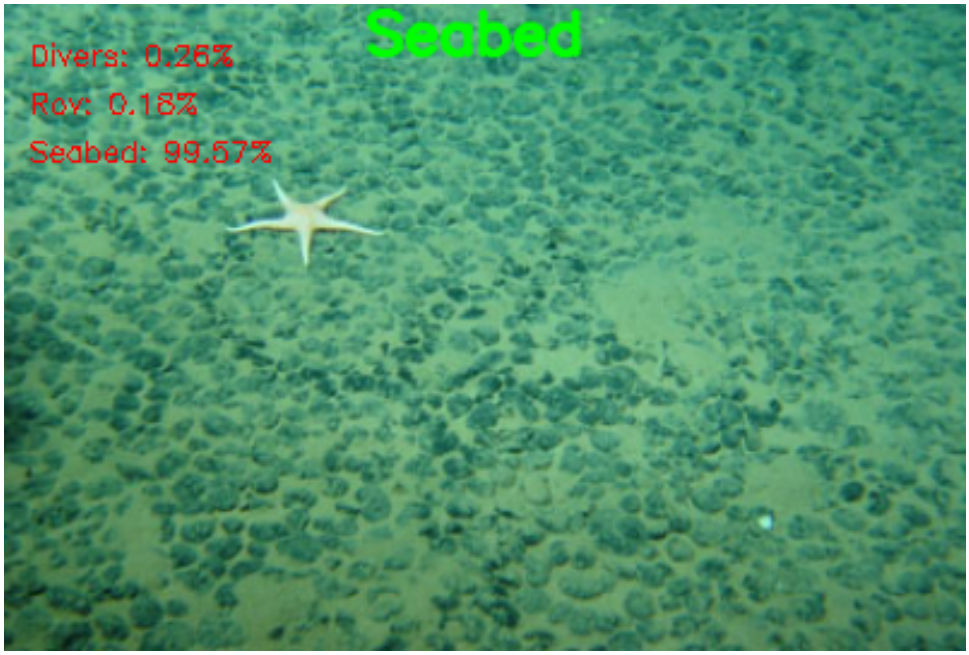

Figure 4.10: Seabed

Chapter 5

Testing Results

In this chapter we will present the testing results of our program as well as the evaluation of the training process of our network. It includes situations with high brightness where our CNN is processing the data and classifies each frame and situations with low or even minimum brightness where our nigh-time mode is on.

5.1 Training Process Evaluation

We begin with our model's evaluation chart (Figure 5.1). We notice 2 different accuracy values and 2 different loss values. As already mentioned in chapter 3, the data-set is divided in training data (75% of the data-set) and testing data (25% of the data-set). Train accuracy and train loss are calculated on the go, during training. These values show how well our network is performing on the data it is being trained. Training accuracy usually keeps increasing throughout training. On the other hand, validation accuracy and validation loss are calculated during the testing process. These numbers show us how good our model is at predicting outputs for inputs it has never seen before.

We notice that both training accuracy and validation accuracy start really low on first epoch, but as the number of epochs increases and our network learns more features, they increase up to 80% for the validation accuracy and up to 88% for the training accuracy. Until both parameters increase to 80% their values are very close to each other, but when validation accuracy reaches 80% it stops increasing. It starts fluctuating around this value. This is a result mainly caused by over-fitting. The model "memorizes" the training examples and becomes ineffective for the test set. As

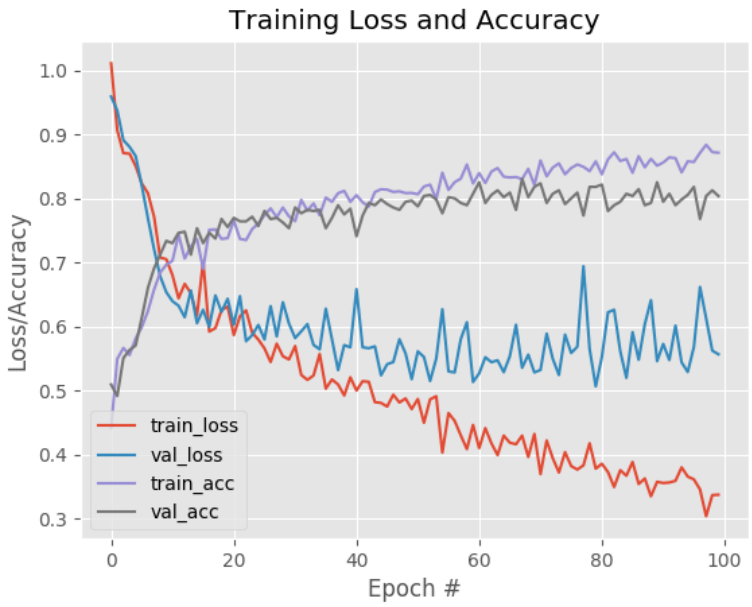

Figure 5.1: Result Plot

our data-set is limited, our model can only learn that much and become that good at predicting. In order to increase the validation accuracy we need a bigger data-set.

On the contrary to accuracy, training and validation losses are not percentage. It is a summation of the errors made for each example in training or validation sets. The loss is usually negative log-likelihood and residual sum of squares for classification and regression respectively. Then naturally, the main objective in a learning model is to reduce (minimize) the loss function's value with respect to the model's parameters by changing the weight vector values through different optimization methods. Loss value implies how well or poorly a certain model behaves after each iteration of optimization. Ideally, we would expect the reduction of loss after each, or several, iteration(s).

Like the accuracy, training loss keeps improving as the number of epochs increases while validation loss initially decreases to a certain point and then starts fluctuating around it.

5.2 Nighttime Mode Results

In situations with low brightness our program focuses on detecting light sources or light reflections within the surveillance area. In the figures 5.2 and 5.3 we present our program's performance on 2 situations with low brightness.

Figure 5.2: Light Detection in Image with Divers

We notice it detects quite well the light sources in both situations and also in the ROV photo it also detects the light reflection from an outer light source on the ROV's arm.

In figures 5.4 and 5.5 we present also the case of minimum brightness when in almost absolute dark suddenly a light appears.

Apart from the detection message we also notice some values on screen like average pixel value, maximum pixel value and threshold value. Those values help us understand how our program performs in different brightness and improve our program through testing.

Finally a registration is made in a .txt file showing the time-stamp if the event. The registration file looks like figure 5.6.

Figure 5.3: Light detection in Image with ROV

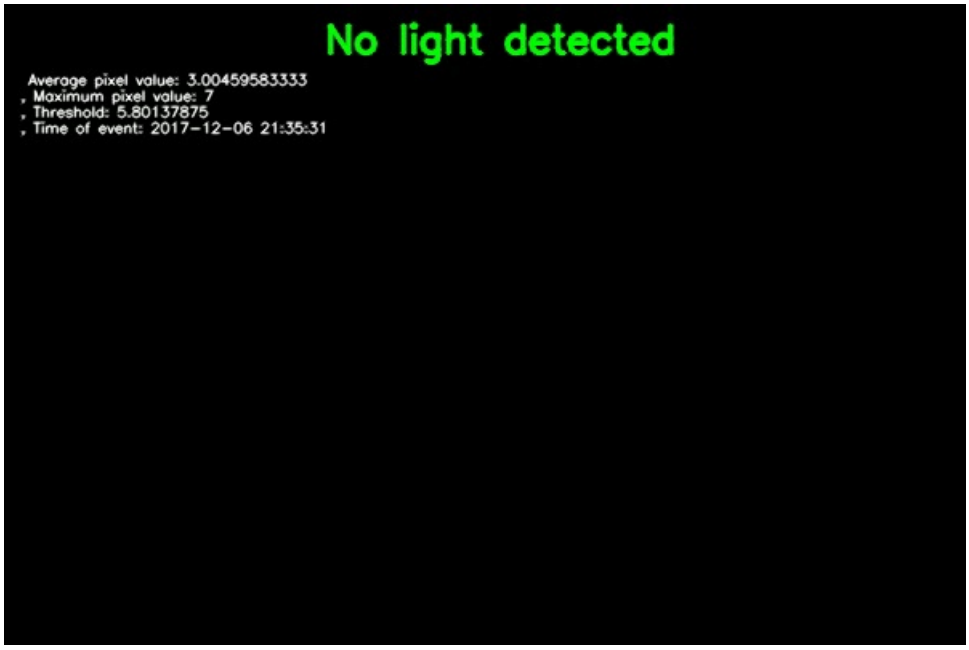

Figure 5.4: No light detection in very dark environment

Figure 5.5: Light detection in very dark environment

```
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:53
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:53
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:53
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
Possibly diver or ROV detected! Time of event: 28-06-2018 12:01:54
```

Figure 5.6: Light detection Record

5.3 Daytime Mode Results

On this mode our program classifies each image to one of the 3 categories (diver, ROV or seabed) and shows a message accordingly.

In figure 5.7, 5.8 and 5.9 we present the case in which a diver is detected, followed by 5.10, 5.11 and 5.12 where a ROV is detected and finally 5.13, 5.14 and 5.15 where it is just the seabed.

Figure 5.7: Light detection in Image with diver (1)

We notice, in all 3 cases of divers, the probability value for the diver label is over 99%. This means our model achieves high level of certainty for those 3 images

In the case of the 3 ROVs is not that high and it scales in every image. The values of 78.13%, 67,08% and 83.84% are still high enough to classify correctly the images but our model is not as "certain" as it was in the divers' example.

In the ROV results we notice similar results to the diver images. The first image is classified as ROV with a probability close to 100% (99.57%). In the other 2 examples the probability is also quite high and surpasses the value of 93% which is a very satisfying result.

Except for the message that let us know in which category our model classified each image we can also see 3 other values. Those 3 percentages show what probability our model gives to each category for the certain image it processed. The category with which it labels the image is obviously the one with the highest percentage.

As already mentioned our model reaches about 80% accuracy. The next 3 figures demonstrate cases where our model prediction is wrong. Figure 5.16 shows the case

Figure 5.8: Light detection in Image with diver (2)

Figure 5.9: Light detection in Image with diver (3)

Figure 5.10: Light detection in Image with ROV (1)

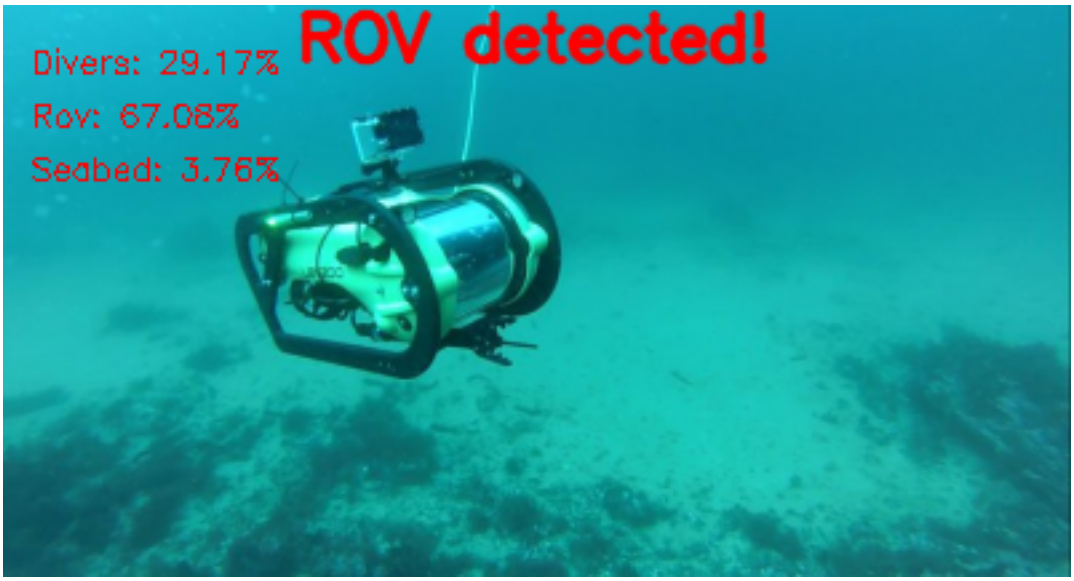

Figure 5.11: Light detection in Image with ROV (2)

where a diver is classified as seabed, figure 5.17 shows a ROV classified as a diver and figure 5.18 shows a seabed image classified as ROV.

In figure 5.16 the diver is well hidden behind a coral and some rocks and he also

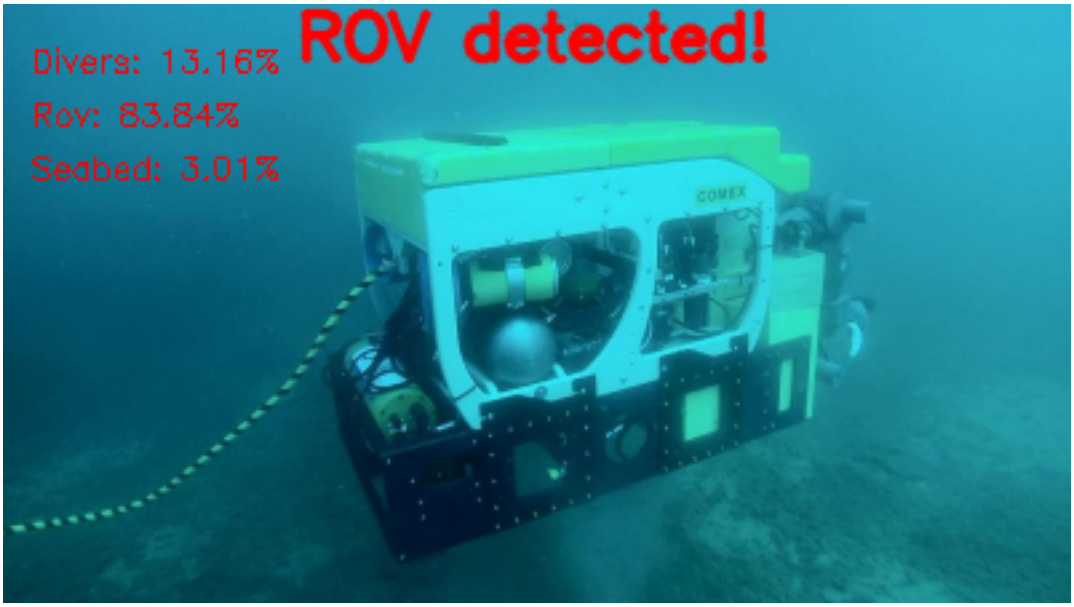

Figure 5.12: Light detection in Image with ROV (3)

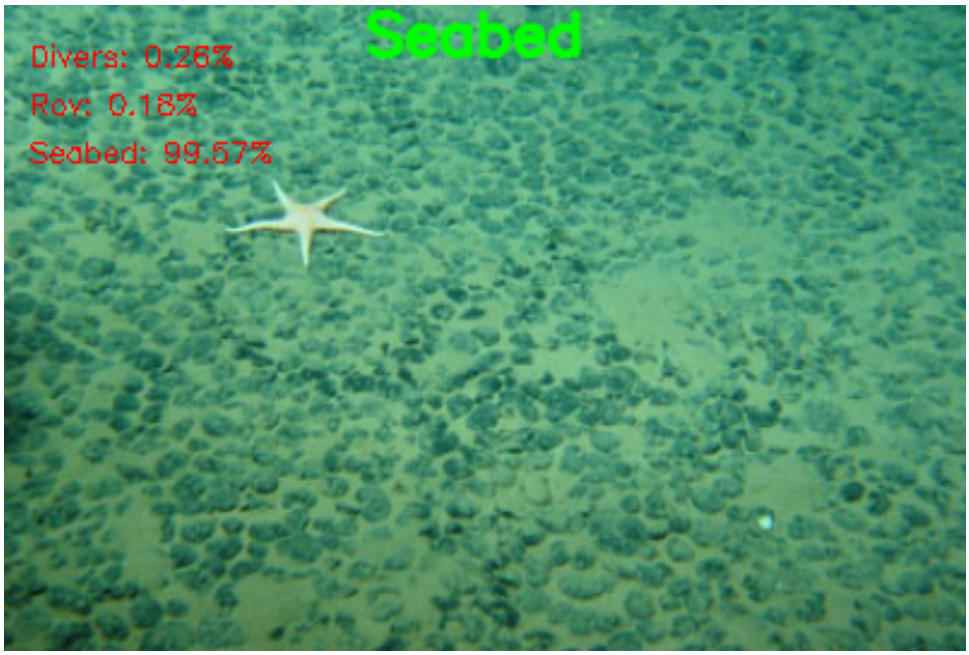

Figure 5.13: Light detection in Image with seabed (1)

holds a camera. This results to a false classification by our model. the image is classified as seabed even though the probability given for this case is 44.75% even lower than 50%. The probability for diver classification is only 7 degrees lower which means our

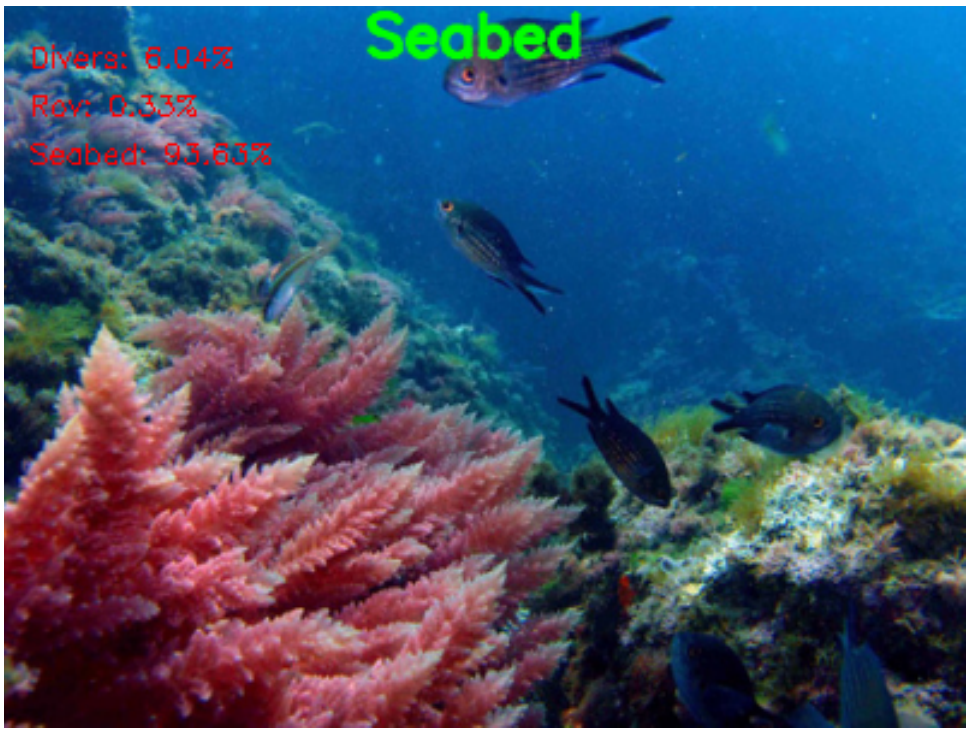

Figure 5.14: Light detection in Image with seabed (2)

model still detects some diver features in the image. Finally, even the ROV probability has a noticeable value which is probably caused by the camera which resembles in a way the ROV.

In figure 5.17 our models' prediction is very far from the truth. This probably happens because of the uncommon shape and structure of this specific vehicle. Most of the ROVs in our data-set look like rectangular boxes while this one differs.

In figure 5.18 the seabed class gets only 22.66%. This is probably a result of the high amount of yellow color in the image. Since most ROV's in our data-set are yellow colored our model sees this image to be a ROV. Even the diver's class gets a relatively noticeable value due to the glasses in the image.

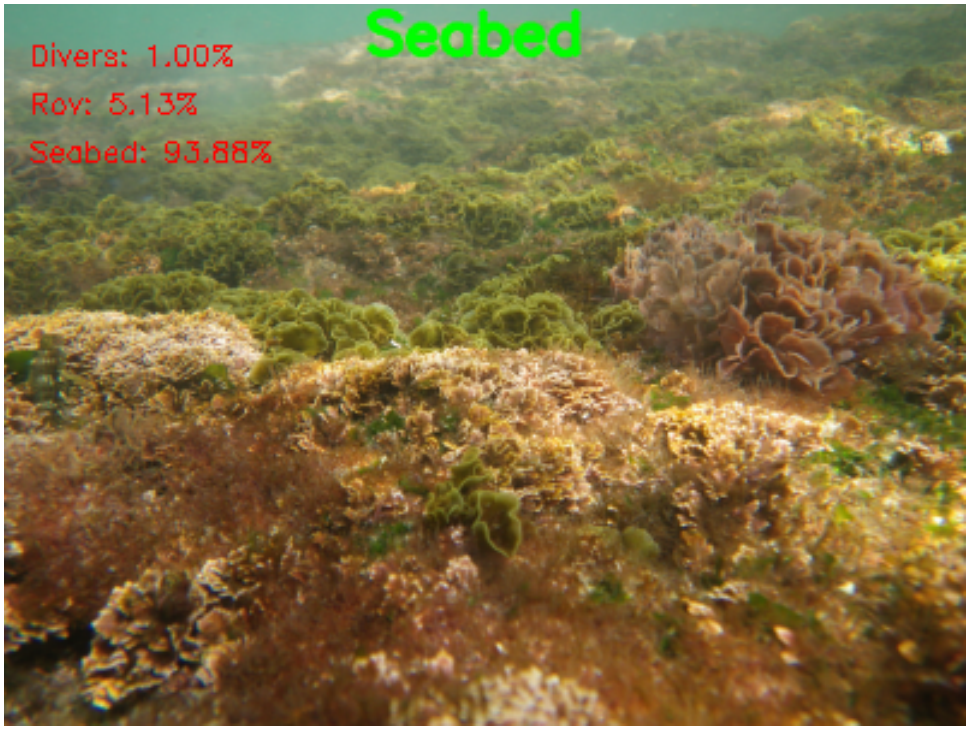

Figure 5.15: Light detection in Image with seabed (3)

Figure 5.16: False classification of a diver

Finally a registration is made in case of an alert, showing the type of alert (Diver or ROV) and the time-stamp of the event. Figures 5.19 and 5.20 show the registrations made for both diver and ROV detection.

Figure 5.17: False classification of a ROV

Figure 5.18: False classification of seabed

```
Diver detected! Time of event: 28-06-2018 11:02:53
Diver detected! Time of event: 28-06-2018 11:03:27
Diver detected! Time of event: 28-06-2018 11:03:54
Diver detected! Time of event: 28-06-2018 11:06:16
Diver detected! Time of event: 28-06-2018 11:06:43
Diver detected! Time of event: 28-06-2018 11:06:53
Diver detected! Time of event: 28-06-2018 11:08:08
Diver detected! Time of event: 28-06-2018 11:08:16
Diver detected! Time of event: 28-06-2018 11:08:23
Diver detected! Time of event: 28-06-2018 11:10:01
Diver detected! Time of event: 28-06-2018 11:10:13
Diver detected! Time of event: 28-06-2018 11:10:22
Diver detected! Time of event: 28-06-2018 11:10:30
Diver detected! Time of event: 28-06-2018 11:10:39
Diver detected! Time of event: 28-06-2018 11:12:42
Diver detected! Time of event: 28-06-2018 11:12:49
Diver detected! Time of event: 28-06-2018 11:12:55
Diver detected! Time of event: 28-06-2018 11:13:23
```

Figure 5.19: Diver detection registration

```
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:36
ROV detected! Time of event: 28-06-2018 11:41:37
```

Figure 5.20: ROV detection registration

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The program with current model achieves an 80% accuracy on daytime mode and up to 100% on nighttime mode. The nighttime mode can be easily set for such high accuracy with few data from the actual environment. The model's accuracy although its quite good it can not be trusted completely at the moment without any human supervision. However it can be easily improved significantly by increasing the amount of training and test data, getting data from the the actual environment in several scenarios. For example with divers with different suits approaching from random directions and different types of ROVs.

Ideally, for each area we want to watch, the model should be trained separately mostly with images from the actual environment. This would make our model insensitive to the background and focus more on potential changes in the area. This idea can be enhanced by subtracting the background image from every image before training as well as during the run-time mode. This way the model will ignore the background and learn more from the changes that happen in it like the presence of an intruder.

6.2 Future Work

Data-set Augmentation

Collecting a much larger data-set will certainly increase model's accuracy and min-

imize over-fitting. Usually neural networks are trained with data-sets of over 100.000 images. A training of this scale though requires also a lot of computing resources so good hardware is needed.

Increase of Image's Resolution

Currently the images we process are 32x32. This is obviously a very low resolution. If we manage to increase the data set then a higher resolution will let our model extract even more details from each image and thus achieve also higher accuracy. As expected an increase in resolution of the processed images will also increase the training time significantly.

Collecting data from the actual environment we plan to install the camera including various scenarios

A more aimed training of the model will result in more accurate predictions in real time situations as well. As mentioned in the conclusions, data from the area of interest will make the model less sensitive on the background and will focus more on whatever changes happen like someone or something entering the watching area. A subtraction of the background image from every frame will probably help too.

More complex CNN architecture

Assuming we applied all above, a more complex network architecture might also help in increasing the model's accuracy. One way to achieve this is adding more layers and filters to the network. Finding a perfect trade-off is a tedious task and requires a lot of tests and experience.

Appendix

Main Program

```
# import the necessary packages
from datetime import datetime
from imutils import contours
from skimage import measure
import numpy as np
import argparse
import imutils
import cv2
from keras.preprocessing.image import img_to_array
from keras.models import load_model

#function with custon created thresholds
def custom_threshold(image):

  # compute the average pixel value and the max pixel
  # value of the image
  maximum = np.max(image)

  # compute the threshold
  if average < 20:
    thresh = average*0.3 + maximum*0.7
```

```
    elif  (average >= 20) and (average < 30):
          thresh = average*0.2 + maximum*0.8
    elif  (average >= 30) and (average < 40):
          thresh = average*0.1 + maximum*0.9
    else :
          thresh = average*0.03 + maximum*0.95

    # return the image threshold
    return thresh, average, maximum

def mesage_on_screen(image, alert, av_str, max_str,
          thresh_str, date):

    #create text to be shown on screen
    font = cv2.FONT_HERSHEY_SIMPLEX
    average_pixel_value = 'Average pixel value: ' + av_str
    maximum_pixel_value = 'Maximum pixel value: ' + max_str
    threshold_value = 'Threshold: ' + thresh_str
    date_text = 'Time of event: ' + date
    cv2.putText(image, average_pixel_value, (10,50), font,
          0.3,(255,255,255),1,cv2.LINE_AA, False)
    cv2.putText(image, maximum_pixel_value, (10,60), font,
          0.3,(255,255,255),1,cv2.LINE_AA, False)
    cv2.putText(image, threshold_value, (10,70), font,
          0.3,(255,255,255),1,cv2.LINE_AA, False)
    cv2.putText(image, date_text, (10,80), font, 0.3,
          (255,255,255),1,cv2.LINE_AA, False)

    if alert == True :
          light_detection = 'Light detected!!!'
          cv2.putText(image, light_detection, (200,30), font,
                0.8,(0,0,255),2,cv2.LINE_AA, False)
```

```
  else :
        light_detection = 'No light detected'
        cv2.putText(image, light_detection, (200,30), font,
                0.8,(0,255,0),2,cv2.LINE_AA, False)

  return image

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
        help="path to trained model model")
args = vars(ap.parse_args())

# load the trained convolutional neural network
print("[INFO] loading network...")
model = load_model(args["model"])

#start video reading/capturing
cap = cv2.VideoCapture(0)

while(cap.isOpened()):

  # Capture frame-by-frame
  ret, frame = cap.read()

  # Checking if frame has been read correctly
  if ret:
    # transforming image to grayscale and checking
    # average pixel value to decide day/night mode
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    average = np.mean(gray)
```

```
    orig = frame.copy()

    # get current date and time
    date = datetime.now().strftime('%d-%m-%Y %H:%M:%S')
    # set alert to False
    alert = False

    average = np.mean(frame)
    if average < 50:
      #resize video to fixed size
      frame = cv2.resize(frame, (600, 400),
        interpolation = cv2.INTER_CUBIC)

      # load the image, convert it to grayscale, use
      # adaptive histogram equalization and
      #remove gaussian noise by blurring it
      gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
      clahe = cv2.createCLAHE(clipLimit=2.0,tileGridSize=(8,8))
      gray = clahe.apply(gray)
      blurred = cv2.GaussianBlur(gray, (11, 11), 0)

      # threshold the image to reveal light regions in the
      # blurred image changing threshold value makes our
      # detection more or less sensitive
      custom_thresh, average, maximum = custom_threshold(gray)
      thresh = cv2.threshold(blurred, custom_thresh, 255,
        cv2.THRESH_BINARY )[1]

      # perform a series of erosions and dilations to
      # remove any small blobs of noise from the
      # thresholded image changing iterations value makes
      # our detection more or less sensitive
```

```
    thresh = cv2.erode(thresh, None, iterations=2)
    thresh = cv2.dilate(thresh, None, iterations=8)

    # perform a connected component analysis on the
    # thresholded image, then initialize a mask to store
    # only the "large" components
    labels = measure.label(thresh, neighbors=8, background=0)
    mask = np.zeros(thresh.shape, dtype="uint8")

    # loop over the unique components for label in
    # np.unique(labels). if this is the background label,
    # ignore it
    if label == 0:
      continue

    # otherwise, construct the label mask and count the
    # number of pixels
    labelMask = np.zeros(thresh.shape, dtype="uint8")
    labelMask[labels == label] = 255
    numPixels = cv2.countNonZero(labelMask)

    # if the number of pixels in the component is
    # sufficiently large, then add it to our mask of
    # "large blobs" and flag it as alert mode
    if numPixels > 200:
      mask = cv2.add(mask, labelMask)
      alert = True

    # find the contours in the mask
    cnts = cv2.findContours(mask.copy(),cv2.RETR_EXTERNAL,
      cv2.CHAIN_APPROX_SIMPLE)
    cnts = cnts[0] if imutils.is_cv2() else cnts[1]
```

```
# loop over the contours
for (i, c) in enumerate(cnts):
  # draw the bright spot on the image
  (x, y, w, h) = cv2.boundingRect(c)
  ((cX, cY), radius) = cv2.minEnclosingCircle(c)
  cv2.circle(frame, (int(cX), int(cY)), int(radius),
    (0, 0, 255), 3)

# convert average, maximum, custom_thresh to strings
av_str = str(average)
max_str = str(maximum)
thresh_str = str(custom_thresh)

#get current date and time
date = datetime.now().strftime('%d-%m-%Y %H:%M:%S')

#checking if light is detected
if alert == True:
  #show values and message on screen
  print 'Light detected!'
  frame = mesage_on_screen(frame, alert, av_str, max_str,
    thresh_str, date)

  # create output text string
  date_text = 'Time of event: ' + date
  text = "Possibly diver or ROV detected! " + date_text

  #append the output string in a txt file
  text_file = open("log.txt", "a")
  text_file.write(text + "\n")
  text_file.close()
```

```
    else :

      #show values and message on screen
      print 'No light detected'
      text, frame = mesage_on_screen(frame, alert, av_str,
        max_str, thresh_str, date)

      #create output video
      out.write(frame)
      cv2.imshow("frame", frame)

  else :
    # pre-process the image for classification
    frame = cv2.resize(frame, (32, 32))
    frame = frame.astype("float") / 255.0
    frame = img_to_array(frame)
    frame = np.expand_dims(frame, axis=0)

    # classify the input image
    (seabed, divers, rov) = model.predict(frame)[0]

    # build the label
    label1 = "{}: {:.2f}%".format("Divers", divers * 100)
    label2 = "{}: {:.2f}%".format("Rov", rov * 100)
    label3 = "{}: {:.2f}%".format("Seabed", seabed * 100)

    # draw the label on the image
    output = imutils.resize(orig, width=400)
    cv2.putText(output, label1, (10, 25),
      cv2.FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1)
    cv2.putText(output, label2, (10, 45),
      cv2.FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1)
```

```
        cv2.putText(output, label3, (10, 65),
          cv2.FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1)

        # set alert mode on/off, define alert type and append
        # message to log.txt file
        if divers > rov and divers > seabed:
          alert = True
          type_of_alert = "Diver"
          date_text = 'Time of event: ' + date
          text = "Diver detected! " + date_text
          message = "Diver detected!"
          cv2.putText(output, message, (100,20),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.8,(0,0,255),2, cv2.LINE_AA, False)

          #append the output string in a txt file
          text_file = open("log.txt", "a")
          text_file.write(text + "\n")
          text_file.close()

        elif rov > divers and rov > seabed:
          alert = True
          type_of_alert = "ROV"
          date_text = 'Time of event: ' + date
          text = "ROV detected! " + date_text
          message = "ROV detected!"
          cv2.putText(output, message, (100,20),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.8,(0,0,255), 2,cv2.LINE_AA, False)

          #append the output string in a txt file
          text_file = open("log.txt", "a")
```

```
        text_file.write(text + "\n")
        text_file.close()

      else :
        alert = False
        type_of_alert = None
        message = "Seabed"
        cv2.putText(output, message, (150,20),
        cv2.FONT_HERSHEY_SIMPLEX, 0.8,(0,255,0),
        2,cv2.LINE_AA, False)

    # show the output image
    cv2.imshow("Output", output)
  else :
    break

  # if q is pressed manually close program
  if cv2.waitKey(1) & 0xFF == ord('q'):
    break

#video release and closing program
cap.release()
cv2.destroyAllWindows()cv2.LINE_AA, False)
```

Bibliography

[1] Gary Bradski and Adrian Kaehler. Learning OpenCV. O'Reilly Media, September 2008

[2] Alexander Mordvintsev and Abid K. OpenCV-Python Tutorials Documentation. September 02, 2017

[3] Wesley J. Chun. Core Python Programming, 2nd Edition. Prentice Hall, September 18, 2006

[4] Alex Martelli. Python in a Nutshell, 2nd Edition. O'Reilly Media, July 2006

[5] Adrian Rosebrock. Practical Python and OpenCV. An Introductory, Example Driven Guide to Image Processing and Computer Vision, 3rd Edition. Pyimagesearch, 2014

[6] Adrian Rosebrock. Deep Learning for Computer Vision with Python. Pyimagesearch, 2017

[7] Vikas Gupta. Deep learning using Keras – The Basics. Learn OpenCV, September 25, 2017

[8] Vikas Gupta. Image Classification using Feedforward Neural Network in Keras. Learn OpenCV, October 23, 2017

[9] Vikas Gupta. Image Classification using Convolutional Neural Networks in Keras. Learn OpenCV, November 29, 2017

[10] Wolfgang Beyer. How to Build a Simple Image Recognition System with TensorFlow (part1). Wolfib.com, December 5, 2016

[11] Wolfgang Beyer. How to Build a Simple Image Recognition System with TensorFlow (part2). Wolfib.com, December 31, 2016

[12] Francois Chollet. Building powerful image classification models using very little data. The Keras Blog, June 5, 2016

[13] Jason Brownlee. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. Machine Learning Mastery, July 3, 2017

[14] Charlie Crawford. An Introduction to Deep Learning. Algorithmia, November 4, 2016

[15] Aarshay Jain. Deep Learning for Computer Vision - Introduction to Convolution Neural Networks. Analytics Vidhya, April 4, 2016

[16] Christos Stergiou and Dimitrios Siganos. Neural Networks. Imperial London College, Department of Computing. www.doc.ic.ac.uk/~ nd/surprise_96/.../report.html, 2018

[17] Sebastien Villon, Marc Chaumont, Gerard Subsol, Sebastien Villeger, Thomas Claverie, and David Mouillot Coral reef fish detection and recognition in underwater videos by supervised machine learning : Comparison between Deep Learning and HOG+SVM methods LIRMM, University of Montpellier/CNRS, France, University of Nimes, France, MARBEC, IRD/Ifremer/University of Montpellier/CNRS, France.

[18] Javier Perez, Aleks C. Attanasio Nataliya Nechyporenko, and Pedro J. Sanz.

A Deep Learning Approach for Underwater Image Enhancement Department of Computer Science an Engineering, Jaume I University Castellon de la Plana, Spain.

[19] Jonathan Horgan and Daniel Toal. Computer Vision Applications in the Navigation of Unmanned Underwater Vehicles University of Limerick, Ireland.

[20] Bastiaan J. Boom, Phoenix X. Huang, Cigdem Beyan, Concetto Spampinato, Simone Palazzo, Jiyin He, Emmanuelle Beauxis-Aussalet, Sun-In Lin, Hsiu-Mei Chou, Gayathri Nadarajan, Yun-Heh Chen-Burge, Jacco van Ossenbruggen, Daniela Giordano, Lynda Hardman, Fang-Pang Lin, Robert B. Fisher Long-term underwater camera surveillance for monitoring and analysis of fish populations. Int. Workshop on Visual observation and Analysis of Animal and Insect Behavior (VAIB), in conjunction with ICPR 2012, Tsukuba Science City, Japan. January 2012

[21] Andreas Geiger, Philip Lenz and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite.

[22] Apple's Computer Vision Machine Learning Team An On-device Deep Neural Network for Face Detection Apple's Machine Learning Journal, Vol. 1, Issue 7 November 2017