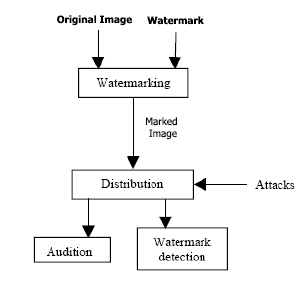
**IV.1 Concept:**

A watermarking system is usually divided into three distinct steps, embedding, attack, and detection. In embedding, an algorithm accepts the host and the data to be embedded, and produces a watermarked image.

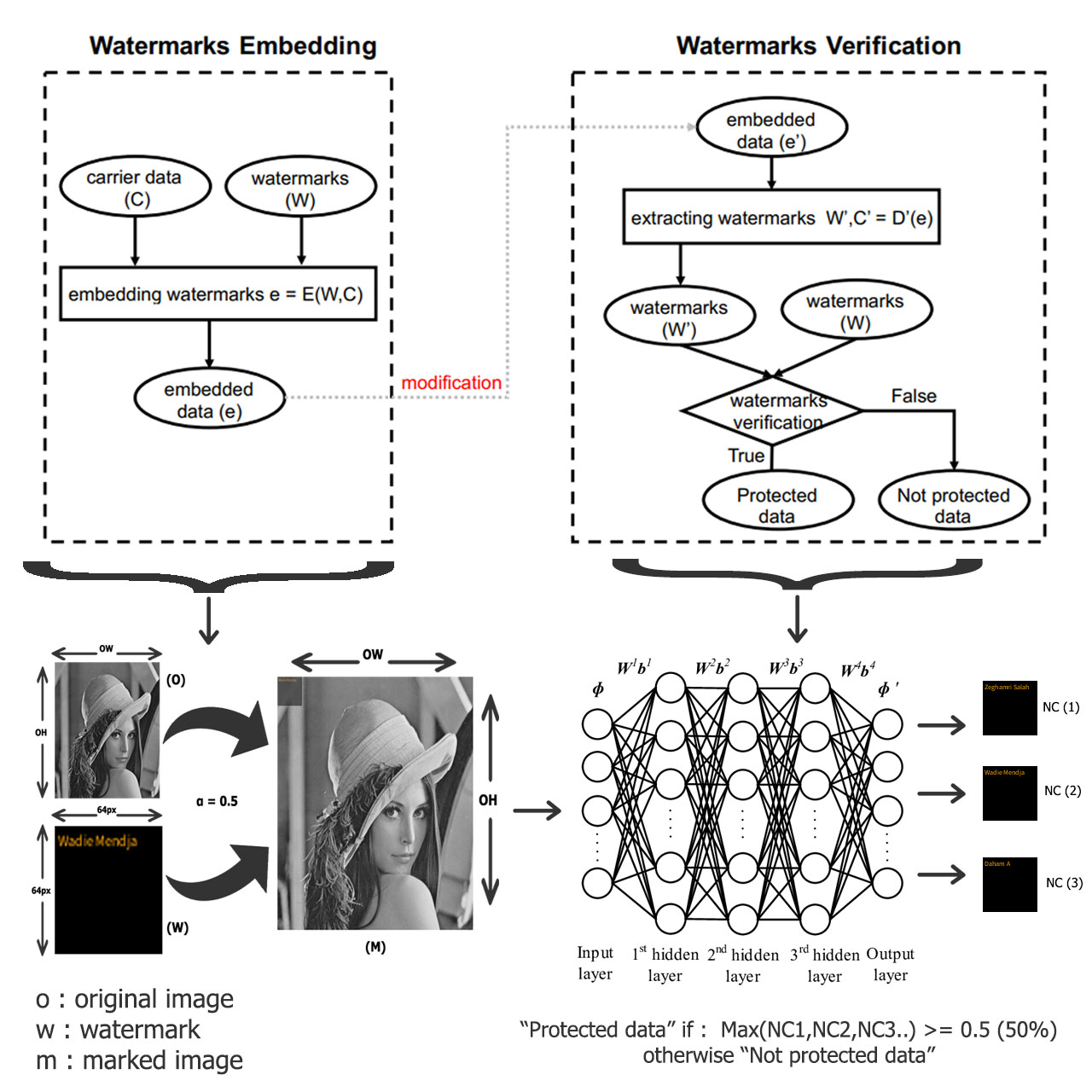
Then the watermarked digital image is transmitted or stored, usually transmitted to another person. If this person makes a modification, this is called an attack. While the modification may not be malicious, the term attack arises from copyright protection application, where third parties may attempt to remove the digital watermark through modification.

Detection (often called extraction) is an algorithm which is applied to the attacked image to attempt to extract the watermark from it. If the image was unmodified during transmission, then the watermark still is present and it may be extracted. In robust digital watermarking applications, the extraction algorithm should be able to produce the watermark correctly, even if the modifications were strong. In fragile digital watermarking, the extraction algorithm should fail if any change is made to the image.

Types of Watermarks: Visible Watermarks – These watermarks are visible. Invisible Watermarks – These watermarks are embedded in the media and use steganography technique. They are not visible by naked eyes.

**Figure IV.1:** Watermarking process

**IV.2 Overall architecture:**

Watermarking procedure is usually divided into two steps: embedding and verification. In the embedding process, an embedding algorithm E embeds pre-defined watermarks W into the carrier data C, which is the data to be protected. After the embedding, the embedded data (e = E(W, C)) are stored or transmitted. During the watermark verification process, a decryption algorithm D attempts to extract.

**Figure IV.2** Overall architecture

NC (Neuron Coverage) bisects a neuron’s state into activated and non-activated. Given an input, a neuron is activated if its output value is above a predefined threshold. NC measures the ratio of activated neurons of a DNN. [46]

**IV.3 Embedding a watermark:**

We developed our app using JavaScript programming language, and this language offers a great image processing and manipulation APIs, so in our case we’re using the “canvas fill text “API to embed certain text as a watermark on the input image. But how this actually works someone might ask?

Well, let us assume that an original image **O** has a width of **OW** and a height of OH and the watermark **W** is basically a 64x64 frame filled with a text (signature) which is going to be overlapped on **O** with a predefined opacity **α** which varies between (0.1 to 1), and with all that being performed a marked image **M** is going to be produced as given in Eq. 1

**m = O + αW** ……………. (1)

Illustration:

**Figure IV.3** Embedding Process

**a) CanvasRenderingContext2D.fillText() method:**

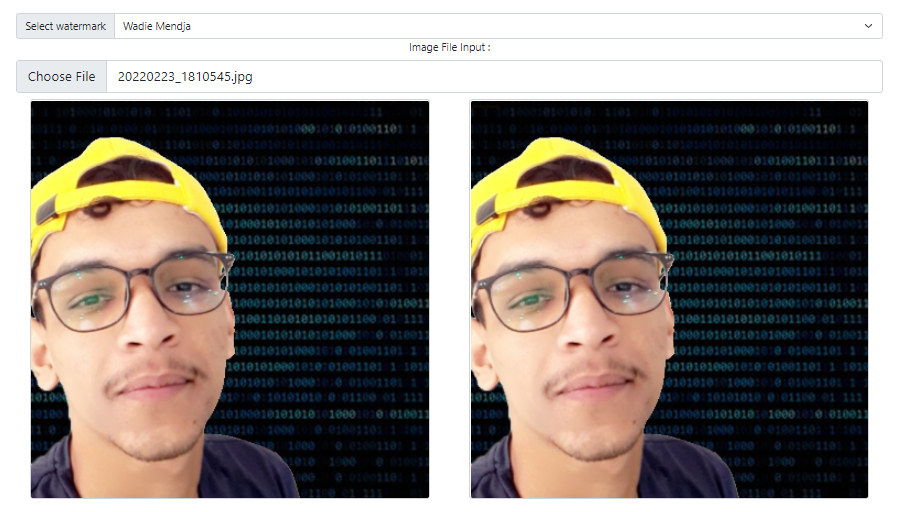
The CanvasRenderingContext2D method fillText(), part of the Canvas 2D API, draws a text string at the specified coordinates, filling the string's characters with the current fillStyle. An optional parameter allows specifying a maximum width for the rendered text, which the user agent will achieve by condensing the text or by using a lower font size.

This method draws directly to the canvas without modifying the current path, so any subsequent fill() or stroke() calls will have no effect on it.

The text is rendered using the font and text layout configuration as defined by the font, textAlign, textBaseline, and direction properties. [47]

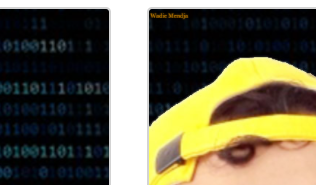
Syntax:

CanvasRenderingContext2D.fillText(text, x, y [, maxWidth]);

Here is an example of how it works, in the screenshot below we have on the left side the original image and the watermarked image on the right side, the “Select Watermark” text field contains the text that we should embed on the image as a watermark, we did that using the syntax that we talked about earlier

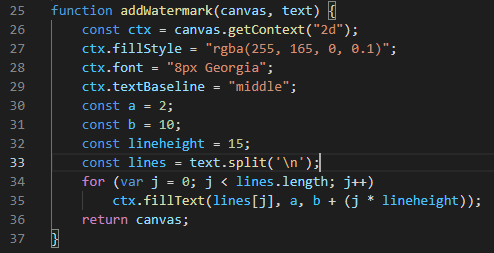
**Figure IV.4** App overview

You may not notice any differences between the two images and that’s because we’re embedding an invisible watermark(α=0.1), but if we try to make the watermark visible(α=1) it’s going to look like that:



**Figure IV.5** Watermak visibility (α=1.0)

And here is the function that does all this work:



**Figure IV.6** the functionaddWatermark

In which the “fillStyle” property specified the color and the opacity (alpha component) of the text that we’re trying to insert into the image, also we have “font” property which specifies the size of the text and its font, the constants “a” & “b” defines the position of the text on the inserted image.

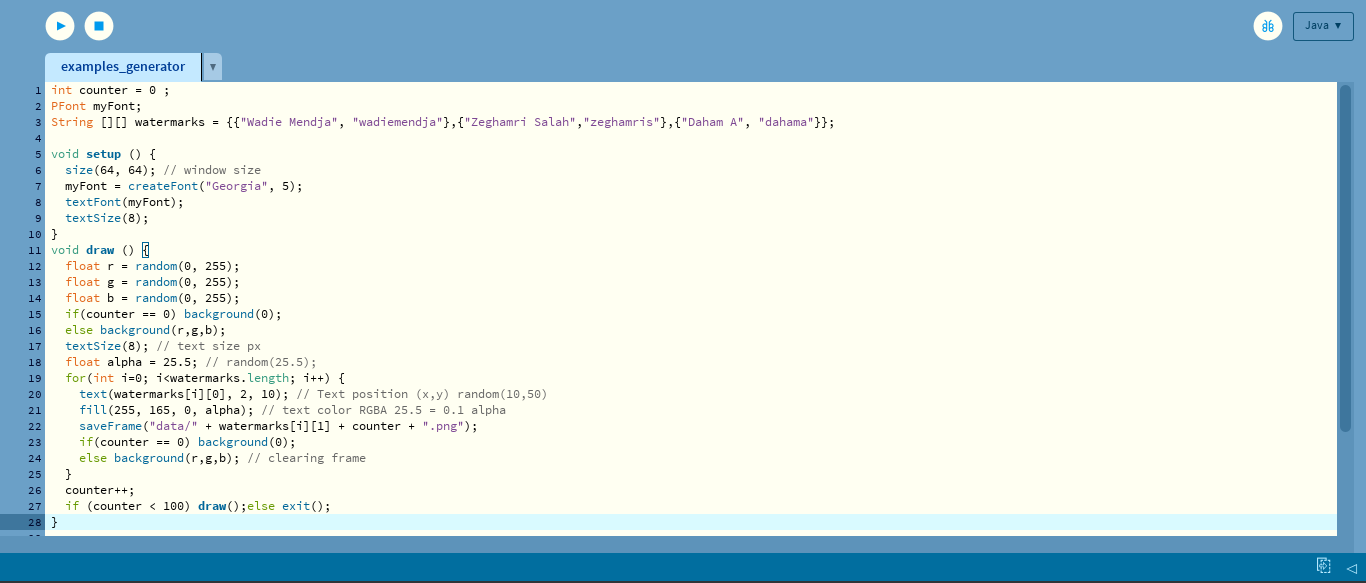
Now that we’re done with the watermarking part lets skip up to the good parts which are generating a dataset of image (watermarks) and training a neural network to be able to detect it.

**IV.4 Generating a Dataset**

Before generating a dataset of watermarks, we should first create them, so in our app we’re using three different watermarks basically images that are filled with the following texts “Wadie Mendja”, “Zeghamri Salah” and “Daham A” which they look like this:

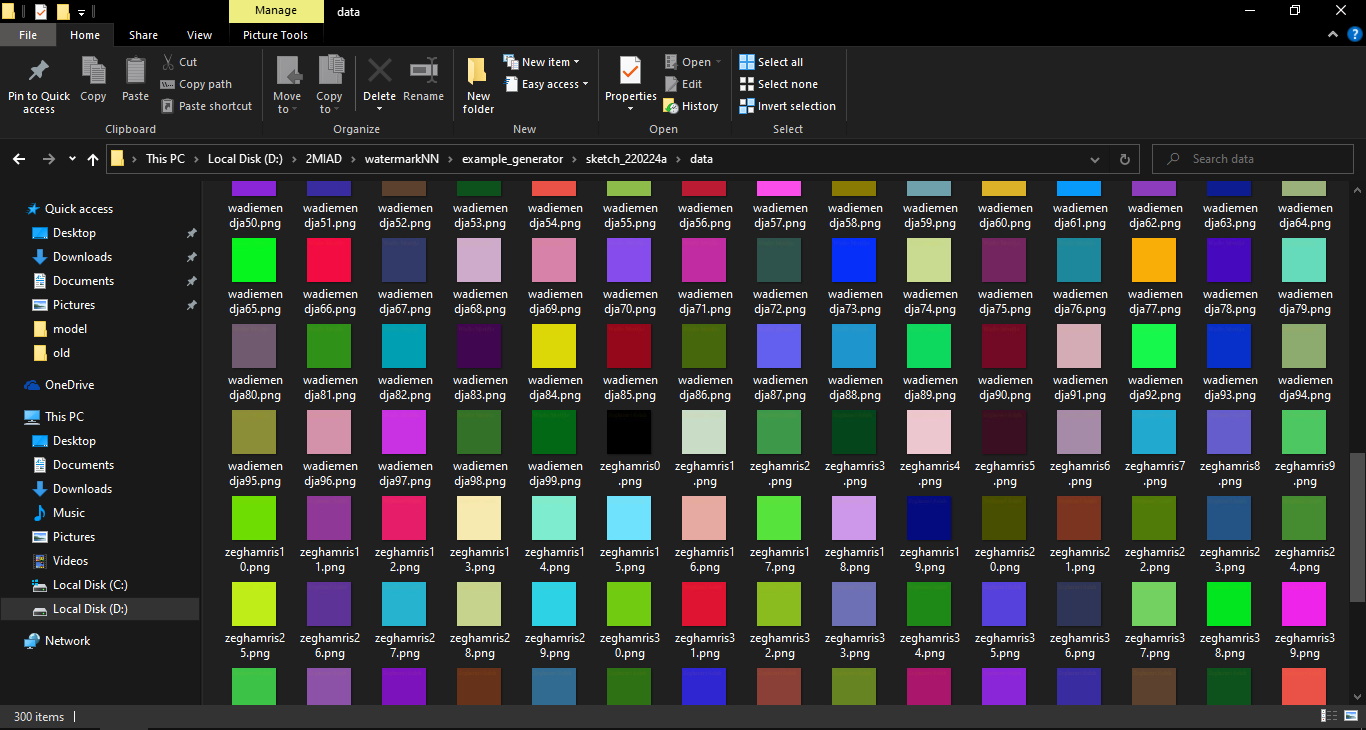


**Figure IV.7** pre-defined watermarks

Those are the original watermarks, now we need to images similar to them by changing the text opacity, position and background color, so for this task we’re using a programming language called processing which is a graphical library and integrated development environment built for the electronic arts that’s going to help us generate multiple frames (images) and save them in our machine so we can use them in the training process, here you can take a look at the program:

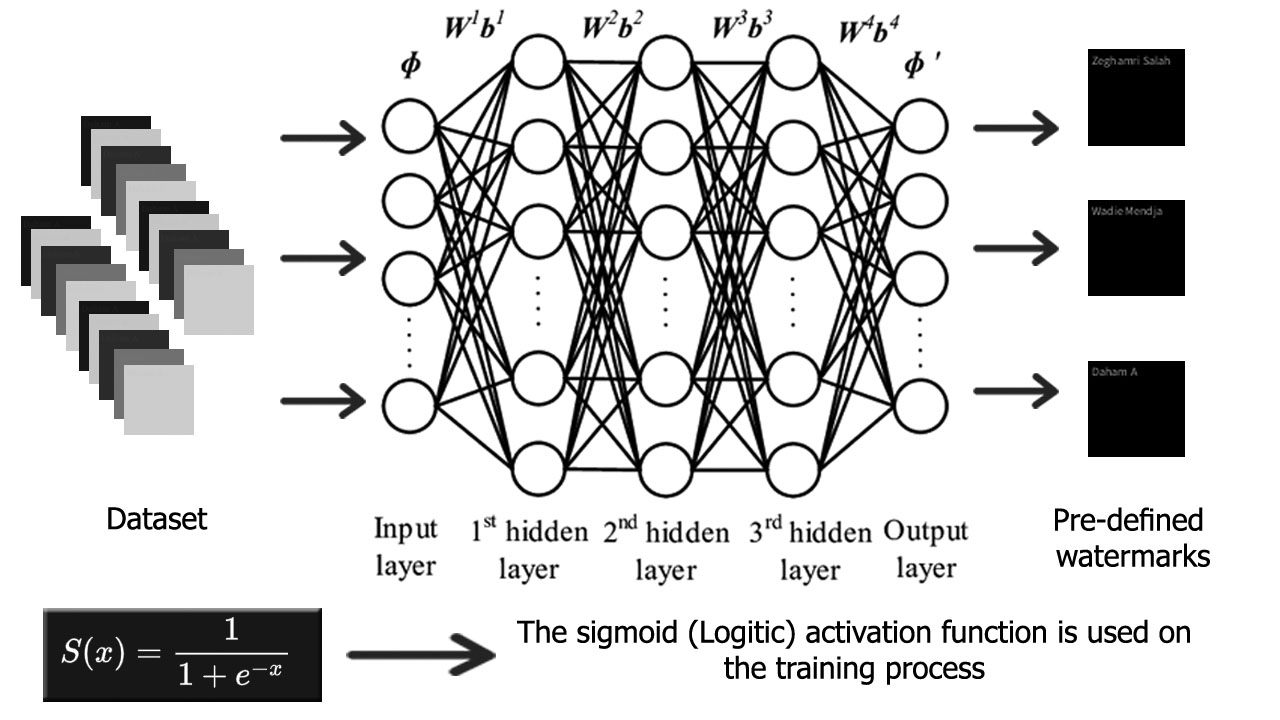
**Figure IV.8** Dataset generator

All what this program does is generate a bunch (100 in this case) of frames (images) with a size of 64x64 pixels that contains the watermarks above that we talked about (Wadie Mendja | Daham A | Zeghamri Salah) but with different sizes, positions, opacities and background colors, and finally save them as PNG files in a folder called “data”, that’s it.

Here is a screenshot of the “data” folder after the execution in done:

**Figure IV.9** Generated dataset

**IV.5 Training the dataset**

Our DNN is going to take each image’s individual pixel (RGBA) value as an input to the first layer of our DNN, the total number of inputs should be 64\*64\*4 which is 16384 inputs and 3 outputs since we have three pre-defined watermarks:

**Figure IV.10** Training process

So, in this particular step we’re going to be using a library called “ml5.js” which uses the TensorFlow framework and has a predefined neural network class which will allow as to train a model and do the classification after the training process, all we have to do is give it the dataset and a label for each particular watermark, in general the steps for using the ml5.neuralNetwork look something like:

**Step 1:** load data or create some data

**Step 2:** Set your neural network options & initialize your neural network

**Step 3:** add data to the neural network

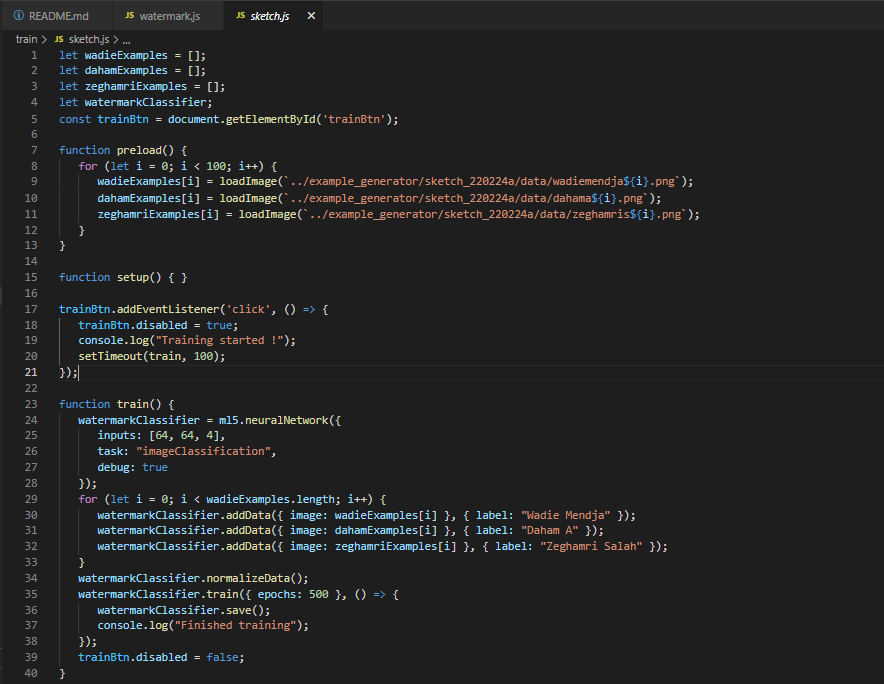
**Step 4:** Normalize your data

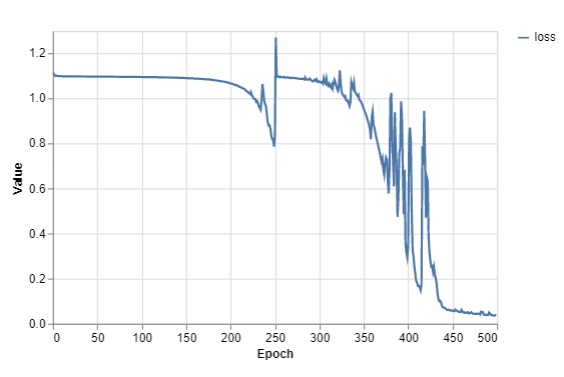
**Step 5:** Train your neural network

**Step 6:** use the trained model to make a classification

**Step 7:** do something with the results

Our dataset (100 sample for each watermark) will be trained with 500 epochs, here is the code that does the training process :

**Figure IV.11** Training code

Training performance:

**Figure IV.12** Training performance

An epoch in machine learning simply means one complete pass of the training dataset through the algorithm. This epochs number is an important hyperparameter for the algorithm. It specifies the number of epochs or complete passes of the entire training dataset passing through the training or learning process of the algorithm. [48]

**IV.6 Watermark extraction**

The extraction phase is divided into two steps:

a) Cropping the watermark area (64x64 pixel) where the watermark is expected to be as show below;

b) The trained DNN is going to take WA as an input (pixels matrix), and in the other hand we got 3 output nodes which well end up taking one of if not depending on the NC factor (confidence score) as given in Eq. 2 and 3:

**NC(i) = AN / TNN** …... (Eq. 2)

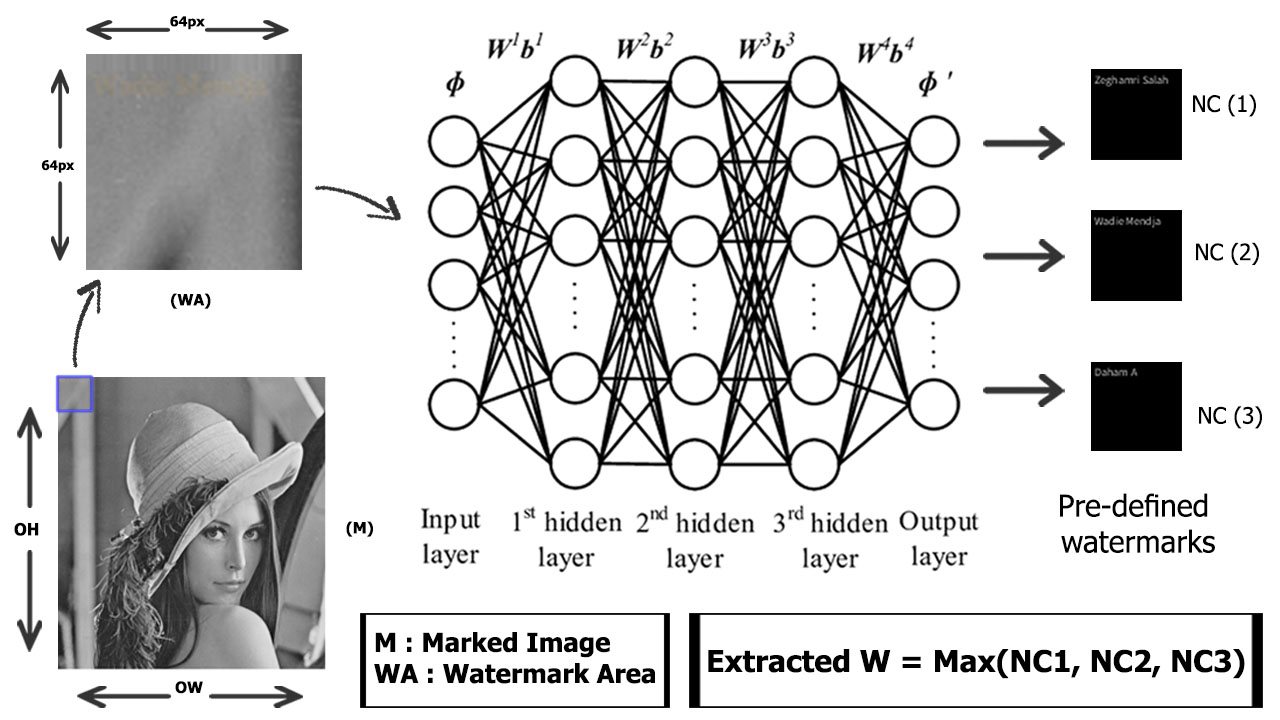
AN: activated neurons

TNN: total number of neurons

The percentage of non-activated neurons is going to be calculated as following:

**Err(i) = 1- NC(i)** …... (Eq. 3)

The minimum confidence for a positive watermark detection is 0.5, detections with a probability less than this value will be discarded as a false-positive result.

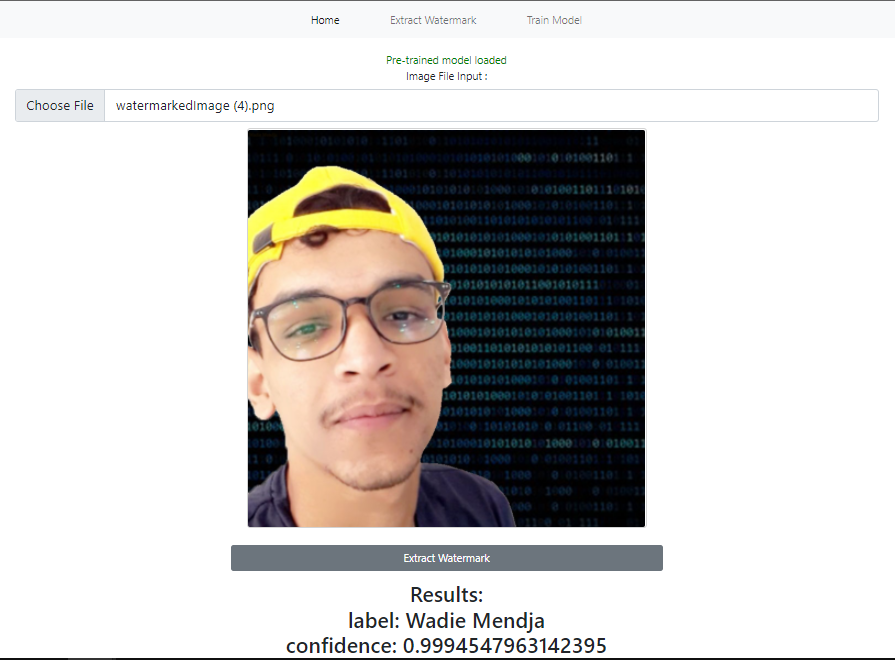
**Figure IV.13** Extraction Process

Results with a confidence score of 99.94% and an error of 0.06% which is a good indicator of a well-trained model, although we applied a compression attack on the image (Vectorial Quantization) but still the results turned out to be pretty good:

**IV.7 Testing**

Now it’s time to try it out and see if it will give us some decent results or not, so to do that we need to load our pretrained model to our app and trigger the “classify” method and see the results. Here is a piece of the code that does that:

**Figure IV.1****4** Extraction code

Results with a confidence score of 99.94% and an error of 0.06% which is a good indicator of a well-trained model, although we applied a compression attack on the image (Vectorial Quantization) but still the results turned out to be pretty good:

**Figure IV.15** Extraction results

**IV.8 Robustness and experiments:**

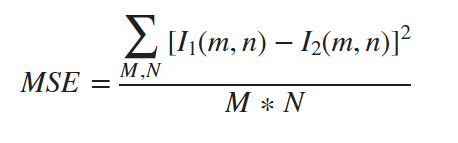
The robustness of the proposed system against different distortions applied to the marked-image is evaluated by analyzing the distortion tolerance range, and for that matter we’re using Microsoft COCO dataset and a software called Photoshop and some other tools to apply the distortion.

For demonstration, the proposed system generalizes the watermarking rules without over-fitting to the training samples, the testing cover-images are not used in the training.

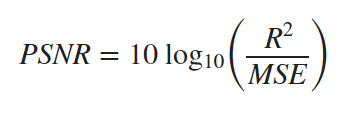
The PSNR block computes the peak signal-to-noise ratio, in decibels, between two images. This ratio is used as a quality measurement between the original and a compressed image. The higher the PSNR, the better the quality of the compressed, or reconstructed image.

The mean-square error (MSE) and the peak signal-to-noise ratio (PSNR) are used to compare image compression quality. The MSE represents the cumulative squared error between the compressed and the original image, whereas PSNR represents a measure of the peak error. The lower the value of MSE, the lower the error.

To compute the PSNR, the block first calculates the mean-squared error using the following equation:



In the previous equation, M and N are the number of rows and columns in the input images. Then the block computes the PSNR using the following equation:



In the previous equation, R is the maximum fluctuation in the input image data type. For example, if the input image has a double-precision floating-point data type, then R is 1. If it has an 8-bit unsigned integer data type, R is 255, etc. [49]

For these particular experiments we’re setting the watermark strength (α) to 0.1.

**a) Compression:**

Image compression is a type of data compression applied to digital images, to reduce their cost for storage or transmission. Algorithms may take advantage of visual perception and the statistical properties of image data to provide superior results compared with generic data compression methods which are used for other digital data.

Image compression may be lossy or lossless. Lossless compression is preferred for archival purposes and often for medical imaging, technical drawings, clip art, or comics. Lossy compression methods, especially when used at low bit rates, introduce compression artifacts. Lossy methods are especially suitable for natural images such as photographs in applications where minor (sometimes imperceptible) loss of fidelity is acceptable to achieve a substantial reduction in bit rate. Lossy compression that produces negligible differences may be called visually lossless. [50]

In this experiment we’re going to use a compression method called Transform coding which is the most commonly used method.

No compression 10% compression 20% compression

30% compression 40% compression 50% compression

**Figure IV.16** TC compression degrees sample

Results:

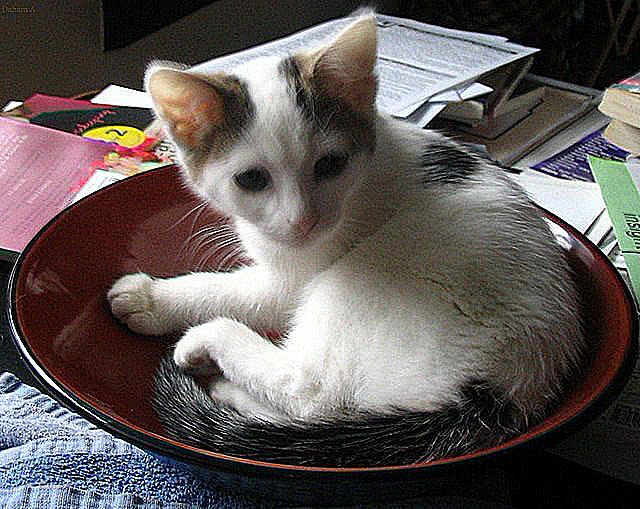
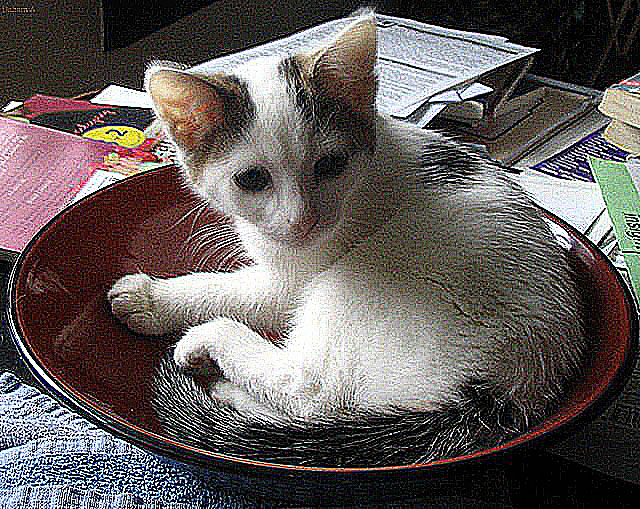
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Compression level (%)** | **NC** | **PSNR (dB)** | **Accuracy (%)** | **Output** |
| 0 (No compression) | 0.9622 | Infinite | 96.22 | Watermark detected |
| 10 | 0.9745 | 52.78 | 97.45 | Watermark detected |
| 15 | 0.9405 | 43.61 | 94.05 | Watermark detected |
| 20 | 0.5665 | 43.61 | 56.65 | Watermark detected |
| 25 | 0.7054 | 43.16 | 70.54 | Watermark detected |
| 30 | 0.6537 | 42.39 | 65.37 | Watermark detected |
| 35 | 0.7917 | 41.65 | 79.17 | Watermark detected |
| 40 | 0.6892 | 41.03 | 68.92 | Watermark detected |
| 45 | 0.4869 | 40.24 | 48.69 | No watermark detected |
| 50 | 0.4233 | 39.81 | 42.33 | No watermark detected |

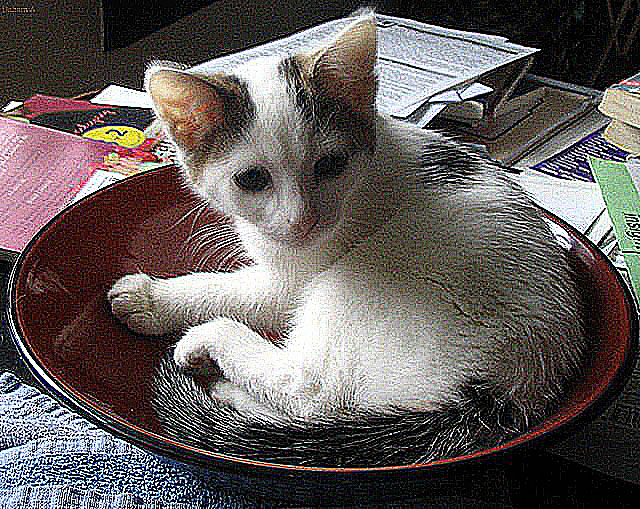
**Table IV.1:** compression resistant experiment

The accuracy is basically NC \* 100.

Our system can resist up to 40% compression with an accuracy between 97.45 and 56.65 %.

**b) Sharpening:** Image sharpening is an effect applied to digital images to give them a sharper appearance. Almost all lenses can benefit from at least a small amount of sharpening, we’re going to try multiple levels of sharpening and observe our system’s response to them:

 Marked-image 100% sharpening

 200% sharpening

**Figure IV.17** Sharpened images

Results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sharpening level (%)** | **NC** | **PSNR (dB)** | **Accuracy (%)** | **Output** |
| Default sharpening | 0.9743 | Infinite | 97.43 | Watermark detected |
| 100% | 0.9954 | 21.67 | 99.54 | Watermark detected |
| 200% | 0.9596 | 15.51 | 95.96 | Watermark detected |
| 400% | 0.9997 | 11.65 | 99.97 | Watermark detected |

**Table IV.2:** sharpening experiment

As a conclusion we could say that the sharpening goes up the accuracy typically goes up with it.

**c)** **Gaussian blur:**

In image processing, a 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. [51]

No gaussian blur gaussian blur: radius 10 pixels

**Figure IV.18** Gaussian blur

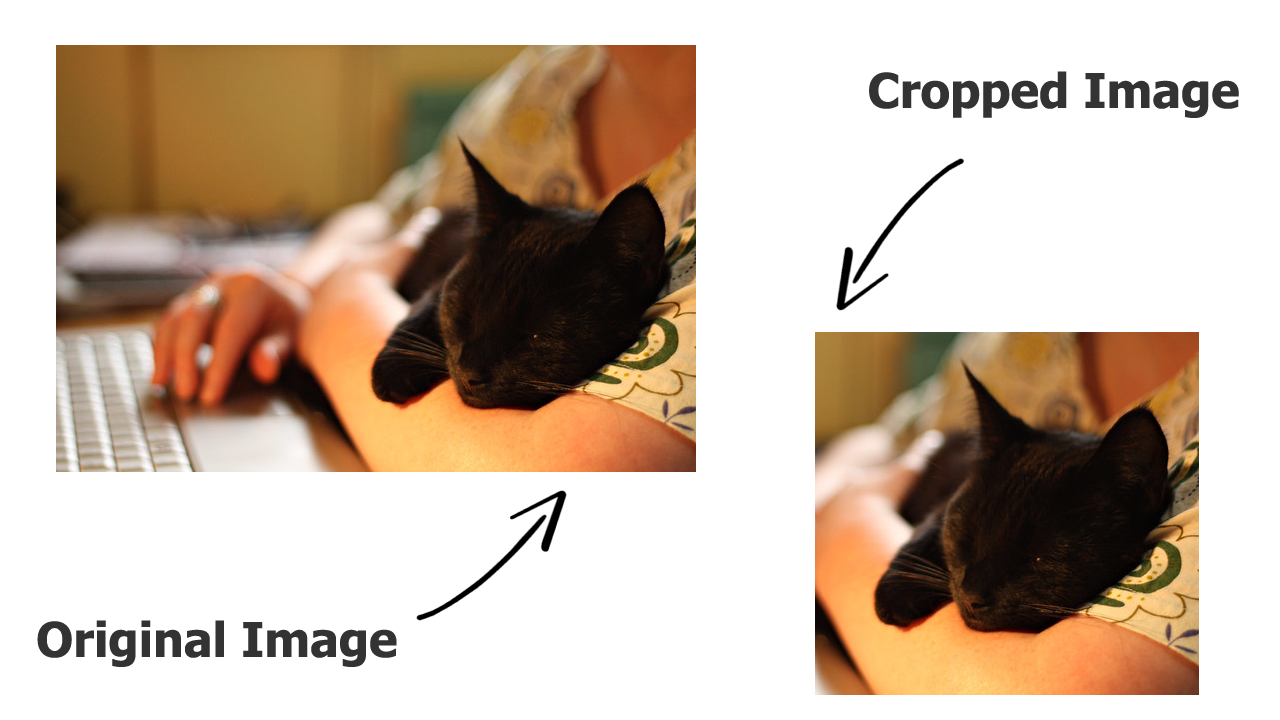
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Gaussian blur radius (pixels)** | **NC** | **PSNR (dB)** | **Accuracy (%)** | **Output** |
| None | 0.8323 | Infinite | 83.23 | Watermark detected |
| 0.5 | 0.8128 | 35.82 | 81.28 | Watermark detected |
| 1 | 0.8136 | 31.60 | 81.36 | Watermark detected |
| 2 | 0.6982 | 28.31 | 69.82 | Watermark detected |
| 3 | 0.7258 | 25.93 | 72.58 | Watermark detected |
| 4 | 0.4202 | 24.20 | 42.02 | No watermark detected |
| 5 | 0.41 | 21.85 | 41.00 | No watermark detected |

**Table IV.3:** Gaussian blur distortion experiment

Our system can resist up to 3 pixels radius gaussian blur with an accuracy between 83.23 to 69.82%.

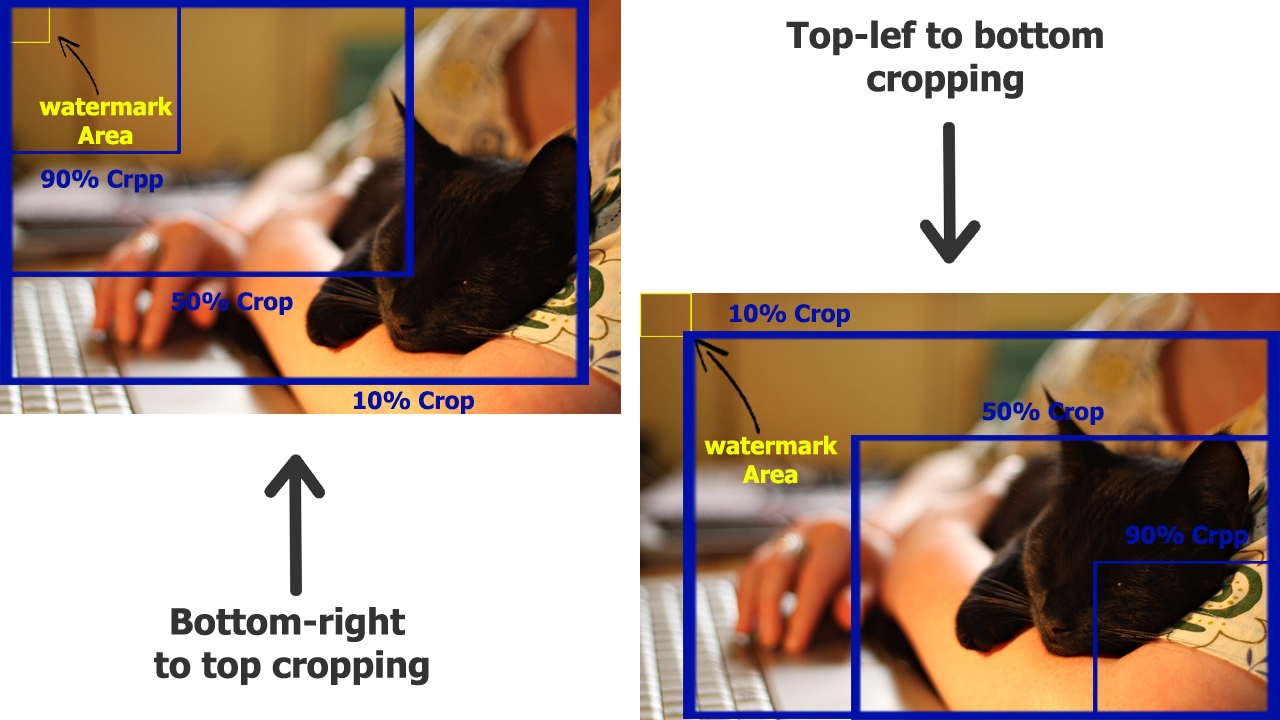
**d)** **Cropping:**

Cropping is the removal of unwanted outer areas from a photographic or illustrated image. The process usually consists of the removal of some of the peripheral areas of an image to remove extraneous trash from the picture, to improve its framing, to change the aspect ratio, or to accentuate or isolate the subject matter from its background.



**Figure IV.19** Cropped and non-cropped image

In this case our embedding algorithm embeds the watermark at the top left side (64x64 area) of the original image, so if marked image gets cropped at that area it will be impossible for us to extract it, but if the cropping happens to be from the bottom right to the top the extraction accuracy would be the same as the original marked image. the figure below shows the type of cropping that would and wouldn’t affect the extraction phase:



**Figure IV.20** Cropping that would and wouldn’t affect extraction

In simple word unless the watermark area does not get cropped the extraction algorithm would work just as well as an extraction with an original marked image.

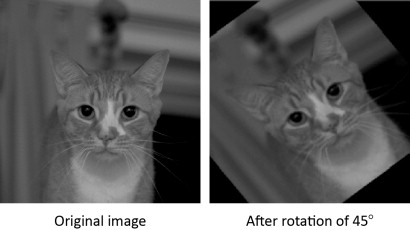
This may sound like a problem but as a solution for this I suggested embedding the watermark in different areas on the original image, for example one at the top and one at middle and another one on the bottom so that even if one or two areas get cropped you can still be able to extract it. The experiment table below applied to first scenario (bottom to top cropping):

|  |  |  |  |
| --- | --- | --- | --- |
| **Cropping** | **NC** | **Accuracy (%)** | **Output** |
| 10% | 0.9718 | 99.16 | Watermark detected |
| 50% | 0.9718 | 17.38 | No watermark detected |
| 90% | 0.9718 | 75.01 | Watermark detected |

**Table IV.4:** Cropping experiment

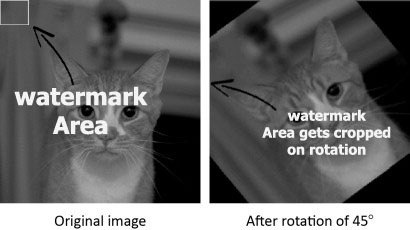
**e)** **Rotation:**

Image rotation is a common image processing routine with applications in matching, alignment, and other image-based algorithms. The input to an image rotation routine is an image, the rotation angle θ, and a point about which rotation is done. The Figure below show an example of an image rotation:



**Figure IV.21** Image rotation exemple

Rotation resistance goes back to the angle of that rotation θ which is going to determine whether or not the watermark is going to be available on the rotated image, the figure below shows an inability of extracting the watermark because of the cropping that happed on the rotation process:



**Figure IV.22** Rotation problem

But our algorithm is able to extract the watermark in stable rotation (vertical and horizontal) 180° and 90° as shown in the experiment table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rotation degree** | **NC** | **PSNR (dB)** | **Accuracy (%)** | **Output** |
| 0° | 0.9916 | Infinite | 99.16 | Watermark detected |
| 45° | 0.1738 | 9.27 | 17.38 | No watermark detected |
| 90° | 0.7501 | 5.45 | 75.01 | Watermark detected |
| 135° | 0.1473 | 9.39 | 14.73 | No watermark detected |
| 180° | 0.6615 | 9.17 | 66.15 | Watermark detected |

**Table IV.5:** Rotation experiment

**f)** **Brightness:**

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, it’s not easy to perceive. Brightness can be simply increased or decreased by simple addition or subtraction, to the image matrix.



**Figure IV.23** An image before and after applying brightness

The experiment table below show the result after applying different degrees of brightness:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Brightness degree** | **NC** | **PSNR (dB)** | **Accuracy (%)** | **Output** |
| 0% | 1 | Infinite | 100 | Watermark detected |
| 50% | 0.9999 | 16.08 | 99.99 | Watermark detected |
| 65% | 0.8685 | 14.01 | 86.85 | Watermark detected |
| 75% | 0.4998 | 12.54 | 49.98 | No watermark detected |
| 100% | 0.1918 | 10.42 | 19.18 | No watermark detected |

**Table IV.6:** Brightness experiment

**IV.9 Conclusion:**

Although we got great results on the experiments that we applied, however there are some cases were the marked-image gets damaged by some strong attacks or compression algorithms in a way that even a well-trained deep neural network cannot be able to detect the embedded watermark or even if it did it will be with a low confidence score (accuracy) which means a higher error, and the higher the error goes, the less we are sure about whether a certain watermark exists in an image or not.