

I) Development process:

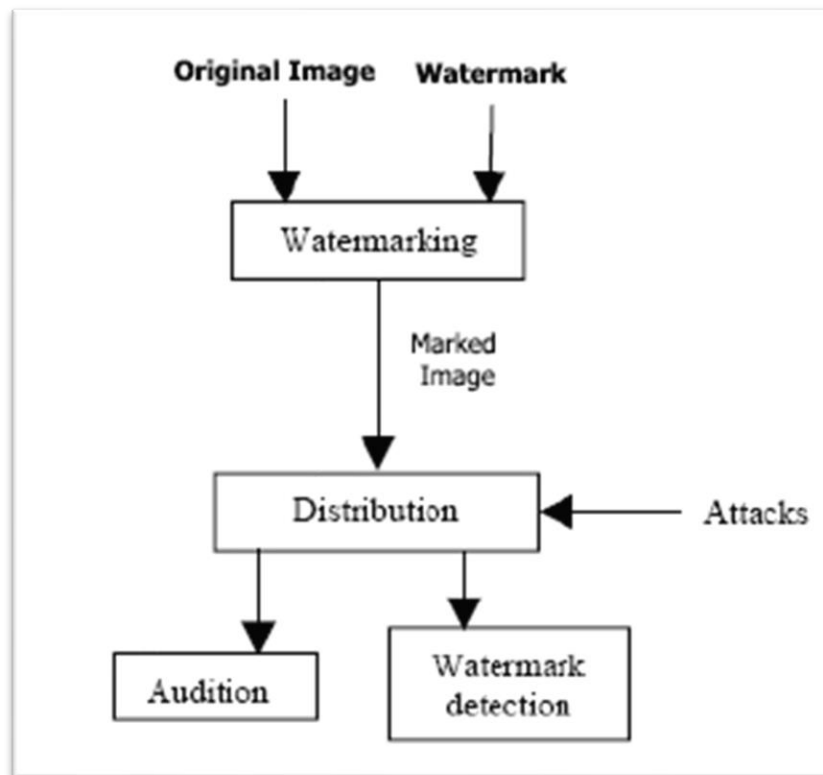
I.1) Concept:

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.



Watermarking process

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.



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. [1]

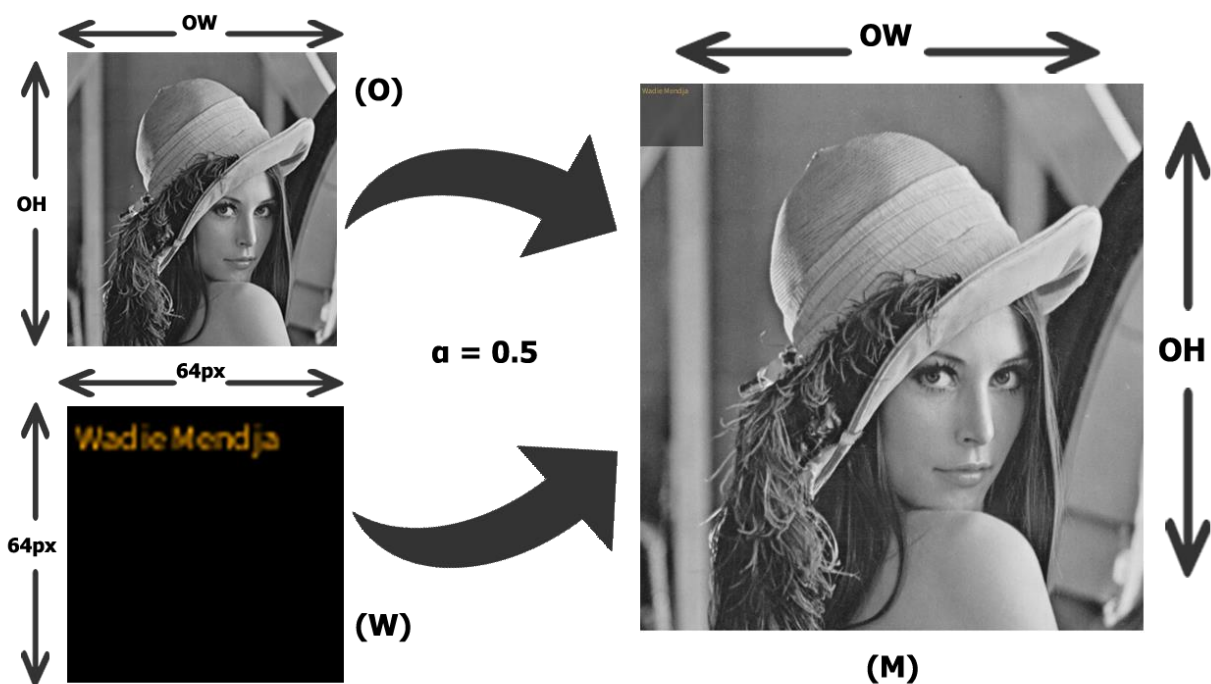
1.3) Embedding a watermark:

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

$$m = O + \alpha W \dots\dots\dots (1)$$

Illustration:



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.

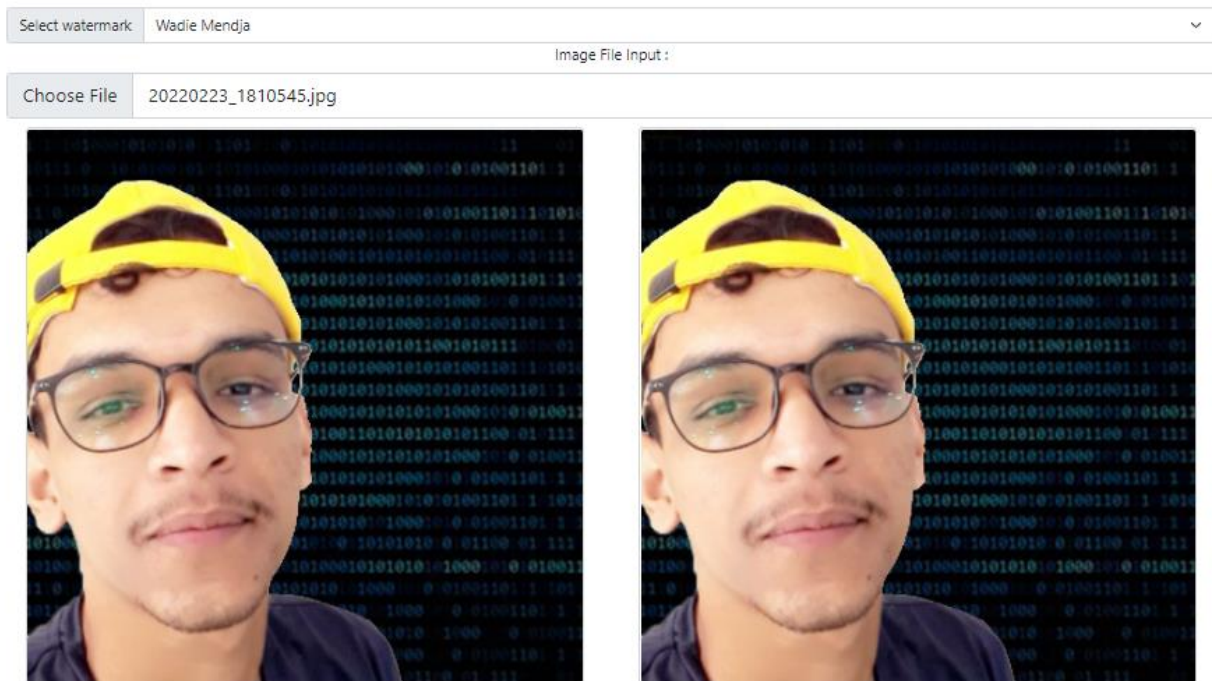
This method draws directly to the canvas without modifying the current path, so any subsequent fill() or stroke() calls will have no effect on it.

The text is rendered using the font and text layout configuration as defined by the font, textAlign, textBaseline, and direction properties. [2]

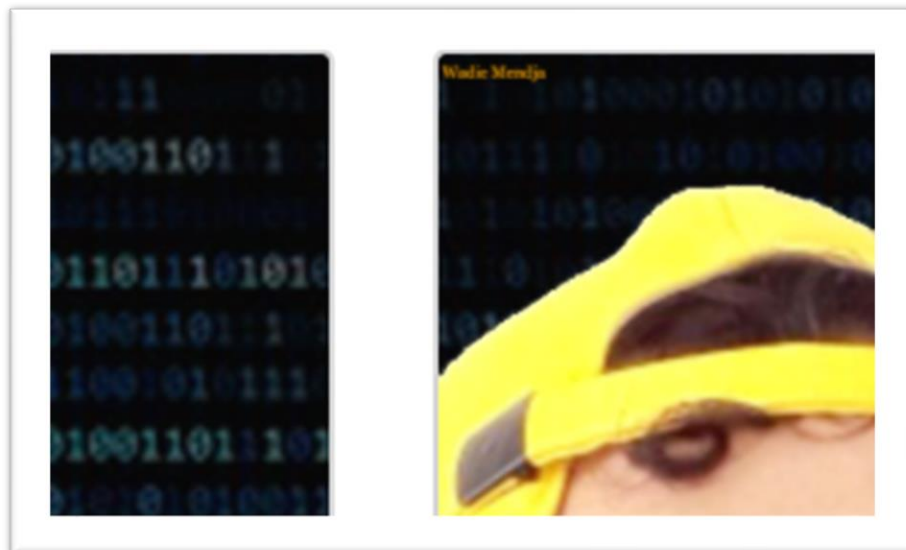
Syntax:

```
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



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



And here is the function that does all this work:

```
25 function addWatermark(canvas, text) {  
26     const ctx = canvas.getContext("2d");  
27     ctx.fillStyle = "rgba(255, 165, 0, 0.1)";  
28     ctx.font = "8px Georgia";  
29     ctx.textBaseline = "middle";  
30     const a = 2;  
31     const b = 10;  
32     const lineHeight = 15;  
33     const lines = text.split('\n');  
34     for (var j = 0; j < lines.length; j++)  
35         ctx.fillText(lines[j], a, b + (j * lineHeight));  
36     return canvas;  
37 }
```

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.


1.4) 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:



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

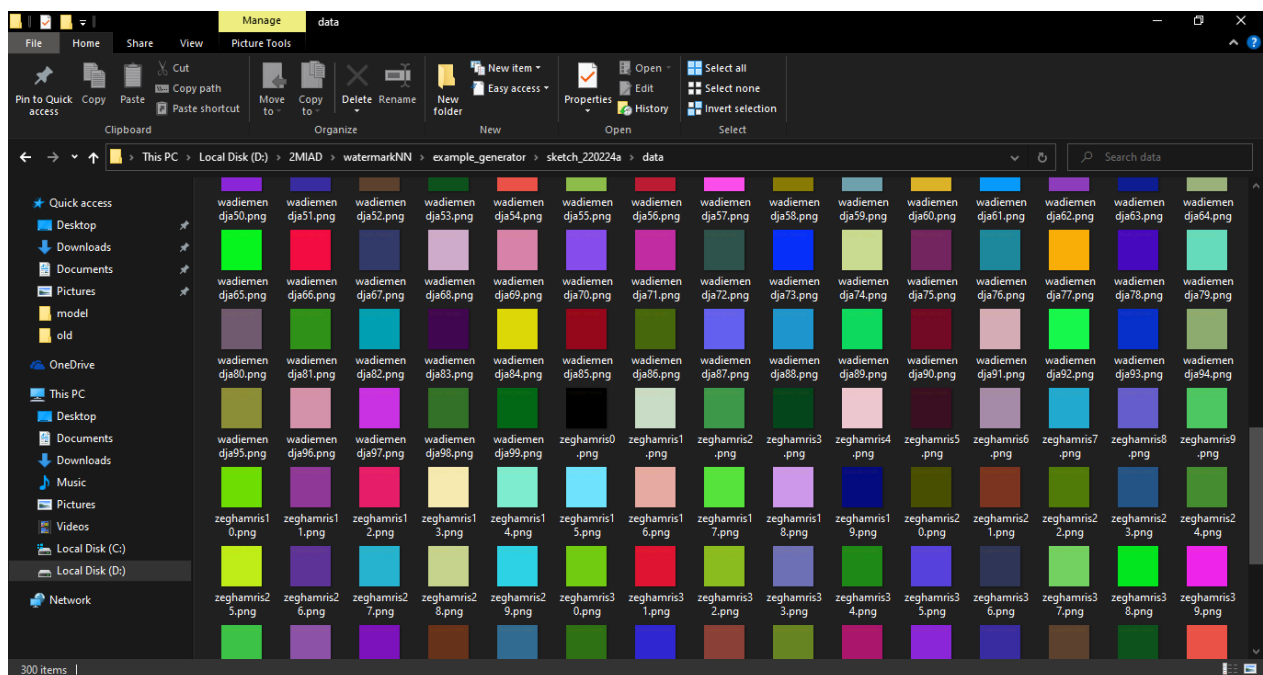
going to help us generate multiple frames (images) and save them in our machine so we can use them in the training process, here you can take a look at the program:



```
1 int counter = 0 ;
2 PFont myFont;
3 String [][] watermarks = {"Wadie Mendja", "wadiemendja"}, {"Zeghamri Salah", "zeghamris"}, {"Daham A", "dahama"};
4
5 void setup () {
6   size(64, 64); // window size
7   myFont = createFont("Georgia", 5);
8   textFont(myFont);
9   textSize(8);
10 }
11 void draw () {
12   float r = random(0, 255);
13   float g = random(0, 255);
14   float b = random(0, 255);
15   if(counter == 0) background(0);
16   else background(r,g,b);
17   textSize(8); // text size px
18   float alpha = 25.5; // random(25.5);
19   for(int i=0; i<watermarks.length; i++) {
20     text(watermarks[i][0], 2, 10); // Text position (x,y) random(10,50)
21     fill(255, 165, 0, alpha); // text color RGBA 25.5 = 0.1 alpha
22     saveFrame("data/" + watermarks[i][1] + counter + ".png");
23     if(counter == 0) background(0);
24     else background(r,g,b); // clearing frame
25   }
26   counter++;
27   if (counter < 100) draw();else exit();
28 }
```

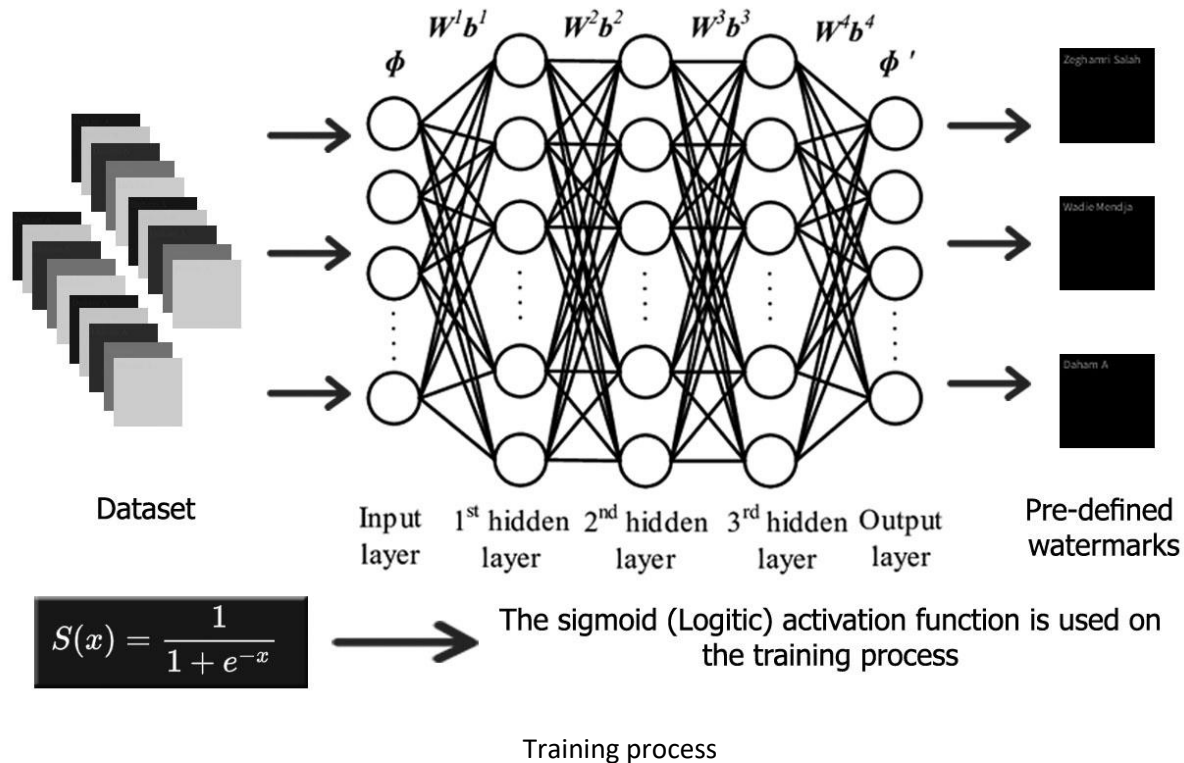
All what this program does is generate a bunch (100 in this case) of frames (images) with a size of 64x64 pixels that contains the watermarks above that we talked about (Wadie Mendja | Daham A | Zeghamri Salah) but with different sizes, positions, opacities and background colors, and finally save them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

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



1.5) 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 $64 \times 64 \times 4$ which is 16384 inputs and 3 outputs since we have three pre-defined watermarks:



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 us 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 4: add data to the neural network

Step 5: Normalize your data

Step 6: Train your neural network

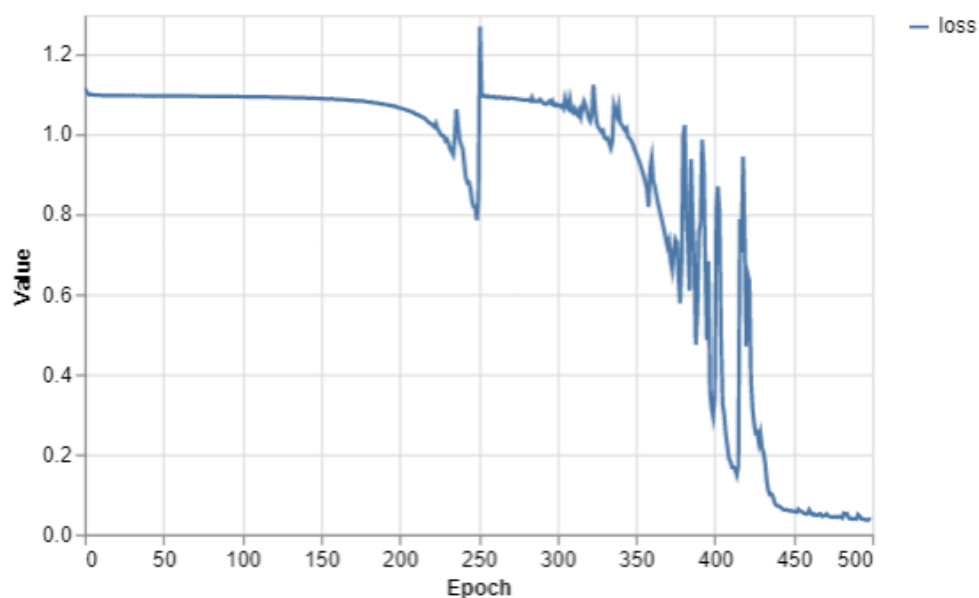
Step 7: use the trained model to make a classification

Step 8: 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:


```
train > JS sketchjs > ...
1 let wadieExamples = [];
2 let dahamExamples = [];
3 let zeghamriExamples = [];
4 let watermarkClassifier;
5 const trainBtn = document.getElementById('trainBtn');
6
7 function preload() {
8   for (let i = 0; i < 100; i++) {
9     wadieExamples[i] = loadImage(`../example_generator/sketch_220224a/data/wadiemendja${i}.png`);
10    dahamExamples[i] = loadImage(`../example_generator/sketch_220224a/data/dahama${i}.png`);
11    zeghamriExamples[i] = loadImage(`../example_generator/sketch_220224a/data/zeghamris${i}.png`);
12  }
13 }
14
15 function setup() { }
16
17 trainBtn.addEventListener('click', () => {
18   trainBtn.disabled = true;
19   console.log("Training started !");
20   setTimeout(train, 100);
21 });
22
23 function train() {
24   watermarkClassifier = ml5.neuralNetwork({
25     inputs: [64, 64, 4],
26     task: "imageClassification",
27     debug: true
28   });
29   for (let i = 0; i < wadieExamples.length; i++) {
30     watermarkClassifier.addData({ image: wadieExamples[i] }, { label: "Wadie Mendja" });
31     watermarkClassifier.addData({ image: dahamExamples[i] }, { label: "Daham A" });
32     watermarkClassifier.addData({ image: zeghamriExamples[i] }, { label: "Zeghamri Salah" });
33   }
34   watermarkClassifier.normalizeData();
35   watermarkClassifier.train({ epochs: 500 }, () => {
36     watermarkClassifier.save();
37     console.log("Finished training");
38   });
39   trainBtn.disabled = false;
40 }
```

Training performance:



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. [3]

1.6) Watermark extraction:

The extraction phase is divided into two steps:

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

$$NC(i) = AN / TNN \dots\dots (Eq. 2)$$

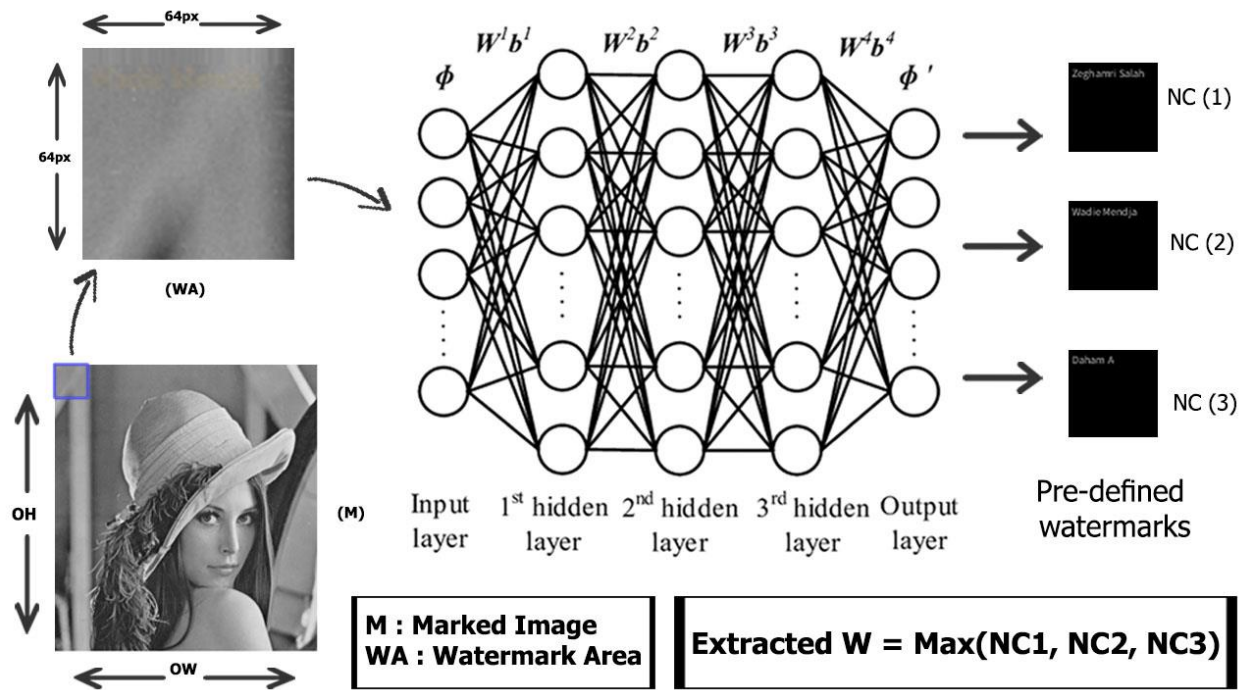
AN: activated neurons

TNN: total number of neurons

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

$$Err(i) = 1 - NC(i) \dots\dots (Eq. 3)$$

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.



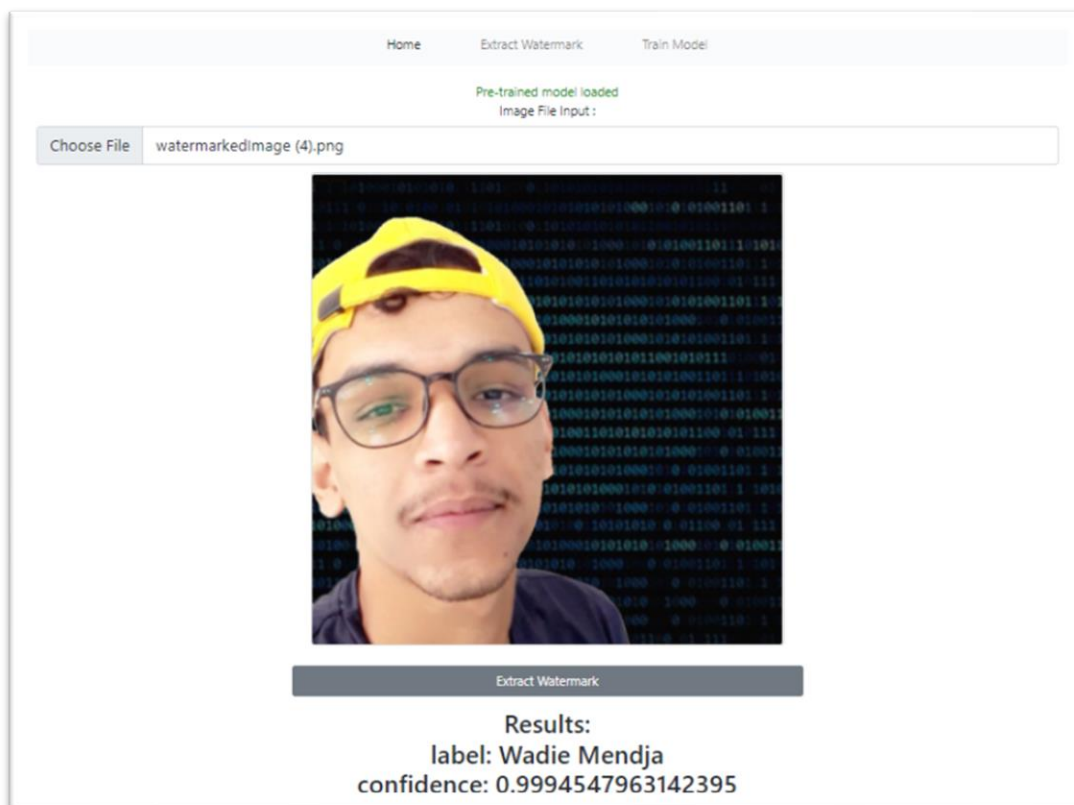
Extraction Process

1.7) 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:

```
46 function extractWatermark() {
47   const canvasImageUrl = canvas.toDataURL();
48   testImage.src = canvasImageUrl;
49   testImage.onload = () => {
50     watermarkClassifier.classify({ image: testImage }, (err, results) => {
51       if (err)
52         console.log(err)
53       else {
54         const label = results[0].label;
55         const confidence = results[0].confidence;
56         if (confidence >= 0.85)
57           resultsDiv.innerHTML = `Results:<br>label: ${label}<br>confidence: ${confidence}`;
58         else resultsDiv.innerHTML = "No watermark detected !";
59         console.log(results);
60         extractWatermarkBtn.disabled = false;
61       }
62     });
63   }
64 }
```

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:



1.7) 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:

$$MSE = \frac{\sum_{M,N} [I_1(m,n) - I_2(m,n)]^2}{M * N}$$

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:

$$PSNR = 10 \log_{10} \left(\frac{R^2}{MSE} \right)$$

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. [4]

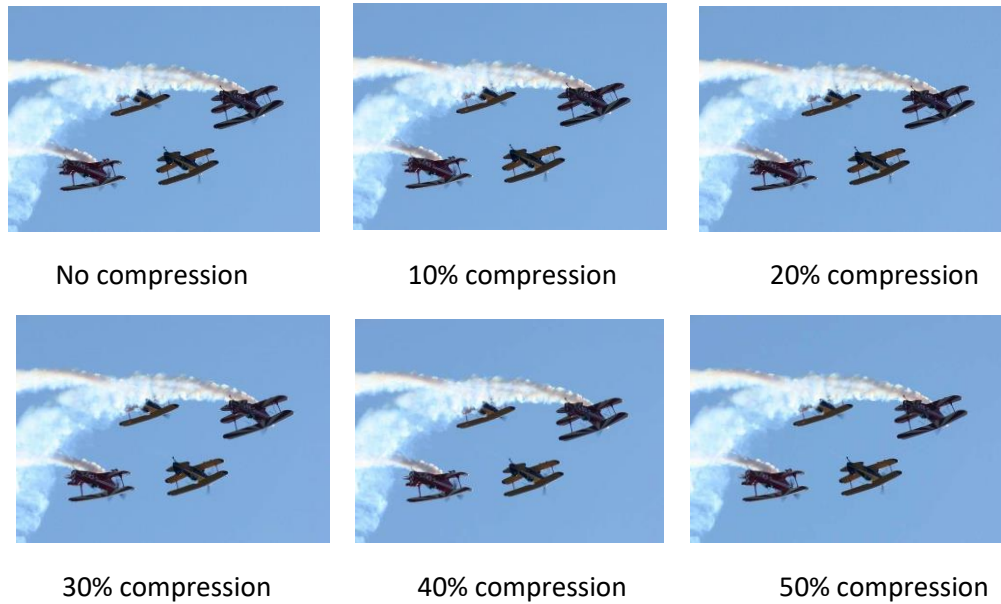
For these particular experiments we're setting the watermark strength (α) to 0.1.

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. [5]

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



Results:

Compression level (%)	Accuracy (%)	PSNR (dB)	Output
0 (No compression)	96.22	Infinite	Watermark detected
10	97.45	52.78	Watermark detected
15	94.05	43.61	Watermark detected
20	56.65	43.61	Watermark detected
25	70.54	43.16	Watermark detected
30	65.37	42.39	Watermark detected
35	79.17	41.65	Watermark detected
40	68.92	41.03	Watermark detected
45	48.69	40.24	No watermark detected
50	42.33	39.81	No watermark detected

Table: compression resistant experiment

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

Results:

Sharpening level (%)	Accuracy (%)	Output
Default sharpening	97.43	Watermark detected
100%	99.54	Watermark detected
200%	95.96	Watermark detected
400%	99.97	Watermark detected

Table: 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. [6]



No gaussian blur



gaussian blur: radius 10 pixels

Gaussian blur radius (pixels)	Accuracy (%)	Output
None	83.23	Watermark detected
0.5	81.28	Watermark detected
1	81.36	Watermark detected
2	69.82	Watermark detected
3	72.58	Watermark detected
4	42.02	No watermark detected
5	41.00	No watermark detected

Table: Gaussian blur distortion experiment

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

I.8) Conclusion:

Although we got great results on the experiments that we applied, however there are some cases where 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.

[1] : Zhiyu Wang et al 2020 J. Phys.: Conf. Ser. 1693 012017

[2] : <https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/>

[3] : <https://radiopaedia.org/articles/epoch-machine-learning>

[4] : <https://www.mathworks.com/help/vision/ref/psnr.html>

[5] : https://en.wikipedia.org/wiki/Image_compression

[6] : https://en.wikipedia.org/wiki/Gaussian_blur