I) Development process:

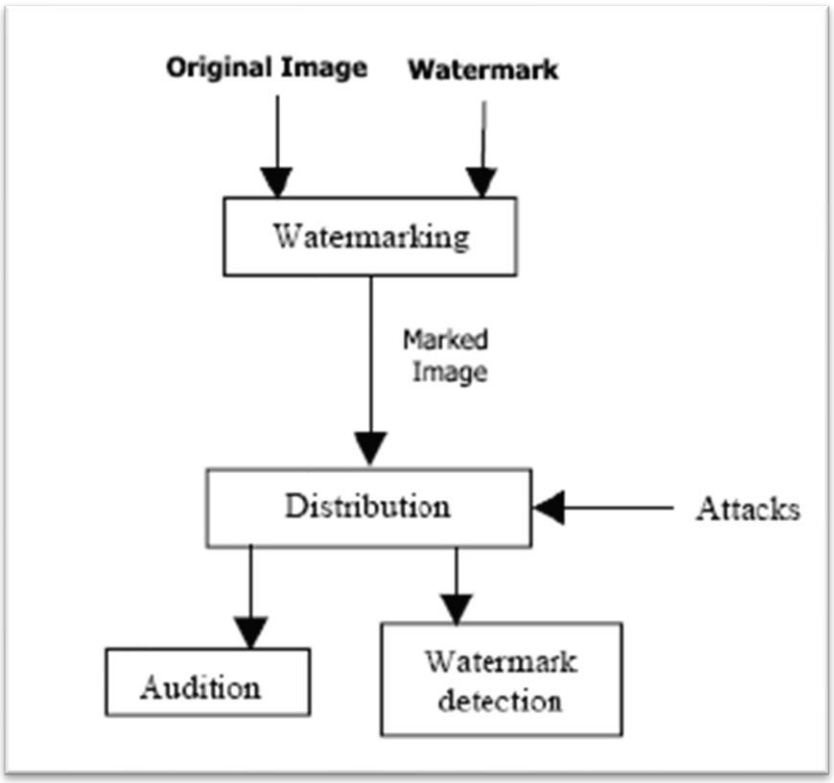
I.1) Concept:

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

I.2) Watermarking an image:

We develop 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.

a) CanvasRenderingContext2D.fillText():

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.

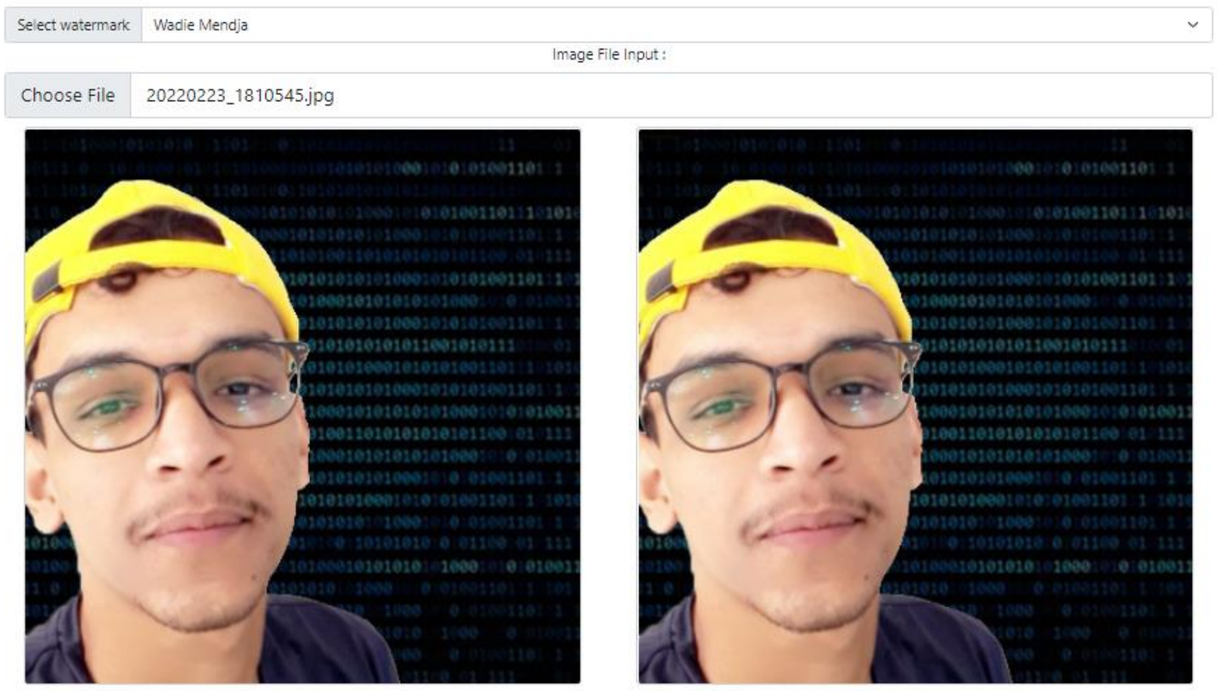
This method draws directly to the canvas without modifying the current path, so any subsequent fill() or stroke() calls will have no effect on it.

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

Syntax:

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

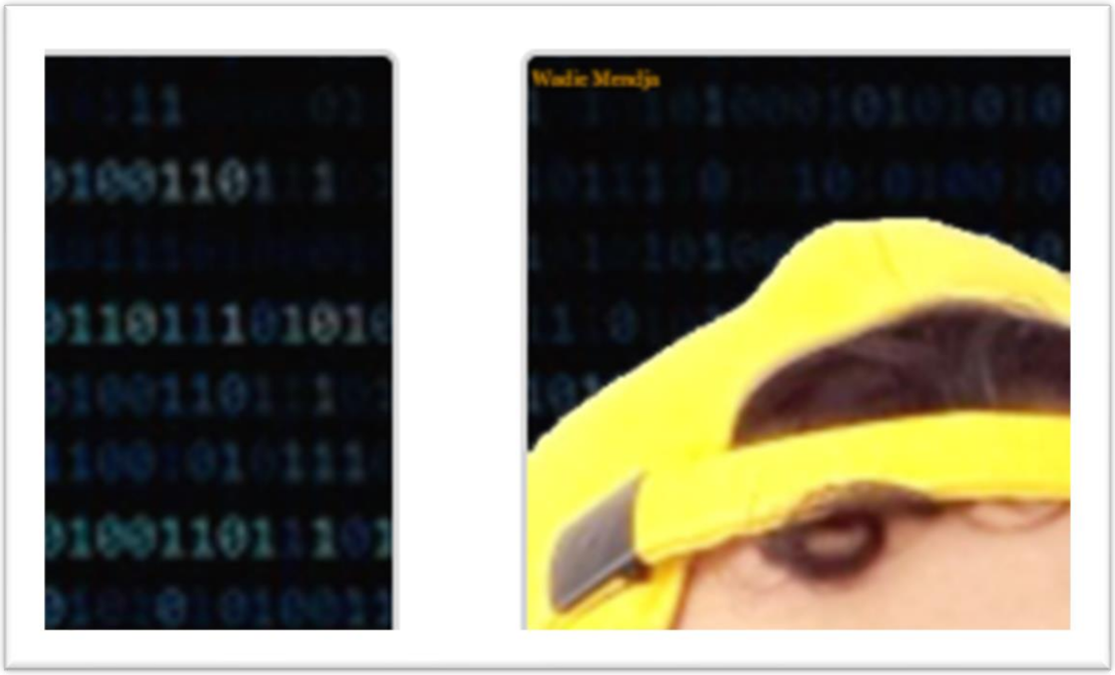
Here is an example of how it works, in the screenshot below we have on the left side the original image and the watermarked image on the right side, the "Select Watermark" text field contains the text that we should embed on the image as a watermark, we did that using the syntax that we talked about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



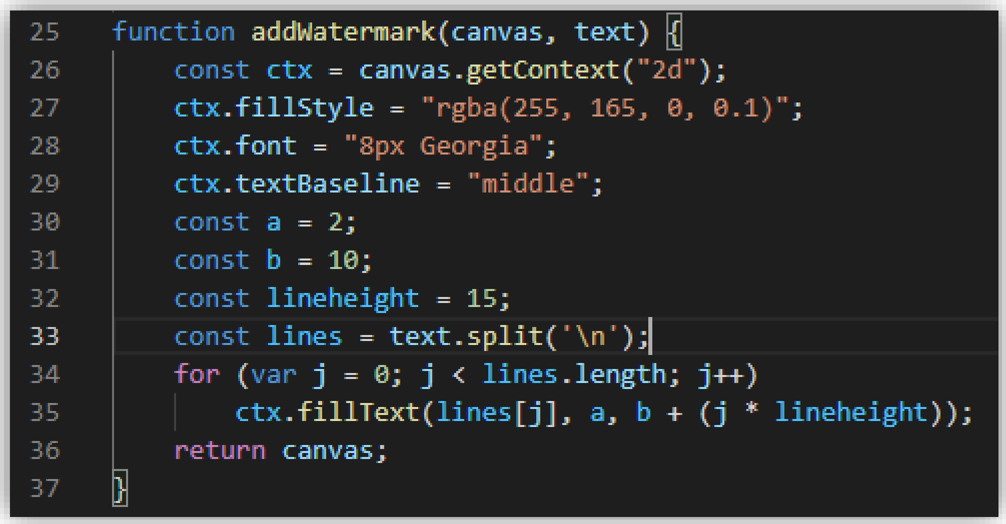
Wadie Mendja

20220223_1810545.jpg

Maybe you didn't notice any difference between the two images and that's because we're embedding an invisible watermark, but if we try to make the watermark visible it's going to look like that:



And here is the function that does all this work:



```
function addWatermark(canvas, text) {
    const ctx = canvas.getContext("2d");
    ctx.fillStyle = "rgba(255, 165, 0, 0.1)";
    ctx.font = "8px Georgia";
    ctx.textBaseline = "middle";
    const a = 2;
    const b = 10;
    const lineheight = 15;
    const lines = text.split('\n');
    for (var j = 0; j < lines.length; j++)
        ctx.fillText(lines[j], a, b + (j * lineheight));
    return canvas;
}
```

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.

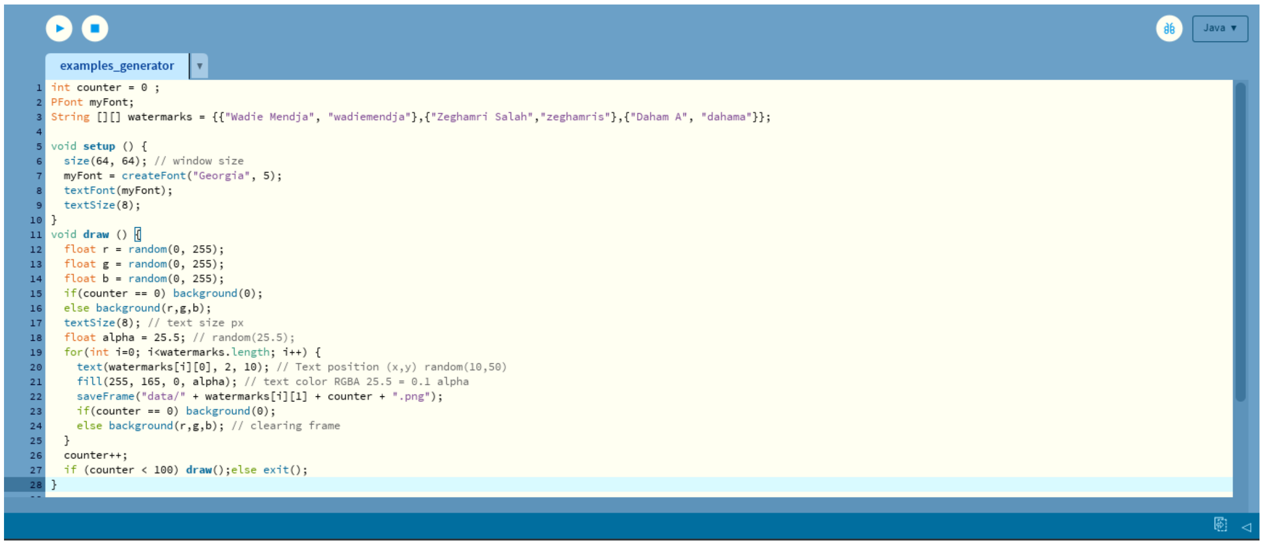
I.3) Generating a dataset:

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:

Wadie Mendja

Zeghamri Salah

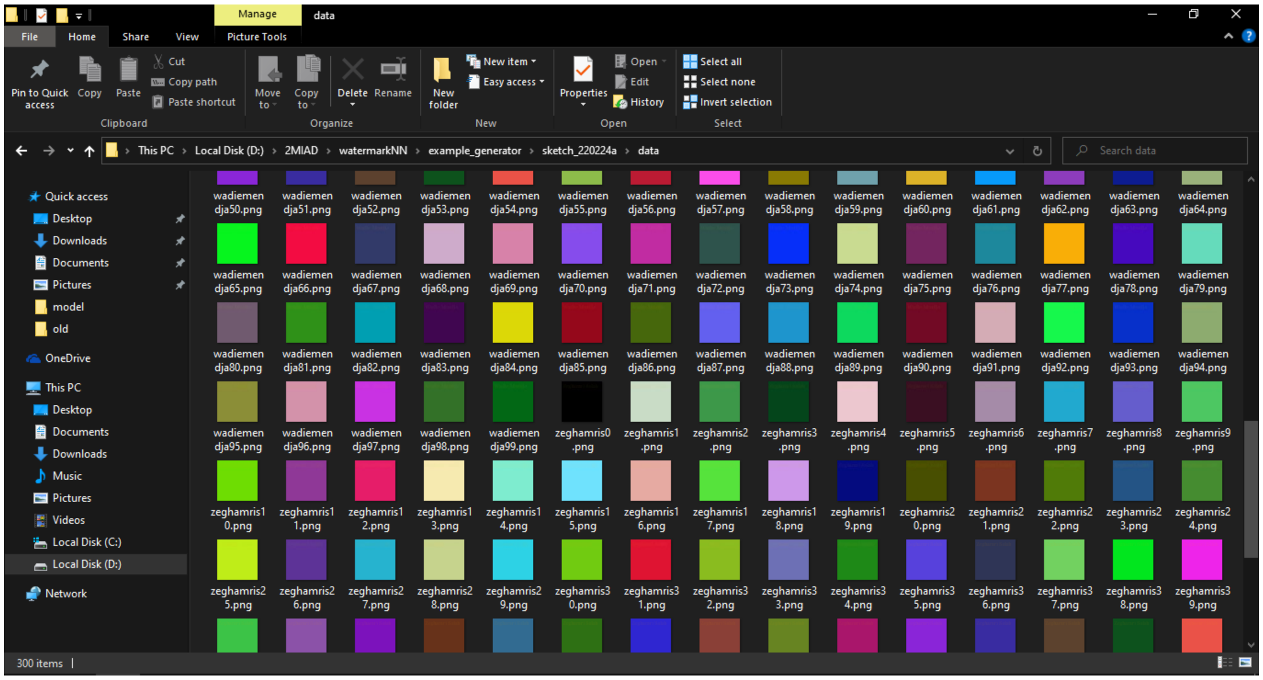
Those are the original watermarks, now we need to images similar to them by changing the text opacity, position and background color, so for this task we’re using a programming language called processing which is a graphical library and integrated development environment built for the electronic arts 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:



```
int counter = 0 ;
PFont myFont;
String [][] watermarks = {{"Wadie Mendja", "wadiemendja"},{"Zeghamri Salah","zeghamris"},{"Daham A", "dahama"}};

void setup () {
  size(64, 64); // window size
  myFont = createFont("Georgia", 5);
  textFont(myFont);
  textSize(8);
}
void draw () {
  float r = random(0, 255);
  float g = random(0, 255);
  float b = random(0, 255);
  if(counter == 0) background(0);
  else background(r,g,b);
  textSize(8); // text size px
  float alpha = 25.5; // random(25.5);
  for(int i=0; i<watermarks.length; i++) {
    text(watermarks[i][0], 2, 10); // Text position (x,y) random(10,50)
    fill(255, 165, 0, alpha); // text color RGBA 25.5 = 0.1 alpha
    saveFrame("data/" + watermarks[i][1] + counter + ".png");
    if(counter == 0) background(0);
    else background(r,g,b); // clearing frame
  }
  counter++;
  if (counter < 100) draw();else exit();
}
```

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:



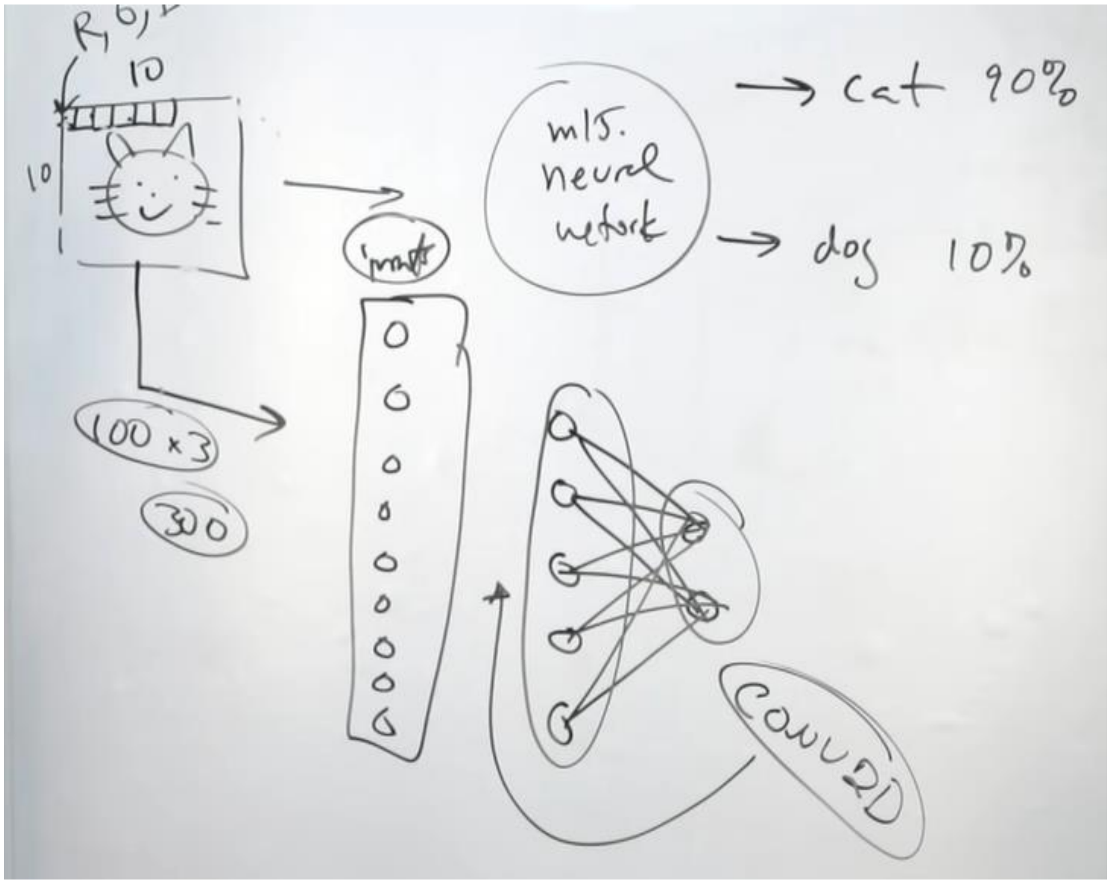
This PC > Local Disk (D:) > 2MIAD > watermarkNN > example_generator > sketch_220224a > data

```
wadiemendja50.png wadiemendja51.png wadiemendja52.png wadiemendja53.png wadiemendja54.png wadiemendja55.png wadiemendja56.png wadiemendja57.png wadiemendja58.png wadiemendja59.png wadiemendja60.png wadiemendja61.png wadiemendja62.png wadiemendja63.png wadiemendja64.png
wadiemendja65.png wadiemendja66.png wadiemendja67.png wadiemendja68.png wadiemendja69.png wadiemendja70.png wadiemendja71.png wadiemendja72.png wadiemendja73.png wadiemendja74.png wadiemendja75.png wadiemendja76.png wadiemendja77.png wadiemendja78.png wadiemendja79.png
wadiemendja80.png wadiemendja81.png wadiemendja82.png wadiemendja83.png wadiemendja84.png wadiemendja85.png wadiemendja86.png wadiemendja87.png wadiemendja88.png wadiemendja89.png wadiemendja90.png wadiemendja91.png wadiemendja92.png wadiemendja93.png wadiemendja94.png
wadiemendja95.png wadiemendja96.png wadiemendja97.png wadiemendja98.png wadiemendja99.png zeghamris0.png zeghamris1.png zeghamris2.png zeghamris3.png zeghamris4.png zeghamris5.png zeghamris6.png zeghamris7.png zeghamris8.png zeghamris9.png
zeghamris10.png zeghamris11.png zeghamris12.png zeghamris13.png zeghamris14.png zeghamris15.png zeghamris16.png zeghamris17.png zeghamris18.png zeghamris19.png zeghamris20.png zeghamris21.png zeghamris22.png zeghamris23.png zeghamris24.png
zeghamris25.png zeghamris26.png zeghamris27.png zeghamris28.png zeghamris29.png zeghamris30.png zeghamris31.png zeghamris32.png zeghamris33.png zeghamris34.png zeghamris35.png zeghamris36.png zeghamris37.png zeghamris38.png zeghamris39.png
```

300 items

I.4) Training the dataset:

Our NN is going to take each image's individual pixel (RGBA) value as an input to the first layer of our NN, the example below shows a classification NN that tell the difference between a dog and a cat:



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 4: add data to the neural network

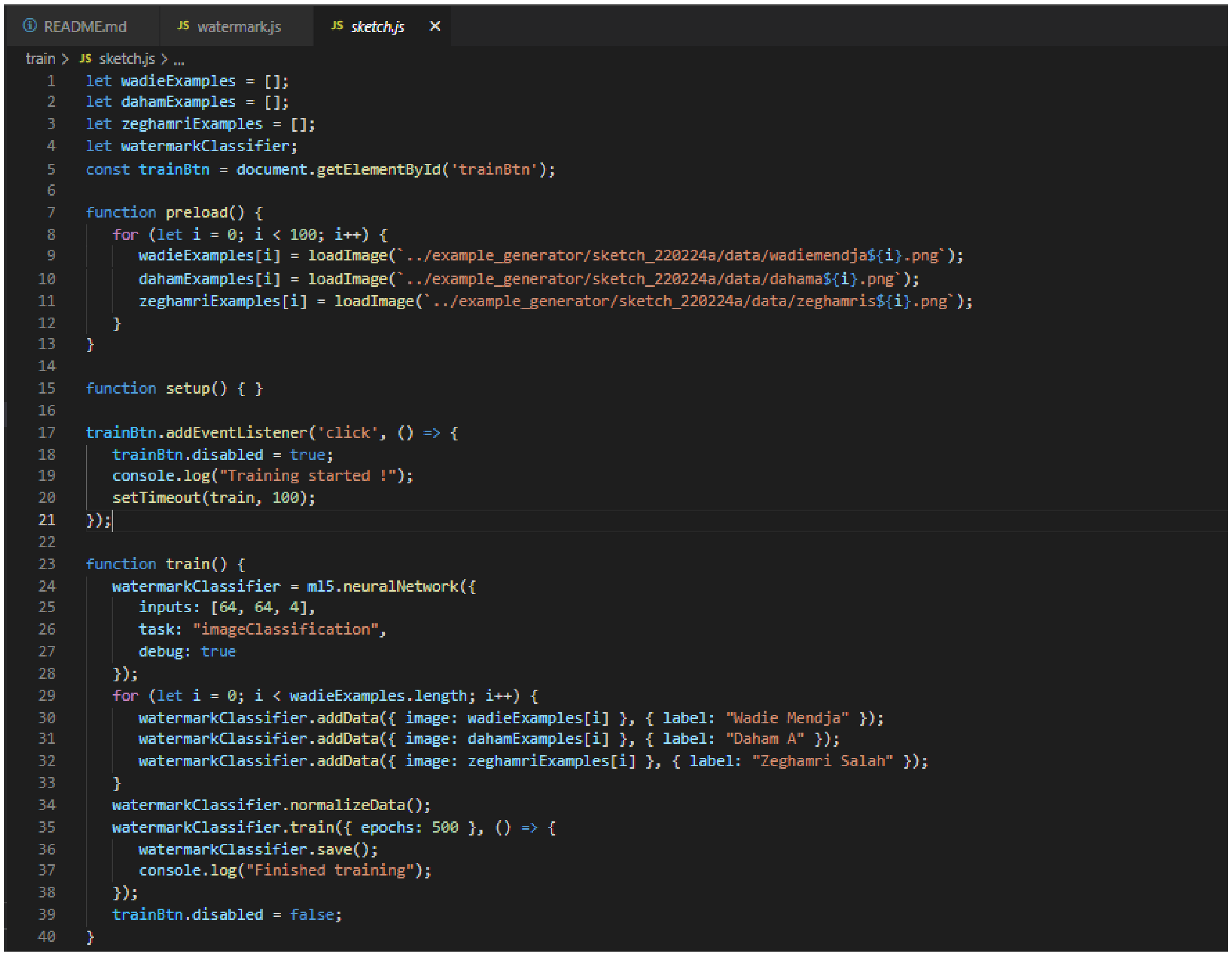
Step 5: Normalize your data

Step 6: Train your neural network

Step 7: use the trained model to make a classification

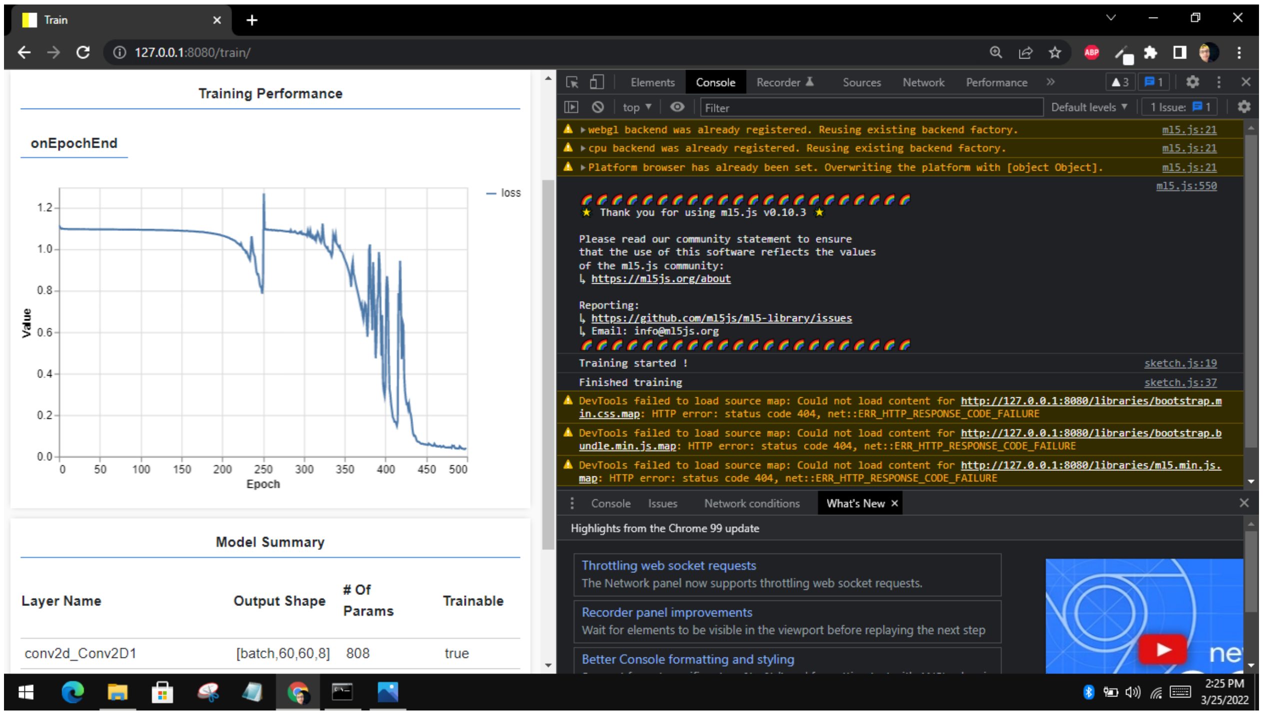
Step 8: do something with the results

Our dataset (100 sample for each watermark) will be trained with 50 epoch (stopping criteria) using sigmoid activation function, here is the code that does the training process:



```javascript
let wadieExamples = [];
let dahamExamples = [];
let zeghamriExamples = [];
let watermarkClassifier;
const trainBtn = document.getElementById('trainBtn');

function preload() {
    for (let i = 0; i < 100; i++) {
        wadieExamples[i] = loadImage(`../example_generator/sketch_220224a/data/wadiemendja${i}.png`);
        dahamExamples[i] = loadImage(`../example_generator/sketch_220224a/data/dahama${i}.png`);
        zeghamriExamples[i] = loadImage(`../example_generator/sketch_220224a/data/zeghamris${i}.png`);
    }
}

function setup() { }

trainBtn.addEventListener('click', () => {
    trainBtn.disabled = true;
    console.log("Training started !");
    setTimeout(train, 100);
});

function train() {
    watermarkClassifier = ml5.neuralNetwork({
        inputs: [64, 64, 4],
        task: "imageClassification",
        debug: true
    });
    for (let i = 0; i < wadieExamples.length; i++) {
        watermarkClassifier.addData({ image: wadieExamples[i] }, { label: "Wadie Mendja" });
        watermarkClassifier.addData({ image: dahamExamples[i] }, { label: "Daham A" });
        watermarkClassifier.addData({ image: zeghamriExamples[i] }, { label: "Zeghamri Salah" });
    }
    watermarkClassifier.normalizeData();
    watermarkClassifier.train({ epochs: 500 }, () => {
        watermarkClassifier.save();
        console.log("Finished training");
    });
    trainBtn.disabled = false;
}
```

Training performance:


Train
127.0.0.1:8080/train/
Training Performance
onEpochEnd
loss
Value
Epoch
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
350
400
450
500
Elements
Console
Recorder
Sources
Network
Performance
top
Filter
Default levels
1 Issue: 1
What's New
Highlights from the Chrome 99 update
Throttling web socket requests
The Network panel now supports throttling web socket requests.
Recorder panel improvements
Wait for elements to be visible in the viewport before replaying the next step
Better Console formatting and styling
2:25 PM
3/25/2022

```
▸webgl backend was already registered. Reusing existing backend factory.    m15.js:21
▸cpu backend was already registered. Reusing existing backend factory.    m15.js:21
▸Platform browser has already been set. Overwriting the platform with [object Object].    m15.js:21
                                                                          m15.js:550
🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈
⭐ Thank you for using ml5.js v0.10.3 ⭐

Please read our community statement to ensure
that the use of this software reflects the values
of the ml5.js community:
↳ https://ml5js.org/about

Reporting:
↳ https://github.com/ml5js/ml5-library/issues
↳ Email: info@ml5js.org
🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈🌈
Training started !                                                        sketch.js:19
Finished training                                                         sketch.js:37
DevTools failed to load source map: Could not load content for http://127.0.0.1:8080/libraries/bootstrap.m
in.css.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
DevTools failed to load source map: Could not load content for http://127.0.0.1:8080/libraries/bootstrap.b
undle.min.js.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
DevTools failed to load source map: Could not load content for http://127.0.0.1:8080/libraries/ml5.min.js.
map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
```

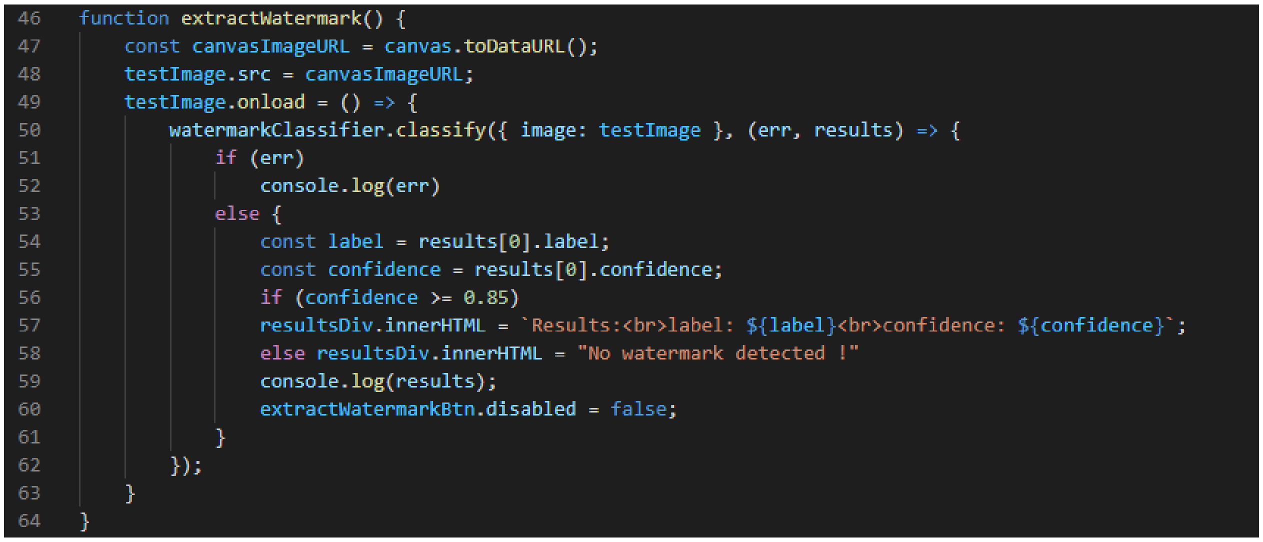
Model Summary

Layer Name	Output Shape	# Of Params	Trainable
conv2d_Conv2D1	[batch,60,60,8]	808	true

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.

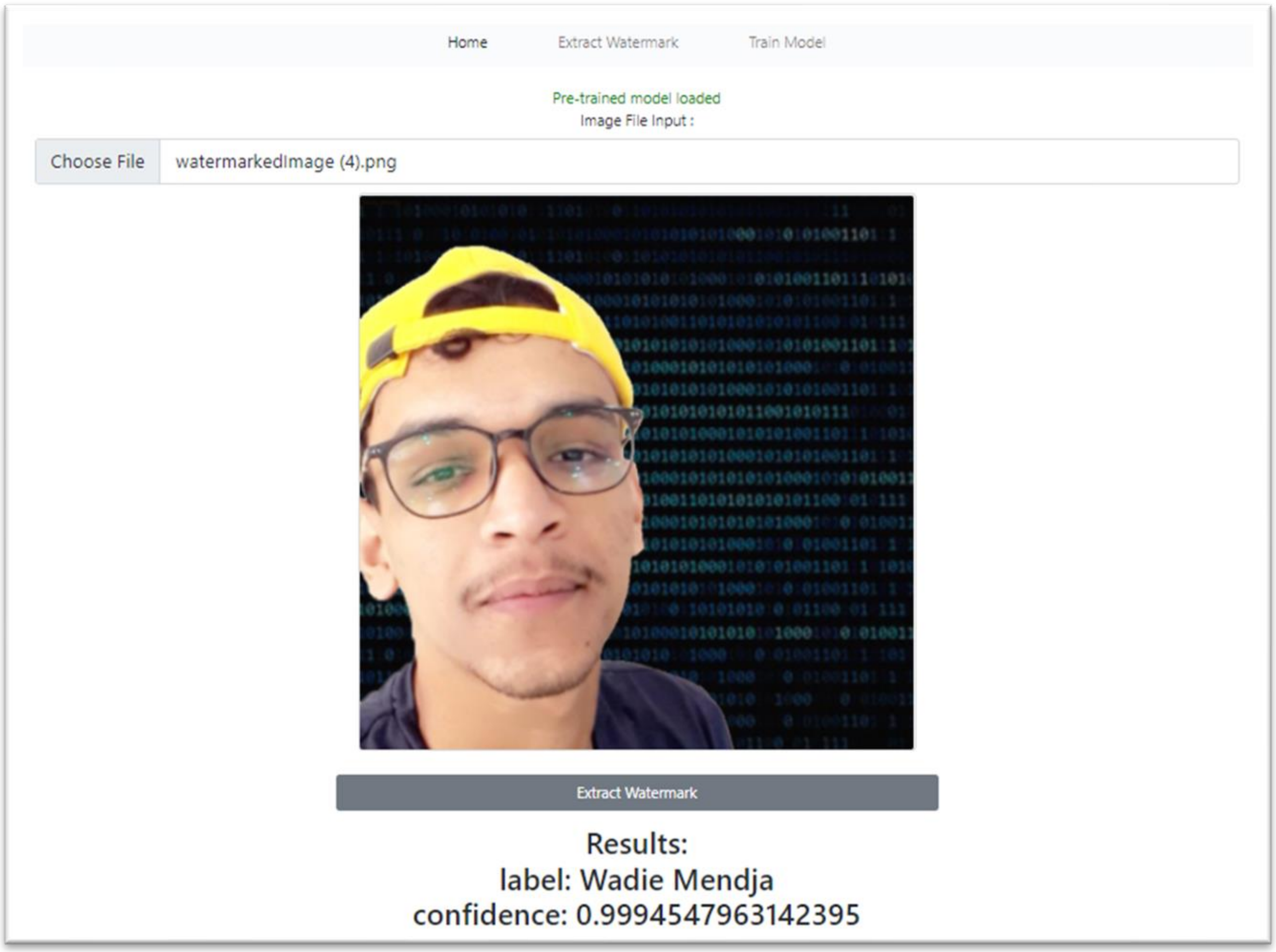
I.5) Watermark extraction:

After we'd finished training our NN and got our model files (weights and meta data files), 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:



```javascript
function extractWatermark() {
    const canvasImageURL = canvas.toDataURL();
    testImage.src = canvasImageURL;
    testImage.onload = () => {
        watermarkClassifier.classify({ image: testImage }, (err, results) => {
            if (err)
                console.log(err)
            else {
                const label = results[0].label;
                const confidence = results[0].confidence;
                if (confidence >= 0.85)
                resultsDiv.innerHTML = `Results:<br>label: ${label}<br>confidence: ${confidence}`;
                else resultsDiv.innerHTML = "No watermark detected !"
                console.log(results);
                extractWatermarkBtn.disabled = false;
            }
        });
    }
}
```

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:



Pre-trained model loaded
Image File Input :
watermarkedImage (4).png

Results:
label: Wadie Mendja
confidence: 0.9994547963142395

I.6) Conclusion:

Although we got great results on the example above that we applied, however there are some cases were the image get damaged by some attack or compression algorithms in a way that even a well-trained neural network cannot be able to detect the embedded watermark or even if it did it will be with a low confidence score 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.