

Algorithmic
Intelligence
The Joy
of Coding
Workbook #2
Creating more
Models

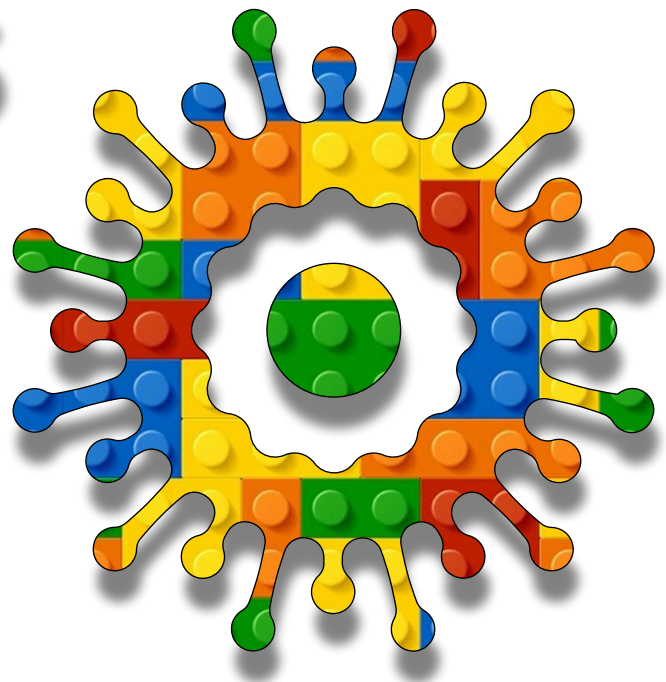




Table of Contents

Module A Unit #7: mouse gesture with ml5.js	6
Sketch A7.1 our starting sketch	7
The data	8
Sketch A7.2 adding the data	9
Sketch A7.3 building the model	11
Sketch A7.4 adding the data training the model	12
Sketch A7.5 training the model	14
Sketch A7.6 epochs	16
Sketch A7.7 mouse vectors	19
Sketch A7.8 classifying the mouse	21
Sketch A7.9 getting the results	24
Sketch A7.10 displaying the results	27
Module A Unit #8: p5.js code snippets #2	32
Sketch A8.1 strings	33
Sketch A8.2 adding two strings	35
Sketch A8.3 strings as variables	36
Sketch A8.4 the keyPressed() function	38
What is the difference between == versus ===	40
Sketch A8.5 the colour color() function	41
Sketch A8.6 sliders	43
Sketch A8.7 another shape, the square	46
Sketch A8.8 the centre of the square	48
Introduction to using the video input	50
Sketch A8.9 video capture	51
Sketch A8.10 returning the canvas	53
Sketch A8.11 video on the canvas	54
Sketch A8.12 hide the video	56
Sketch A8.13 flipping the video	58
Introduction to the pixels	60
Sketch A8.14 new starting sketch	61
Sketch A8.15 pixel array formula	62
Sketch A8.16 changing the pixels	64
Sketch A8.17 orange pixels	66
Sketch A8.18 return of the video	68
Sketch A8.19 pixelating the image	70
Sketch A8.20 brightness grey scale	73
Module A Unit #9: cluster classification with ml5.js	76
The index.html file	77
Sketch A9.1 starting sketch	78
Sketch A9.2 creating the model	79
Sketch A9.3 adding the options	80
Sketch A9.4 training data	81
Sketch A9.5 target label	83

Sketch A9.6 collect training data	85
Sketch A9.7 button	87
Sketch A9.8 training	90
Sketch A9.9 normalisation	93
Sketch A9.10 adding a bit of colour	96
Sketch A9.11 a state of mind	99
Sketch A9.12 classifying	102
Module A Unit #10: cluster regression	107
The index.html file	108
Sketch A10.1 basic starting model	109
Sketch A10.2 assigning a value	110
Sketch A10.3 assigning the keys	111
Sketch A10.4 drawing the circles	112
Sketch A10.5 defining the inputs	114
Sketch A10.6 collecting the data	116
Sketch A10.7 training button	118
Sketch A10.8 training	121
Sketch A10.9 predicting	124
Sketch A10.10 prediction results	127
Sketch A10.11 some refinements	131
Module A Unit #11: colour predictor	136
The index.html	137
Sketch A11.1 creating the data	138
Sketch A11.2 Sliders	139
Sketch A11.3 adding the neural network	141
Sketch A11.4 adding the data	143
Sketch A11.5 normalising the data	145
Sketch A11.6 training the neural network	147
Sketch A11.7 classifying the colour	150
Sketch A11.8 the best prediction	153
Module A Unit #12: pixel predictor	159
The index.html	160
Sketch A12.1 starting sketch	161
Sketch A12.2 hiding the video	163
Sketch A12.3 mirrored	164
Sketch A12.4 video ready	166
Sketch A12.5 the size of the video	167
Sketch A12.6 resizing the video	169
Sketch A12.7 even smaller	171
Sketch A12.8 loading the pixels	174
Sketch A12.9 enlarged pixels	176
Sketch A12.10 neural network	178
Sketch A12.11 adding the slider output	180
Sketch A12.12 adding the buttons	183
Sketch A12.13 the training button	186

Sketch A12.14 getting the inputs	189
Sketch A12.15 adding an example	192
Sketch A12.16 training the model	195
Sketch A12.17 finished training	198
Sketch A12.18 predicting the movement	201
Sketch A12.19 predict ready	204
Module A Unit #13: saving and loading	211
Part #1: saving the data	212
Sketch A13.1 this is our starting sketch	213
Sketch A13.2 we need a button to press	215
Sketch A13.3 adding a function	217
Part #2: loading the data	220
Sketch A13.4 loading the file	221
Sketch A13.5 checking the size	223
Sketch A13.6 drawing the points	225
Part #3: saving the model	227
Sketch A13.7 we start again	229
Sketch A13.8 loading the data	231
Sketch A13.9 adding the data and drawing	233
Sketch A13.10 creating another button	236
Part #4: loading the model	239
Sketch A13.11 loading the model	240
Sketch A13.12 a callback	241
Sketch A13.13 predicting	242
Sketch A13.14 comparing	244
Cluster example Module A Unit #9	247
Additional notes	250

Algorithmic Intelligence Module A Unit #7

mouse gesture classification



Module A Unit #7: mouse gesture with ml5.js

The two examples so far have been **regression** tasks, so now we need a **classification** task. We will try to identify which way the mouse is moving, either up, down, left, or right. First, we will create some synthetic data to train the model on.

The model is then trained on this data over a number of epochs until we are happy with the result. We are going to be careful that there is no underfitting or overfitting.

When the model is trained, we can test it by moving the mouse in each direction to see how well it performs. Remember that this is a relatively simple example and has many drawbacks and omissions, but it demonstrates a simple **classification** task.

! keep the index.html file as it was for the previous three units. We will be using ml5.js as before, so make sure that you have the line of code in the index.html file.



Sketch A7.1 our starting sketch

We start with our basic sketch.

```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
}
```



The data

We are using synthetic data once again. This time, rather than generating it, we will hard-code it. You will notice that it is an **array** of **objects**. Each object has three elements:

- A) the x component of a vector,
- B) the y component of the vector, and
- C) the label indicating which direction it is going in.

You may notice that the units are between **0** and **1**. We have effectively normalised the data already, so we have no need to do it again. We have two lots of data for each movement: **left**, **right**, **up**, and **down**. This is a very small dataset, but we will see how well it performs once we start training it.

Each bit of data is a vector, which is the amount the mouse has moved from the relative position of (**0**, **0**).



Sketch A7.2 adding the data

Add in the data as shown below. I have kept it very simple and very obvious. Either **+1**, **-1**, **+0.1**, **-0.1** depending on the relevant direction.

```
let data = [  
  {x: 1, y: 0.1, label: "right"},  
  {x: 1, y: -0.1, label: "right"},  
  {x: -1, y: 0.1, label: "left"},  
  {x: -1, y: -0.1, label: "left"},  
  {x: 0.1, y: 1, label: "down"},  
  {x: -0.1, y: 1, label: "down"},  
  {x: 0.1, y: -1, label: "up"},  
  {x: -0.1, y: -1, label: "up"}  
]
```

```
function setup()  
{  
  createCanvas(400, 400)  
}  
  
function draw()  
{  
  background(220)  
}
```

Notes

I hope this seems fairly straightforward. The format is for a JSON-type array.

Challenges

1. You could adjust some of the values, making them more random, so that they are not so obvious.
2. Increase the size of the dataset.
3. You could think about how you would collect and then save the data to be loaded (maybe for another time).

Code Explanation

<code>let data = [. . .]</code>	Create an array of objects
<code>{x: 1, y: 0.1, label: "right"},</code>	This is an object with two vectors and a label, this moves the coordinates to the right and slightly downwards
<code>{x: -0.1, y: -1, label: "up"}</code>	This is an object with two vectors and a label, this moves the coordinates slightly to the left and upwards



Sketch A7.3 building the model

You should be familiar with building the model now. We are going to create a neural network model and call it `nn`. We will give the neural network the following options:

- It is a **classification** task.
- Set debug to **true**, which will show the progress of the training (you can set it to **false** later).

```
let nn

let data = [
  { x: 1, y: 0.1, label: "right"},
  { x: 1, y: -0.1, label: "right"},
  { x: -1, y: 0.1, label: "left"},
  { x: -1, y: -0.1, label: "left"},
  { x: 0.1, y: 1, label: "down"},
  { x: -0.1, y: 1, label: "down"},
  { x: 0.1, y: -1, label: "up"},
  { x: -0.1, y: -1, label: "up"}
]

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: "classification",
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}

function draw()
{
  background(220)
}
```

Notes

We are building the model just as we have done previously.



Sketch A7.4 adding the data training the model

The `for()` loop (`let item of data`) will pull all the datapoints in the `data` array into another array called `items`. We can then create an array of inputs based on the `x` and `y` values. The output array can then collect all the labels that correspond with those input vectors. We then add this dataset to the neural network model `nn.addData(inputs, outputs)`.

```
let nn
let data = [
  { x: 1, y: 0.1, label: "right"},
  { x: 1, y: -0.1, label: "right"},
  { x: -1, y: 0.1, label: "left"},
  { x: -1, y: -0.1, label: "left"},
  { x: 0.1, y: 1, label: "down"},
  { x: -0.1, y: 1, label: "down"},
  { x: 0.1, y: -1, label: "up"},
  { x: -0.1, y: -1, label: "up"}
]

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: "classification",
    debug: true
  }
  nn = ml5.neuralNetwork(options)
  for (let item of data)
  {
    let inputs = [item.x, item.y]
    let outputs = [item.label]
    nn.addData(inputs, outputs)
  }
}

function draw()
{
  background(220)
```

```
}
```

Notes

It pulls all the `x`, `y` and `label` values from the array and adds this data to the neural network model (`nn`). This is similar to the `regression` tasks, but there is a slightly different approach for `classification` tasks. No need to normalise the data (already done).



Sketch A7.5 training the model

After we have added the data, we are going to train it with `nn.train()`, with a callback `finishedTraining` which will let us know when it has finished. The callback is a function; it will help us to keep track of what is happening. To help us at this stage, we will console log the `status`, which is a boolean variable. The default is `training`, and when it has finished training, the `status` will change to `ready`.

```
let nn
let status = "training"
let data = [
  { x: 1, y: 0.1, label: "right"},
  { x: 1, y: -0.1, label: "right"},
  { x: -1, y: 0.1, label: "left"},
  { x: -1, y: -0.1, label: "left"},
  { x: 0.1, y: 1, label: "down"},
  { x: -0.1, y: 1, label: "down"},
  { x: 0.1, y: -1, label: "up"},
  { x: -0.1, y: -1, label: "up"}
]

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: "classification",
    debug: true
  }
  nn = ml5.neuralNetwork(options)
  for (let item of data)
  {
    let inputs = [item.x, item.y]
    let outputs = [item.label]
    nn.addData(inputs, outputs)
  }
  nn.train(finishedTraining)
}

function finishedTraining()
```

```

{
  status = "ready"
  console.log(status)
}

function draw()
{
  background(220)
}

```

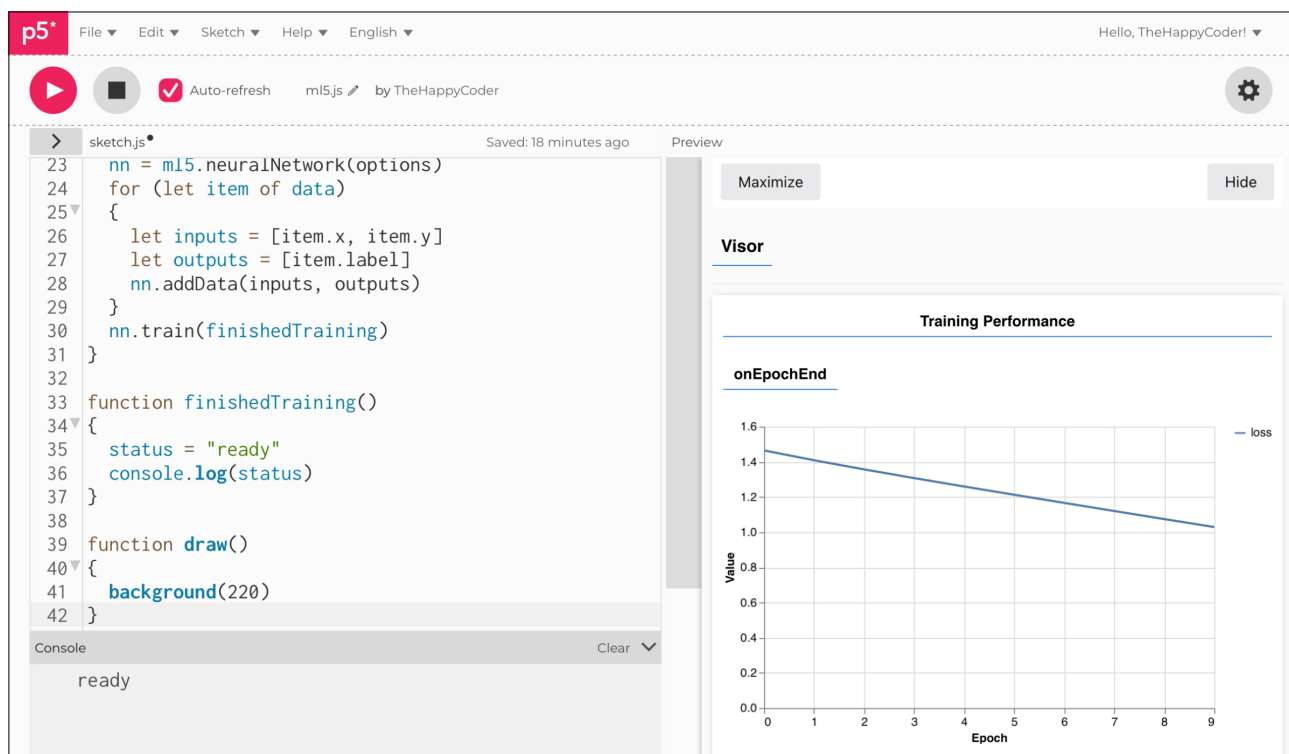
Notes

The **status** should go from **training** to **ready** once it has finished training (you won't see the word training). We also can see that **10** epochs are nowhere near enough, so we need to increase that **hyperparameter**.

Code Explanation

let status = "training"	This is a string variable that is initialised to "training"
status = "ready"	The string variable value is now "ready"

Figure A7.5





Sketch A7.6 epochs

Clearly, the loss function was still going down, so we will try **250** epochs and see if that works. We can just add it straight into the `nn.train()` function, just a shorthand formatting version.

```
let nn
let status = "training"
let data = [
  { x: 1, y: 0.1, label: "right"},
  { x: 1, y: -0.1, label: "right"},
  { x: -1, y: 0.1, label: "left"},
  { x: -1, y: -0.1, label: "left"},
  { x: 0.1, y: 1, label: "down"},
  { x: -0.1, y: 1, label: "down"},
  { x: 0.1, y: -1, label: "up"},
  { x: -0.1, y: -1, label: "up"}
]

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: "classification",
    debug: true
  }
  nn = ml5.neuralNetwork(options)
  for (let item of data)
  {
    let inputs = [item.x, item.y]
    let outputs = [item.label]
    nn.addData(inputs, outputs)
  }
  nn.train({epochs: 250}, finishedTraining)
}

function finishedTraining()
{
```



```
status = "ready"
console.log(status)
}

function draw()
{
  background(220)
}
```

Notes

You will see that it was still going down even after 250 epochs, so it may continue to reduce; however, it is probably overfitting after, say, 100 epochs. The reason for such a high number of epochs compared to our other examples could be that it is a tiny dataset. If you move your mouse over the chart, it gives you the value of the loss function. My effort was 0.009, which is pretty low.

Challenge

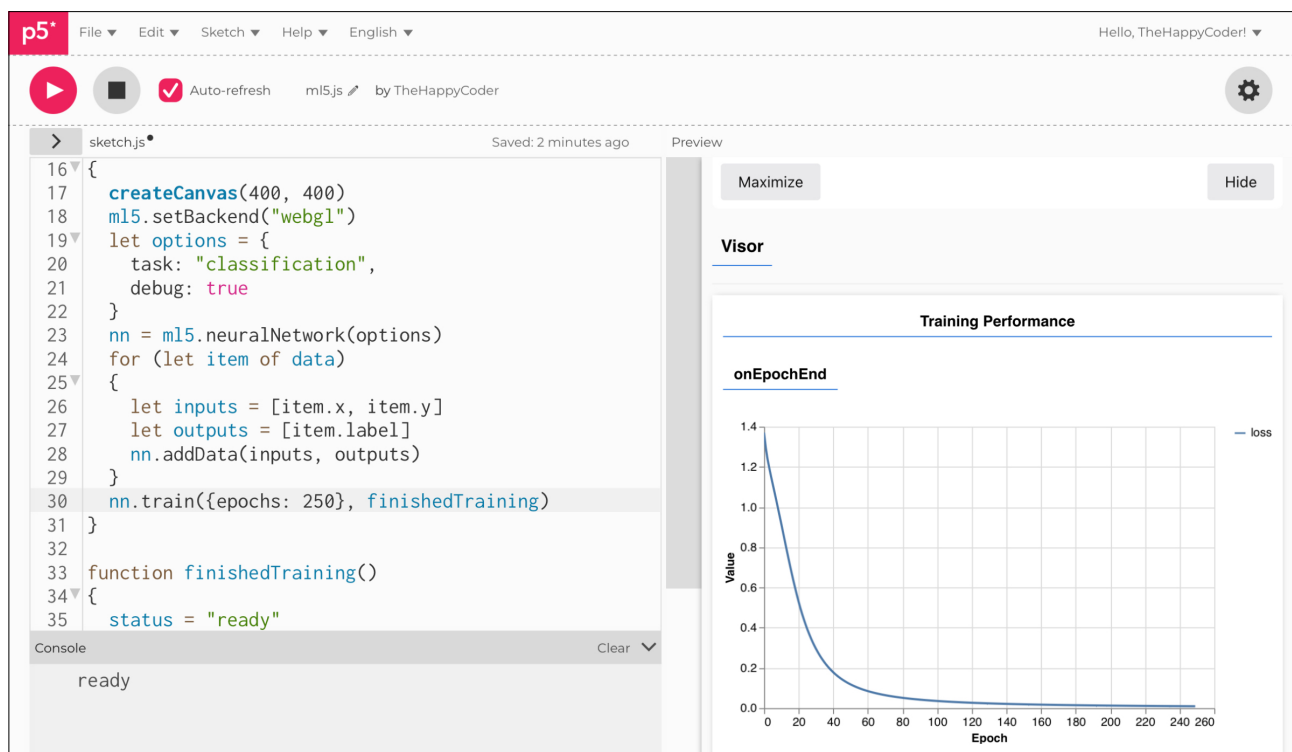
Try an even larger number of epochs.

Code Explanation

```
nn.train({epochs: 250},  
finishedTraining)
```

Specifying the number of epochs within the training function

Figure A7.6





Sketch A7.7 mouse vectors

What we want to do now is move the mouse in such a way that we can use the model to **predict** what movement it has made, either **up**, **down**, **left**, or **right**. So we need two variables for the start and end of the mouse movement. The movement starts when the mouse is clicked and keeps going while it is dragged (and then stops dragging). So we have two vectors for the **start** and for the **end**.

! Suggest making the debug false otherwise it will keep on training every time you enter more lines of code, we know it is working as is anyway.

```
let nn
let status = "training"
let start
let end
let data = [
  { x: 1, y: 0.1, label: "right"},
  { x: 1, y: -0.1, label: "right"},
  { x: -1, y: 0.1, label: "left"},
  { x: -1, y: -0.1, label: "left"},
  { x: 0.1, y: 1, label: "down"},
  { x: -0.1, y: 1, label: "down"},
  { x: 0.1, y: -1, label: "up"},
  { x: -0.1, y: -1, label: "up"}
]

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: "classification",
    debug: false
  }
  nn = ml5.neuralNetwork(options)
  for (let item of data)
  {
    let inputs = [item.x, item.y]
    let outputs = [item.label]
    nn.addData(inputs, outputs)
```

```

    }
    nn.train({epochs: 250}, finishedTraining)
  }

function finishedTraining()
{
  status = "ready"
  console.log(status)
}

function draw()
{
  background(220)
}

function mousePressed()
{
  start = createVector(mouseX, mouseY)
}

function mouseDragged()
{
  end = createVector(mouseX, mouseY)
}

```

Notes

Nothing will happen just yet; we are just collecting data from the mouse.

Code Explanation

<code>start = createVector(mouseX, mouseY)</code>	We create a vector (called start) as soon as we click on the canvas
<code>end = createVector(mouseX, mouseY)</code>	A final vector is created as we drag the mouse across the canvas



Sketch A7.8 classifying the mouse

We create another function to input the data into the model (which has been trained on the synthetic dataset). This function is called when the mouse is released after it has finished dragging. The key elements are described below:

- A) The direction (**dir**) of the movement is done by subtracting the two vectors (**end** and **start**).
- B) We normalise them so their magnitudes are less than **1**.
- C) We then get the **x** and **y** components from the direction (**dir.x**, **dir.y**) vector.
- D) We then have these as our new **inputs** to **classify** as either **up**, **down**, **left**, or **right** and put them into the model.
- E) The **classification** takes two arguments: one is the **inputs** and the other, **gotResults**, is the output.

```
let nn
let status = "training"
let start
let end
let data = [
  { x: 1, y: 0.1, label: "right"},
  { x: 1, y: -0.1, label: "right"},
  { x: -1, y: 0.1, label: "left"},
  { x: -1, y: -0.1, label: "left"},
  { x: 0.1, y: 1, label: "down"},
  { x: -0.1, y: 1, label: "down"},
  { x: 0.1, y: -1, label: "up"},
  { x: -0.1, y: -1, label: "up"}
]

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: "classification",
    debug: false
  }
  nn = ml5.neuralNetwork(options)
  for (let item of data)
  {
```

```

    let inputs = [item.x, item.y]
    let outputs = [item.label]
    nn.addData(inputs, outputs)
  }
  nn.train({epochs: 250}, finishedTraining)
}

function finishedTraining()
{
  status = "ready"
  console.log(status)
}

function draw()
{
  background(220)
}

function mousePressed()
{
  start = createVector(mouseX, mouseY)
}

function mouseDragged()
{
  end = createVector(mouseX, mouseY)
}

function mouseReleased()
{
  let dir = p5.Vector.sub(end, start)
  dir.normalize()
  let inputs = [dir.x, dir.y]
  nn.classify(inputs, gotResults)
}

```

Notes

! Please note you will get a script error if you run this.

When we `classify` the movement of the mouse, we give it the inputs (`dir.x`, `dir.y`) plus a callback. This callback is a function which will carry the result. Next, we need to create a function called, you guessed it, `gotResults()` to make use of the result.

Code Explanation

<code>let dir = p5.Vector.sub(end, start)</code>	We subtract the two vectors, end and start
<code>dir.normalize()</code>	The subtraction of those two vectors (called dir) is normalised to be between 0 and 1
<code>let inputs = [dir.x, dir.y]</code>	The inputs into the classify function are the x and y components of the vector
<code>nn.classify(inputs, gotResults)</code>	We give the inputs to the classify function and also give it a callback (gotResults)



Sketch A7.9 getting the results

As we create the callback function `gotResults()`, we can see how well we are doing by putting the results in the console for now. The `status` changes from `training` to `ready`, and now it is expressed as one of the labels `left`, `right`, `up`, or `down`.

```
let nn
let status = "training"
let start
let end

let data = [
  { x: 1, y: 0.1, label: "right"},
  { x: 1, y: -0.1, label: "right"},
  { x: -1, y: 0.1, label: "left"},
  { x: -1, y: -0.1, label: "left"},
  { x: 0.1, y: 1, label: "down"},
  { x: -0.1, y: 1, label: "down"},
  { x: 0.1, y: -1, label: "up"},
  { x: -0.1, y: -1, label: "up"}
]

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: "classification",
    debug: false
  }
  nn = ml5.neuralNetwork(options)
  for (let item of data)
  {
    let inputs = [item.x, item.y]
    let outputs = [item.label]
    nn.addData(inputs, outputs)
  }
  nn.train({epochs: 250}, finishedTraining)
}
```



```
function finishedTraining()
{
  status = "ready"
  console.log(status)
}

function draw()
{
  background(220)
}

function mousePressed()
{
  start = createVector(mouseX, mouseY)
}

function mouseDragged()
{
  end = createVector(mouseX, mouseY)
}

function mouseReleased()
{
  let dir = p5.Vector.sub(end, start)
  dir.normalize()
  let inputs = [dir.x, dir.y]
  nn.classify(inputs, gotResults)
}

function gotResults(results)
{
  status = results[0].label
  console.log(status)
}
```

Notes

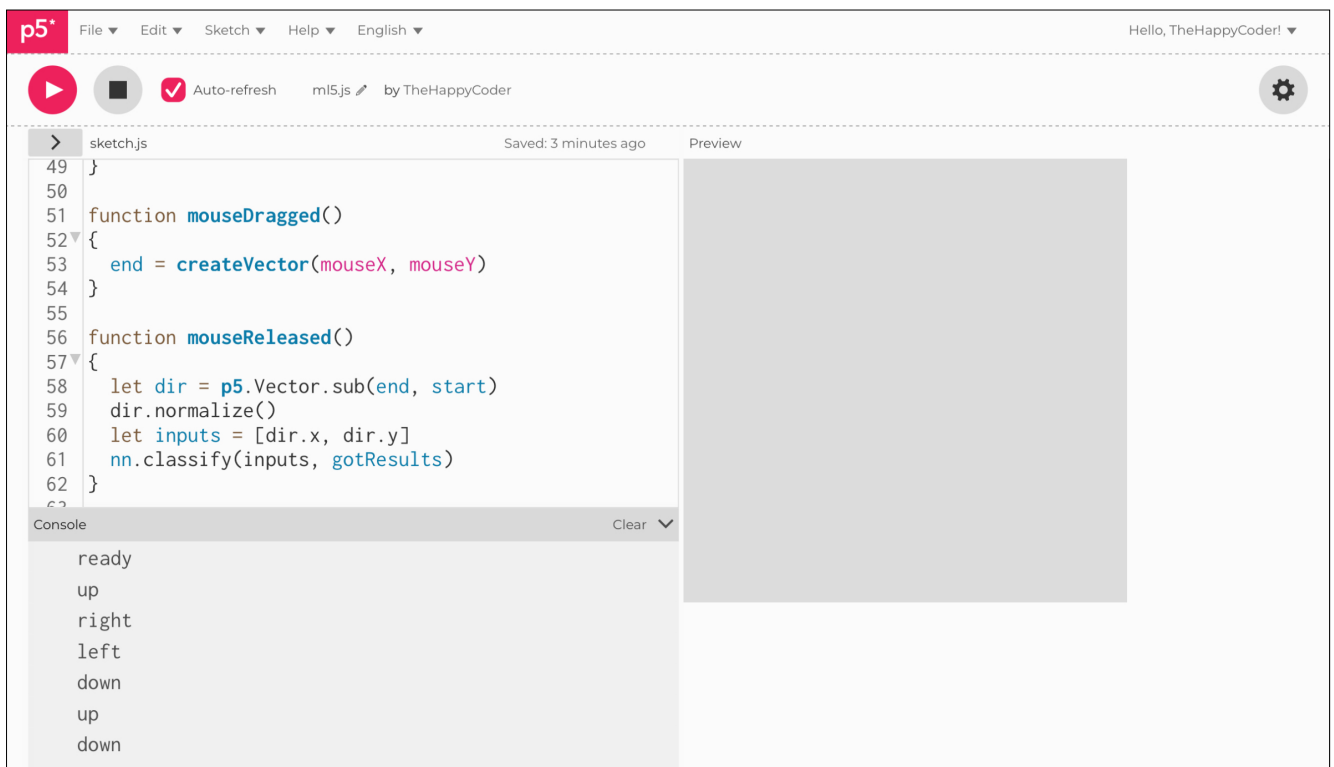
This seems to work quite well. Remember to hold the button down as you move the mouse, and when you release the button, you should get the correct movement.

Code Explanation

```
status = results[0].label
```

We can now see the result of the mouse movement (drag) in the console

Figure A7.9





Sketch A7.10 displaying the results

We want to see this in action. We want the result on the canvas and also draw a line showing the movement of the mouse. We will do all this in the `draw()` function.

! Comment out (remove) the console logs

```
let nn
let status = "training"
let start
let end

let data = [
  { x: 1, y: 0.1, label: "right"},
  { x: 1, y: -0.1, label: "right"},
  { x: -1, y: 0.1, label: "left"},
  { x: -1, y: -0.1, label: "left"},
  { x: 0.1, y: 1, label: "down"},
  { x: -0.1, y: 1, label: "down"},
  { x: 0.1, y: -1, label: "up"},
  { x: -0.1, y: -1, label: "up"}
]

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: "classification",
    debug: false
  }
  nn = ml5.neuralNetwork(options)
  for (let item of data)
  {
    let inputs = [item.x, item.y]
    let outputs = [item.label]
    nn.addData(inputs, outputs)
  }
  nn.train({epochs: 250}, finishedTraining)
```

```

}

function finishedTraining()
{
  status = "ready"
  // console.log(status)
}

function draw()
{
  background(220)
  textAlign(CENTER, CENTER)
  textSize(64)
  text(status, width/2, height/2)
  if (start && end)
  {
    strokeWeight(8)
    line(start.x, start.y, end.x, end.y)
  }
}

function mousePressed()
{
  start = createVector(mouseX, mouseY)
}

function mouseDragged()
{
  end = createVector(mouseX, mouseY)
}

function mouseReleased()
{
  let dir = p5.Vector.sub(end, start)
  dir.normalize()
  let inputs = [dir.x, dir.y]
  nn.classify(inputs, gotResults)
}

```

```
function gotResults(results)
{
  status = results[0].label
  // console.log(status)
}
```

Notes

We don't need the console logs anymore as we are writing straight to the canvas.

Challenge

Add some colour or other event.

Code Explanation

<code>textAlign(CENTER, CENTER)</code>	Puts the complete text in the centre of the text co-ordinates
<code>textSize(64)</code>	Nice and big text
<code>text(status, width/2, height/2)</code>	The status has the string values
<code>if (start && end)</code>	Condition, draws the line when we have a start AND end vector
<code>line(start.x, start.y, end.x, end.y)</code>	Draw the line to those two vectors

Figure A7.10a
we are training



Figure A7.10b
we are ready (training complete)



Figure A7.10c
move and release upwards

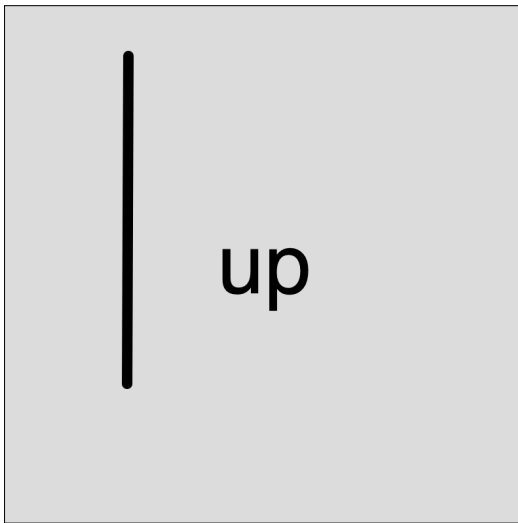


Figure A7.10d
and then down

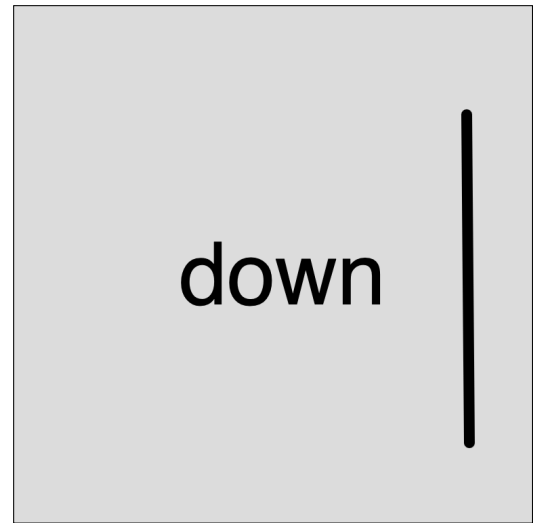


Figure A7.10e
move to the left

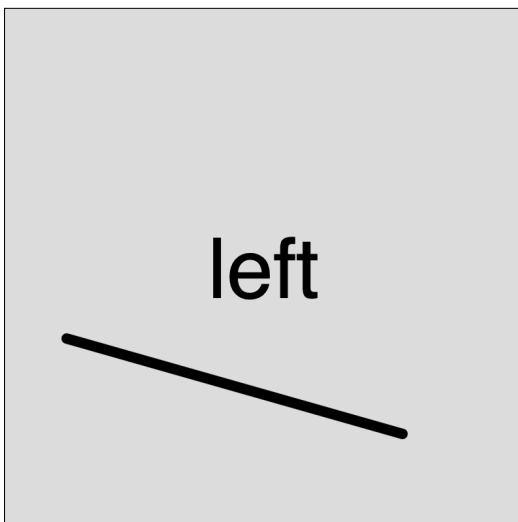
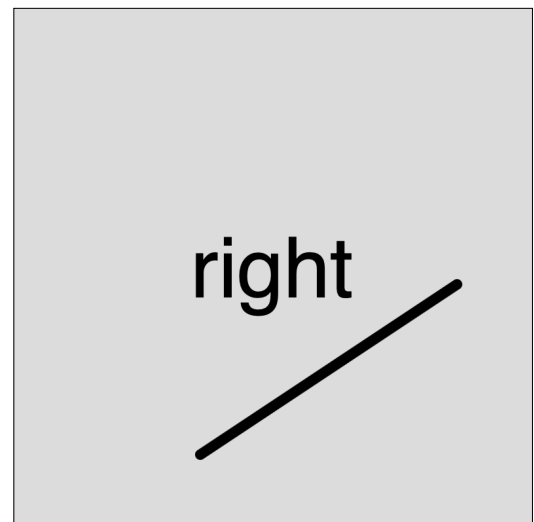


Figure A7.10f
and now to the right



Algorithmic Intelligence Module A Unit #8 p5.js code snippets 2



Module A Unit #8: p5.js code snippets #2

Some more useful snippets and information relevant to the next module. Just work through them, play, experiment, and make sure you can follow the logic. Create something interesting yourself; it is the best way to learn.

We will be exploring how to incorporate video into our coding, in particular the webcam (if you have one), and also what we can do to all the pixels on the canvas.



Sketch A8.1 strings

We have floats (numbers with a decimal place) and integers (numbers without a decimal place, whole numbers). Another data type is the **string**, which is either a letter, word, series of letters (and numbers), or even a number. It is then treated as text rather than an integer or a float. A **string** will have speech marks. You can usually use either single or double speech marks but never both in the same string. We will use it as a single letter or as a word or phrase.

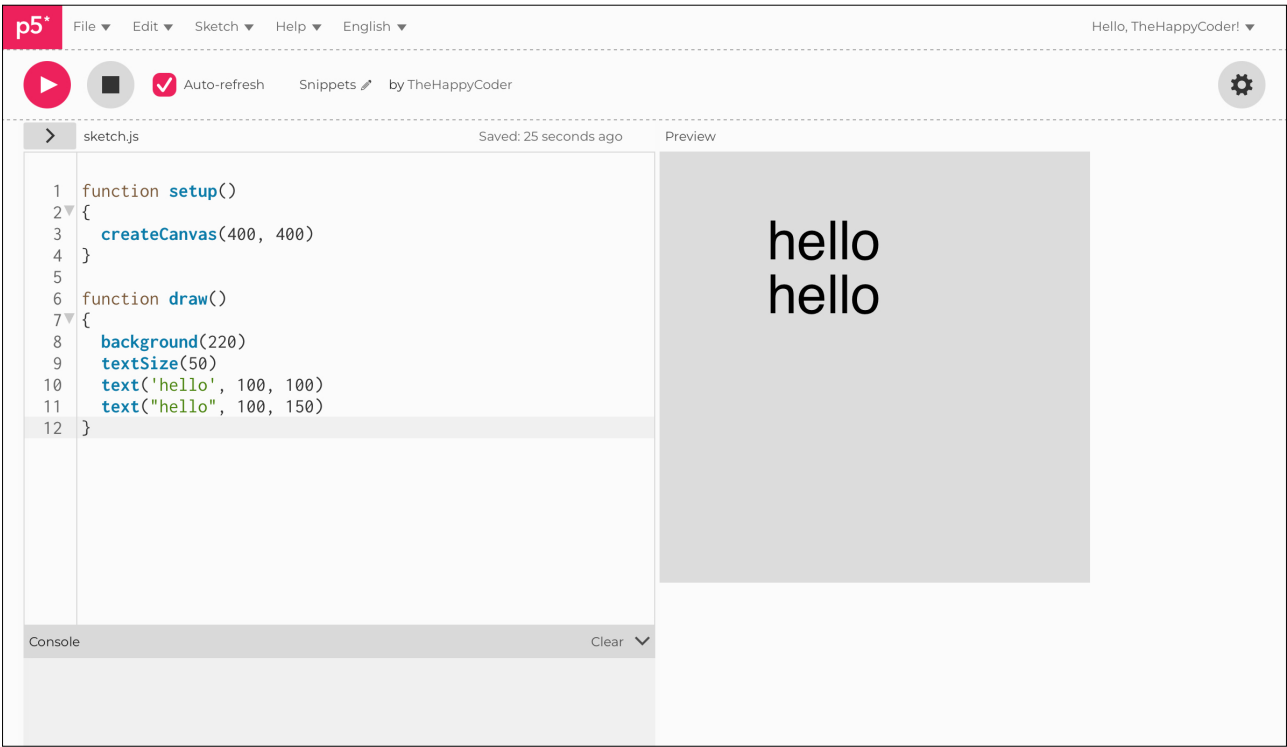
```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  textSize(50)
  text('hello', 100, 100)
  text("hello", 100, 150)
}
```

Notes

Either single or double speech marks work.

Figure A8.1





Sketch A8.2 adding two strings

We can add strings together.

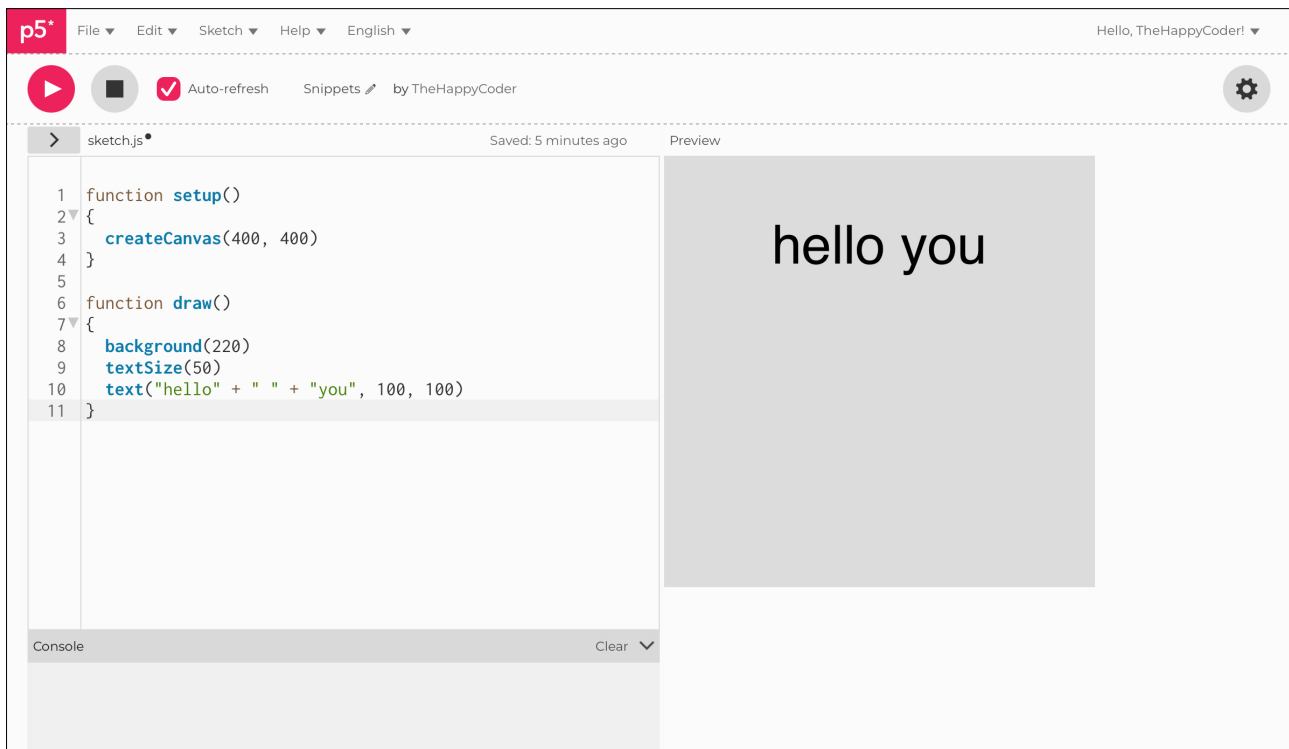
```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  textSize(50)
  text("hello" + " " + "you", 100, 100)
}
```

Notes

Just adding two strings together leaves no gap, so leave an empty string in between.

Figure A8.2





Sketch A8.3 strings as variables

! Starting a new sketch.

When we include a number and treat it as a string, it does not behave like a number anymore (except in certain situations).

```
let x = "20"
let y

function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  textSize(50)
  y = x + 16
  text(x, 100, 100)
  text(y, 100, 150)
  text(x + y, 100, 200)
}
```

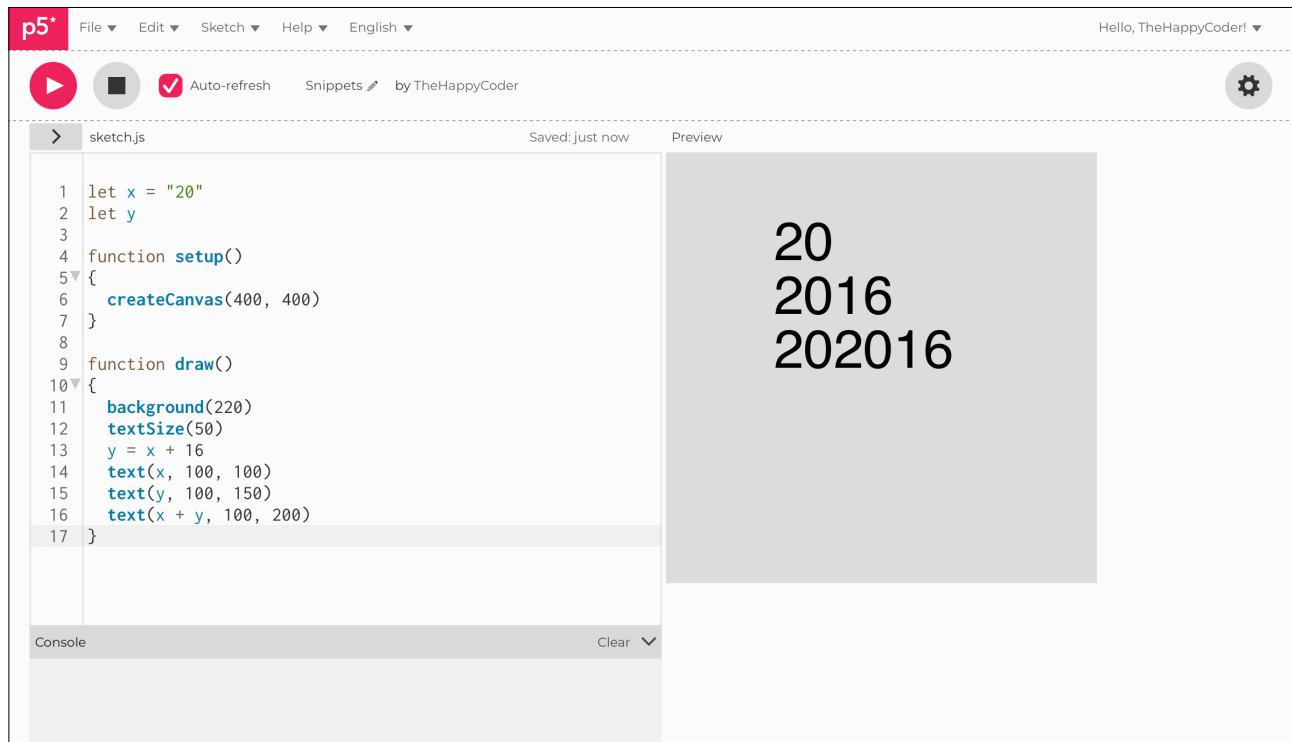
Notes

They behave quite differently.

Challenge

Play around with this concept and get used to how strings and integers work. Be careful, as strings can suddenly start behaving as integers or floats.

Figure A8.3





Sketch A8.4 the keyPressed() function

! Starting a new sketch

You can use the keyboard as well as the mouse to interact with the sketch. Here we change the colour of the circle from black to white and back again by pressing the **w** key for white fill, and the **b** key for black fill.

! You do need to click on the canvas after you have, otherwise it will think you are still typing your code.

```
let value = 0

function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(200)
  fill(value)
  circle(width/2, height/2, 200)
}

function keyPressed()
{
  if (key === 'w')
  {
    value = 255
  }
  else if (key === 'b')
  {
    value = 0
  }
}
```

Notes

This only accepts lowercase. If that is a problem whereby there might be a mixture of uppercase and lowercase, then we can use another function to always make it uppercase.

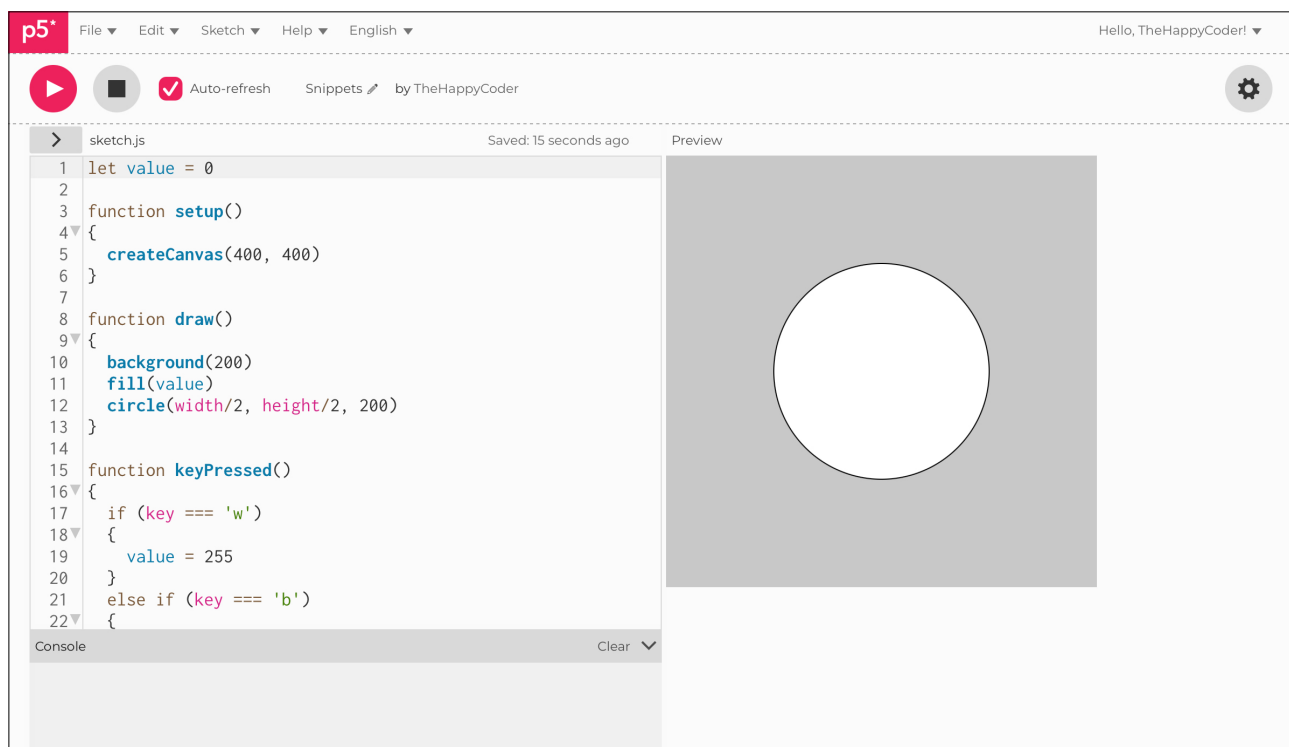
Challenge

Try other letters or use ENTER or LEFT_ARROW as the keys.

Code Explanation

function keyPressed()	Waits for a key to be pressed
if (key === 'w')	If the lowercase w is pressed
else if (key === 'b')	Or the lowercase b is pressed

Figure A8.4





What is the difference between `==` versus `===`

In JavaScript, both `==` and `===` are used to compare values, but they have subtle differences, although most of the time either will work.

A) `==` (Loose Equality)

Compares values after performing type coercion (automatic conversion between data types). For example, `'5' == 5` will evaluate to true because JavaScript converts the string "5" to the number 5 before comparison.

B) `===` (Strict Equality)

Compares both the values and the data types of the operands. For example, `'5' === 5` will evaluate to false because the data types are different (string and number).

Use `===` whenever possible, as it provides stricter and more predictable comparisons. Use `==` only when you specifically intend to perform type coercion. By using `===`, you can avoid unexpected behaviour and write more robust and maintainable JavaScript code.

This may seem academic, but there is a difference, even if it sometimes feels a subtle one. I just wanted you to be aware that there is a difference and a reason behind there being sometimes a `==` sign and at other times a `===`. If unsure, just use `==` unless you find there is a problem.



Sketch A8.5 the colour `color()` function

! Starting a new sketch.

This built-in function `color()`, is very useful when you want to carry the **RGB** colour as a single variable rather than three separate variables. This is illustrated below.

```
let c

function setup()
{
  createCanvas(400, 400)
  c = color(255, 204, 0)
}

function draw()
{
  background(220)
  fill(c)
  circle(width/2, height/2, 200)
}
```

Notes

The above example simply carries the **RGB** in a single variable.

Challenges

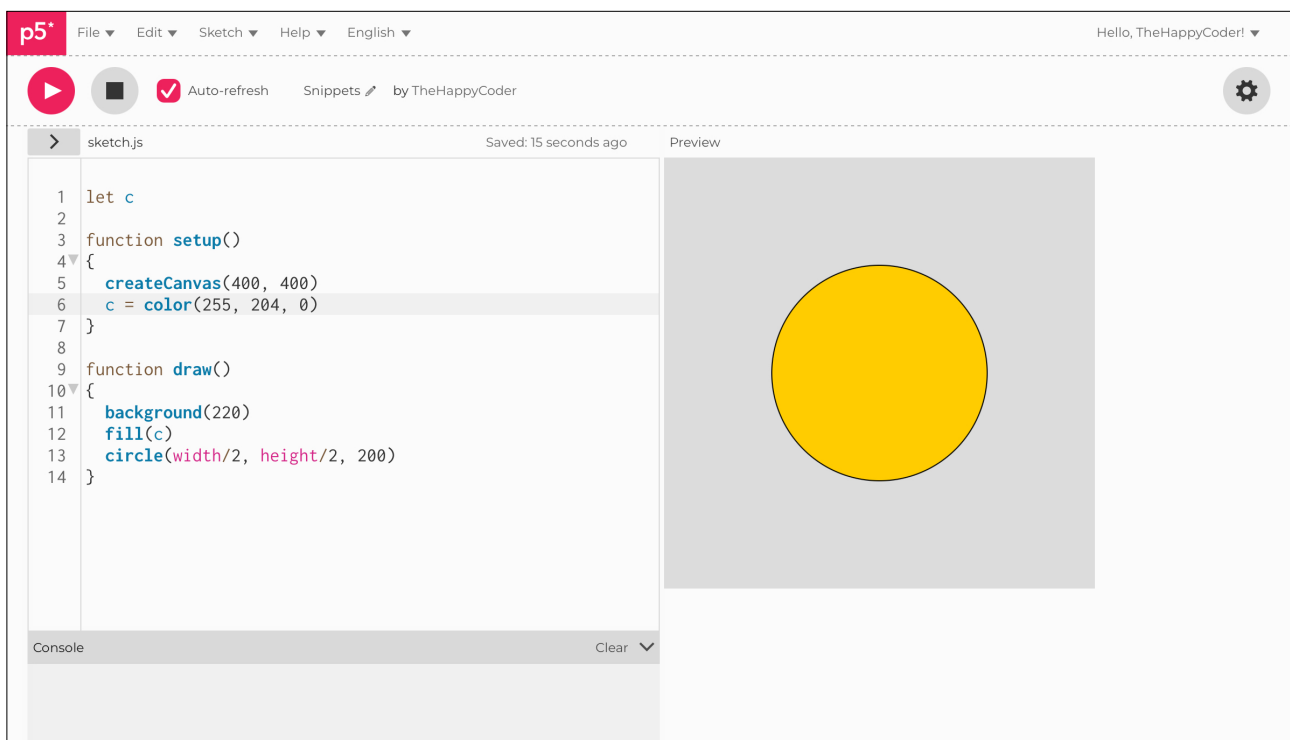
1. Try other colours.
2. Try some alpha.

Code Explanation

```
c = color(255, 204, 0)
```

The variable holds the three rgb values

Figure A8.5





Sketch A8.6 sliders

! Starting a new sketch.

A slider is another useful interactive element we can use alongside the button already covered.

```
let slider

function setup()
{
  createCanvas(400, 400)
  slider = createSlider(0, 255, 0)
  slider.position(100, 50)
  slider.size(200)
}

function draw()
{
  background(150)
  let c = slider.value()
  fill(c)
  circle(width/2, height/2, 200)
}
```

Notes

This just gives us a greyscale value to fill the circle.

Challenges

1. Have three sliders, one for each **R**, **G**, and **B**.
2. Move the slider around, with different sizes and values (min, max, and initial values).

Code Explanation

<code>let slider</code>	Slider variable name
<code>slider = createSlider(0, 255, 0)</code>	Creating the slider, with the minimum value, the maximum value and the initial starting value
<code>slider.position(100, 50)</code>	Where to position it on the canvas
<code>slider.size(200)</code>	The size of the slider
<code>let c = slider.value()</code>	Taking the value of the slider

Figure A8.6a

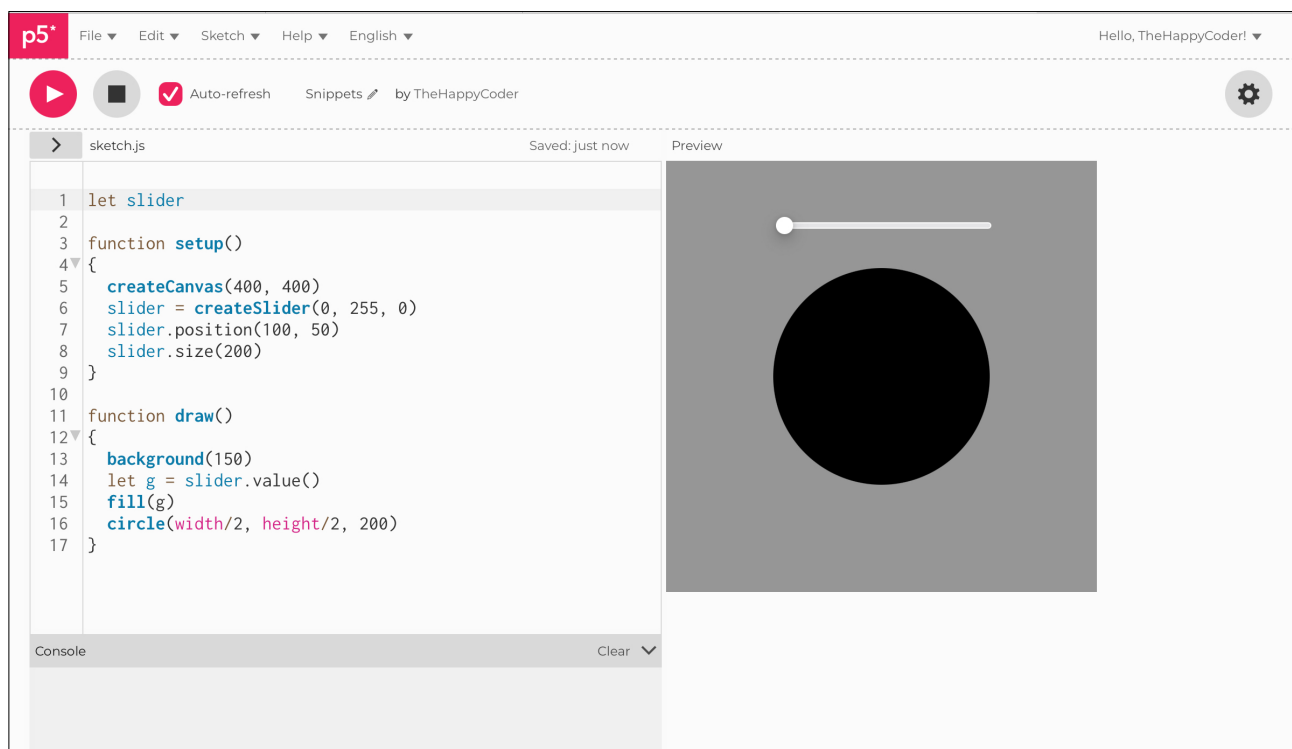
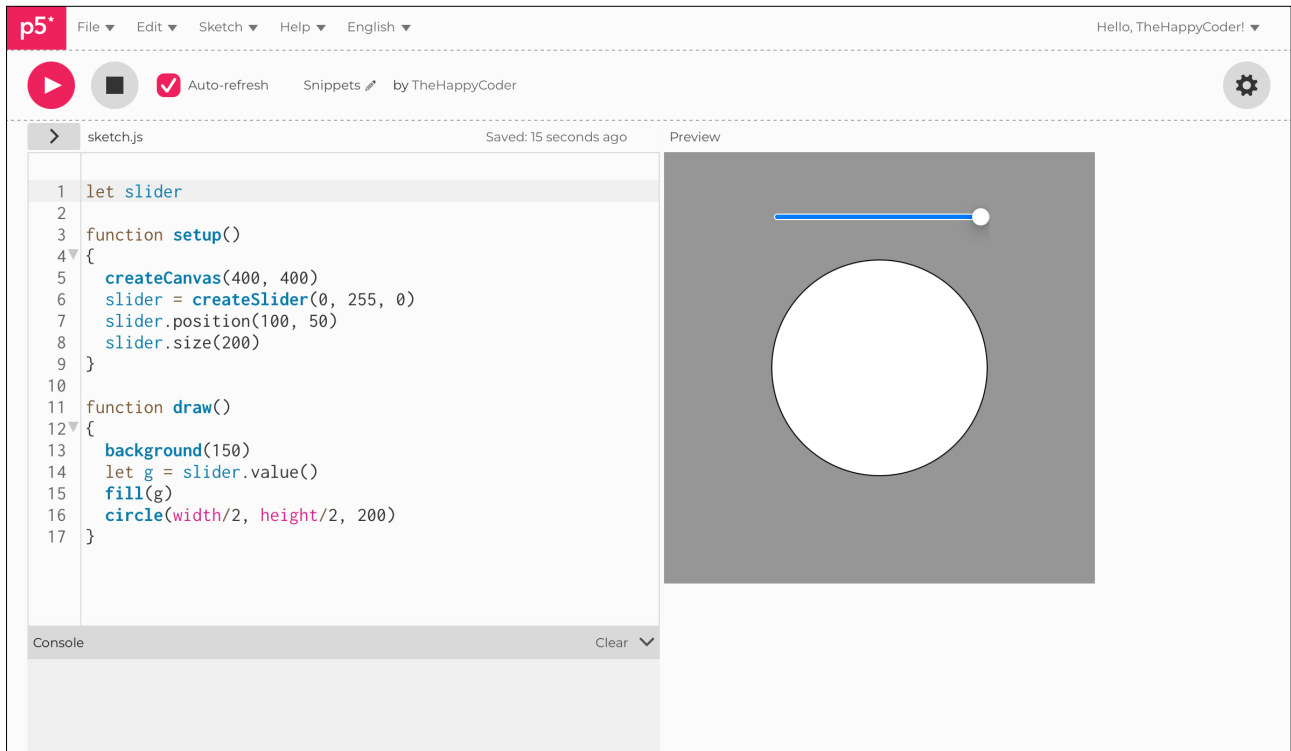


Figure A8.6b





Sketch A8.7 another shape, the square

! Starting a new sketch.

Introducing the **square**, we have drawn it in the centre of the canvas with a side length of **100** pixels.

```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  square(width/2, height/2, 100)
}
```

Notes

You will notice that, although we gave it the co-ordinates for the centre, it drew the square with the top left-hand corner in the middle of the canvas.

Challenge

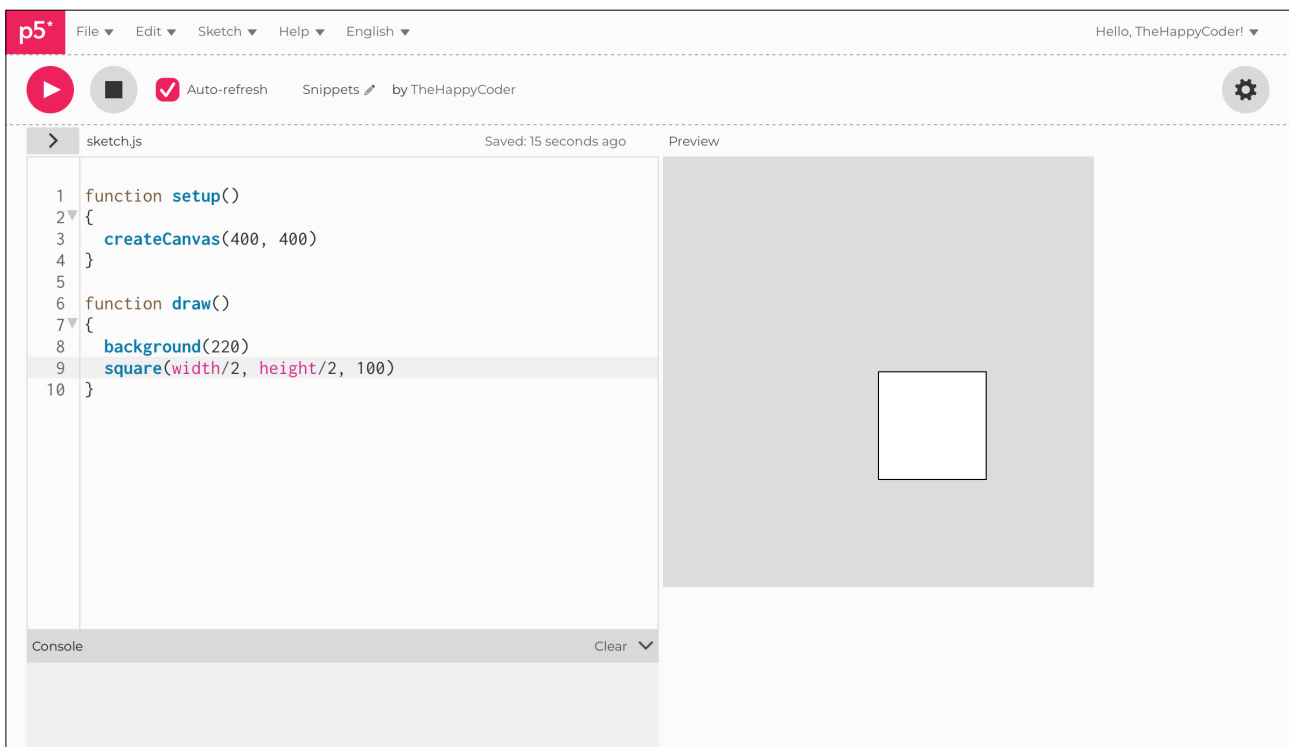
How do you think you could change that so that the square is in the middle?

Code Explanation

```
square(width/2, height/2, 100)
```

Square in the middle of the canvas with a side length of 100

Figure A8.7





Sketch A8.8 the centre of the square

We can add a function called `rectMode()` that can move the origin co-ordinates of the `square` to the centre.

```
function setup()
{
  createCanvas(400, 400)
  rectMode(CENTER)
}

function draw()
{
  background(220)
  square(width/2, height/2, 100)
}
```


Notes

The square is now in the middle of the canvas.

Challenge

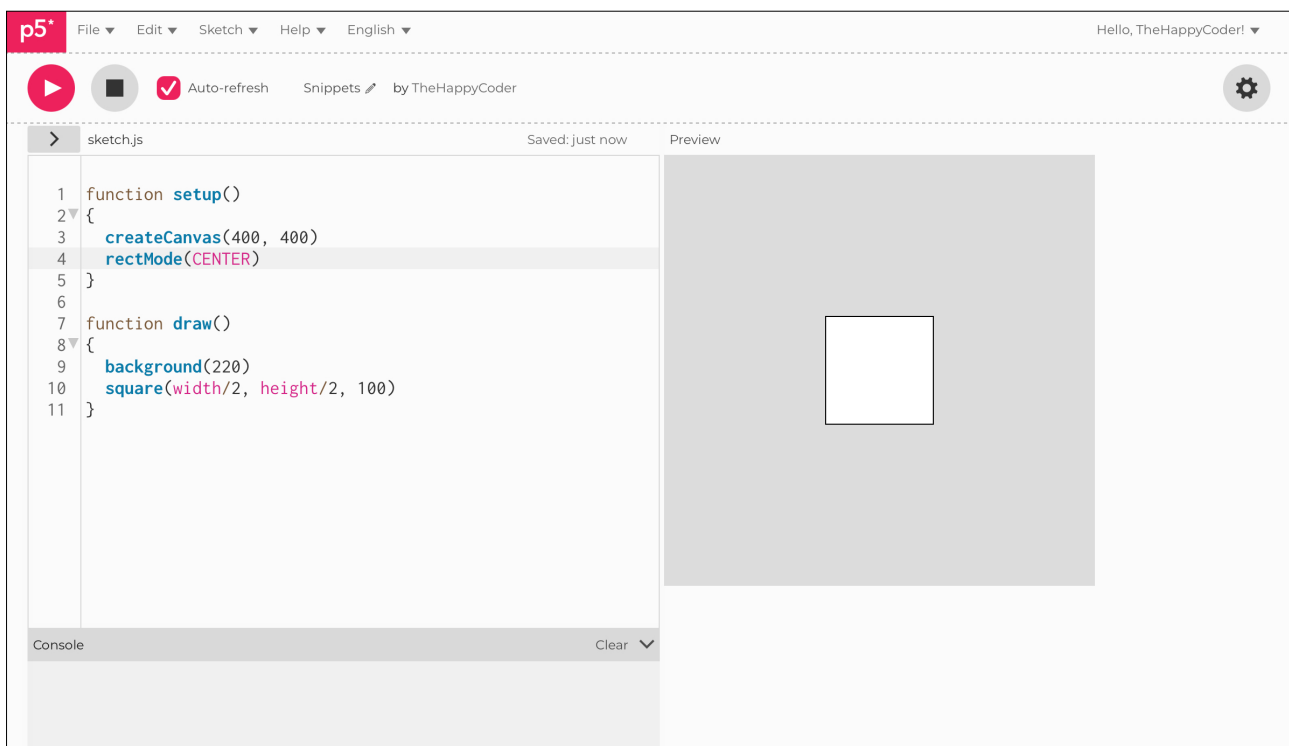
Draw lots of squares with different positions and dimensions.

Code Explanation

`rectMode(CENTER)`

Function to move the co-ordinates to the centre of the square

Figure A8.8





Introduction to using the video input

We can use the built-in webcam of your computer in our code if you have one; if not, then you will need to plug one in or skip this module. Most laptops, computers, and tablets have a webcam these days. You could even use your smartphone at a push if necessary (why not?).

The image comes in the size and ratio of **640x480** pixels. We will look at ways we can integrate it into our code, and in further units, we will use the webcam for machine learning.

! A few of the following units use a webcam of sorts, especially the pre-trained units.



Sketch A8.9 video capture

! Starting a new sketch.

The `createCapture(VIDEO)` function will ask the computer for access to your webcam; you will need to give it permission.

```
let video

function setup()
{
  noCanvas()
  video = createCapture(VIDEO)
}
```

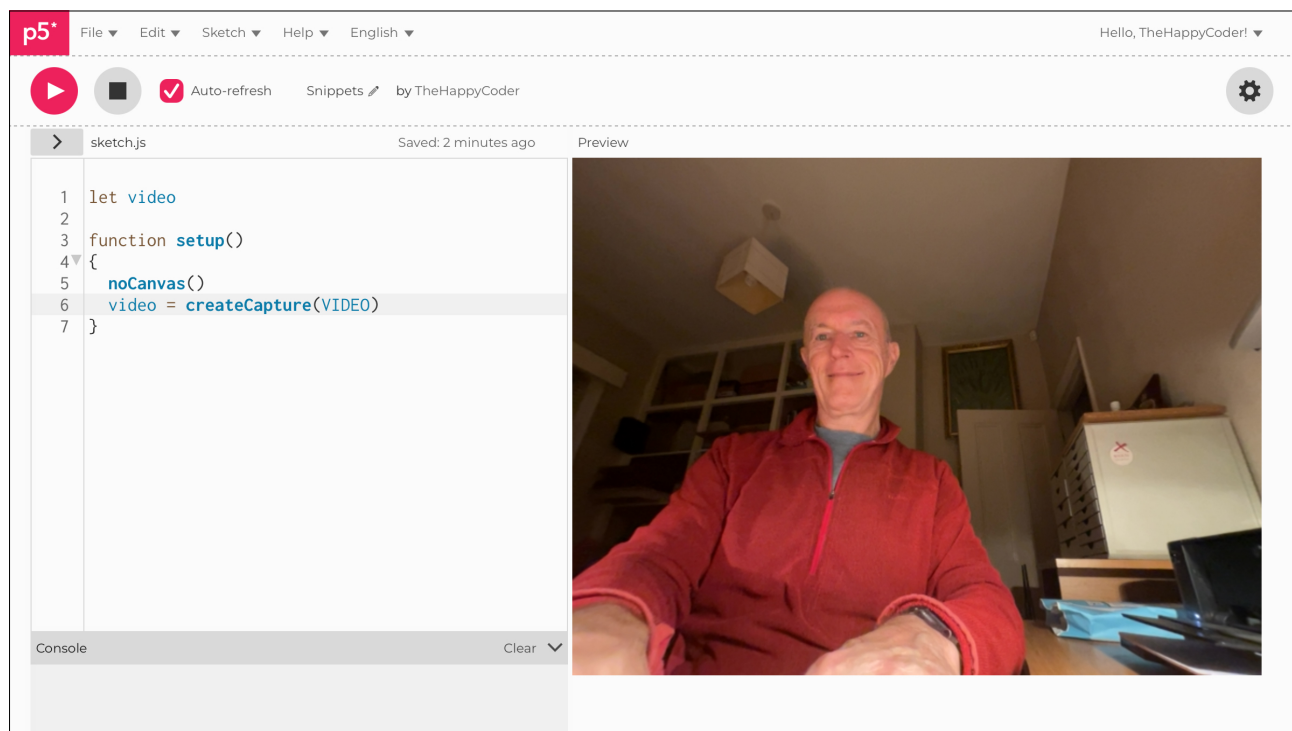
Notes

The default size is **640x480** pixels displayed in the window.

Code Explanation

<code>let video</code>	Create a variable to hold the video image
<code>noCanvas()</code>	Remove the canvas
<code>video = createCapture(VIDEO)</code>	Create the video from the webcam and attribute it to the variable

Figure A8.9





Sketch A8.10 returning the canvas

Replace the `noCanvas()` with `createCanvas(640, 480)` so we have a canvas that we can fill with the video image. Add the background.

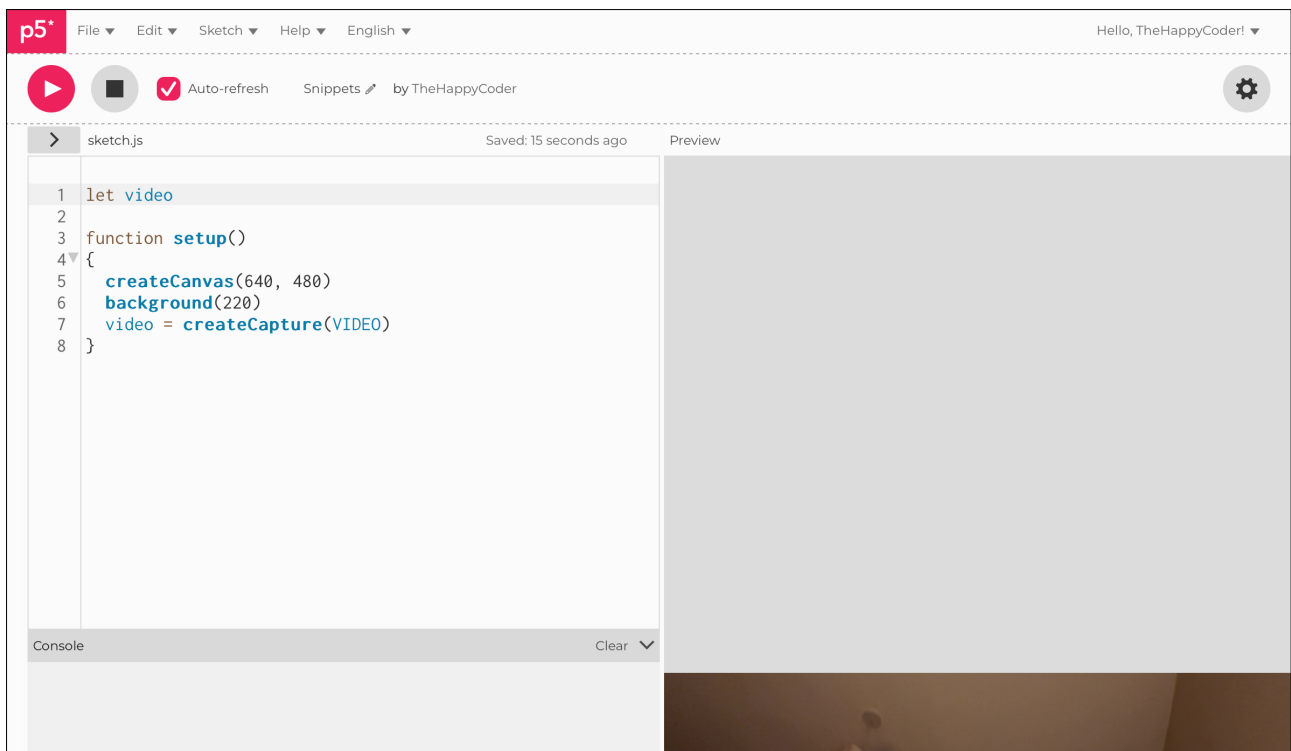
```
let video

function setup()
{
  createCanvas(640, 480)
  background(220)
  video = createCapture(VIDEO)
}
```

Notes

Unfortunately, we get the canvas with the video image below it.

Figure A8.10





Sketch A8.11 video on the canvas

Now we are going to add the video to the canvas with the `image()` function. We set the co-ordinates to the top left-hand corner of `(0, 0)`.

```
let video

function setup()
{
  createCanvas(640, 480)
  background(220)
  video = createCapture(VIDEO)
}

function draw()
{
  image(video, 0, 0)
}
```

Notes

This creates two images; the top one is drawn onto the canvas, and the bottom one is the live video stream.

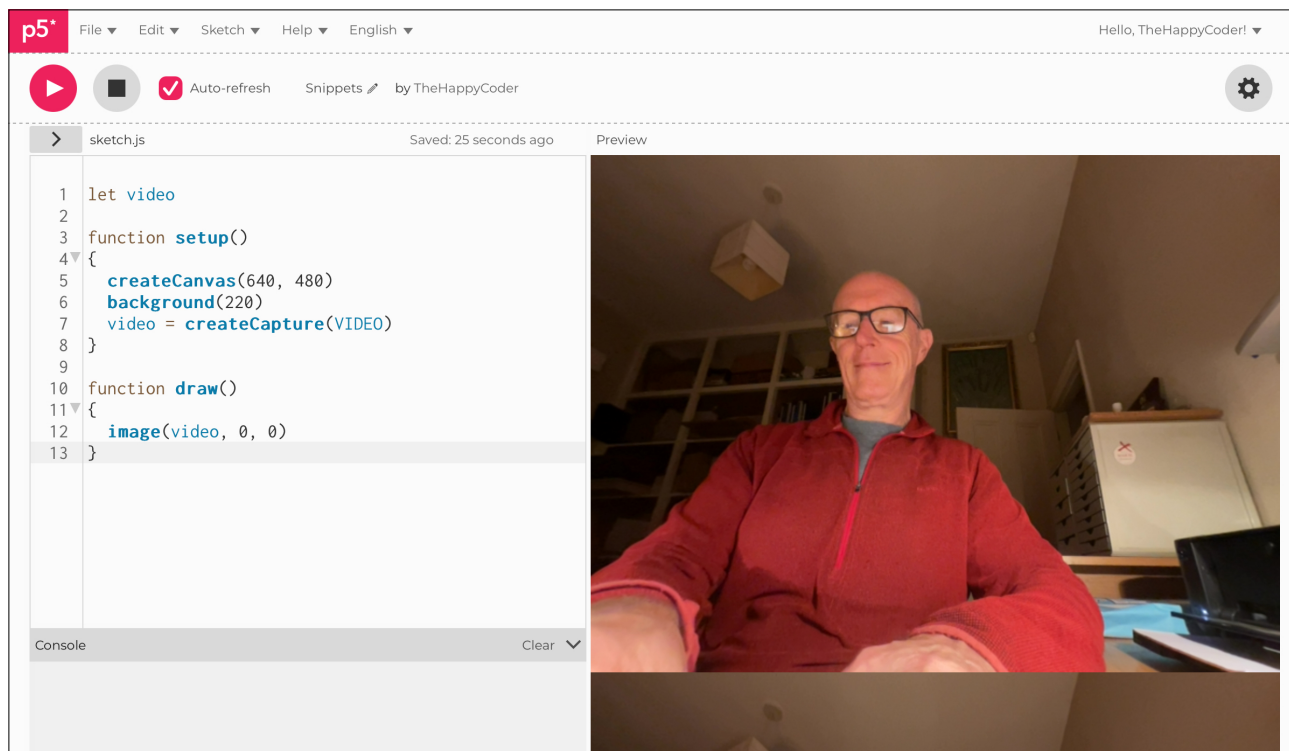
Challenges

1. Add dimensions to the image function: `image(video, 0, 0, 320, 240)`. You should end up with an image half the size (actually a quarter of the size!).
2. Change the co-ordinates of the image function to: `image(video, 200, 200, 320, 240)`. It has now moved it across the canvas.

Code Explanation

<code>image(video, 0, 0)</code>	Draws an image to the canvas, in this case it is a video and its co-ordinates are (0, 0)
---------------------------------	--

Figure A8.11





Sketch A8.12 hide the video

We can hide the video and just have it on the canvas.

```
let video

function setup()
{
  createCanvas(640, 480)
  background(220)
  video = createCapture(VIDEO)
  video.hide()
}

function draw()
{
  image(video, 0, 0)
}
```


Notes

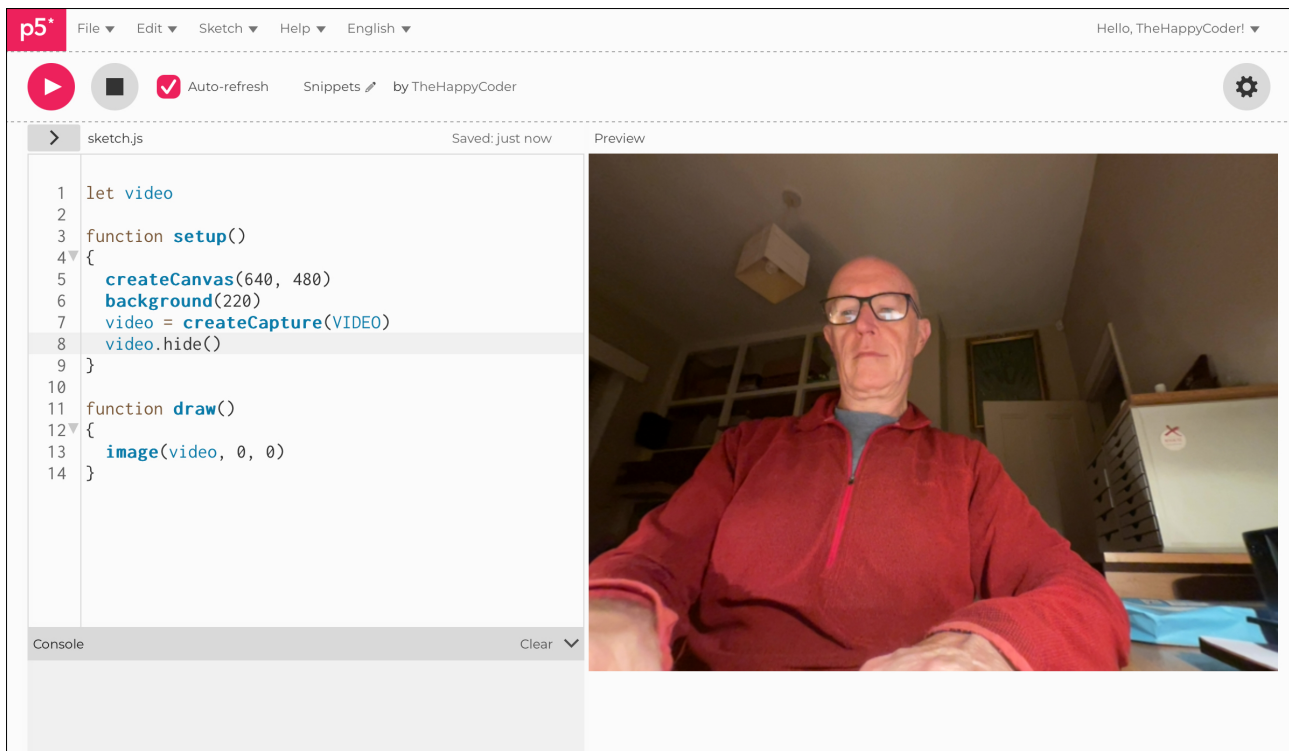
We now have the one video on the canvas.

Code Explanation

`video.hide()`

This hides the streaming video

Figure A8.12





Sketch A8.13 flipping the video

We can change the video so that it mirrors your movements, as if you were looking in a mirror. This feels a little more intuitive.

```
let video

function setup()
{
  createCanvas(640, 480)
  background(220)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
}

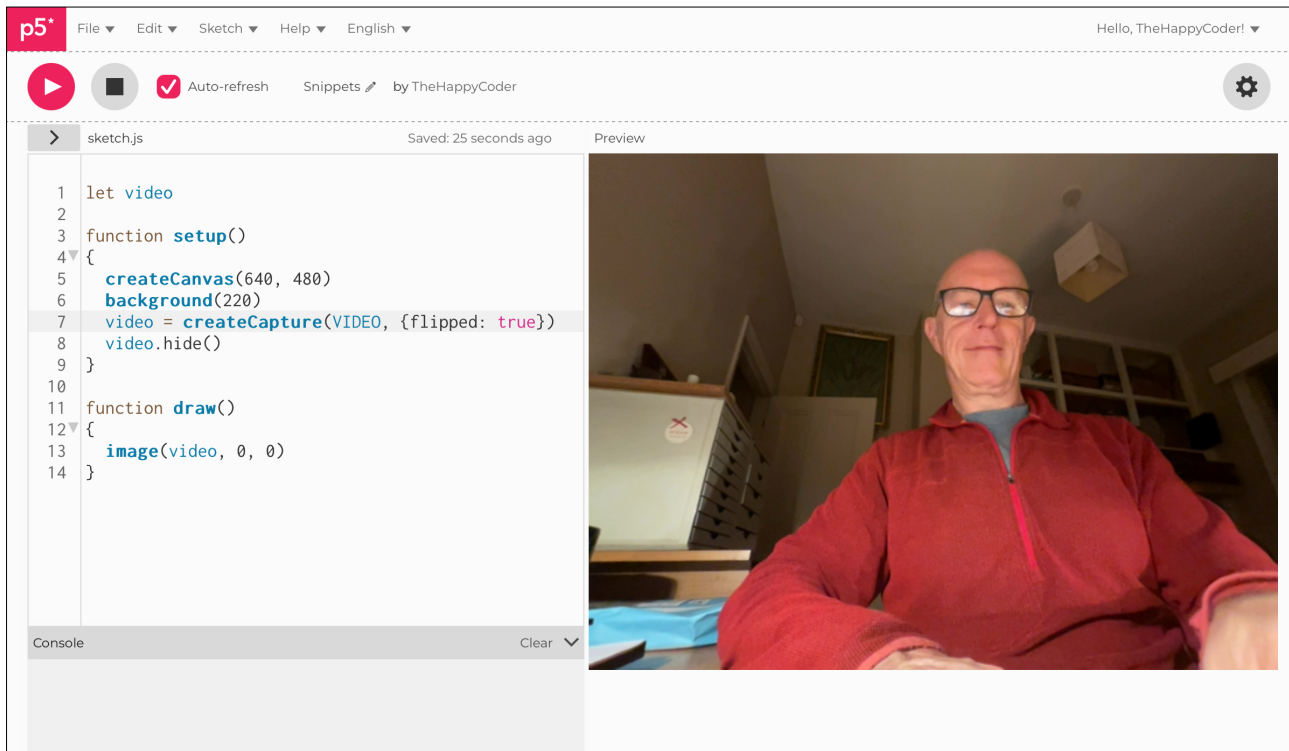
function draw()
{
  image(video, 0, 0)
}
```

Code Explanation

```
video = createCapture(VIDEO, {flipped: true})
```

This reverses the image to make it more like a mirror

Figure A8.13





Introduction to the pixels

A canvas is made up of lots of pixels. All are very tiny and hard to see with the naked eye. In a **400x400** canvas, there are **160,000** pixels. Each pixel has four channels: a **red**, a **green**, a **blue**, and an **alpha**. The alpha is the transparency, and all have values in the range **0** to **255**.

Yet we can manipulate the pixels and even the channels by picking out the individual elements that make up a pixel. We could take all the pixels and make them green, and so on. There are two main functions we will use: the first is called **loadPixels()**, which will look at all the pixels in the canvas or the image and store them in a pixel array (all the red, green, blue, and alpha for every pixel) in one long array (**640,000** elements). So to go from one pixel to another, we have to jump over **four** in the array.

Once we have done something to the pixels, we then use the other main function called **updatePixels()** to display the new, changed pixels on the canvas.

! We have to address pixel density because there are actually more channels than four with HD, more on that later.



Sketch A8.14 new starting sketch

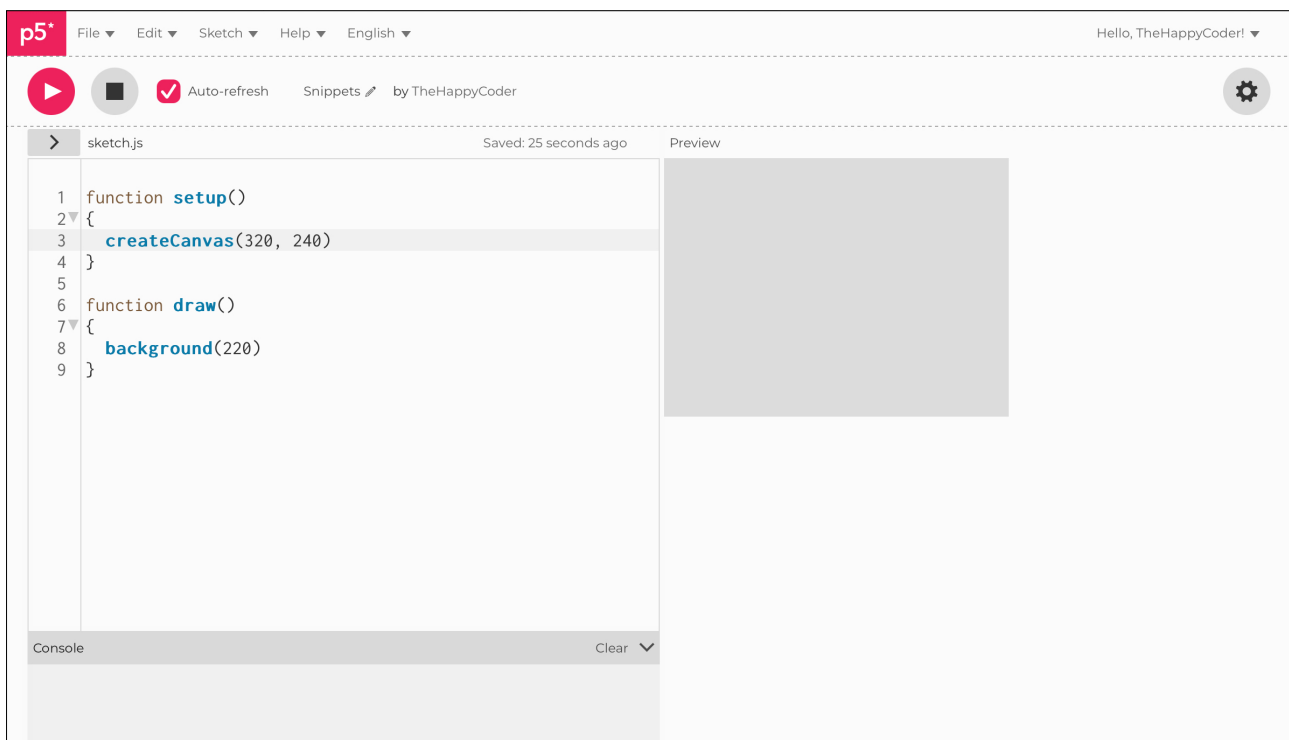
! Starting a new sketch.

We will have a canvas size of **320** x **240**, which is half the dimensions of the video image.

```
function setup()
{
  createCanvas(320, 240)
}

function draw()
{
  background(220)
}
```

Figure A8.14





Sketch A8.15 pixel array formula

The function `loadPixels()` gets all the data points from all the pixels and puts them into a pixel array. Every pixel has four values (as mentioned), but we want every fourth index, where the index is just for referencing the next pixel, not every element (of which there are four per pixel). Hence, we jump every four elements in the pixel array.

We have a simple algorithm for getting every pixel, which is every fourth element in the pixel array. We can then pick out the four elements from each pixel and change them (if we want to), followed by updating the new pixel array with the `updatePixels()` function.

To go through all the pixels in the array, we have a nested loop where we start with the first line of pixels and work our way down. So that is why we start the first nested loop with the `y` co-ordinate and then the `x` co-ordinates. For each `y` value, we loop through all the `x` values before going onto the next `y` value (one line at a time).

```
let x
let y
let index

function setup()
{
  createCanvas(320, 240)
}

function draw()
{
  background(220)
  loadPixels()
  for (y = 0; y < height; y++)
  {
    for (x = 0; x < width; x++)
    {
      index = (x + (y * width)) * 4
    }
  }
  updatePixels()
}
```

Notes

Nothing changes with the canvas; you should get the grey background. The formula makes more sense when you plug in some values for **y** and for **x**. All you are doing is giving the index value to every fourth element in the array. Next, we will break down the four elements in a pixel further into components of **red**, **green**, **blue**, and **alpha**.

Challenge

Put in the values for **y = 0**, **x = 0**, then **x = 1**, **x = 2**, and so on, and you will, hopefully, see the logic of this algorithm.

Code Explanation

<code>loadPixels()</code>	Loads all the pixels into an array
<code>index = (x + (y * width)) * 4</code>	A simple formula for working through the pixel array four at a time
<code>updatePixels()</code>	Updates the array and returns them to the canvas



Sketch A8.16 changing the pixels

Here we can give each pixel element, the **red**, the **green**, the **blue**, and the **alpha** new values. We have a grey canvas where the **red**, **green**, and **blue** all have the same value of **220**. We usually just have the one value for greyscale, but in reality, all three are the same; the **alpha** is **255** by default.

This may seem a bit confusing, but in the pixel array, which has all the reds, greens, blues, and alphas for all the pixels. The **index** we have created is for the actual pixel; so **index + 0** is the first element, which is **red**, **index + 1** is the second, which is **green**, **index + 2** is the third element, which is **blue**, and **index + 3** is the fourth element, which is the **alpha**. Here we are going to give you the values of the greyscale and then next change them to an orange colour.

! I have added an extra line of code **pixelDensity(1)** because there are extra elements in the pixel array if the display is of a higher density, and you get funny results. You can remove it if it causes any problems.

```
let x
let y
let index

function setup()
{
  createCanvas(320, 240)
  pixelDensity(1)
}

function draw()
{
  background(220)
  loadPixels()
  for (y = 0; y < height; y++)
  {
    for (x = 0; x < width; x++)
    {
      index = (x + (y * width)) * 4
      pixels[index + 0] = 220
      pixels[index + 1] = 220
      pixels[index + 2] = 220
      pixels[index + 3] = 255
    }
  }
}
```



```
}  
  updatePixels()  
}
```

Notes

At present, you simply get the same grey canvas.

Code Explanation

<code>pixels[index + 0] = 220</code>	The red element of the pixel value
<code>pixels[index + 1] = 220</code>	The green element of the pixel value
<code>pixels[index + 2] = 220</code>	The blue element of the pixel value
<code>pixels[index + 3] = 255</code>	The alpha element of the pixel value



Sketch A8.17 orange pixels

Now we can play around with the pixel element values.

```
let x
let y
let index

function setup()
{
  createCanvas(320, 240)
  pixelDensity(1)
}

function draw()
{
  background(220)
  loadPixels()
  for (y = 0; y < height; y++)
  {
    for (x = 0; x < width; x++)
    {
      index = (x + (y * width)) * 4
      pixels[index + 0] = 255
      pixels[index + 1] = 100
      pixels[index + 2] = 0
      pixels[index + 3] = 255
    }
  }
  updatePixels()
}
```

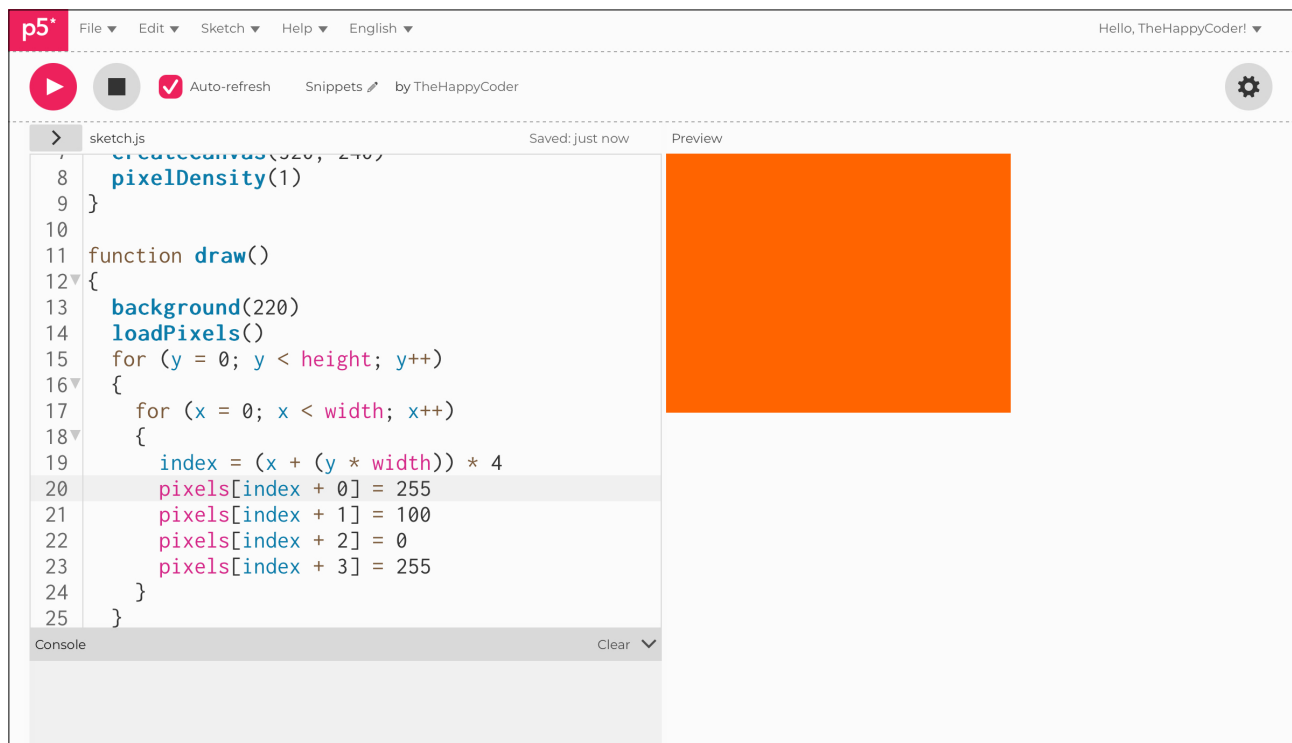
Notes

We now have a nice, bright orange canvas.

Challenge

Try other combinations of colours and alpha.

Figure A8.17





Sketch A8.18 return of the video

Adding in a lot of familiar code to return the video to the canvas, we introduce the `video.size()` function in `setup()` and a new pixel array for the video. The `loadPixels()` function takes the pixels from the canvas (not the video). The pixel array is updated with the video pixels array through the nested loop and then updates the pixels in the canvas.

```
let video

let x
let y
let index

function setup()
{
  createCanvas(320, 240)
  pixelDensity(1)
  video = createCapture(VIDEO, {flipped: true})
  video.size(320, 240)
  video.hide()
}

function draw()
{
  background(220)
  video.loadPixels()
  loadPixels()
  for(y = 0; y < height; y++)
  {
    for(x = 0; x < width; x++)
    {
      index = (x + (y * width)) * 4
      pixels[index + 0] = video.pixels[index + 0]
      pixels[index + 1] = video.pixels[index + 1]
      pixels[index + 2] = video.pixels[index + 2]
      pixels[index + 3] = video.pixels[index + 3]
    }
  }
  updatePixels()
}
```

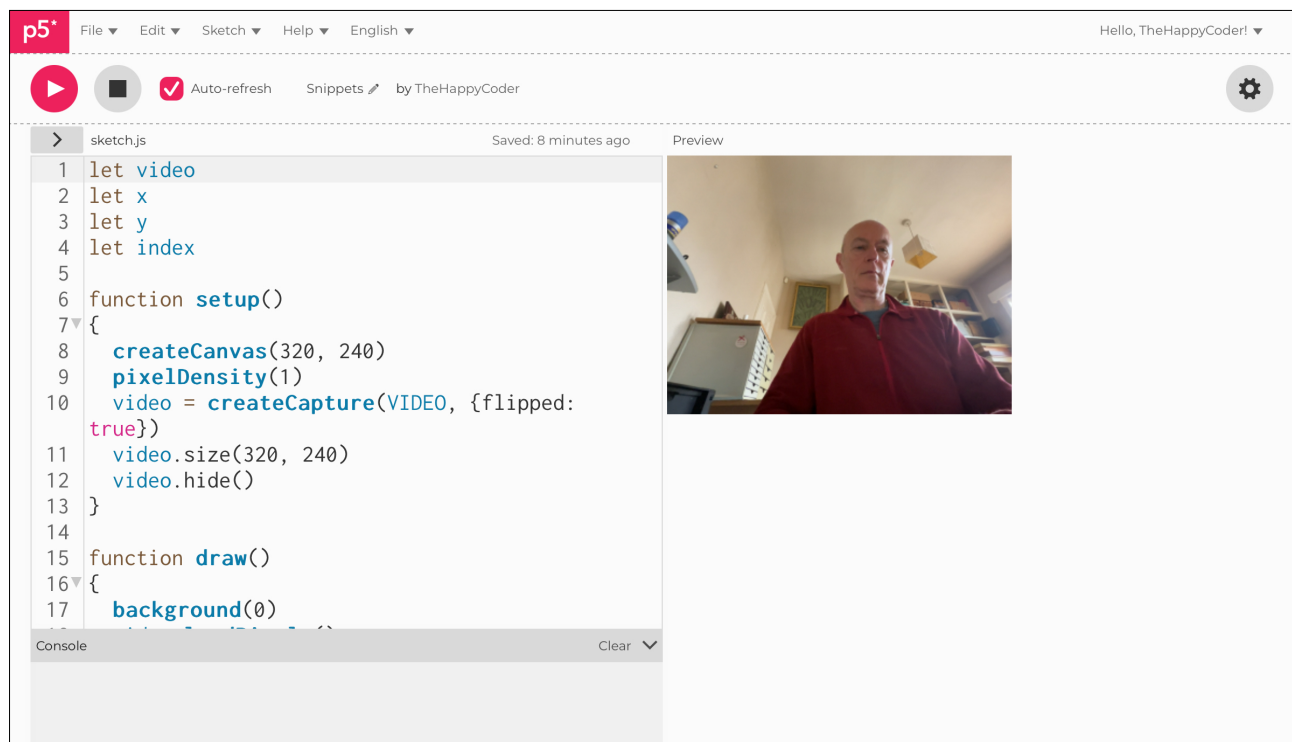
Notes

We now have an image drawn onto the canvas. This means we can manipulate the image.

Code Explanation

<code>video.size(320, 240)</code>	Alter the size of the video image
<code>video.loadPixels()</code>	Loading all the video pixels into a new array
<code>pixels[index + 0] = video.pixels[index + 0]</code>	Changing the pixel array with the new pixel array values from the video

Figure A8.18





Sketch A8.19 pixelating the image

Another useful technique is to pixelate the image. We will use this in one of the units later. All the changes are highlighted in blue. It looks a lot, but they are mostly just new variables. We change the size of the video by 1/16th. Then we scale it back up to fill the canvas. The pixels are drawn from the new video size, which is a lot smaller, and if you could see it, it would be quite blurry.

! Don't forget to remove the `updatePixels()` near the end of the code.

```
let video
let x
let y
let index

let r
let g
let b
let a
let vScale = 16

function setup()
{
  createCanvas(320, 240)
  pixelDensity(1)
  video = createCapture(VIDEO, {flipped: true})
  video.size(width / vScale, height / vScale)
  video.hide()
}

function draw()
{
  background(220)
  video.loadPixels()
  loadPixels()
  for(y = 0; y < video.height; y++)
  {
    for(x = 0; x < video.width; x++)
    {
      index = (x + (y * video.width)) * 4
      r = video.pixels[index + 0]
```

```
    g = video.pixels[index + 1]
    b = video.pixels[index + 2]
    a = video.pixels[index + 3]
    fill(r, g, b, a)
    square(x * vScale, y * vScale, vScale)
  }
}

// updatePixels()
}
```

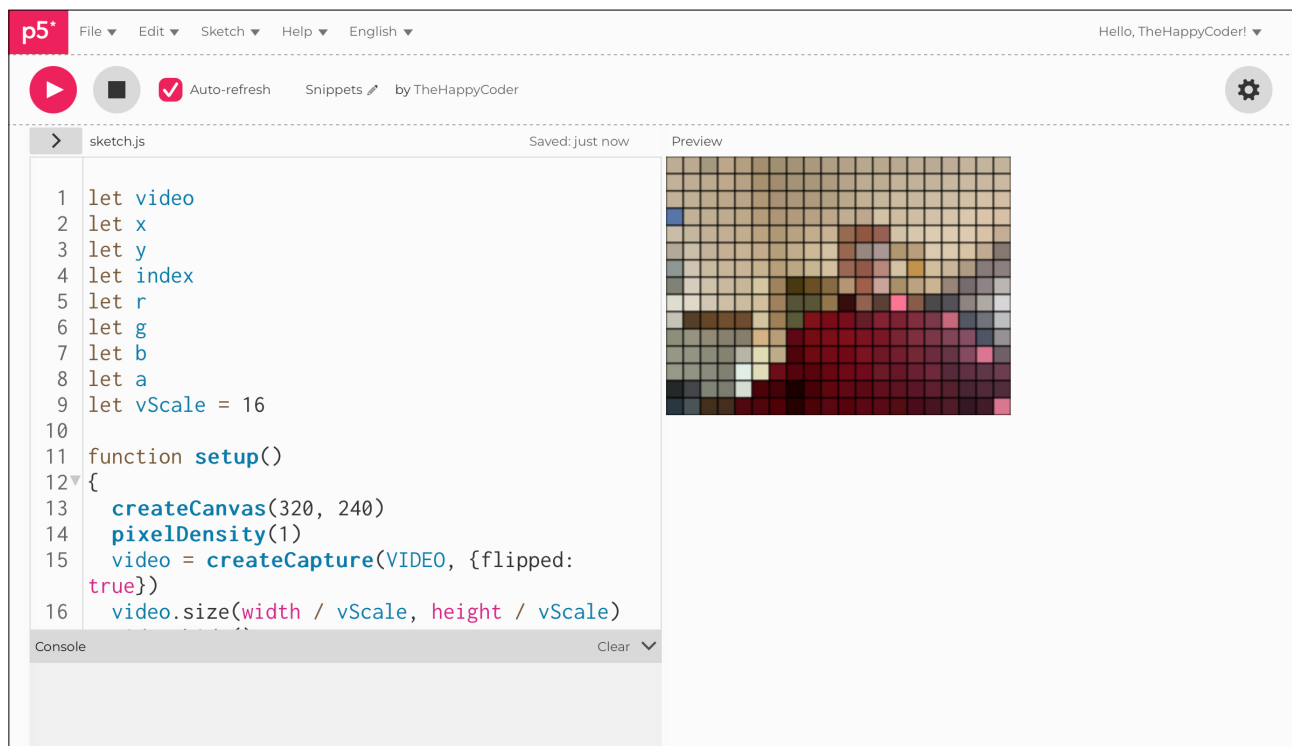
Notes

This takes the original video image, scans each pixel for the red, green, blue, alpha and then scales it up to fill the canvas. A square is then filled with the colour of the original image. We reduce the image by vScale, analyse the image and then scale it back up to full canvas size.

Challenges

1. Change the vScale (you may get an error message)
2. Change the shape to a circle

Figure A8.19





Sketch A8.20 brightness grey scale

We can now take the average of the three colours in each pixel and call that value **bright**. We halve the size of the squares. I have added **floor()** when calculating the video size as it kept throwing up errors at certain **vScale** values. We also removed the lines around the boxes.

```
let video
let x
let y
let index
let r
let g
let b

let vScale = 8
let bright

function setup()
{
  createCanvas(320, 240)
  pixelDensity(1)
  video = createCapture(VIDEO, {flipped: true})
  video.size(floor(width/vScale), floor(height/vScale))
  video.hide()
  noStroke()
}

function draw()
{
  background(0)
  video.loadPixels()
  loadPixels()
  for(y = 0; y < video.height; y++)
  {
    for(x = 0; x < video.width; x++)
    {
      index = (x + (y * video.width)) * 4
      r = video.pixels[index + 0]
      g = video.pixels[index + 1]
```

```

    b = video.pixels[index + 2]
    a = video.pixels[index + 3]
    bright = (r + g + b)/3
    fill(bright)
    square(x * vScale, y * vScale, vScale)
  }
}
}

```

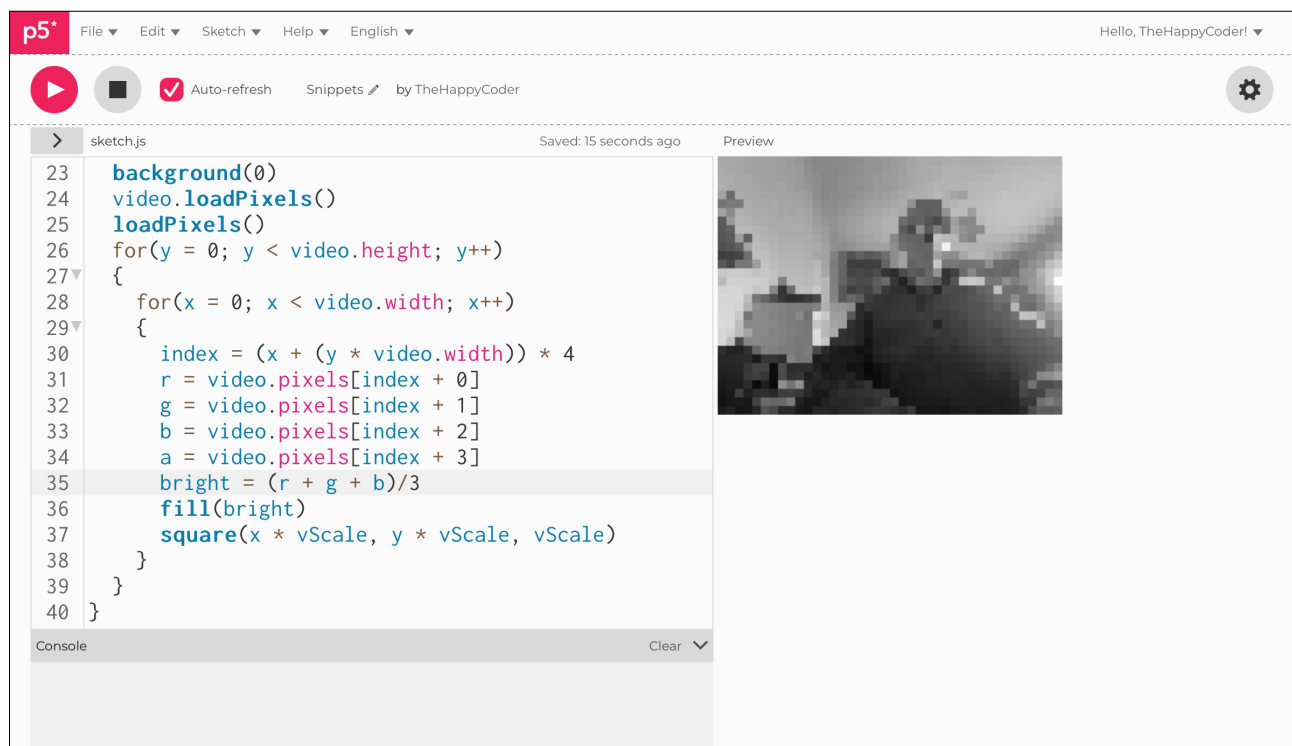
Notes

Now we have added the video in as before, but we have added the values of the **red**, **green**, and **blue** and divided by three to get the average brightness. This is not the same as the alpha (which we have ignored).

Code Explanation

<code>video.size(floor(width/vScale), floor(height/vScale))</code>	Take the floor value for the dimensions of the video size
<code>bright = (r + g + b)/3</code>	Average (bright) the colours in each pixel
<code>fill(bright)</code>	Fill that square with that average, bright

Figure A8.20



Algorithmic Intelligence Module A Unit #9 cluster classification



Module A Unit #9: cluster classification with ml5.js

This is a supervised learning model. We are going to click on the canvas in four areas. In each quadrant, we will create a set of ten circles clustered together; each set of circles will be labelled **A**, **B**, **C**, and **D**. We will then train it with ml5.js, and when the training is complete, we will click on the canvas, and it will predict which lettered circle should be at that point on the canvas.

A) This is a **classification** task.

B) It will classify each circle as **A**, **B**, **C**, or **D**.



The index.html file

The `index.html` file needs the `ml5.js` library.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch A9.1 starting sketch

We have our bog-standard starting sketch.

```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
}
```



Sketch A9.2 creating the model

Getting our neural network set up. We are going to use the `ml5.js` default `hyperparameter` settings for the moment.

```
let nn

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork()
}

function draw()
{
  background(220)
}
```

Notes

This just checks that everything is working as it should when you run it.

Challenge

Think about giving your model a different name.

Code Explanation

<code>let nn</code>	We declare a variable name for our neural network
<code>ml5.setBackend("webgl")</code>	The <code>setBackend()</code> function allows this to run in all browsers, you could try "cpu" or even "gpu" depending on your machine
<code>nn = ml5.neuralNetwork()</code>	We define our neural network



Sketch A9.3 adding the options

We use the options to clarify what we are doing and what the inputs and outputs are. We tell it what task it is going to perform, **classification** or **regression**. We will define the inputs and outputs.

```
let nn

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")

  let options = {
    inputs: ['x', 'y'],
    outputs: ['label'],
    task: 'classification'
  }
  nn = ml5.neuralNetwork(options)
}

function draw()
{
  background(220)
}
```

Notes

The **options** give you the possibility of fine-tuning the neural network. In this case, we will have two inputs, the **x** and **y** co-ordinates and four outputs, **A**, **B**, **C**, and **D**, which are the **labels**. Sometimes you don't even need to specify the inputs and outputs because the library can work it out anyway. It is important to specify the task, though. In this case, it is a **classification** task.

Code Explanation

<code>let options = {...}</code>	Defining the options
<code>inputs: ['x', 'y']</code>	The inputs are an array of x and y values
<code>outputs: ['label']</code>	The outputs are labels A, B, C or D
<code>task: 'classification'</code>	The task is a classification
<code>nn = ml5.neuralNetwork(options)</code>	The options are attributed to the neural network



Sketch A9.4 training data

We are going to create a `mousePressed()` function to collect training data. We need to move the `background()` into `setup()` and remove the `draw()` function because we will do all the drawing in the `mousePressed()` function. We will start with a default letter **A** and place it in a blue circle.

```
let nn

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['label'],
    task: 'classification'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}

// function draw()
// {
//   background(220)
// }

function mousePressed()
{
  fill(0, 0, 255, 100)
  circle(mouseX, mouseY, 25)
  textAlign(CENTER, CENTER)
  stroke(0)
  text('A', mouseX, mouseY)
}
```

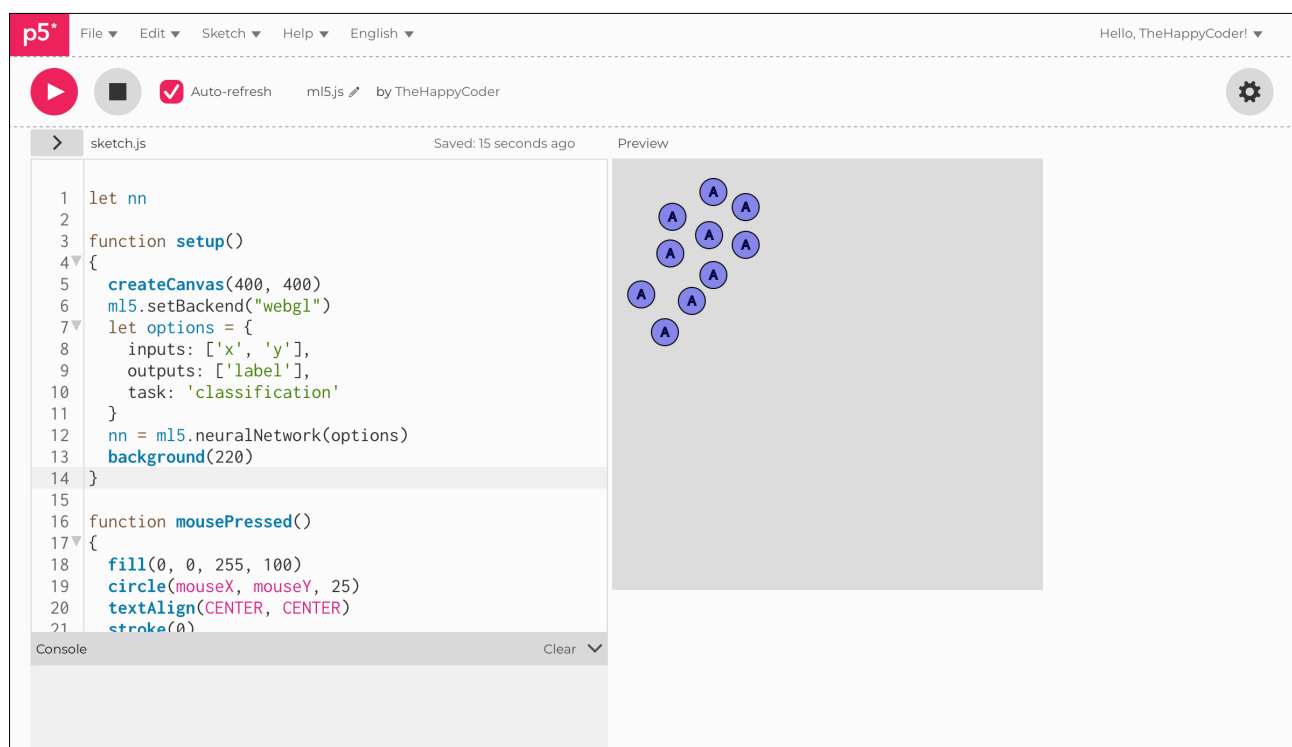
Notes

If you click on the canvas, you should get a light blue circle with the letter **A** in the centre of it, like this. This will form part of your data collection. For the future sketches, there will be no `draw()` function.

Code Explanation

<code>function mousePressed()</code>	This function waits for the mouse to be pressed and then actions the code in the function.
<code>fill(0, 0, 255, 100)</code>	Gives a blue colour and the fourth argument is the alpha making a it a little transparent
<code>textAlign(CENTER, CENTER)</code>	This puts the letter in the very centre of the circle
<code>text('A', mouseX, mouseY)</code>	We place the text at the same co-ordinates as the circle, at mouseX and mouseY

Figure A9.4





Sketch A9.5 target label

We already have the letter **A** by default, but we want to collect three more data points with target labels **B**, **C**, and **D**, for this, we will need a keyboard input and the `keyPressed()` function. When you press the letter on the keyboard (**A**, **B**, **C**, or **D**), you activate that target **label** letter. The function `toUpperCase()` will make it uppercase for you, so you don't need to set Caps Lock; the function does it for you.

```
let nn

let targetLabel = 'A'

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['label'],
    task: 'classification'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}

function keyPressed()
{
  targetLabel = key.toUpperCase()
}

function mousePressed()
{
  fill(0, 0, 255, 100)
  circle(mouseX, mouseY, 25)
  textAlign(CENTER, CENTER)
  stroke(0)
  text(targetLabel, mouseX, mouseY)
}
```

Notes

Have about **10** (data points) per letter and try to cluster them together as illustrated in the image below. The target label has the default letter **A**, but to get the other target labels, press the **B**, **C**, or **D** keys first before clicking on the canvas.

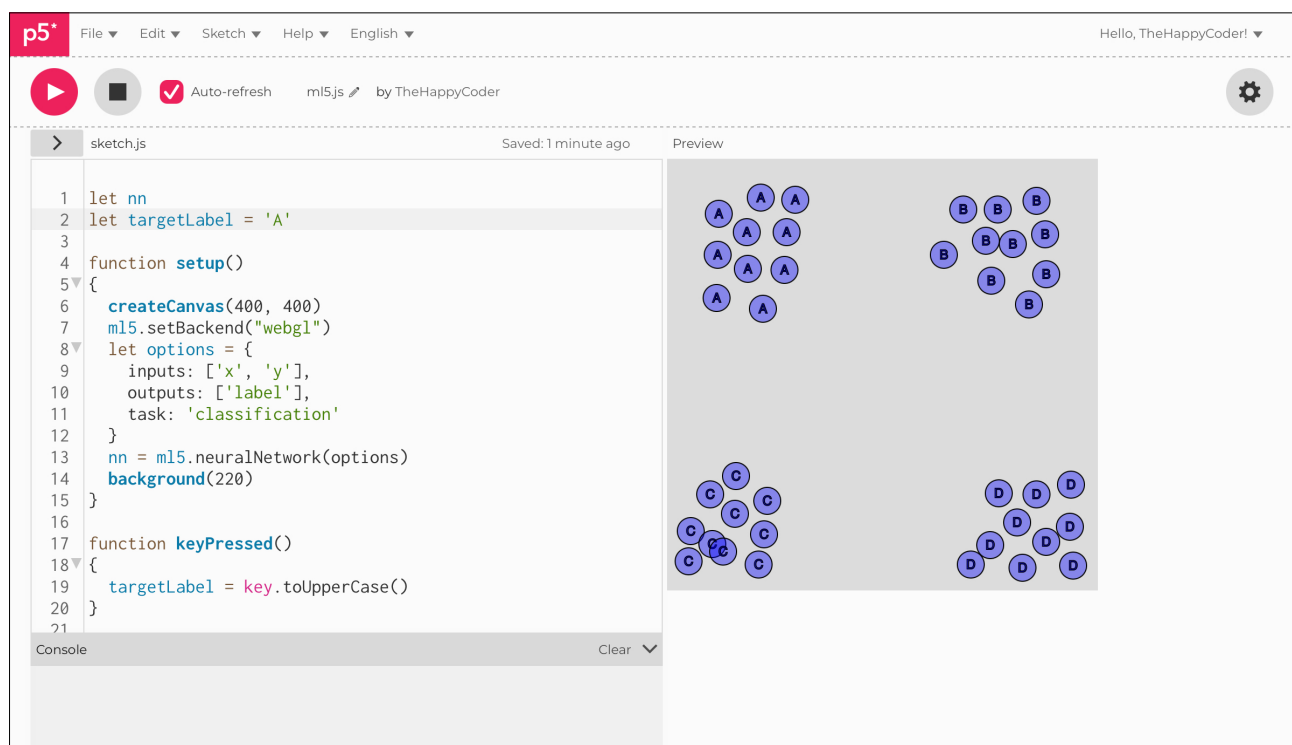
Challenges

1. You could have any labels for your targets.
2. You could have more or less than the four suggested.

Code Explanation

<code>let targetLabel = 'A'</code>	Defines a variable for the target label and initialises it to A
<code>function keyPressed()</code>	This function waits for a key to be pressed before actioning the code in the function
<code>targetLabel = key.toUpperCase()</code>	Whatever key is pressed that is the new target label until another key is pressed, it capitalises any key pressed
<code>text(targetLabel, mouseX, mouseY)</code>	Now it puts the default key A unless another is pressed

Figure A9.5





Sketch A9.6 collect training data

Now we are going to do a bit of rearranging because we want to collect the data points every time we click on the canvas. To do that, we move the inputs and outputs object arrays into the `mousePressed()` function and then add the data to the neural network each time the mouse is clicked, inputting the target `label`. We use the `ml5.js` function `nn.addData()`.

```
let nn
let targetLabel = 'A'

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    // inputs: ['x', 'y'],
    // outputs: ['label'],
    task: 'classification'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}

function keyPressed()
{
  targetLabel = key.toUpperCase()
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  let target = {
    label: targetLabel
  }
  nn.addData(inputs, target)
  fill(0, 0, 255, 100)
```

```
circle(mouseX, mouseY, 25)
textAlign(CENTER, CENTER)
stroke(0)
text(targetLabel, mouseX, mouseY)
}
```

Notes

Just to clarify, every time you click and add a circle (with a target label letter) to the canvas, that data is passed to the neural network. You may notice that we didn't call the outputs by that name. This is because we created a new variable name (**target**) for the output.

Challenge

If you want to see the data produced, then add this line of code: **console.log(inputs, targetLabel)** at the end of the **mousePressed()** function. This is not essential but interesting (do one for each letter).

Code Explanation

```
nn.addData(inputs, target)
```

This function adds the data to the neural network. Inputs are the x and y co-ordinates and the output is called target which is the label (letter)



Sketch A9.7 button

We are going to use the data to train our model, but first we are going to create a button that, when we press it, we `train()` the model. But first, we will just get a console message to make sure the button is working.

```
let nn
let targetLabel = 'A'
let button

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: 'classification'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key.toUpperCase()
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  let target = {
    label: targetLabel
  }
  nn.addData(inputs, target)
```

```
fill(0, 0, 255, 100)
circle(mouseX, mouseY, 25)
textAlign(CENTER, CENTER)
stroke(0)
text(targetLabel, mouseX, mouseY)
}
```

```
function train()
{
  console.log('button working')
}
```

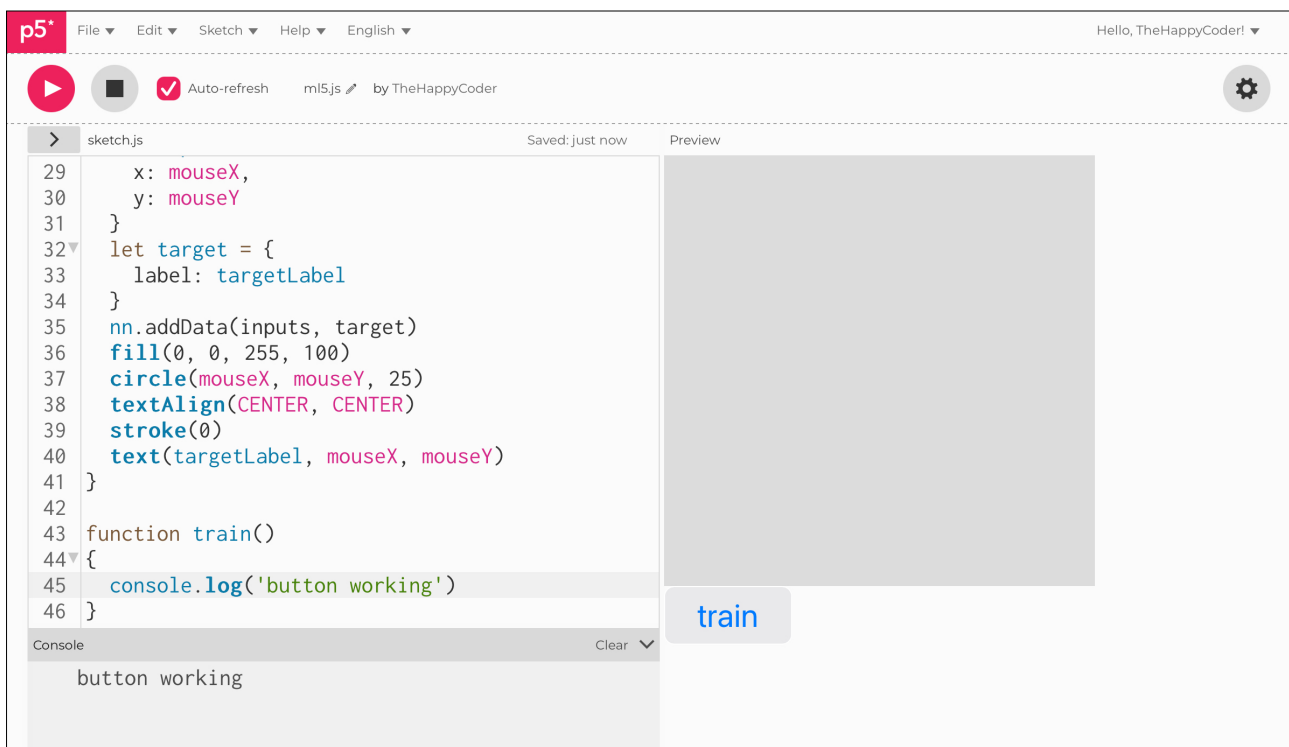

Notes

This just creates the button and checks to see if it is working, so you should get a message to that effect in the console. This is just temporary until we use the button to start the training process after we have our four sets of ten data points (target labels).

Code Explanation

<code>button.mousePressed(train)</code>	We activate the <code>train()</code> function when we click the button
<code>console.log('button working')</code>	This is where we are going to train our neural network

Figure A9.7





Sketch A9.8 training

If the button works OK, then remove the console log and add the **debug** in the neural network options. In the **train()** function, we start training the model. Add the data points (circles) for **A**, **B**, **C**, and **D** before clicking on the train button.

! We need to add a comma (,) after the word **'classification'**.

```
let nn
let targetLabel = 'A'
let button

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: 'classification',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key.toUpperCase()
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  let target = {
    label: targetLabel
  }
```

```
}  
nn.addData(inputs, target)  
fill(0, 0, 255, 100)  
circle(mouseX, mouseY, 25)  
textAlign(CENTER, CENTER)  
stroke(0)  
text(targetLabel, mouseX, mouseY)  
}  
  
function train()  
{  
  nn.train()  
}
```

Notes

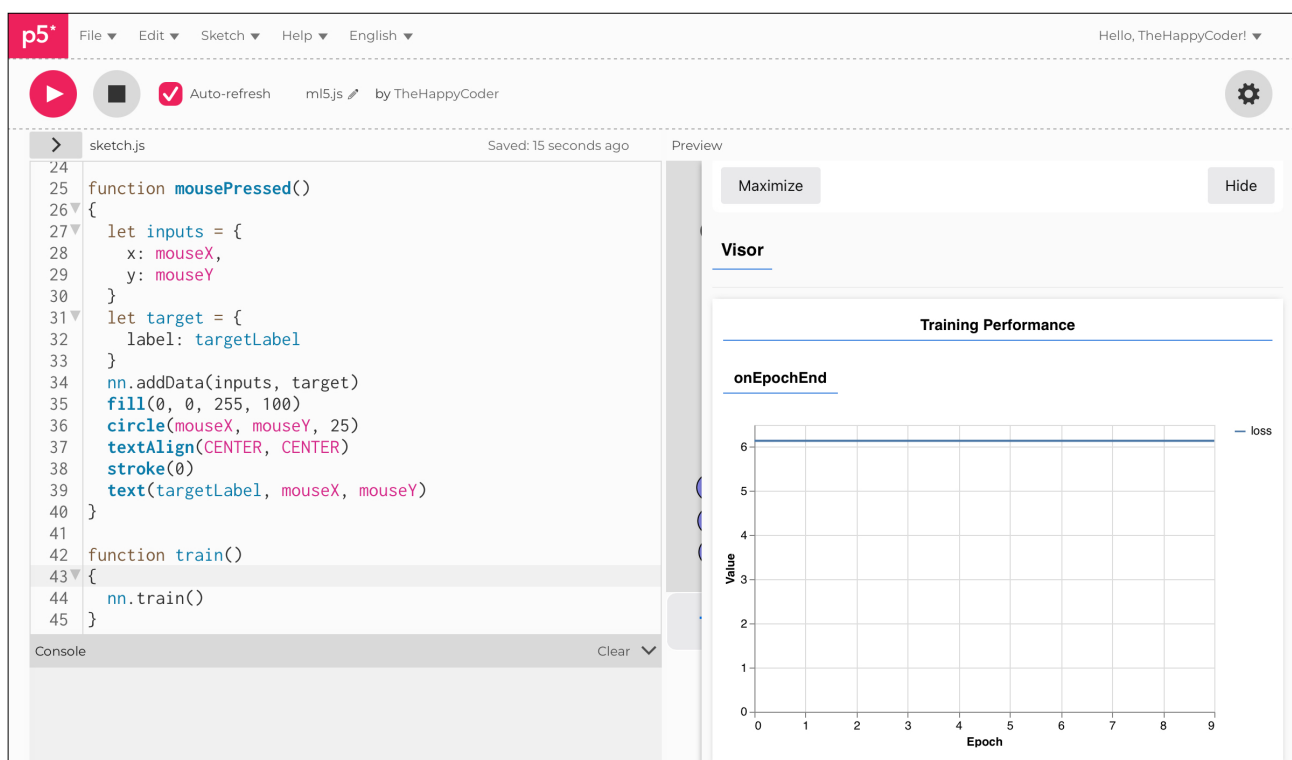
When we do this, we get the following results, which are extremely poor. In this example, it was around 6 or 7, not good at all. There is a very good reason for this, and it is all to do with the size of the data points we have collected. The model works on the basis that all the values (including the inputs) are small; this is called **normalisation**. This was deliberate to illustrate what difference it makes and how essential it is. The neural network only really works with values between **0** and **1** (or **-1** and **+1**). We have **x** and **y** values in the region of up to **400**. We use the normalisation function **normalizeData()**.

Code Explanation

```
nn.train()
```

Instead of a console log we train the neural network instead

Figure A9.8





Sketch A9.9 normalisation

We need to improve things a little bit. Two things we are going to do: one is to normalise the data, which means putting the values between **0** and **1**; for that, we use the `normalizeData()` function. The other thing is to specify the number of epochs. In the example from the training performance above, there were only ten epochs; we are increasing the number to **100**. We also need a callback function: `finishedTraining()`.

```
let nn
let targetLabel = 'A'
let button

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: 'classification',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key.toUpperCase()
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  let target = {
```

```
    label: targetLabel
  }
  nn.addData(inputs, target)
  fill(0, 0, 255, 100)
  circle(mouseX, mouseY, 25)
  textAlign(CENTER, CENTER)
  stroke(0)
  text(targetLabel, mouseX, mouseY)
}

function train()
{
  nn.normalizeData()
  let options = {
    epochs: 100
  }
  nn.train(options, finishedTraining)
}

function finishedTraining()
{
  console.log('finished training')
}
```

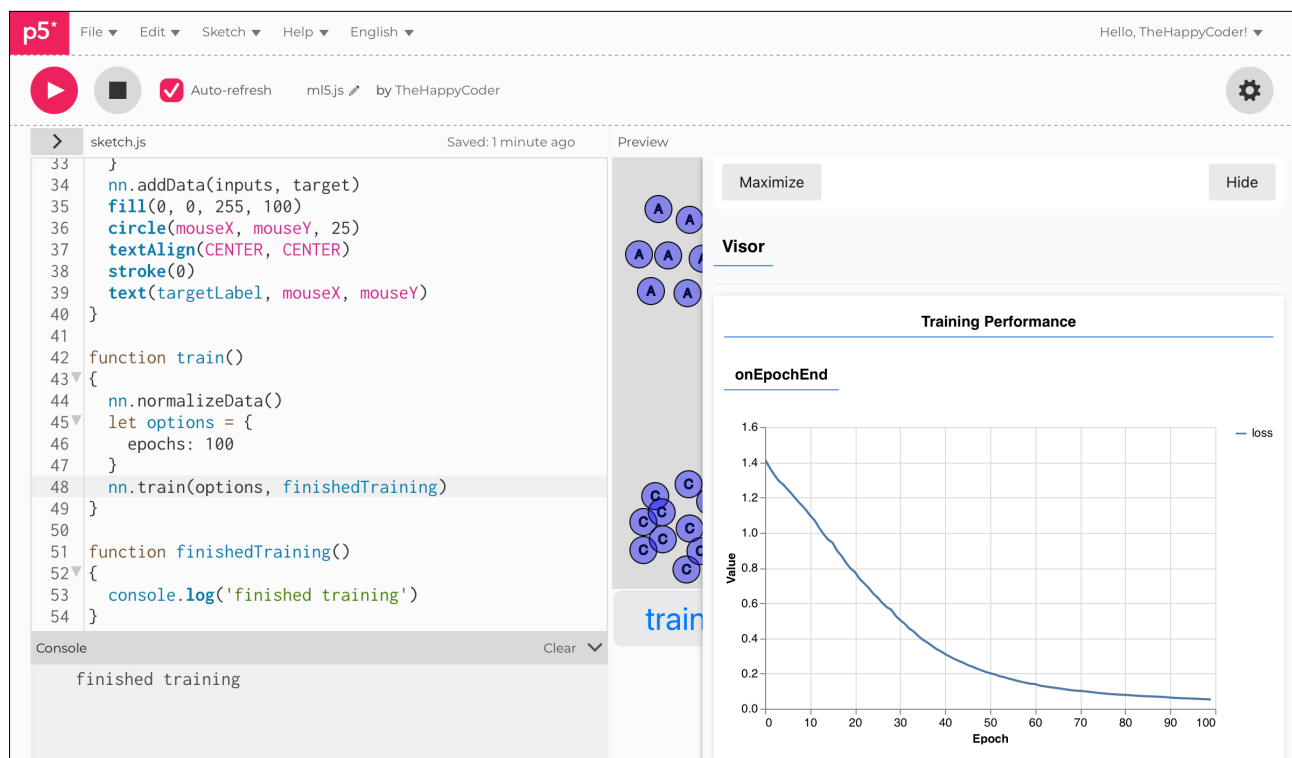
Notes

You will get a **finished training** comment in the console; also, the graph will look considerably better, very close to zero.

Code Explanation

<code>nn.normalizeData()</code>	Normalises the data between 0 and 1
<code>epochs: 100</code>	Go through the whole training data 100 times rather than just the 10

Figure A9.9





Sketch A9.10 adding a bit of colour

Before we go any further, I want to add a bit of colour to the proceedings. Each letter will have a given colour:

A is blue

B is red

C is green

D is yellow

This will give a more visual prediction later.

! Replace the `fill(0, 0, 255, 100)` with the highlighted code

```
let nn
let targetLabel = 'A'
let button

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: 'classification',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key.toUpperCase()
}

function mousePressed()
{
  let inputs = {
```



```

    x: mouseX,
    y: mouseY
  }
  let target = {
    label: targetLabel
  }
  nn.addData(inputs, target)

  if (targetLabel === 'A')
  {
    fill(0, 0, 255, 100)
  }
  if (targetLabel === 'B')
  {
    fill(255, 0, 0, 100)
  }
  if (targetLabel === 'C')
  {
    fill(0, 255, 0, 100)
  }
  if (targetLabel === 'D')
  {
    fill(255, 255, 0, 100)
  }

  circle(mouseX, mouseY, 25)
  textAlign(CENTER, CENTER)
  stroke(0)
  text(targetLabel, mouseX, mouseY)
}

function train()
{
  nn.normalizeData()
  let options = {
    epochs: 100
  }
  nn.train(options, finishedTraining)
}

```

```
function finishedTraining()
{
  console.log('finished training')
}
```

Notes

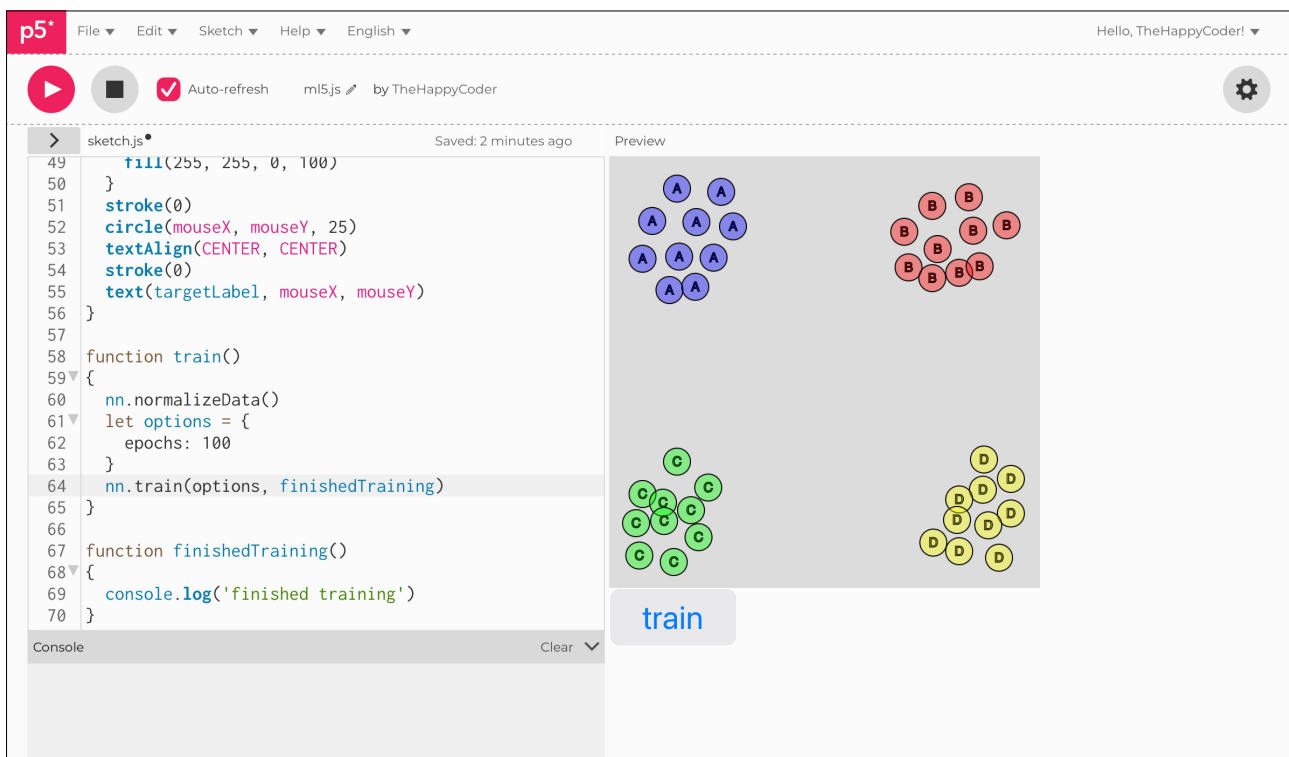
We have a separate colour for each cluster of data points; we just use several `if()` statements. The `===` means it has to be exactly that string.

Code Explanation

```
if (targetLabel === 'A')
```

If the label is A then use the following colour fill

Figure A9.10





Sketch A9.11 a state of mind

We create a simple variable expression called **state**; (same as status in previous model) it helps us keep track of where we are in the process. We will have three states. The first state is to collect the data (**collection**), we then train the model on that data (**training**), and finally we predict what happens when we click somewhere on the canvas (**predicting**). In **mousePressed()**, we move a whole block of code inside an **if()** statement.

```
let nn
let targetLabel = 'A'
let button
let state = 'collection'

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: 'classification',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key.toUpperCase()
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
```

```

if (state == 'collection')
{
    let target = {
        label: targetLabel
    }
    nn.addData(inputs, target)
    if (targetLabel === 'A')
    {
        fill(0, 0, 255, 100)
    }
    if (targetLabel === 'B')
    {
        fill(255, 0, 0, 100)
    }
    if (targetLabel === 'C')
    {
        fill(0, 255, 0, 100)
    }
    if (targetLabel === 'D')
    {
        fill(255, 255, 0, 100)
    }
    stroke(0)
    circle(mouseX, mouseY, 25)
    textAlign(CENTER, CENTER)
    stroke(0)
    text(targetLabel, mouseX, mouseY)
}
}

function train()
{
    state = 'training'
    nn.normalizeData()
    let options = {
        epochs: 100
    }
    nn.train(options, finishedTraining)
}

```

```
}

function finishedTraining()
{
  console.log('finished training')
  state = 'prediction'
}
```

Notes

A bit of refactoring, not really adding lots of new code. The change of state is something like changing the mode setting on a piece of equipment.

Code Explanation

let state = 'collection'	Define and initialise variable string to collection
state = 'training'	Change the string to training
state = 'prediction'	Change again to prediction



Sketch A9.12 classifying

The final stage is the prediction or classification. We want to predict what every pixel of the canvas would be if you were to create a new data point. We create a new callback function called `gotResults()` but only after we have run the classifier.

When you have trained the model, click on the canvas (anywhere) and as you move the mouse around, you will start colouring in the predicted colour for that particular coordinate on the canvas. We are filling the circle with the predicted colour at that coordinate on the canvas.

```
let nn
let targetLabel = 'A'
let button
let state = 'collection'

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    task: 'classification',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key.toUpperCase()
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
```

```

    y: mouseY
  }
  if (state == 'collection')
  {
    let target = {
      label: targetLabel
    }
    nn.addData(inputs, target)
    if (targetLabel === 'A')
    {
      fill(0, 0, 255, 100)
    }
    if (targetLabel === 'B')
    {
      fill(255, 0, 0, 100)
    }
    if (targetLabel === 'C')
    {
      fill(0, 255, 0, 100)
    }
    if (targetLabel === 'D')
    {
      fill(255, 255, 0, 100)
    }
    circle(mouseX, mouseY, 25)
    textAlign(CENTER, CENTER)
    stroke(0)
    text(targetLabel, mouseX, mouseY)
  }
  else if (state == 'prediction')
  {
    nn.classify(inputs, gotResults)
  }
}

function train()
{
  state = 'training'

```

```
nn.normalizeData()
let options = {
  epochs: 100
}
nn.train(options, finishedTraining)
}
```

```
function finishedTraining()
{
  console.log('finished training')
  state = 'prediction'
}
```

```
function gotResults(results)
{
  mousePressed()
  if (results[0].label === 'A')
  {
    fill(0, 0, 255, 100)
  }
  if (results[0].label === 'B')
  {
    fill(255, 0, 0, 100)
  }
  if (results[0].label === 'C')
  {
    fill(0, 255, 0, 100)
  }
  if (results[0].label === 'D')
  {
    fill(255, 255, 0, 100)
  }
  noStroke()
  circle(mouseX, mouseY, 25)
}
```


Notes

Each coordinate will have a predicted colour value depending on where that particular pixel point is. The results will classify it and draw the corresponding coloured circle. Think of it as filling in all the blanks where you had no data; it has interpreted the rest of the canvas for you.

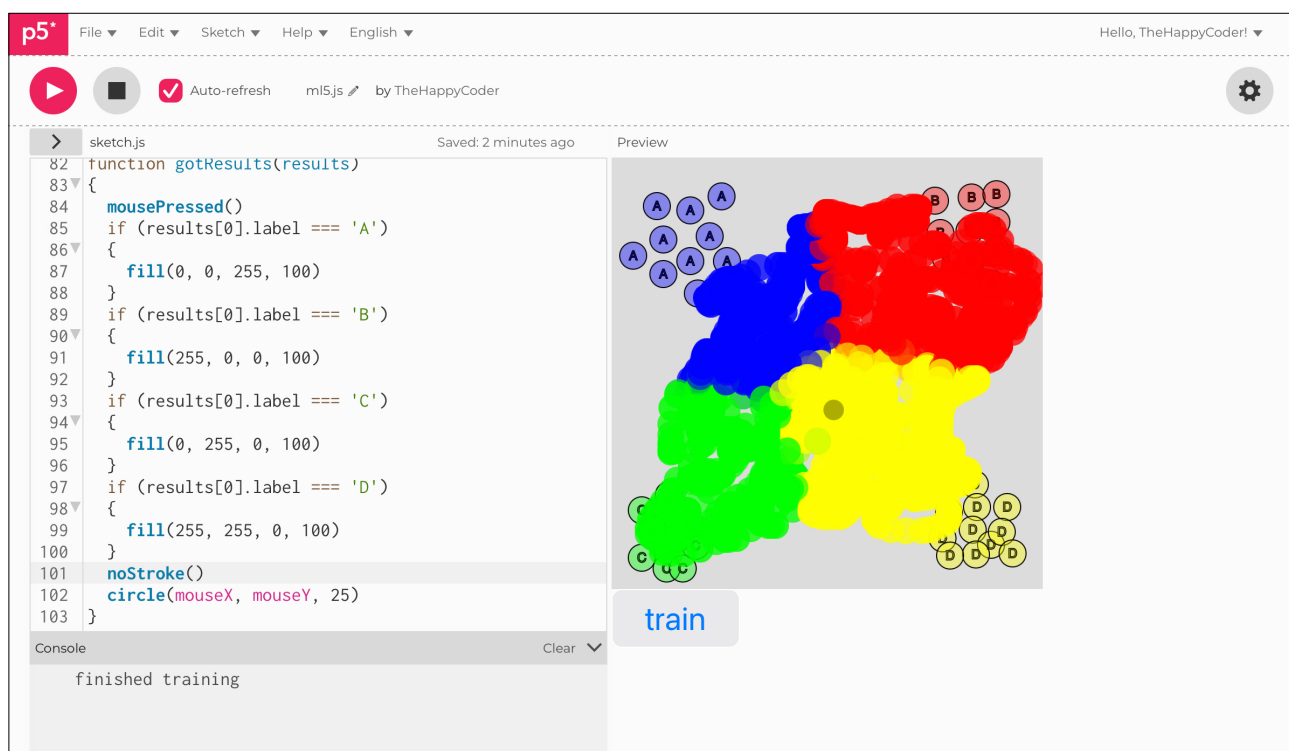
Challenge

Try different shapes of data points e.g a swirl, concentric circles (you may need a lot more data points). Also, try drawing pixels instead of circles (although it might take longer to fill the canvas).

Code Explanation

<code>nn.classify(inputs, gotResults)</code>	The classify() function takes the inputs and returns the results whether it is A, B, C or D
<code>if (results[0].label === 'A')</code>	It makes the prediction for that x, y value input and checks to see whether it is A, B, C or D and colours appropriately

Figure A9.12



Algorithmic Intelligence Module A Unit #10 cluster regression



Module A Unit #10: cluster regression

In this module, we will describe a simple model for **regression**. This is very similar to the **classification** one you have just done. The main difference is that the outcome (output) is a value between **0** and **255**, compared to the **classification** of the labels **A**, **B**, **C**, or **D**.

Here, we will click on the canvas to create a cluster of white circles and a cluster of black circles. We will train it on those data points and then predict what shade of grey the canvas is between those data points. It is, in effect, going to interpolate using a **Neural Network**.



The index.html file

The `index.html` file and the requisite `ml5.js` library.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch A10.1 basic starting model

This is our starting sketch. We are creating the basic neural network model, similar to the previous one. There are two inputs, **x** and **y**, and one output, **val**, which will be the grey value between **0** and **255**. The task is **regression**, not **classification**.

```
let nn

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}
```

Notes

The main points here are:

- A) The task is now **regression**.
- B) The inputs are still **x**, and **y**.
- C) The output is a value called **val**.

Notice that the inputs are an array of two elements.

Code Explanation

task: 'regression'	Defines this as a regression task
--------------------	-----------------------------------



Sketch A10.2 assigning a value

We want to assign a value to the circle (rather than a label). To keep it simple, we are using **w** for white (with a value of **255**) and **b** for black (assigning a value of **0**). These are the fill colours (and values); they will be the data we train the model on.

```
let nn

let targetLabel = 'w'
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}
```

Notes

We will have two target labels, **w** for white and **b** for black.

Code Explanation

<code>let targetLabel = 'w'</code>	We define and initialise the target label as w
<code>let col = {w: 255, b: 0}</code>	This is defining the values of the target labels as an object array



Sketch A10.3 assigning the keys

When we want to change the letter of the `targetLabel`, we use the `keyPressed()` function to define the label and attribute a value to the data point. So, with a `w` we will give it a value of `255`, and with `b`, we will give it a value of `0`.

```
let nn
let targetLabel = 'w'
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}

function keyPressed()
{
  targetLabel = key
}
```

Notes

All that happens at this stage is that we are changing the value of the target label.

Code Explanation

`targetLabel = key`

The variable `targetLabel` will take whatever letter you type in. Which should be the letter `b` or `w` for this exercise



Sketch A10.4 drawing the circles

There is quite a lot happening here. We have a `targetLabel` of `w` which has an initial value. The `col()` function takes the value from the object array `col = {w: 255, b: 0}` and gives it to the `targetVal` variable, which in turn is used to `fill()` the circle. We aren't bothering with the letters in the circles this time, just the colour. We introduce the `mousePressed()` function to draw the circles when we click on the canvas.

```
let nn
let targetLabel = 'w'
let state = 'collection'
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}

function keyPressed()
{
  targetLabel = key
}

function mousePressed()
{
  let targetVal = col[targetLabel]
  fill(targetVal)
```



```

circle(mouseX, mouseY, 25)
}

```

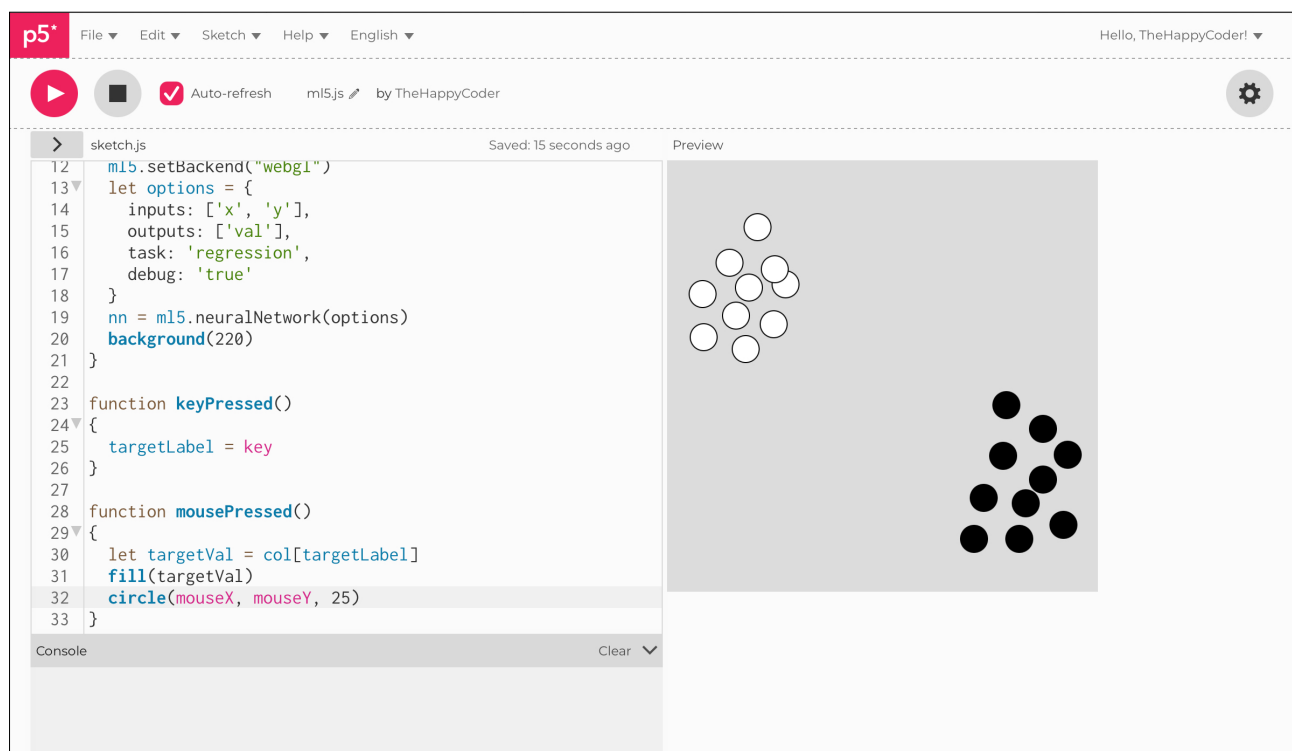
Notes

When you click on the canvas, you get the white circle by default. When you press the letter **b** and then click on the canvas, the circle is black (0). This continues until you press **w**, then it will fill the circle with white (255).

Code Explanation

<code>let targetVal = col[targetLabel]</code>	Takes the value of the target label, it is an object array hence the square brackets []
<code>fill(targetVal)</code>	Fills the circle with the value of the target label
<code>circle(mouseX, mouseY, 25)</code>	Draws the circle at the coordinates of the mousePressed()

Figure A10.4





Sketch A10.5 defining the inputs

We want to collect the data points from the canvas, through the `mousePressed()` function. The input data is the `mouseX` and `mouseY` coordinates.

```
let nn
let targetLabel = 'w'
let state = 'collection'
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}

function keyPressed()
{
  targetLabel = key
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }

  let targetVal = col[targetLabel]
```

```
fill(targetVal)
circle(mouseX, mouseY, 25)
}
```

Notes

All we have done is define the inputs, which you should be familiar with.



Sketch A10.6 collecting the data

We introduce the variable **state** to keep track of proceedings. We initialise it to **collection** because we first need to collect the data. We need the inputs for the data collection to be an array, so we create a data object to pass into the neural network. We create an **if()** statement inside the **mousePressed()** function and put some lines of code within that **if()** statement. We only want to draw the circles when we are collecting the data.

```
let nn
let targetLabel = 'w'
let state = 'collection'
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
}

function keyPressed()
{
  targetLabel = key
}

function mousePressed()
{
  let inputs = {
    x: mouseX,
```

```

    y: mouseY
  }
  if (state == 'collection')
  {
    let targetVal = col[targetLabel]
    let target = {
      val: targetVal
    }
    nn.addData(inputs, target)
    fill(targetVal)
    circle(mouseX, mouseY, 25)
  }
}

```

Notes

The data passed into the neural network are two object arrays, the inputs and the target value. We can write these objects as one line of code if we wish: **target = {val: targetVal}**. If there are many, it is nicer to see them as a list.

Code Explanation

<code>if (state == 'collection')</code>	Check what state we are in
<code>let target = {val: targetVal}</code>	We take the value of the target label as an object array
<code>nn.addData(inputs, target)</code>	We can now pass in the data to the neural network ready for training



Sketch A10.7 training button

Let us create a button to press to train the model. For the moment, we have an empty `train()` function, but we will soon put some code in. We only want to train it after we have finished adding the data points.

```
let nn
let targetLabel = 'w'
let state = 'collection'
let button
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)

  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key
}

function mousePressed()
{

```

```
let inputs = {
  x: mouseX,
  y: mouseY
}
if (state == 'collection')
{
  let targetVal = col[targetLabel]
  let target = {
    val: targetVal
  }
  nn.addData(inputs, target)
  fill(targetVal)
  circle(mouseX, mouseY, 25)
}
```

```
function train()
{
  console.log('button working')
}
```

Notes

We have a button that doesn't do anything except tell you it is working when you click on it!

Challenge

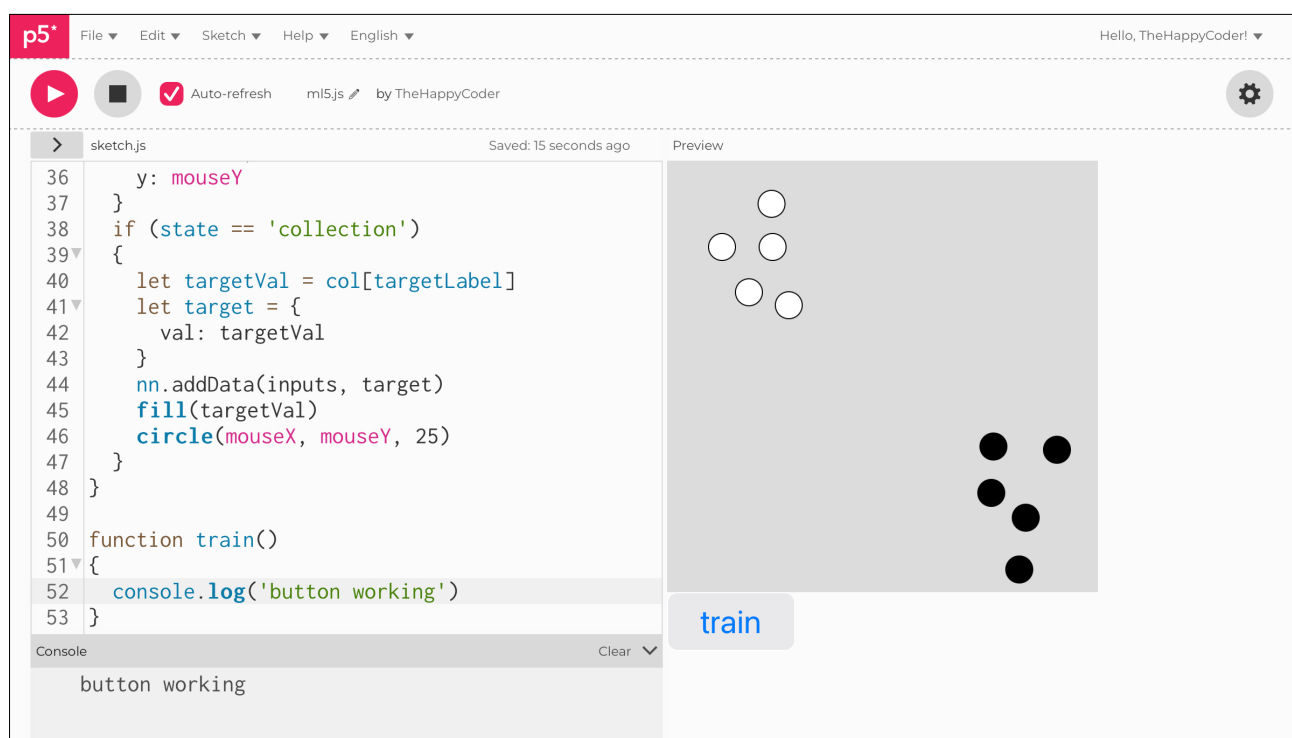
Add some colour to the button.

Code Explanation

`button.mousePressed(train)`

When the button is pressed it calls the `train()` function

Figure A10.7





Sketch A10.8 training

Here we will add the code for training the model and a callback for when training is finished. The **state** is now **training**. We can also add the next function, **finishedTraining()**, and change the state to **prediction**. The console will tell us when it has finished training and is ready to predict.

```
let nn
let targetLabel = 'w'
let state = 'collection'
let button
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key
}

function mousePressed()
```

```
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  if (state == 'collection')
  {
    let targetVal = col[targetLabel]
    let target = {
      val: targetVal
    }
    nn.addData(inputs, target)
    fill(targetVal)
    circle(mouseX, mouseY, 25)
  }
}
```

```
function train()
{
  state = 'training'
  nn.normalizeData()
  nn.train(finishedTraining)
}
```

```
function finishedTraining()
{
  console.log('finished training')
  state = 'prediction'
}
```

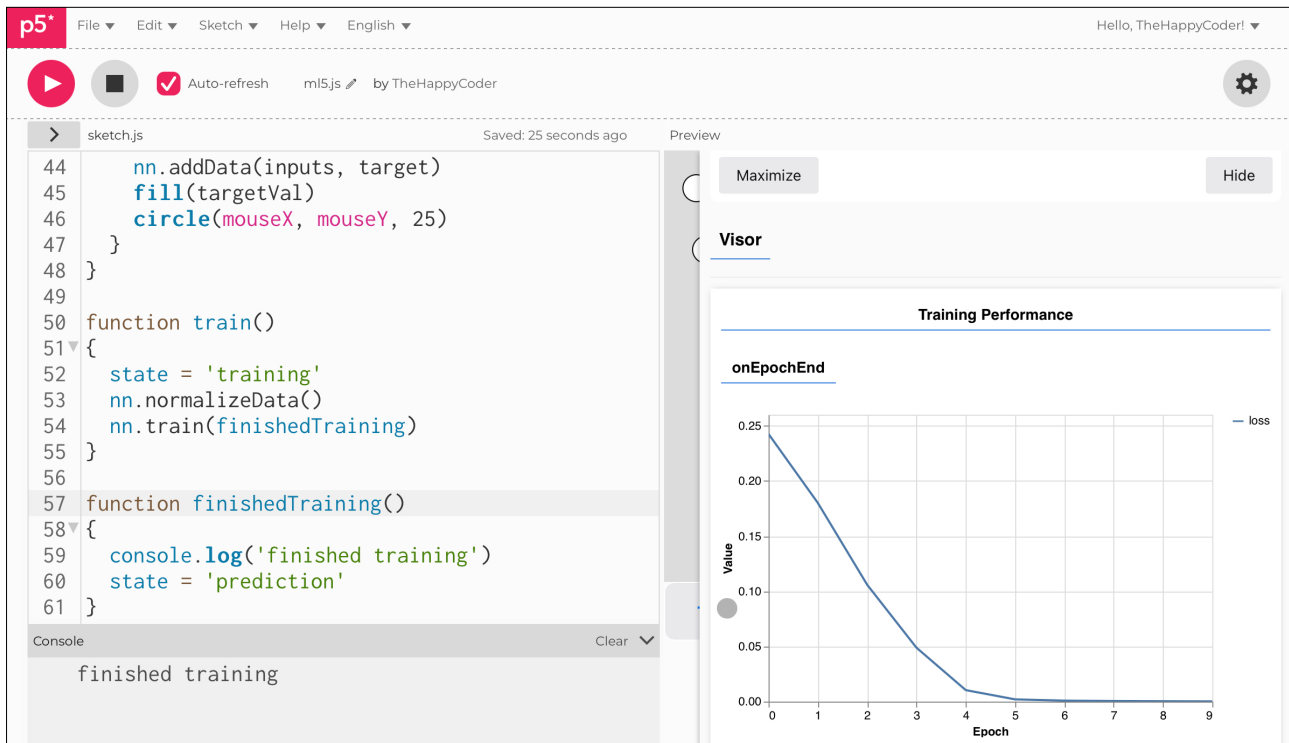
Notes

As you train the model, you may well find that the loss curve plummets to zero in less than ten epochs. This is probably because there may not be enough data points.

Challenges

1. Try more data points.
2. Try just one of each data point.

Figure A10.8





Sketch A10.9 predicting

The next part is to predict the results, to determine the colour of the pixel when we draw our circle on the canvas. We will want a greyscale of colour, not just black or white, because a regression task gives us a sliding scale of values.

We have set the state to prediction already. Next, we add an `if()` statement to check to see if there is a change of `state` to `prediction` inside the `mousePressed()` function to start making those predictions with the `predict()` function. We will pick up the results in the callback function `gotResults()`.

! Don't run this yet, we haven't added the callback function yet.

```
let nn
let targetLabel = 'w'
let state = 'collection'
let button
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{

```

```

    targetLabel = key
  }

function mousePressed()
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  if (state == 'collection')
  {
    let targetVal = col[targetLabel]
    let target = {
      val: targetVal
    }
    nn.addData(inputs, target)
    fill(targetVal)
    circle(mouseX, mouseY, 25)
  }
  else if (state == 'prediction')
  {
    nn.predict(inputs, gotResults)
  }
}

function train()
{
  state = 'training'
  nn.normalizeData()
  nn.train(finishedTraining)
}

function finishedTraining()
{
  console.log('finished training')
  state = 'prediction'
}

```

Notes

After we have finished **collecting** the data and finished **training** the model, we are then in **prediction** mode. Next is the callback function **gotResults()**, where we see what our results look like. The lack of the **gotResults()** function will give you an error.

Code Explanation

```
nn.predict(inputs, gotResults)
```

We receive the inputs and have a callback function for the results



Sketch A10.10 prediction results

We need to add in the function `gotResults()` so we can see the results or **predictions**. After the training is completed, you will see the results when you click on the canvas. The circles should be a bit grey in between where your data points are; for each circle, it should fill with the predicted colour value.

```
let nn
let targetLabel = 'w'
let state = 'collection'
let button
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key
}

function mousePressed()
```

```
{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  if (state == 'collection')
  {
    let targetVal = col[targetLabel]
    let target = {
      val: targetVal
    }
    nn.addData(inputs, target)
    fill(targetVal)
    circle(mouseX, mouseY, 25)
  }
  else if (state == 'prediction')
  {
    nn.predict(inputs, gotResults)
  }
}
```

```
function train()
```

```
{
  state = 'training'
  nn.normalizeData()
  nn.train(finishedTraining)
}
```

```
function finishedTraining()
```

```
{
  console.log('finished training')
  state = 'prediction'
}
```

```
function gotResults(results)
```

```
{
  fill(floor(results[0].value))
  circle(mouseX, mouseY, 25)
```



```
}
```

Notes

This is the difference between **classification** and **regression**. You can see the gradual change in the fill colour of the circles as you click between the trained clusters of circles.

Challenge

You can look inside the results using the `console.log(results)` as shown in the second image below.

Code Explanation

<code>results[0].value</code>	This pulls the value from the prediction results array at index [0]
<code>fill(floor(results[0].value))</code>	We use the floor so it is an integer

Figure A10.10a

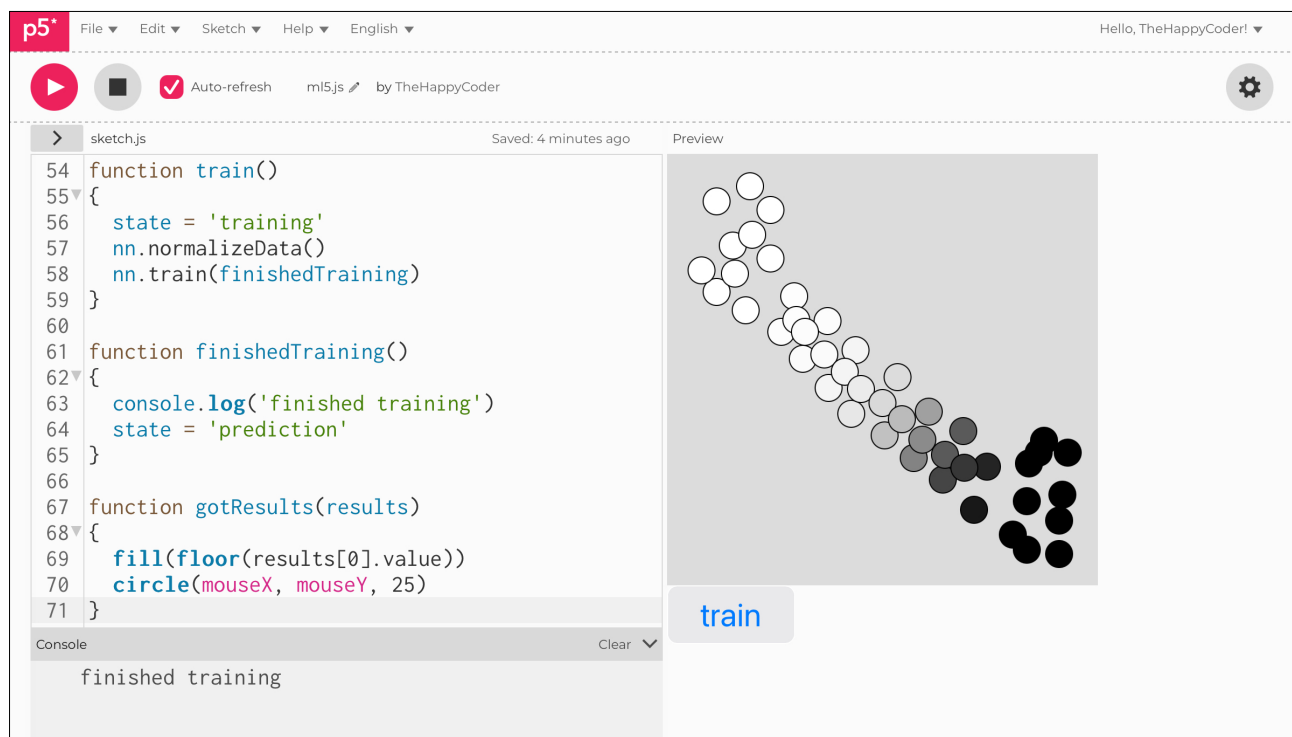
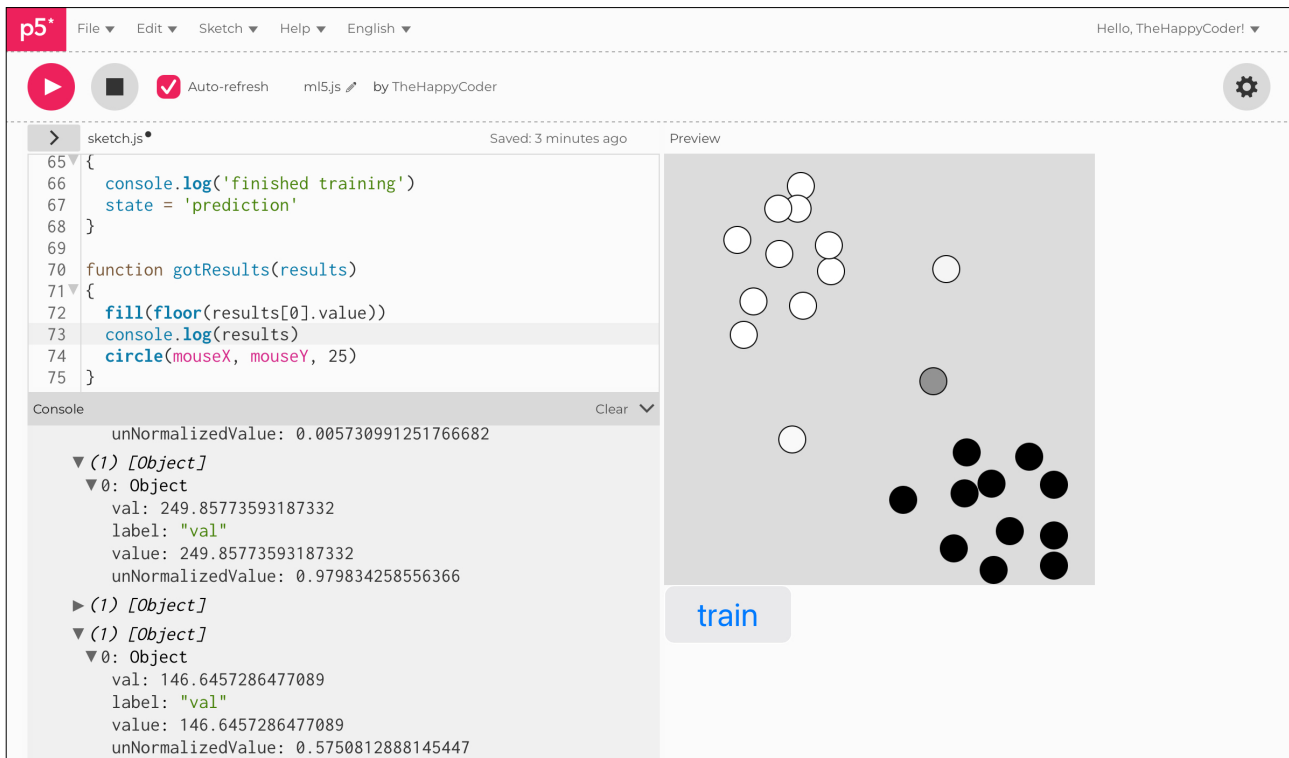


Figure A10.10b: with console.log(results)





Sketch A10.11 some refinements

I have highlighted some refinements. This will keep the original circles and their colour, but now when you click on the canvas, it will draw the shades of grey on the canvas.

! Comment out debugging to hide the loss chart.

```
let nn
let targetLabel = 'w'
let state = 'collection'
let button
let col = {
  w: 255,
  b: 0
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  let options = {
    inputs: ['x', 'y'],
    outputs: ['val'],
    task: 'regression',
    // debug: 'true'
  }
  nn = ml5.neuralNetwork(options)
  background(220)
  button = createButton('train')
  button.style('font-size', '30px')
  button.mousePressed(train)
}

function keyPressed()
{
  targetLabel = key
}

function mousePressed()
```

```

{
  let inputs = {
    x: mouseX,
    y: mouseY
  }
  if (state == 'collection')
  {
    let targetVal = col[targetLabel]
    let target = {
      val: targetVal
    }
    nn.addData(inputs, target)
    fill(targetVal)
    circle(mouseX, mouseY, 25)
  }
  else if (state == 'prediction')
  {
    nn.predict(inputs, gotResults)
  }
}

function train()
{
  state = 'training'
  nn.normalizeData()
  nn.train(finishedTraining)
}

function finishedTraining()
{
  console.log('finished training')
  state = 'prediction'
}

function gotResults(results)
{
  mousePressed()
  {

```

```

fill(floor(results[0].value), 100)
noStroke()
circle(mouseX, mouseY, 30)
}
}

```

Notes

We have added a little transparency and removed the stroke, and created a larger circle.

Challenge

Consider how you might colour each pixel (not easy).

Figure A10.11a as we move the mouse

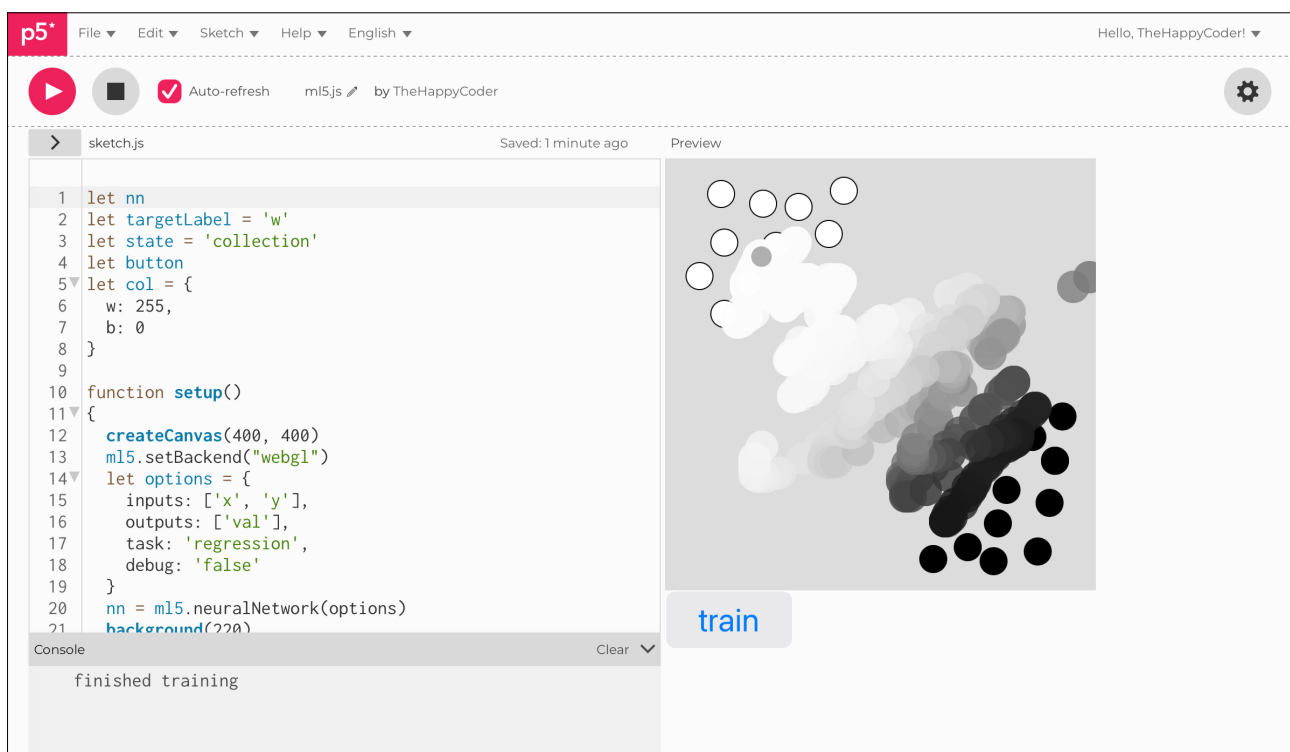
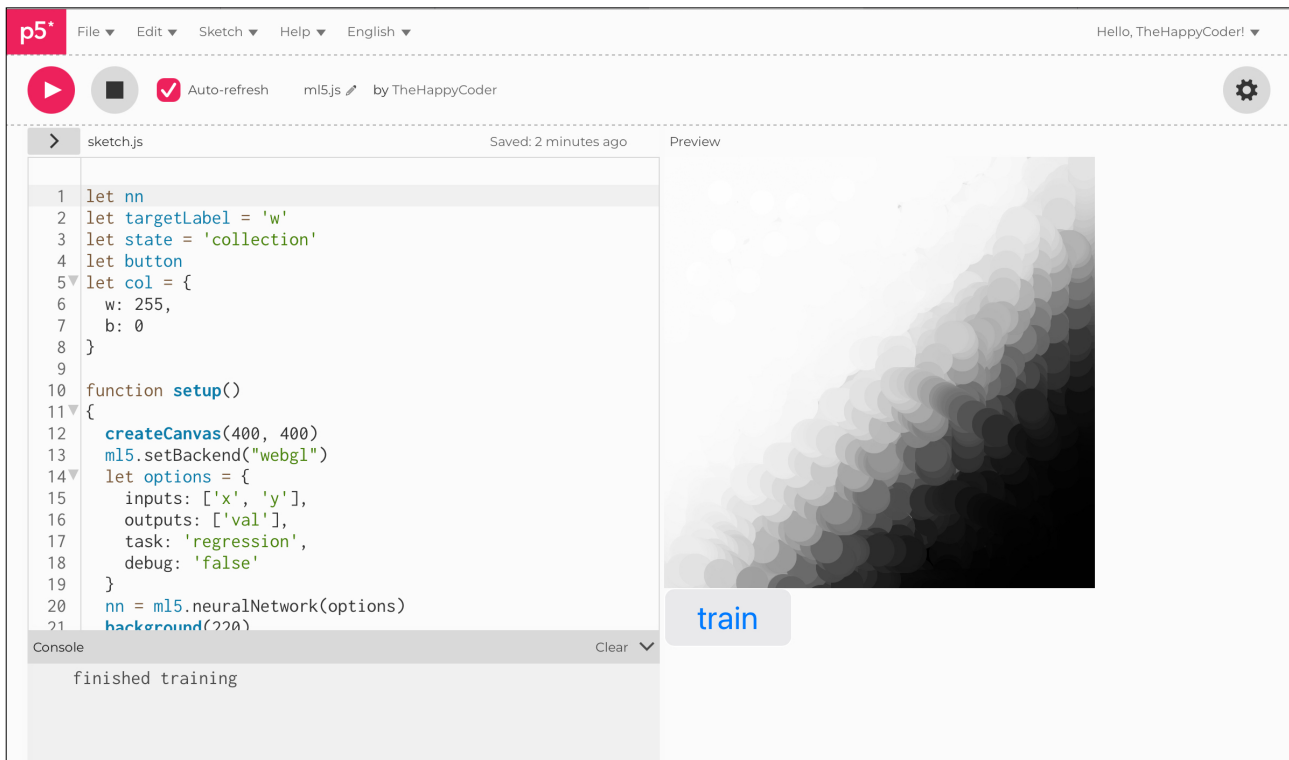


Figure A10.11b final look



Algorithmic Intelligence Module A Unit #11 colour predictor



Module A Unit #11: colour predictor

Another seemingly simple exercise is to train the model to identify **reddish**, **greenish**, and **blueish** colours. For this, we will provide a small synthetic dataset in the form of a **JSON-style** file. We will train the neural network on that data and see if it can learn these three classifications for a variety of combinations of the RGB values.

This is, therefore, a **classification** task as we want it to select, through the built-in softmax function, the most likely colour group. We will use three sliders to change the amount of **red**, **green**, and **blue**.



The index.html

A reminder to add the `ml5.js` line of code if you haven't already.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch A11.1 creating the data

We are going to create three labels: **red-ish**, **green-ish**, and **blue-ish**. For each of these three labels, we are going to give them the **RGB** numbers that are pretty close. For example, **red-ish** is (255, 0, 0), (254, 0, 0), and (253, 0, 0), and do the same for the **green-ish** and **blue-ish**.

```
let data = [
  { r: 255, g: 0, b: 0, colour: "red-ish" },
  { r: 254, g: 0, b: 0, colour: "red-ish" },
  { r: 253, g: 0, b: 0, colour: "red-ish" },
  { r: 0, g: 255, b: 0, colour: "green-ish" },
  { r: 0, g: 254, b: 0, colour: "green-ish" },
  { r: 0, g: 253, b: 0, colour: "green-ish" },
  { r: 0, g: 0, b: 255, colour: "blue-ish" },
  { r: 0, g: 0, b: 254, colour: "blue-ish" },
  { r: 0, g: 0, b: 253, colour: "blue-ish" },
]

function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
}
```

Notes

This is another small dataset; in reality, you would have a much larger dataset based on people's options (labels), but this is simple enough to be useful.

Challenge

You could add more data.

Code Explanation

```
let data = [{r:, g:, b:, colour:},]
```

Creating a series of objects in an array called data. These will be the input data



Sketch A11.2 Sliders

We are going to create three sliders to control the colour of the background. We will give it an initial colour of red.

```
let data = [
  { r: 255, g: 0, b: 0, colour: "red-ish" },
  { r: 254, g: 0, b: 0, colour: "red-ish" },
  { r: 253, g: 0, b: 0, colour: "red-ish" },
  { r: 0, g: 255, b: 0, colour: "green-ish" },
  { r: 0, g: 254, b: 0, colour: "green-ish" },
  { r: 0, g: 253, b: 0, colour: "green-ish" },
  { r: 0, g: 0, b: 255, colour: "blue-ish" },
  { r: 0, g: 0, b: 254, colour: "blue-ish" },
  { r: 0, g: 0, b: 253, colour: "blue-ish" },
]

let r
let g
let b
let rSlider
let gSlider
let bSlider

function setup()
{
  createCanvas(400, 400)
  rSlider = createSlider(0, 255, 255).position(10, 20)
  gSlider = createSlider(0, 255, 0).position(10, 40)
  bSlider = createSlider(0, 255, 0).position(10, 60)
}

function draw()
{
  r = rSlider.value()
  g = gSlider.value()
  b = bSlider.value()
  background(r, g, b)
}
```

Notes

The slider has three arguments:

- A) the first is the minimum value,
- B) the second is the maximum value, and
- C) the third is the starting value.

We give the red slider `rSlider` a starting value of `255`, which is the maximum.

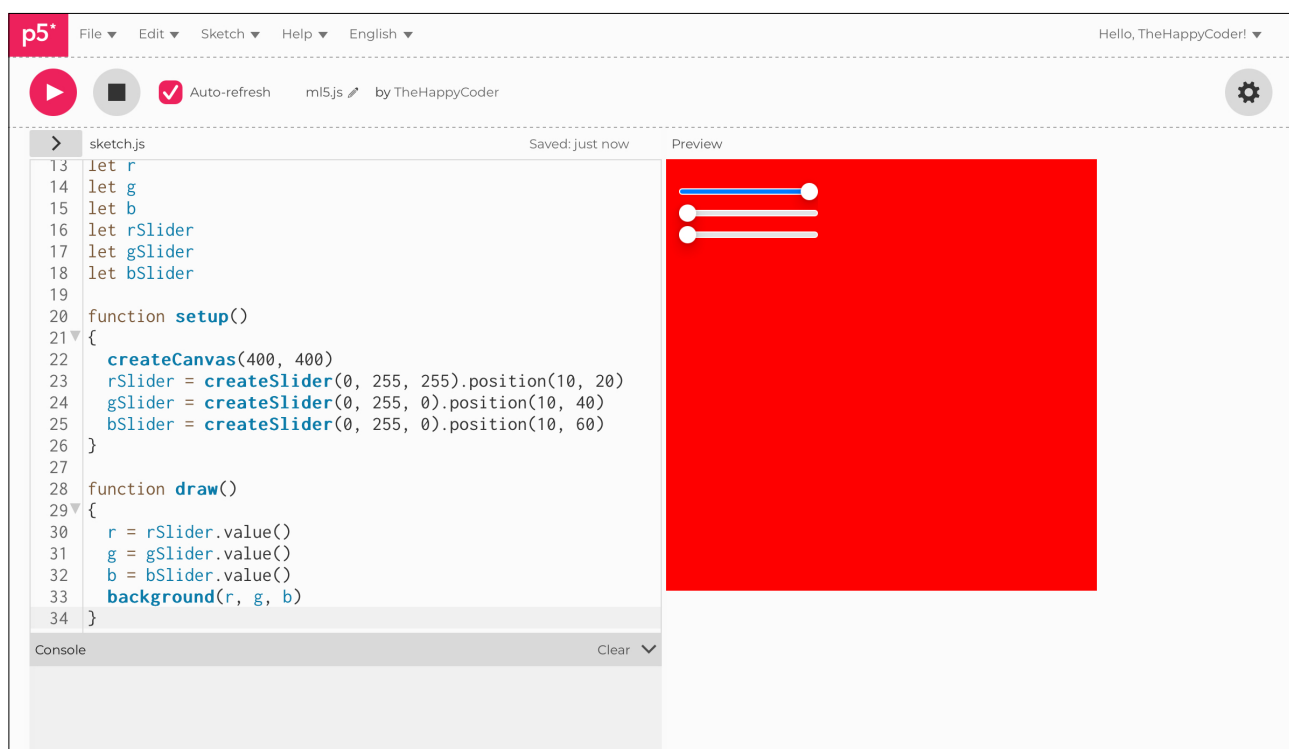
Challenge

Have one of the others start at a different value.

Code Explanation

<code>createSlider(0, 255, 255)</code>	The third argument is the value of the slider
<code>createSlider(...).position(10, 20)</code>	The position is the x (10 pixels) and y (20 pixels) co-ordinates of the slider

Figure A11.2





Sketch A11.3 adding the neural network

Let's introduce our neural network, as we have done numerous times before.

```
let data = [
  { r: 255, g: 0, b: 0, colour: "red-ish" },
  { r: 254, g: 0, b: 0, colour: "red-ish" },
  { r: 253, g: 0, b: 0, colour: "red-ish" },
  { r: 0, g: 255, b: 0, colour: "green-ish" },
  { r: 0, g: 254, b: 0, colour: "green-ish" },
  { r: 0, g: 253, b: 0, colour: "green-ish" },
  { r: 0, g: 0, b: 255, colour: "blue-ish" },
  { r: 0, g: 0, b: 254, colour: "blue-ish" },
  { r: 0, g: 0, b: 253, colour: "blue-ish" },
]

let r
let g
let b
let rSlider
let gSlider
let bSlider

let nn

function setup()
{
  createCanvas(400, 400)
  rSlider = createSlider(0, 255, 255).position(10, 20)
  gSlider = createSlider(0, 255, 0).position(10, 40)
  bSlider = createSlider(0, 255, 0).position(10, 60)

  let options = {
    task: 'classification',
    debug: true
  }
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
}

function draw()
```

```
{  
  r = rSlider.value()  
  g = gSlider.value()  
  b = bSlider.value()  
  background(r, g, b)  
}
```

Notes

As you can see, we have a **classification** task and set the debug to **true**.



Sketch A11.4 adding the data

Now, to add the data to the neural network. We cycle through the **data** array, giving the set of inputs and the colour label to a variable called **item**. This holds the value of the **red**, **green**, and **blue** as well as the colour label. However, we only pull out the **r**, **g**, and **b** values. The **colour** values are the outputs for each set of corresponding **r**, **g**, **b** values.

```
let data = [
  { r: 255, g: 0, b: 0, colour: "red-ish" },
  { r: 254, g: 0, b: 0, colour: "red-ish" },
  { r: 253, g: 0, b: 0, colour: "red-ish" },
  { r: 0, g: 255, b: 0, colour: "green-ish" },
  { r: 0, g: 254, b: 0, colour: "green-ish" },
  { r: 0, g: 253, b: 0, colour: "green-ish" },
  { r: 0, g: 0, b: 255, colour: "blue-ish" },
  { r: 0, g: 0, b: 254, colour: "blue-ish" },
  { r: 0, g: 0, b: 253, colour: "blue-ish" },
]

let r
let g
let b
let rSlider
let gSlider
let bSlider
let nn

function setup()
{
  createCanvas(400, 400)
  rSlider = createSlider(0, 255, 255).position(10, 20)
  gSlider = createSlider(0, 255, 0).position(10, 40)
  bSlider = createSlider(0, 255, 0).position(10, 60)
  let options = {
    task: 'classification',
    debug: true
  }
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
```

```

    for (let i = 0; i < data.length; i++)
    {
        let item = data[i]
        let inputs = [item.r, item.g, item.b]
        let outputs = [item.colour]
        nn.addData(inputs, outputs)
    }
}

function draw()
{
    r = rSlider.value()
    g = gSlider.value()
    b = bSlider.value()
    background(r, g, b)
}

```

Notes

Using the `item` variable seems a bit redundant, but you are taking them from the complete array. It means you don't accidentally change any values in the dataset. This repeats for the length of the dataset.

Challenge

You could replace `item` with `data[i]` and remove all references to the `item` variable.

Code Explanation

<code>let item = data[i]</code>	For each index if the dataset array push it into the item variable
<code>let inputs = [item.r, item.g, item.b]</code>	Each set of inputs are added
<code>let outputs = [item.colour]</code>	The corresponding output is added
<code>nn.addData(inputs, outputs)</code>	Both the inputs and outputs are added to the neural network



Sketch A11.5 normalising the data

Next, we will normalise the data because we don't want very large and very small numbers in the neural network calculations.

```
let data = [
  { r: 255, g: 0, b: 0, colour: "red-ish" },
  { r: 254, g: 0, b: 0, colour: "red-ish" },
  { r: 253, g: 0, b: 0, colour: "red-ish" },
  { r: 0, g: 255, b: 0, colour: "green-ish" },
  { r: 0, g: 254, b: 0, colour: "green-ish" },
  { r: 0, g: 253, b: 0, colour: "green-ish" },
  { r: 0, g: 0, b: 255, colour: "blue-ish" },
  { r: 0, g: 0, b: 254, colour: "blue-ish" },
  { r: 0, g: 0, b: 253, colour: "blue-ish" },
]

let r
let g
let b
let rSlider
let gSlider
let bSlider
let nn

function setup()
{
  createCanvas(400, 400)
  rSlider = createSlider(0, 255, 255).position(10, 20)
  gSlider = createSlider(0, 255, 0).position(10, 40)
  bSlider = createSlider(0, 255, 0).position(10, 60)
  let options = {
    task: 'classification',
    debug: true
  }
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  for (let i = 0; i < data.length; i++)
  {
```

```
    let item = data[i]
    let inputs = [item.r, item.g, item.b]
    let outputs = [item.colour]
    nn.addData(inputs, outputs)
  }
  nn.normalizeData()
}

function draw()
{
  r = rSlider.value()
  g = gSlider.value()
  b = bSlider.value()
  background(r, g, b)
}
```



Sketch A11.6 training the neural network

We will now train our neural network, setting the batch size to **16** and the epochs to **128**.

```
let data = [
  { r: 255, g: 0, b: 0, colour: "red-ish" },
  { r: 254, g: 0, b: 0, colour: "red-ish" },
  { r: 253, g: 0, b: 0, colour: "red-ish" },
  { r: 0, g: 255, b: 0, colour: "green-ish" },
  { r: 0, g: 254, b: 0, colour: "green-ish" },
  { r: 0, g: 253, b: 0, colour: "green-ish" },
  { r: 0, g: 0, b: 255, colour: "blue-ish" },
  { r: 0, g: 0, b: 254, colour: "blue-ish" },
  { r: 0, g: 0, b: 253, colour: "blue-ish" },
]

let r
let g
let b
let rSlider
let gSlider
let bSlider
let nn

function setup()
{
  createCanvas(400, 400)
  rSlider = createSlider(0, 255, 255).position(10, 20)
  gSlider = createSlider(0, 255, 0).position(10, 40)
  bSlider = createSlider(0, 255, 0).position(10, 60)
  let options = {
    task: 'classification',
    debug: true
  }
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  for (let i = 0; i < data.length; i++)
  {
```

```

    let item = data[i]
    let inputs = [item.r, item.g, item.b]
    let outputs = [item.colour]
    nn.addData(inputs, outputs)
  }
  nn.normalizeData()

  const trainingOptions = {
    batchSize: 16,
    epochs: 128
  }
  nn.train(trainingOptions, finishedTraining)
}

function finishedTraining()
{
  console.log("finished training")
}

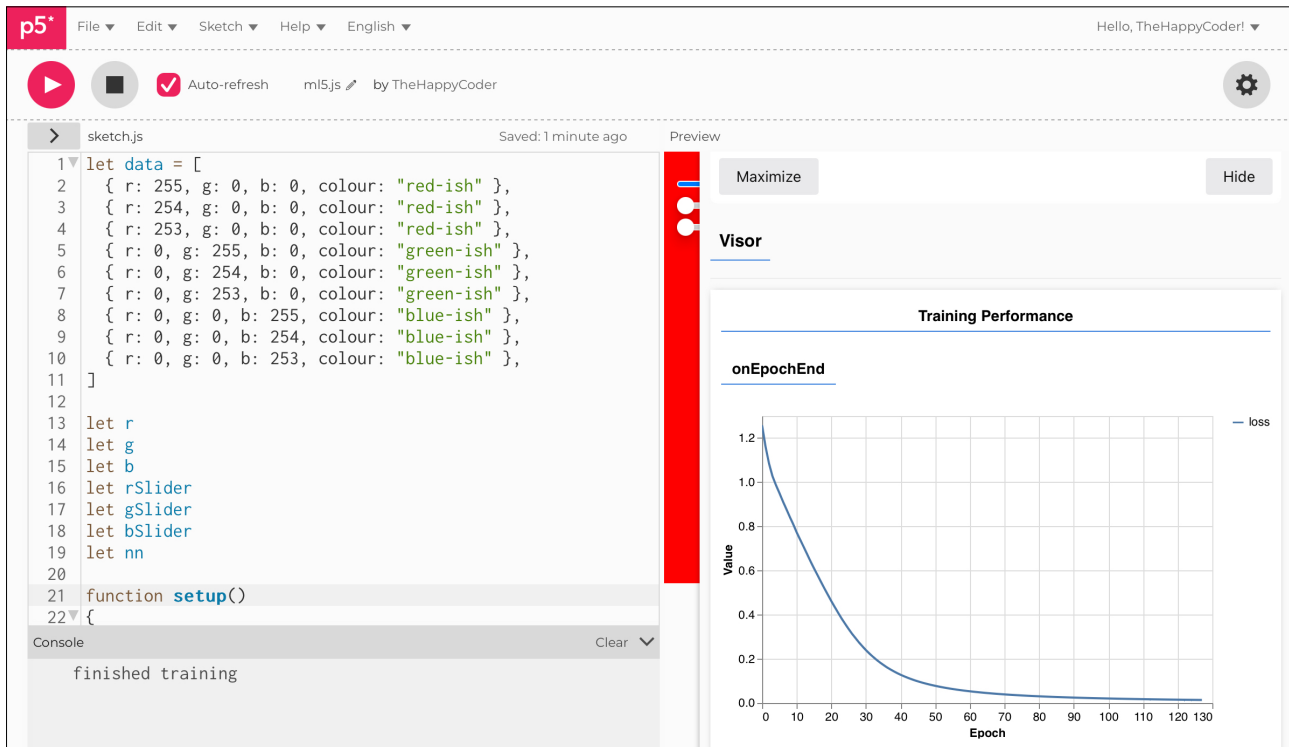
function draw()
{
  r = rSlider.value()
  g = gSlider.value()
  b = bSlider.value()
  background(r, g, b)
}

```

Notes

We do get a very nice curved loss function.

Figure A11.6





Sketch A11.7 classifying the colour

We can now classify the colour as **red-ish**, **green-ish**, or **blue-ish**. We create a **classify()** function where we input the new slider values and classify the output, which is then sent to another function called **gotResults()**. There we can have a look at the results to understand what data we want. To look inside the data, we **console.log** the results.

```
let data = [
  { r: 255, g: 0, b: 0, colour: "red-ish" },
  { r: 254, g: 0, b: 0, colour: "red-ish" },
  { r: 253, g: 0, b: 0, colour: "red-ish" },
  { r: 0, g: 255, b: 0, colour: "green-ish" },
  { r: 0, g: 254, b: 0, colour: "green-ish" },
  { r: 0, g: 253, b: 0, colour: "green-ish" },
  { r: 0, g: 0, b: 255, colour: "blue-ish" },
  { r: 0, g: 0, b: 254, colour: "blue-ish" },
  { r: 0, g: 0, b: 253, colour: "blue-ish" },
]

let r
let g
let b
let rSlider
let gSlider
let bSlider
let nn

function setup()
{
  createCanvas(400, 400)
  rSlider = createSlider(0, 255, 255).position(10, 20)
  gSlider = createSlider(0, 255, 0).position(10, 40)
  bSlider = createSlider(0, 255, 0).position(10, 60)
  let options = {
    task: 'classification',
    debug: true
  }
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
```

```

for (let i = 0; i < data.length; i++)
{
    let item = data[i]
    let inputs = [item.r, item.g, item.b]
    let outputs = [item.colour]
    nn.addData(inputs, outputs)
}
nn.normalizeData()
const trainingOptions = {
    batchSize: 16,
    epochs: 128
}
nn.train(trainingOptions, finishedTraining)
}

function finishedTraining()
{
    console.log("finished training")
    classify()
}

function classify()
{
    const input = [r, g, b]
    nn.classify(input, gotResults)
}

function gotResults(results)
{
    console.log(results)
}

function draw()
{
    r = rSlider.value()
    g = gSlider.value()
    b = bSlider.value()
    background(r, g, b)
}

```

```
}
```

Notes

In the console, you get the following (see image below):

- A) An array with three objects
- B) Each object has a label and confidence score
- C) The highest confidence score is the top one `result[0]`
- D) The next highest is `result[1]`
- E) The third is `result[2]`
- F) We want the top one at `index [0]`

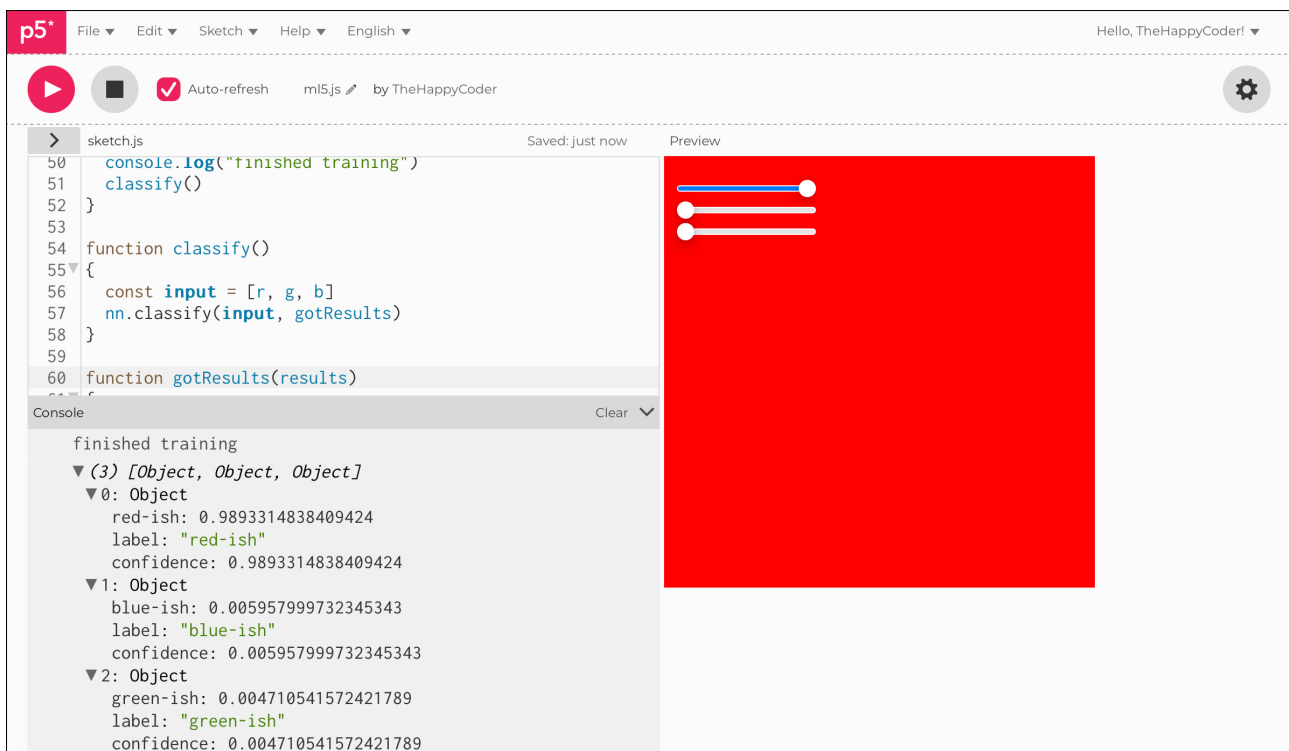
This means we only get the default RGB values; next, we need to get the new values when we move the sliders

Code Explanation

```
const input = [r, g, b]
```

We use the default settings for the initial slider which is red

Figure A11.7





Sketch A11.8 the best prediction

! Remove the `console.log()` in the `gotResults()` function as we don't need that information now.

Here we will be extracting the top result (`index[0]`) with the highest confidence score and then printing the result as text on the canvas. We initialise the `label` as a string (`let label = " "`). We add the `classify()` function in the `gotResults()` function to check for updates to the slider, so that every time you move the sliders you get an instant update. We only train it once but classify repeatedly. If that is unclear, work through the logic!

```
let data = [
  { r: 255, g: 0, b: 0, colour: "red-ish" },
  { r: 254, g: 0, b: 0, colour: "red-ish" },
  { r: 253, g: 0, b: 0, colour: "red-ish" },
  { r: 0, g: 255, b: 0, colour: "green-ish" },
  { r: 0, g: 254, b: 0, colour: "green-ish" },
  { r: 0, g: 253, b: 0, colour: "green-ish" },
  { r: 0, g: 0, b: 255, colour: "blue-ish" },
  { r: 0, g: 0, b: 254, colour: "blue-ish" },
  { r: 0, g: 0, b: 253, colour: "blue-ish" },
]

let r
let g
let b
let rSlider
let gSlider
let bSlider
let nn
let label = " "

function setup()
{
  createCanvas(400, 400)
  rSlider = createSlider(0, 255, 255).position(10, 20)
  gSlider = createSlider(0, 255, 0).position(10, 40)
  bSlider = createSlider(0, 255, 0).position(10, 60)
  let options = {
```

```

    task: 'classification',
    debug: true
  }
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  for (let i = 0; i < data.length; i++)
  {
    let item = data[i]
    let inputs = [item.r, item.g, item.b]
    let outputs = [item.colour]
    nn.addData(inputs, outputs)
  }
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 16,
    epochs: 128
  }
  nn.train(trainingOptions, finishedTraining)
}

function finishedTraining()
{
  console.log("finished training")
  classify()
}

function classify()
{
  const input = [r, g, b]
  nn.classify(input, gotResults)
}

function gotResults(results)
{
  // console.log(results)
  label = results[0].label
  classify()
}

```

```
function draw()
{
  r = rSlider.value()
  g = gSlider.value()
  b = bSlider.value()
  background(r, g, b)

  textSize(64)
  text(label, width/4, height/2)
}
```

Notes

You will perhaps notice that the results aren't perfect despite the loss function looking pretty good. We have only provided a small amount of data, but also we haven't offered any change to most of the other **hyperparameters**, which is something we could do.

Challenges

1. Change the number of hidden layers and nodes,
2. Change the batch size,
3. Change the learning rate
4. Create a JSON. File for all the named colours

Code Explanation

<code>let label = " "</code>	Define the label as a string
<code>label = results[0].label</code>	Pulling the label from the classified results

Figure A11.8a

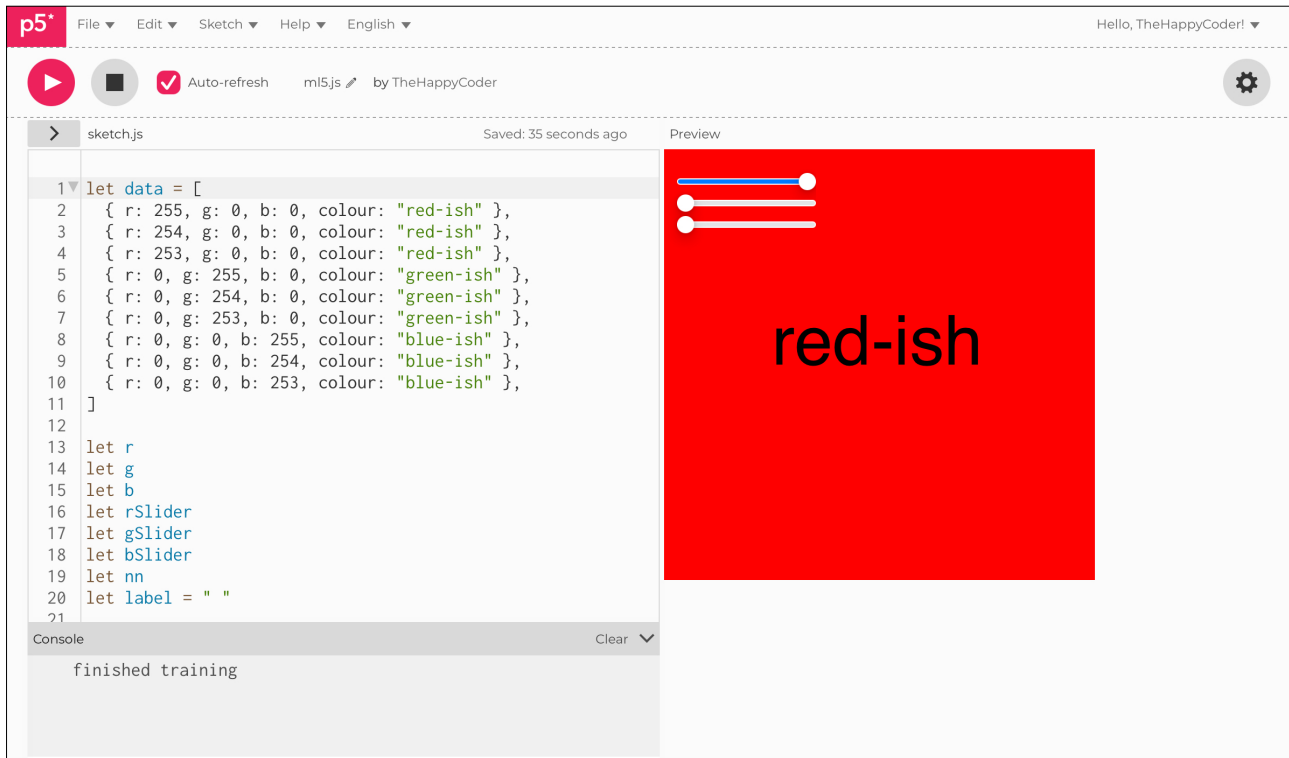


Figure A11.8b

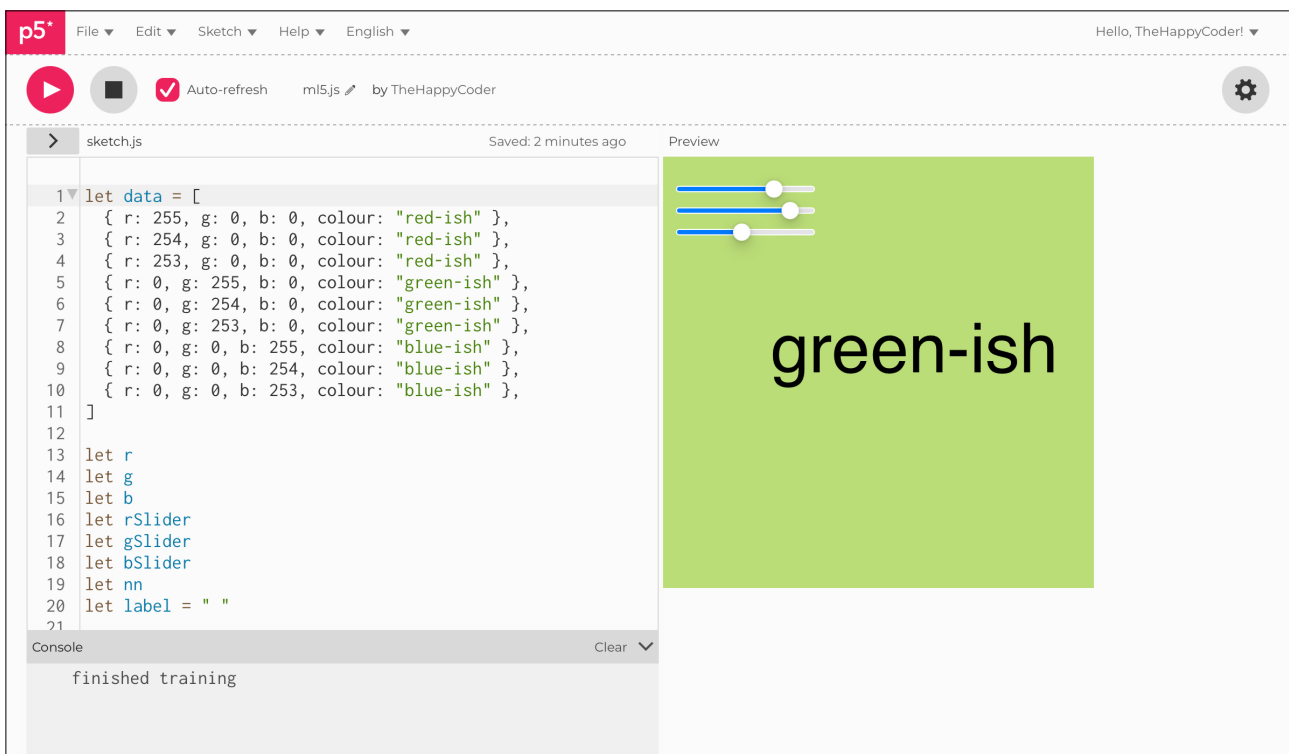
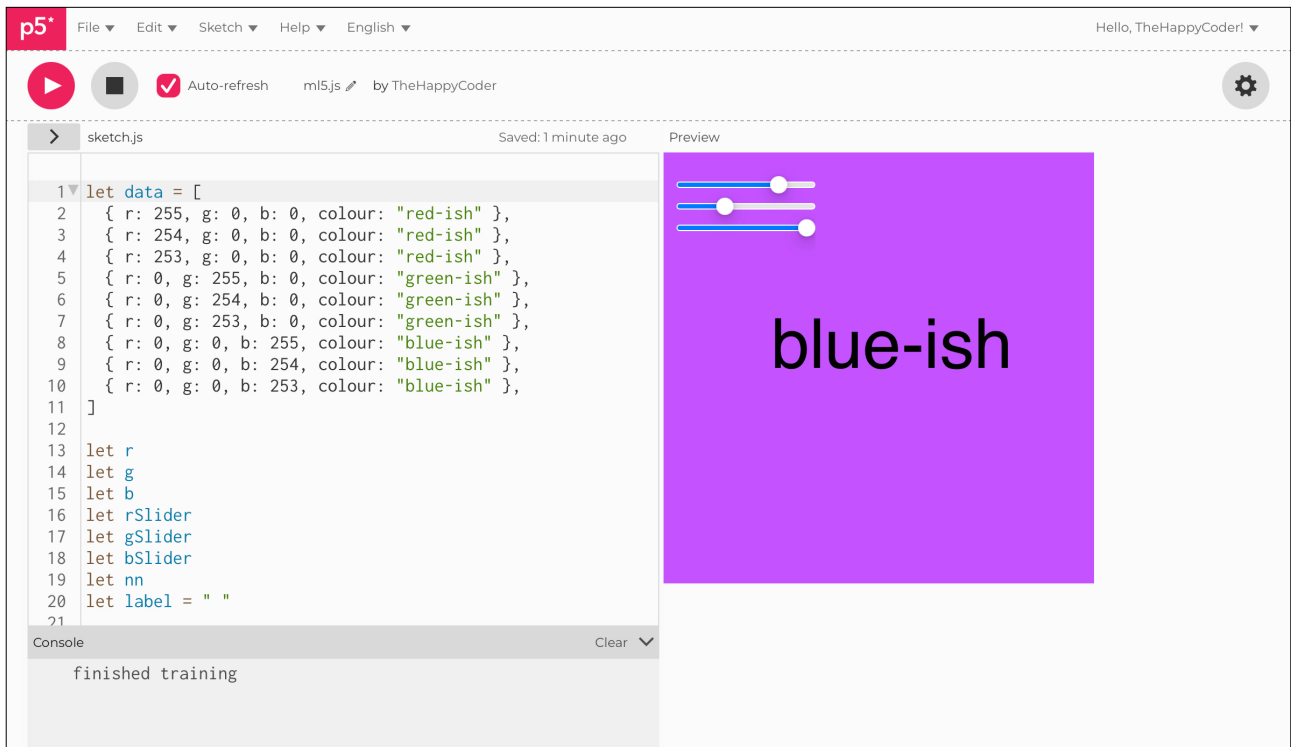


Figure A11.8c



Algorithmic Intelligence Module A Unit #12 pixel predictor



Module A Unit #12: pixel predictor

We will use a **webcam** from your device. If you don't have one, then you either miss this unit out or attach one manually. The image will be reduced, flipped (mirrored), and then pixelated; this enables the model to be trained on a dataset that isn't too large.

Once it is trained, you will be able to move a circle across the canvas through moving your head from side to side, alternatively holding your left or right hand up. It will use images of you in certain positions and use this data to train the model. The model is pretty similar to before.



The index.html

Just a reminder that your index.html file needs the ml5.js added.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```




Sketch A12.1 starting sketch

Give p5.js permission to access your webcam. What you should get instead of the canvas is a video of yourself. It will be reversed (not mirrored).

```
let video

function setup()
{
  noCanvas()
  video = createCapture(VIDEO)
}

function draw()
{
  background(220)
}
```

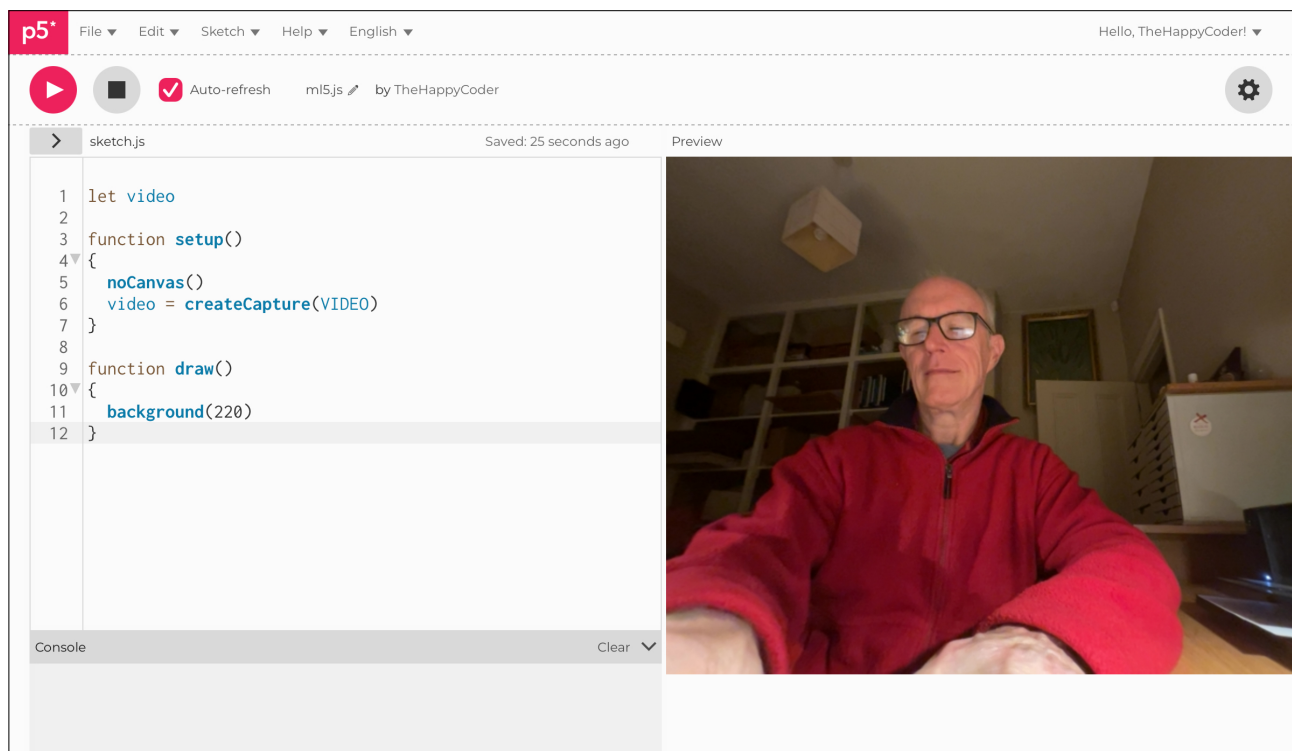
Notes

You should have a nice video image of yourself; the dimensions should be 640 x 480.

Code Explanation

<code>noCanvas()</code>	We tell the <code>setup()</code> function we don't want a canvas
<code>video = createCapture(VIDEO)</code>	We do want a video from the webcam

Figure A12.1





Sketch A12.2 hiding the video

To use the video image, we want it on the canvas. So replace `noCanvas()` with `createCanvas(640, 480)` and hide the video in `setup()`. This will place the video on the canvas.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

You should have one video image exactly as before.

Challenges

1. Comment out `// video.hide()`
2. Change the size of the canvas to see that you have placed the video on the canvas, hence the dimensions.
3. Change `image(video, 0, 0)` to `image(video, 100, 100)`.

Code Explanation

<code>video.hide()</code>	Hides the video (but not the one drawn on the canvas)
<code>image(video, 0, 0)</code>	This places the image, in this case a video at origin (0, 0)



Sketch A12.3 mirrored

We want to flip the video so that it mirrors our movements; it makes it much more intuitive when we come to train the model.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

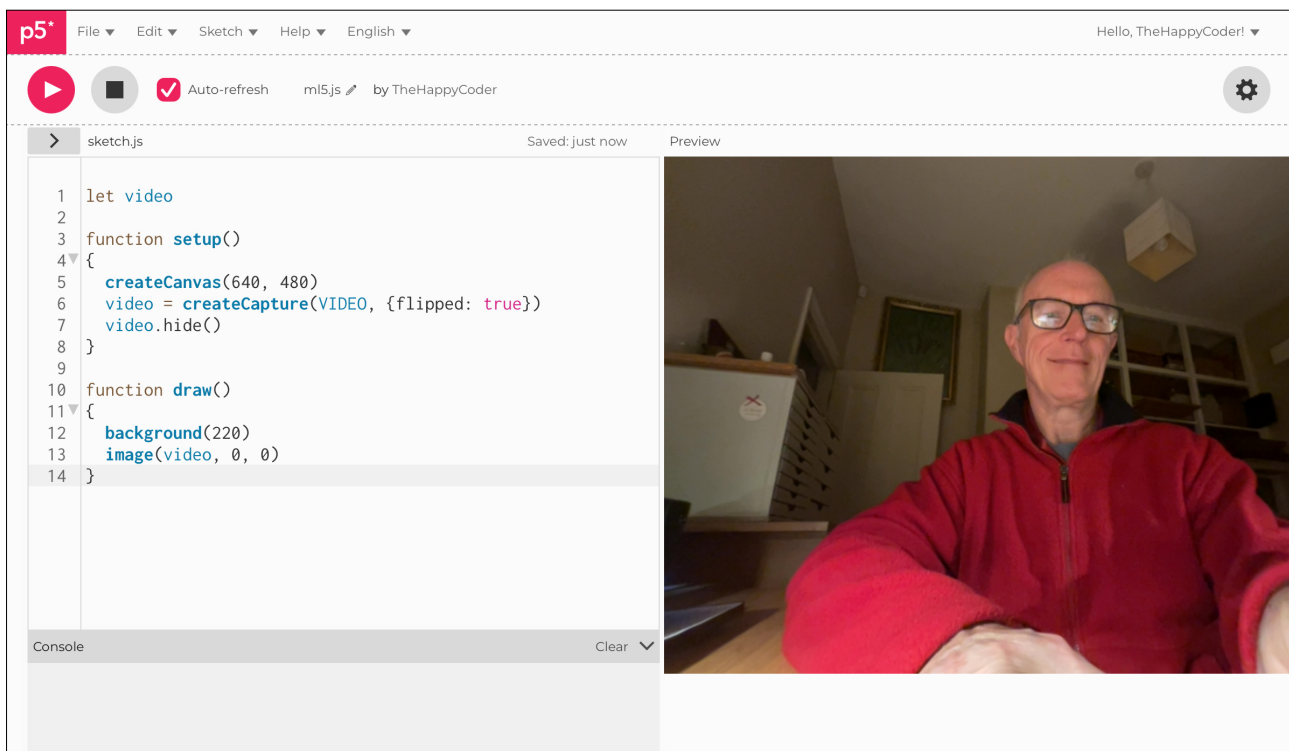
Now you have a mirrored image from the webcam.

Code Explanation

```
{flipped: true}
```

Flips the video so that it behaves like a mirror

Figure A12.3 mirrored





Sketch A12.4 video ready

We do not want to try to use the video before we have the video available. This is the asynchronous nature of coding, in the past we could either have preloaded or create a boolean variable and have a callback to check the video is ready. Now we use `async` and `await`. This works with version 1.x and version 2.x of p5.js.

```
let video

async function setup()
{
  createCanvas(640, 480)
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

Everything should work just as before. This is an important step before we start adding data to the neural network.

Code Explanation

<code>async function setup()</code>	This helps with the asynchronous nature of the code
<code>video = await createCapture(VIDEO, {flipped: true})</code>	Returns boolean true when ready



Sketch A12.5 the size of the video

At present, we have an image coming in at 640 x 480 pixels. You are drawing that onto the canvas of the same dimensions (and ratio). If we change the canvas to 400 x 400, we crop the image, losing some of it. We want to use a square image, and then we can simplify the pixelation later. First, let's change the canvas size.

```
let video
let ready = false

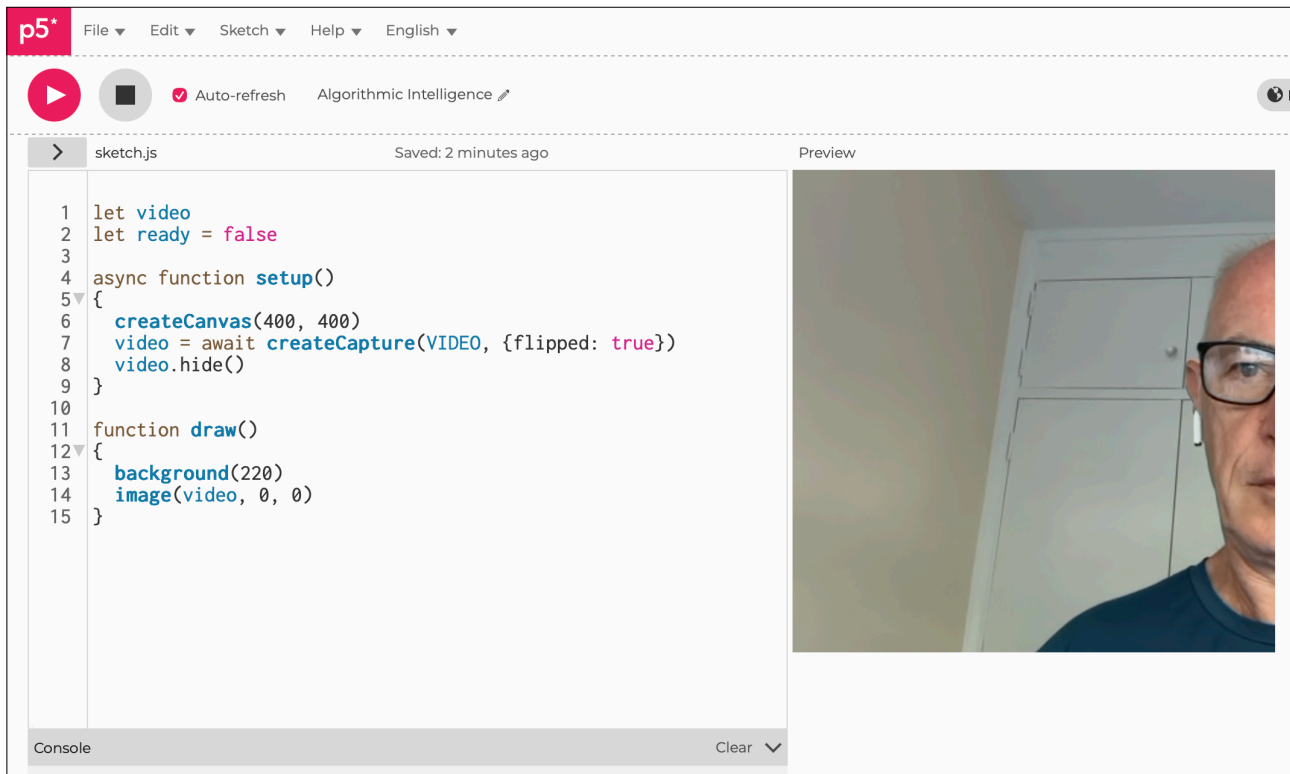
async function setup()
{
  createCanvas(400, 400)
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

We only get part of the image from the camera. We want the full image, even if it is distorted and squashed into a square.

Figure A12.5





Sketch A12.6 resizing the video

We can squash the video to a new size. This will distort the video, but isn't a problem for this exercise. In fact, we will be squashing it a whole lot more. We create a variable called `videoSize` and use the function `video.size()` to change the dimensions. We will set the dimensions to `400` for now.

```
let video

let videoSize = 400

async function setup()
{
  createCanvas(400, 400)
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

The video is now fully 400 x 400, although it is much squashier. Using the same dimensions for the width and height (`videoSize`) is just for convenience. You could have different values. Using a variable means you can alter and change it at any time.

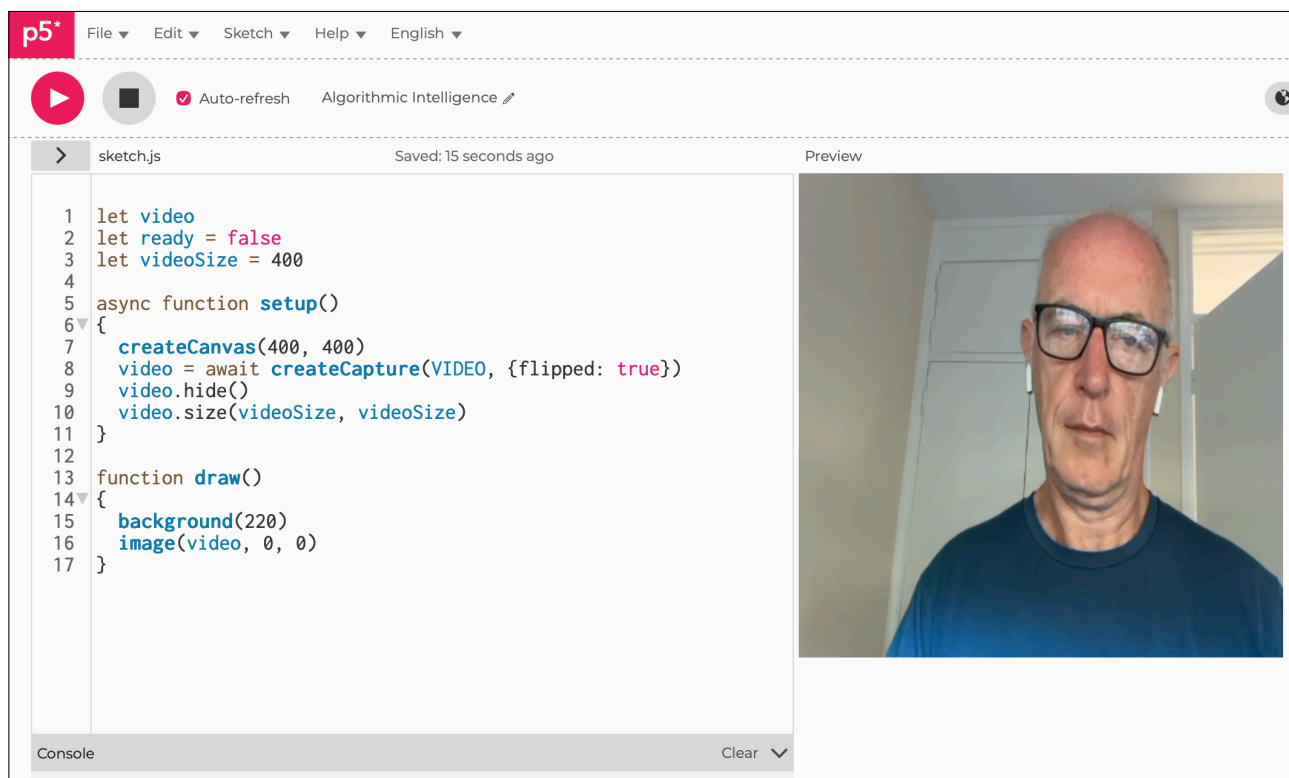
Challenge

1. Try other dimensions

Code Explanation

<code>let videoSize = 400</code>	Set the dimensions of the video
<code>video.size(videoSize, videoSize)</code>	Specifies the dimensions of the video not the canvas drawing it onto

Figure A12.6





Sketch A12.7 even smaller

What we want to do is change the size to **10 x 10**, yes, that small! This means the software behind producing the video is going to try and create a **10 x 10** video image from a **640 x 480** video image. We reduce the **videoSize** to **10**.

```
let video

let videoSize = 10

async function setup()
{
  createCanvas(400, 400)
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

The video is now really small, almost too small to see; it is now 10 pixels by 10 pixels. The second figure, A9.7b, below shows what it does look like close up. You will notice that it seems grainy and indistinct. This is because it is trying to approximate the larger image.

Challenge

Try something a little bigger, say 50 pixels.

Figure A12.7a can you see it?

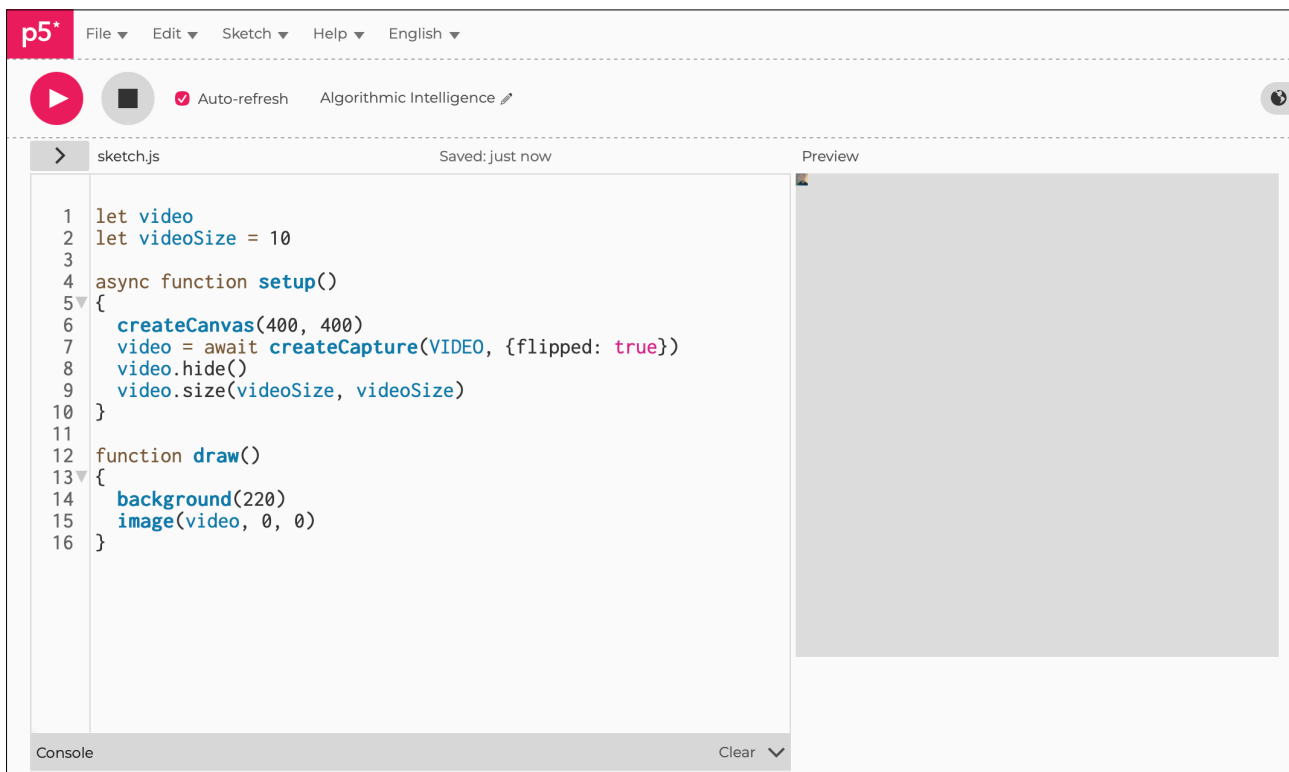
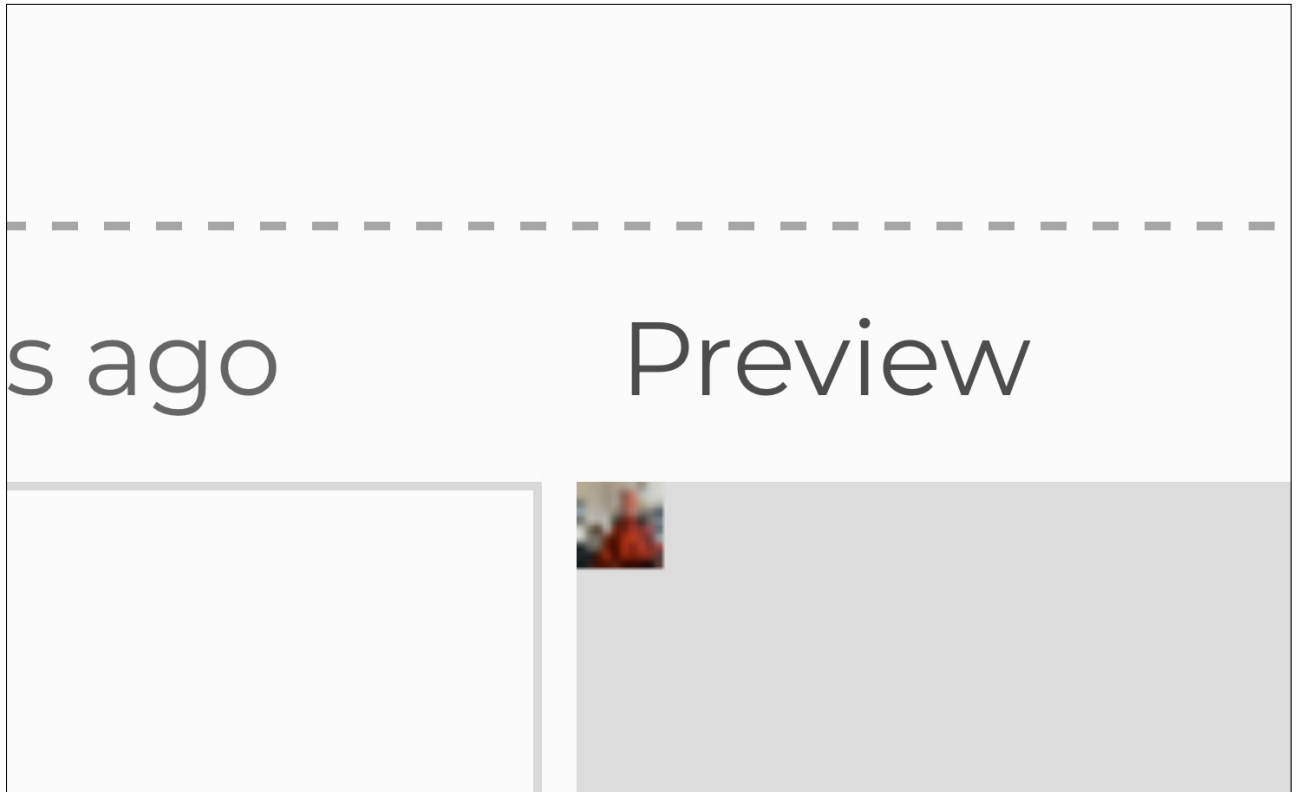


Figure A12.7b yes, it really is me!





Sketch A12.8 loading the pixels

We want to take those pixels which equate to 100 data points (10 by 10), except that there are four channels for each pixel: the red, green, blue, and alpha (transparency), which means we actually have 400 data points, even though we will ignore the alpha.

The function `video.loadPixels()` does just that; it loads the video into a pixel array which we will access through a nested loop. Remove the `image(video, 0, 0)` and replace it with the `loadPixels()`. A full explanation of how we derive the nested loop for getting the values of the pixels in an image is covered in the [coding snippets unit](#). Check back there if you are unsure.

```
let video
let videoSize = 10

async function setup()
{
  createCanvas(400, 400)
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
}

function draw()
{
  background(220)
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
      let index = (x + y * video.width) * 4
      let r = video.pixels[index + 0]
      let g = video.pixels[index + 1]
      let b = video.pixels[index + 2]
    }
  }
}
```

Notes

This means we have, currently, no video to display; you get a blank canvas. The nested loop takes the **r** (red) value from the pixel, the **g** (green) value from that pixel, and the **b** (blue) value of the pixel from the pixel array. What we can now do is scale or magnify each pixel and colour in a square with those pixel values. In short, each pixel has four elements of the pixel in the array; that is why we multiply by four on each loop, the fourth being the alpha value (which we will not bother to use).

Code Explanation

<code>video.loadPixels()</code>	Loads all the pixels (with four elements per pixel) into an array
<code>for (let x = 0; x < video.width; x++)</code>	Goes through all 10 values of x
<code>for (let y = 0; y < video.height; y++)</code>	For each value of x work through each 10 values of y
<code>let index = (x + y * video.width) * 4</code>	Each index starting from the top left hand corner, going across to the bottom right hand corner
<code>let r = video.pixels[index + 0]</code>	This is the index for the first element (red) of the pixel colour
<code>let g = video.pixels[index + 1]</code>	This is the index for the second element (green) of the pixel colour
<code>let b = video.pixels[index + 2]</code>	This is the index for the third element (blue) of the pixel colour



Sketch A12.9 enlarged pixels

We are going to draw **10** by **10** squares on the canvas. Each square will be given the **r**, **g**, **b** values for the corresponding pixel in the **10** x **10** pixel video image. If you remember, it was a bit grainy and blurry; we are effectively going to scale up that tiny image to fill the canvas. The width of the square (**w**) is simply the **width** of the canvas divided by the **videoSize**. This means if you want to make changes to either of those variables, they will calculate them rather than hard-coding the values.

```
let video
let videoSize = 10

function setup()
{
  createCanvas(400, 400)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
}

function draw()
{
  background(220)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
      let index = (x + y * video.width) * 4
      let r = video.pixels[index + 0]
      let g = video.pixels[index + 1]
      let b = video.pixels[index + 2]
      noStroke()
      fill(r, g, b)
      square(x * w, y * w, w)
    }
  }
}
```


Notes

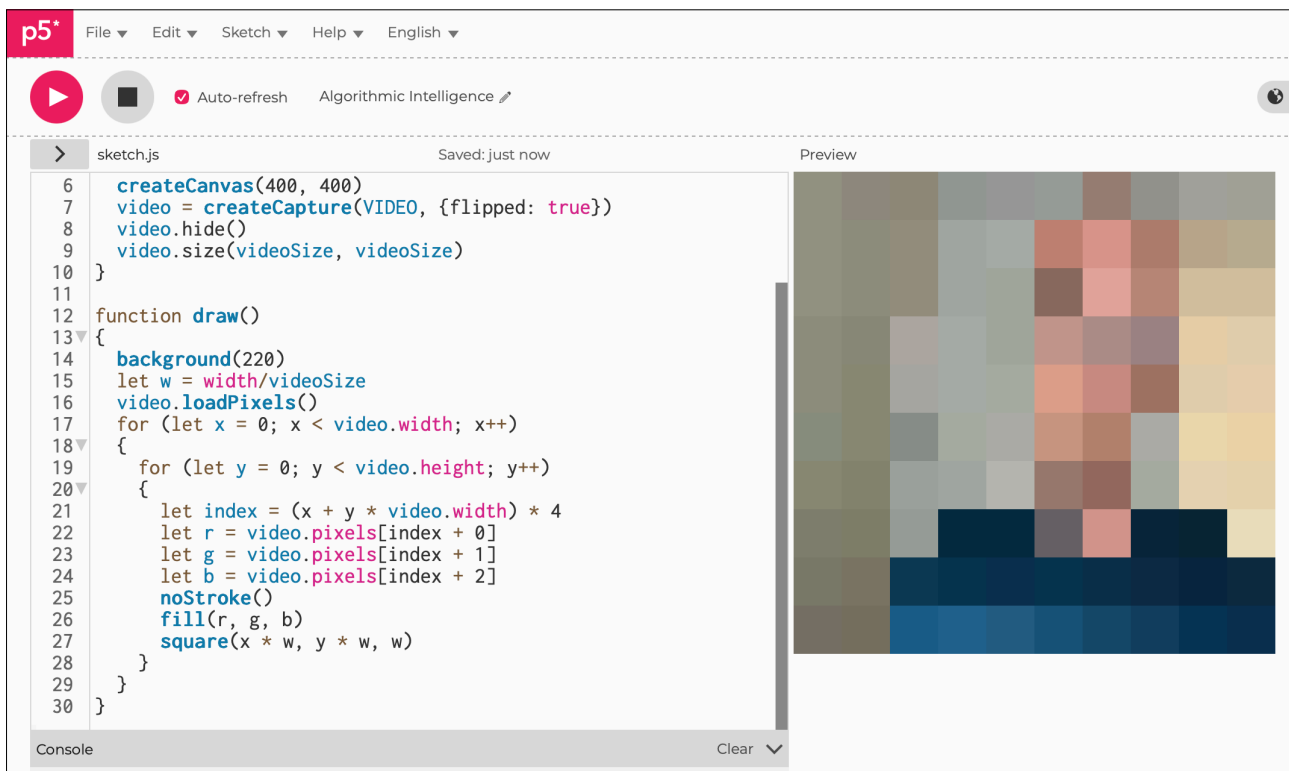
We now have a larger version of that tiny image of 10 x 10 pixels on a canvas of 10 by 10 squares.

Code Explanation

square(x * w, y * w, w)

Calculating the position and size of each square

Figure A12.9 it really is me





Sketch A12.10 neural network

Now we can add our neural network. The inputs are all the pixels (**totalPixels**) in the image of **videoSize** (10), and each pixel has an **RGB** value, hence **videoSize** x **videoSize** x 3. We have all the pixels as the inputs, hence why we want a very small image. For the output, we have one output (this will become clear later). This will include a learning rate lower than the default. The task is **regression** because the output will be a single value. We will also have the loss function charted.

```
let nn

let video
let videoSize = 10

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)

  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}

function draw()
{
  background(220)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
```

```
    let index = (x + y * video.width) * 4
    let r = video.pixels[index + 0]
    let g = video.pixels[index + 1]
    let b = video.pixels[index + 2]
    noStroke()
    fill(r, g, b)
    square(x * w, y * w, w)
  }
}
```

Notes

The effect hasn't changed, but we have the main elements in place to run this example.



Sketch A12.11 adding the slider output

We want to create a circle which we can control by moving our head from side to side or left hand to right hand. We will have the position of your head (or hand) as the input pixels and the position of the circle as the output. We use a slider to position our circle as we train it. We move the circle to the left and move our head to the left and then save the data/image pixels (several times), repeat by moving the circle to the right and our head to the right (read left hand right hand). Again, repeat saving several examples.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}

function draw()
{
  background(220)
  let w = width/videoSize
```

```
video.loadPixels()
for (let x = 0; x < video.width; x++)
{
  for (let y = 0; y < video.height; y++)
  {
    let index = (x + y * video.width) * 4
    let r = video.pixels[index + 0]
    let g = video.pixels[index + 1]
    let b = video.pixels[index + 2]
    noStroke()
    fill(r, g, b)
    square(x * w, y * w, w)
  }
}

sliderPos = slider.value()
fill(0, 0, 255)
circle(sliderPos, height/2, 50)
}
```

Notes

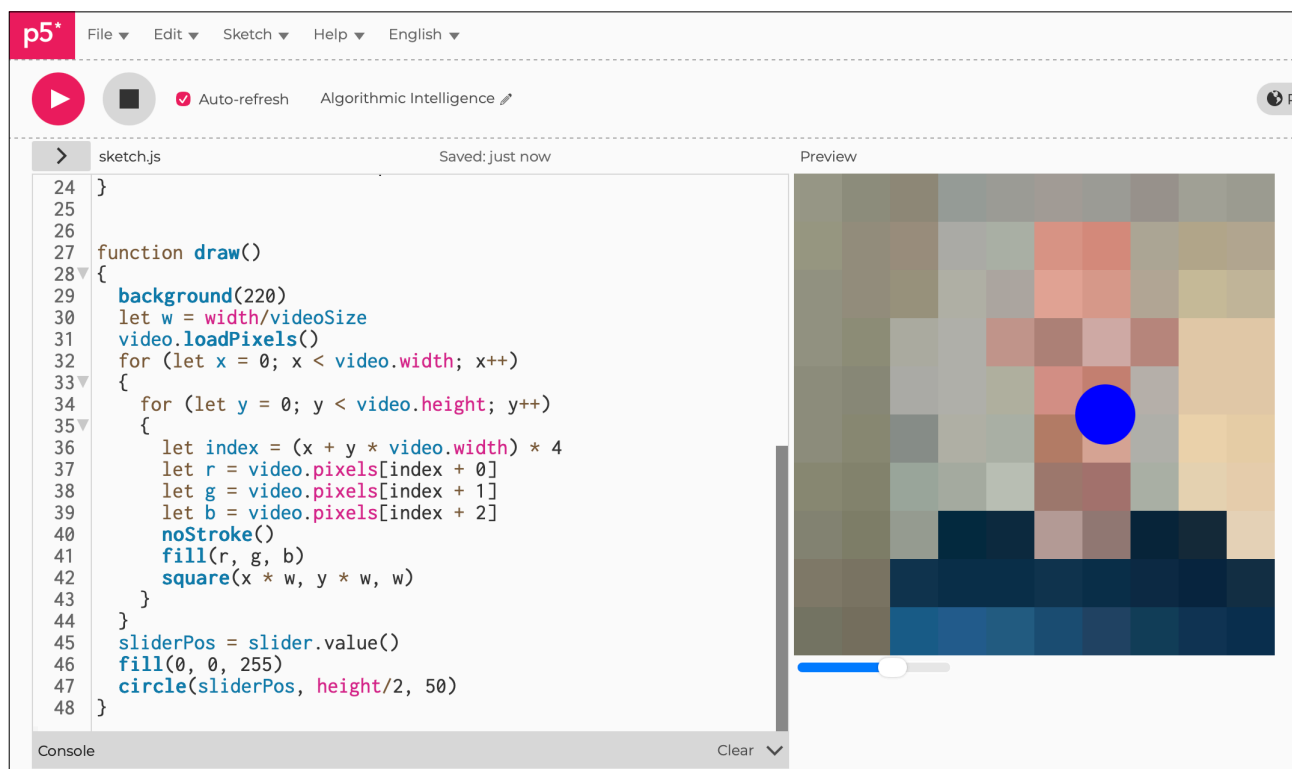
! Move the slider and the blue circles moves with it.

We draw the circle on the canvas; we also initialise it to zero, which is the left-hand side, obviously. Note we are only moving the blue circle along the x-axis; the y-value is fixed, hence one value is output.

Code Explanation

<code>let sliderPos = 0</code>	The position of the slider
<code>let slider</code>	Naming the slider
<code>slider = createSlider(0, width, 0)</code>	Creating the slider
<code>sliderPos = slider.value()</code>	Gives you the value of the slider position

Figure A12.11





Sketch A12.12 adding the buttons

We need a button so we can add the examples. We will need to name the buttons because we will be creating two of them.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider
let buttonAddExample

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  buttonAddExample = createButton('add example')
  buttonAddExample.style('font-size', '20px')
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}

function draw()
{
  background(220)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
```

```
{
  for (let y = 0; y < video.height; y++)
  {
    let index = (x + y * video.width) * 4
    let r = video.pixels[index + 0]
    let g = video.pixels[index + 1]
    let b = video.pixels[index + 2]
    noStroke()
    fill(r, g, b)
    square(x * w, y * w, w)
  }
}
sliderPos = slider.value()
fill(0, 0, 255)
circle(sliderPos, height/2, 50)
}
```


Notes

No action from the button yet.

Figure A12.12





Sketch A12.13 the training button

While we are in the business of adding buttons, we will create a training button to use later.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider
let buttonAddExample
let buttonTrain

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  buttonAddExample = createButton('add example')
  buttonAddExample.style('font-size', '20px')
  buttonTrain = createButton('train')
  buttonTrain.style('font-size', '20px')
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}

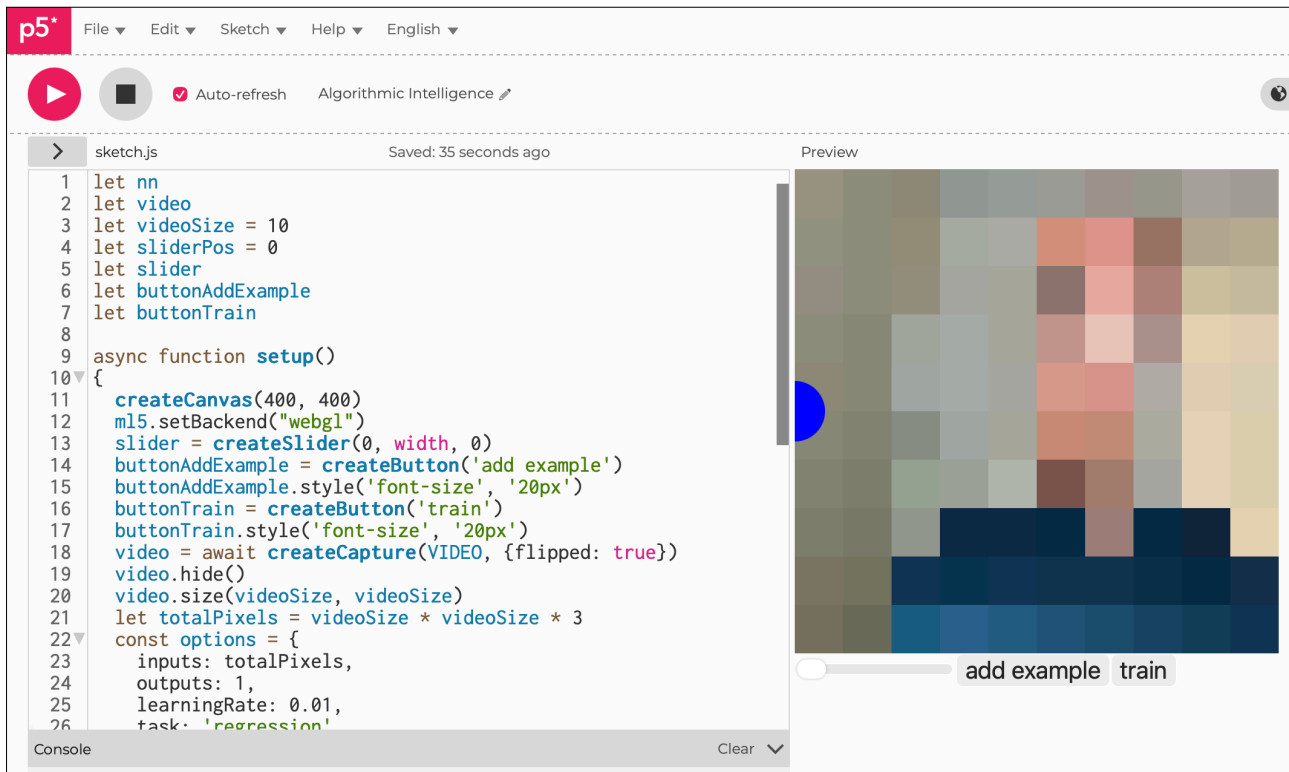
function draw()
{
  background(220)
```

```
let w = width/videoSize
video.loadPixels()
for (let x = 0; x < video.width; x++)
{
  for (let y = 0; y < video.height; y++)
  {
    let index = (x + y * video.width) * 4
    let r = video.pixels[index + 0]
    let g = video.pixels[index + 1]
    let b = video.pixels[index + 2]
    noStroke()
    fill(r, g, b)
    square(x * w, y * w, w)
  }
}
sliderPos = slider.value()
fill(0, 0, 255)
circle(sliderPos, height/2, 50)
}
```

Notes

Now we have the two buttons and a slider. We didn't specify the positions, they just add them on in order of creation.

Figure A12.13





Sketch A12.14 getting the inputs

Now we have the two buttons and a slider, we can do something with them. The aim is to put the slider to zero, move our head to the left (or left hand), and effectively take several snapshots (adding about five examples each), moving the slider to the other edge of the canvas, and repeat with our head in that position (or right hand). So we end up with ten (or however many) datasets. We will do this in stages; the first stage is to get those inputs by creating a function called `getInputs()`.

We are going to load the pixels; remember, it is taking the tiny (10 x 10) video. We need an empty array to store these inputs. There are three values for each pixel: the red (r), green (g), and blue (b) values, which are `index + 0`, `index + 1`, and `index + 2`, respectively, for each pixel.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider
let buttonAddExample
let buttonTrain

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  buttonAddExample = createButton('add example')
  buttonAddExample.style('font-size', '20px')
  buttonTrain = createButton('train')
  buttonTrain.style('font-size', '20px')
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
```

```

}
nn = ml5.neuralNetwork(options)
}

function draw()
{
  background(220)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
      let index = (x + y * video.width) * 4
      let r = video.pixels[index + 0]
      let g = video.pixels[index + 1]
      let b = video.pixels[index + 2]
      noStroke()
      fill(r, g, b)
      square(x * w, y * w, w)
    }
  }
  sliderPos = slider.value()
  fill(0, 0, 255)
  circle(sliderPos, height/2, 50)
}

```

```

function getInputs()
{
  video.loadPixels()
  let inputs = []
  for (let i = 0; i < video.width * video.height; i++)
  {
    let index = i * 4
    inputs.push(video.pixels[index + 0])
    inputs.push(video.pixels[index + 1])
    inputs.push(video.pixels[index + 2])
  }
}

```

```
    return inputs
}
```

Notes

! Nothing will happen just yet.

Quite a few lines of code which look very similar to the `draw()` function. We are using the pixels from the tiny video, hence `video.pixels[]` rather than just `pixels[]`.

Code Explanation

<code>let inputs = []</code>	This is the inputs array
<code>for (let i = 0; i < video.width * video.height; i++)</code>	The video.width times the video.height gives us all the pixels
<code>let index = i * 4</code>	We jump every four elements in the pixel array to get the next pixel
<code>inputs.push(video.pixels[index + 0])</code>	The red value is pushed into the inputs array
<code>inputs.push(video.pixels[index + 1])</code>	The green value is pushed into the inputs array
<code>inputs.push(video.pixels[index + 2])</code>	The blue value is pushed into the inputs array
<code>return inputs</code>	Returns the complete array



Sketch A12.15 adding an example

We want to add the data to the neural network. We want the inputs for the position of the slider. We create another variable called **pos**, which returns the value of the slider. We want to add it every time we click on the add example button with the mouse.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider
let buttonAddExample
let buttonTrain
let pos = 0

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  buttonAddExample = createButton('add example')
  buttonAddExample.style('font-size', '20px')
  buttonTrain = createButton('train')
  buttonTrain.style('font-size', '20px')
  video = await createCapture(VIDEO0, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}

function draw()
```



```

{
  background(220)
  buttonAddExample.mousePressed(addExample)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
      let index = (x + y * video.width) * 4
      let r = video.pixels[index + 0]
      let g = video.pixels[index + 1]
      let b = video.pixels[index + 2]
      noStroke()
      fill(r, g, b)
      square(x * w, y * w, w)
    }
  }
  sliderPos = slider.value()
  fill(0, 0, 255)
  circle(sliderPos, height/2, 50)
}

function getInputs()
{
  video.loadPixels()
  let inputs = []
  for (let i = 0; i < video.width * video.height; i++)
  {
    let index = i * 4
    inputs.push(video.pixels[index + 0])
    inputs.push(video.pixels[index + 1])
    inputs.push(video.pixels[index + 2])
  }
  return inputs
}

function addExample()

```

```

{
  pos = slider.value()
  let inputs = getInputs()
  nn.addData(inputs, [pos])
}

```

Notes

! Still nowt to see.

We add the pixel data combined with the slider position for each example.

Code Explanation

let pos = 0	We need a new variable otherwise we end up with two sets of data from the slider position
buttonAddExample.mousePressed(addExample)	We call the addExample function
pos = slider.value()	The pos variable gets the slider position
let inputs = getInputs()	We bring the pixel dataset
nn.addData(inputs, [pos])	Adding the data, the inputs (pixels) and the output (pos as an array)



Sketch A12.16 training the model

We have now collected the training data examples, and we want to train the model on that data. So, we create another function called `trainModel()`. This normalises the data and then trains on it. We will train it for **50 epochs**.

! If you run it now, you will get an error.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider
let buttonAddExample
let buttonTrain
let pos = 0

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  buttonAddExample = createButton('add example')
  buttonAddExample.style('font-size', '20px')
  buttonTrain = createButton('train')
  buttonTrain.style('font-size', '20px')
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}
```

```

function draw()
{
  background(220)
  buttonAddExample.mousePressed(addExample)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
      let index = (x + y * video.width) * 4
      let r = video.pixels[index + 0]
      let g = video.pixels[index + 1]
      let b = video.pixels[index + 2]
      noStroke()
      fill(r, g, b)
      square(x * w, y * w, w)
    }
  }
  sliderPos = slider.value()
  fill(0, 0, 255)
  circle(sliderPos, height/2, 50)
}

function getInputs()
{
  video.loadPixels()
  let inputs = []
  for (let i = 0; i < video.width * video.height; i++)
  {
    let index = i * 4
    inputs.push(video.pixels[index + 0] / 255)
    inputs.push(video.pixels[index + 1] / 255)
    inputs.push(video.pixels[index + 2] / 255)
  }
  return inputs
}

```

```

function addExample()
{
  pos = slider.value()
  let inputs = getInputs()
  nn.addData(inputs, [pos])
  buttonTrain.mousePressed(trainModel)
}

function trainModel()
{
  nn.normalizeData()
  nn.train(finishedTraining)
}

```

Notes

In the `train()` function, we have a callback function for when the training is finished, called `finishedTraining`. We don't have the `finishedTraining()` function so you will get an error.

Code Explanation

<code>buttonTrain.mousePressed(trainModel)</code>	When the button to train has been pressed it calls the <code>trainModel</code> function
<code>nn.normalizeData()</code>	Data is then normalised
<code>nn.train(finishedTraining)</code>	Trained over 50 epochs and then the callback



Sketch A12.17 finished training

When we have finished training the model, we want to predict the result. To do that, we have a function and call it `predict()`.

! If you run it now, you will still get an error.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider
let buttonAddExample
let buttonTrain
let pos = 0

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  buttonAddExample = createButton('add example')
  buttonAddExample.style('font-size', '20px')
  buttonTrain = createButton('train')
  buttonTrain.style('font-size', '20px')
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}
```

```

function draw()
{
  background(220)
  buttonAddExample.mousePressed(addExample)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
      let index = (x + y * video.width) * 4
      let r = video.pixels[index + 0]
      let g = video.pixels[index + 1]
      let b = video.pixels[index + 2]
      noStroke()
      fill(r, g, b)
      square(x * w, y * w, w)
    }
  }
  sliderPos = slider.value()
  fill(0, 0, 255)
  circle(sliderPos, height/2, 50)
}

function getInputs()
{
  video.loadPixels()
  let inputs = []
  for (let i = 0; i < video.width * video.height; i++)
  {
    let index = i * 4
    inputs.push(video.pixels[index + 0] / 255)
    inputs.push(video.pixels[index + 1] / 255)
    inputs.push(video.pixels[index + 2] / 255)
  }
  return inputs
}

```

```
function addExample()
{
  pos = slider.value()
  let inputs = getInputs()
  nn.addData(inputs, [pos])
  buttonTrain.mousePressed(trainModel)
}

function trainModel()
{
  nn.normalizeData()
  nn.train(finishedTraining)
}

function finishedTraining()
{
  predict()
}
```

Notes

Finished training, which means we are now ready to predict. No `predict()` function hence the error.



Sketch A12.18 predicting the movement

Now we need to predict the result after training. We use the pixels from the video, so that when we move our heads from left to right (left or right hand), it can predict the position of the circle based on our trained model. Here we create the `predict()` function.

! Don't run it just yet.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider
let buttonAddExample
let buttonTrain
let pos = 0

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  buttonAddExample = createButton('add example')
  buttonAddExample.style('font-size', '20px')
  buttonTrain = createButton('train')
  buttonTrain.style('font-size', '20px')
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
  nn = ml5.neuralNetwork(options)
}
```

```

function draw()
{
  background(220)
  buttonAddExample.mousePressed(addExample)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
      let index = (x + y * video.width) * 4
      let r = video.pixels[index + 0]
      let g = video.pixels[index + 1]
      let b = video.pixels[index + 2]
      noStroke()
      fill(r, g, b)
      square(x * w, y * w, w)
    }
  }
  sliderPos = slider.value()
  fill(0, 0, 255)
  circle(sliderPos, height/2, 50)
}

function getInputs()
{
  video.loadPixels()
  let inputs = []
  for (let i = 0; i < video.width * video.height; i++)
  {
    let index = i * 4
    inputs.push(video.pixels[index + 0] / 255)
    inputs.push(video.pixels[index + 1] / 255)
    inputs.push(video.pixels[index + 2] / 255)
  }
  return inputs
}

```

```

function addExample()
{
  pos = slider.value()
  let inputs = getInputs()
  nn.addData(inputs, [pos])
  buttonTrain.mousePressed(trainModel)
}

function trainModel()
{
  nn.normalizeData()
  nn.train(finishedTraining)
}

function finishedTraining()
{
  predict()
}

function predict()
{
  let inputs = getInputs()
  nn.predict(inputs, gotPosition)
}

```

Notes

In the `predict()` function, we collect our head (or hand) movement and feed it into the neural network, and have a callback for the results called `gotPosition`. An error because of the lack of `gotPosition()` function.

Code Explanation

<code>let inputs = getInputs()</code>	The pixels are imputed from the video
<code>nn.predict(inputs, gotPosition)</code>	The callback receives the result (output)



Sketch A12.19 predict ready

When we have a prediction, we then need to draw a different circle (in red). We don't want to draw anything while we are getting data or training; hence, the boolean `predictReady` is set to `false` until we have done the prediction. In the `draw()` function, we need to remove the blue circle (used for training) and replace it with the red circle (for prediction) instead. We set `predictReady` to be `true`, get the result, and send the red circle to the predicted position (`pos`). We predict again because we want to see the red circle move when we move our head (or hand) from left to right. The circle should respond to the changes.

```
let nn
let video
let videoSize = 10
let sliderPos = 0
let slider
let buttonAddExample
let buttonTrain
let pos = 0
let predictReady = false

async function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  slider = createSlider(0, width, 0)
  buttonAddExample = createButton('add example')
  buttonAddExample.style('font-size', '20px')
  buttonTrain = createButton('train')
  buttonTrain.style('font-size', '20px')
  video = await createCapture(VIDEO, {flipped: true})
  video.hide()
  video.size(videoSize, videoSize)
  let totalPixels = videoSize * videoSize * 3
  const options = {
    inputs: totalPixels,
    outputs: 1,
    learningRate: 0.01,
    task: 'regression',
    debug: true
  }
```

```

}
nn = ml5.neuralNetwork(options)
}

function draw()
{
  background(220)
  buttonAddExample.mousePressed(addExample)
  let w = width/videoSize
  video.loadPixels()
  for (let x = 0; x < video.width; x++)
  {
    for (let y = 0; y < video.height; y++)
    {
      let index = (x + y * video.width) * 4
      let r = video.pixels[index + 0]
      let g = video.pixels[index + 1]
      let b = video.pixels[index + 2]
      noStroke()
      fill(r, g, b)
      square(x * w, y * w, w)
    }
  }
}

```

```

if (predictReady == true)
{
  fill(255, 0, 0)
  circle(pos, height/2, 50)
}
else
{
  sliderPos = slider.value()
  fill(0, 0, 255)
  circle(sliderPos, height/2, 50)
}

```

```

}

function getInputs()
{

```

```

video.loadPixels()
let inputs = []
for (let i = 0; i < video.width * video.height; i++)
{
  let index = i * 4
  inputs.push(video.pixels[index + 0])
  inputs.push(video.pixels[index + 1])
  inputs.push(video.pixels[index + 2])
}
return inputs
}

```

```

function addExample()
{
  pos = slider.value()
  let inputs = getInputs()
  nn.addData(inputs, [pos])
  buttonTrain.mousePressed(trainModel)
}

```

```

function trainModel()
{
  nn.normalizeData()
  nn.train(finishedTraining)
}

```

```

function finishedTraining()
{
  predict()
}

```

```

function predict()
{
  let inputs = getInputs()
  nn.predict(inputs, gotPosition)
}

```

```

function gotPosition(results)

```

```
{
  predictReady = true
  pos = (results[0].value)
  predict()
}
```

Notes

We introduce the `if()...else` so we only draw the red circle when we are ready and not before; otherwise, we draw the blue.

Challenges

1. Make the video size smaller (smaller pixels = more of them).
2. Add other hyperparameters.
3. Make other things happen in response to your training.

Code Explanation

<code>let predictReady = false</code>	Initialise the predictReady as false
<code>function gotPosition(results)</code>	Argument is the result
<code>predictReady = true</code>	Boolean true
<code>pos = (results[0].value)</code>	Get the current result from the output (pos) array
<code>predict()</code>	Repeat
<code>if (predictReady == true)</code>	If the boolean is true then draw the red circle at the predicted position otherwise (else) continue to draw the blue circle

Figure A12.19a: loss chart

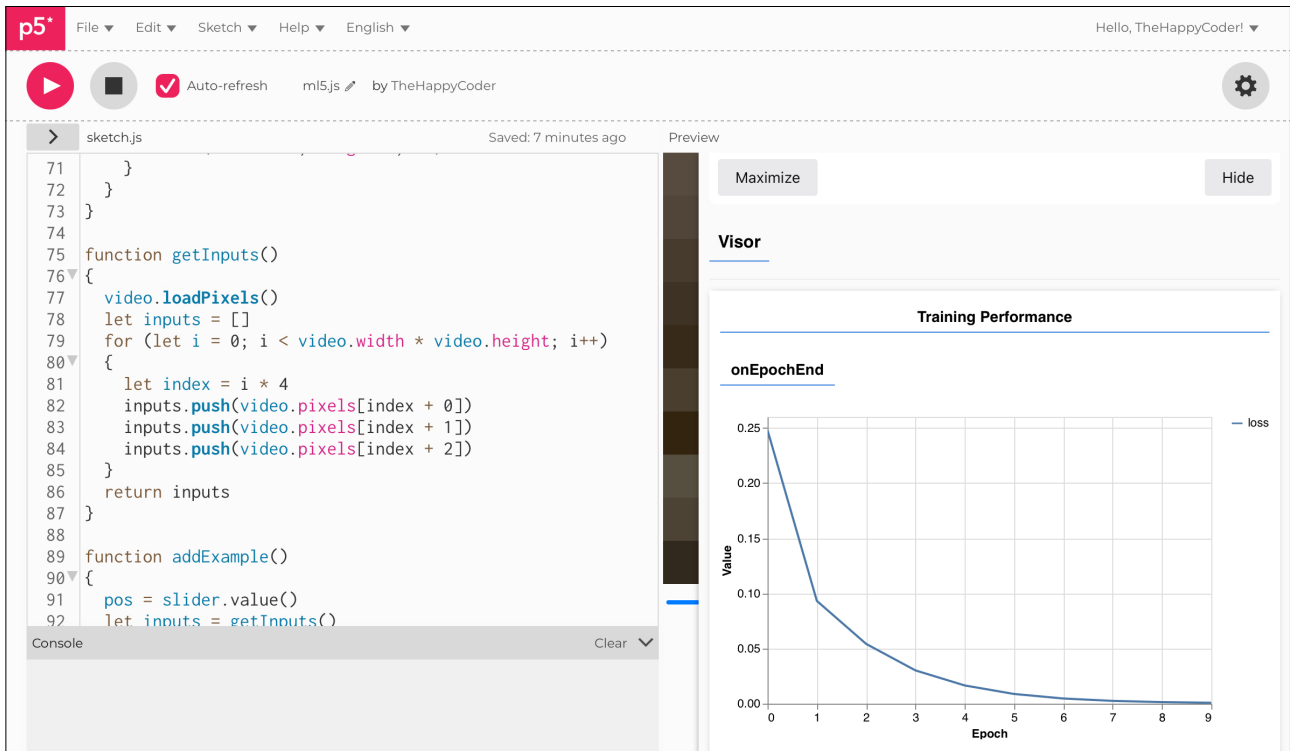


Figure A12.19b: to the left

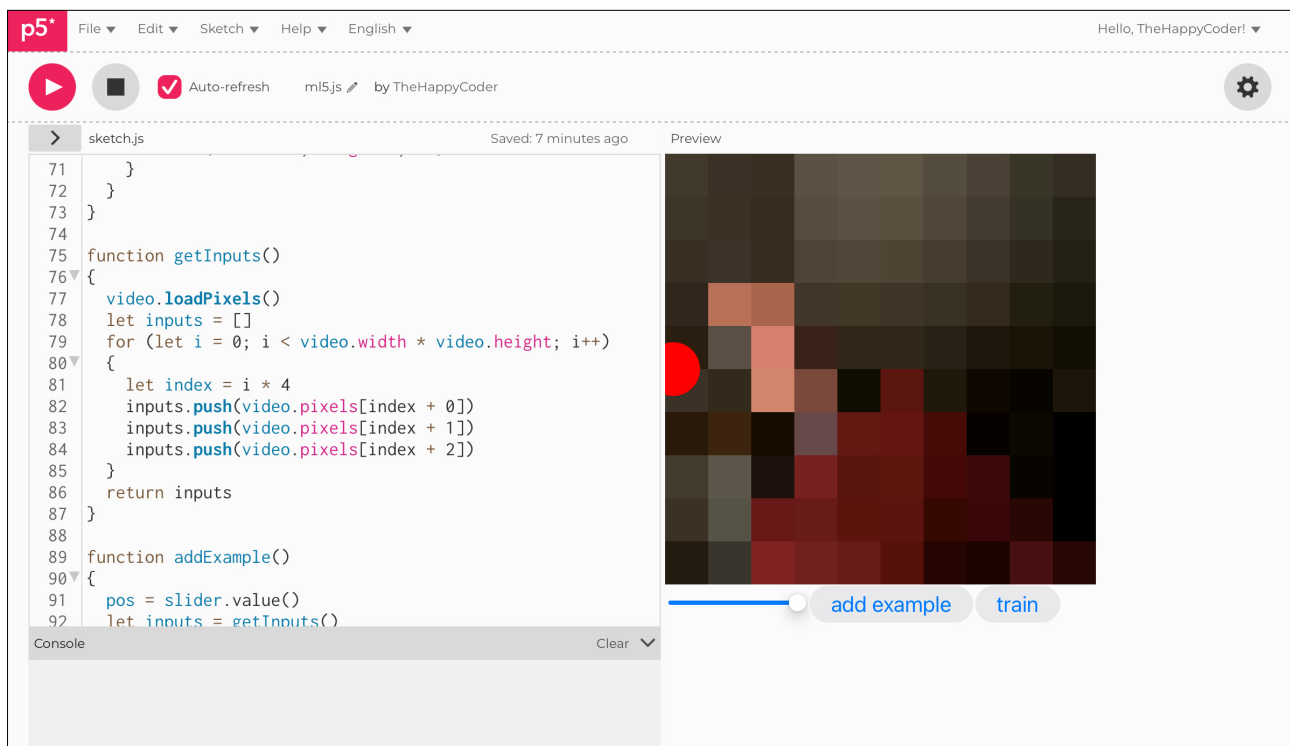
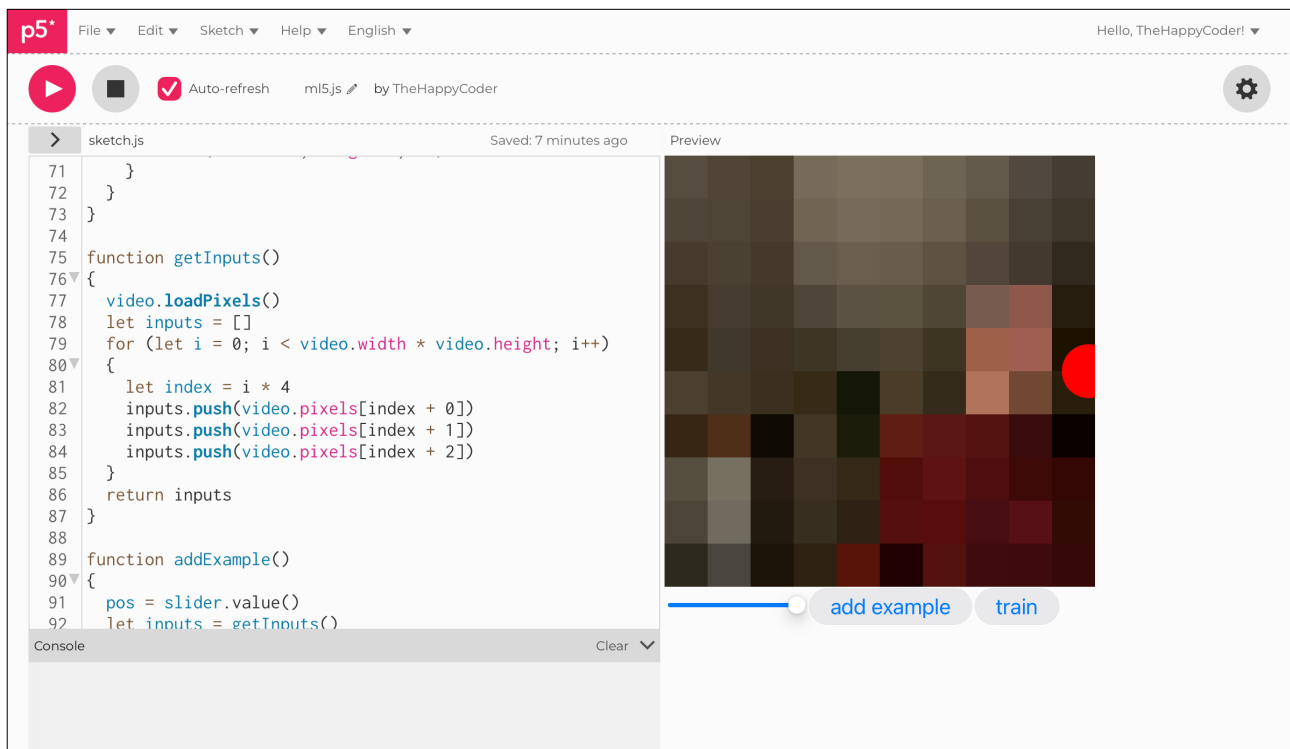


Figure A12.19c: to the right



Algorithmic Intelligence Module A Unit #13 Saving and Loading Data and Models



Module A Unit #13: saving and loading

We are going to save to, and then load the data points from, a JSON file rather than generating those data points each time. This is so that we can then see the impact of changing the hyperparameters without changing the dataset every time.

There are four parts to this unit:

Part 1: saving the data

Part 2: loading the saved data

Part 3: saving the model

Part 4: loading the saved model



Part #1: saving the data

We have used new synthetic data generated each time we run the code. It was good for an initial demonstration of machine learning, but if we want to see how the **hyperparameters** impact the training, we need a fixed dataset, not one that changes every time we run the model.

In this unit, we will save some data and then load it so that we always use the same data when experimenting with our model. You can use this approach for any synthetic dataset that you might generate.

Just another reminder to have the latest p5.js version installed and the latest ml5.js version included. Some of the images may show old versions of either, I may not have got round to changing the images but the code will probably work on both anyway but is now tested on the latest of both.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch A13.1 this is our starting sketch

We have the data points drawn and a `console.log()` of the data so we can see what's inside the `data` array. Notice we have used the `abs()` function inside the `floor()` function. This removes the negative values and keeps everything on the canvas.

```
const number = 30
const spread = 30
let data = []

function setup()
{
  createCanvas(400, 400)
  background(220)
  trainingData()
}

function trainingData()
{
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      noStroke()
      let x = floor(abs(i + random(-spread, spread)))
      let y = floor(abs(height - (i + random(-spread, spread))))
      data.push(x, y)
      circle(x, y, 5)
    }
  }
  console.log(data)
}
```

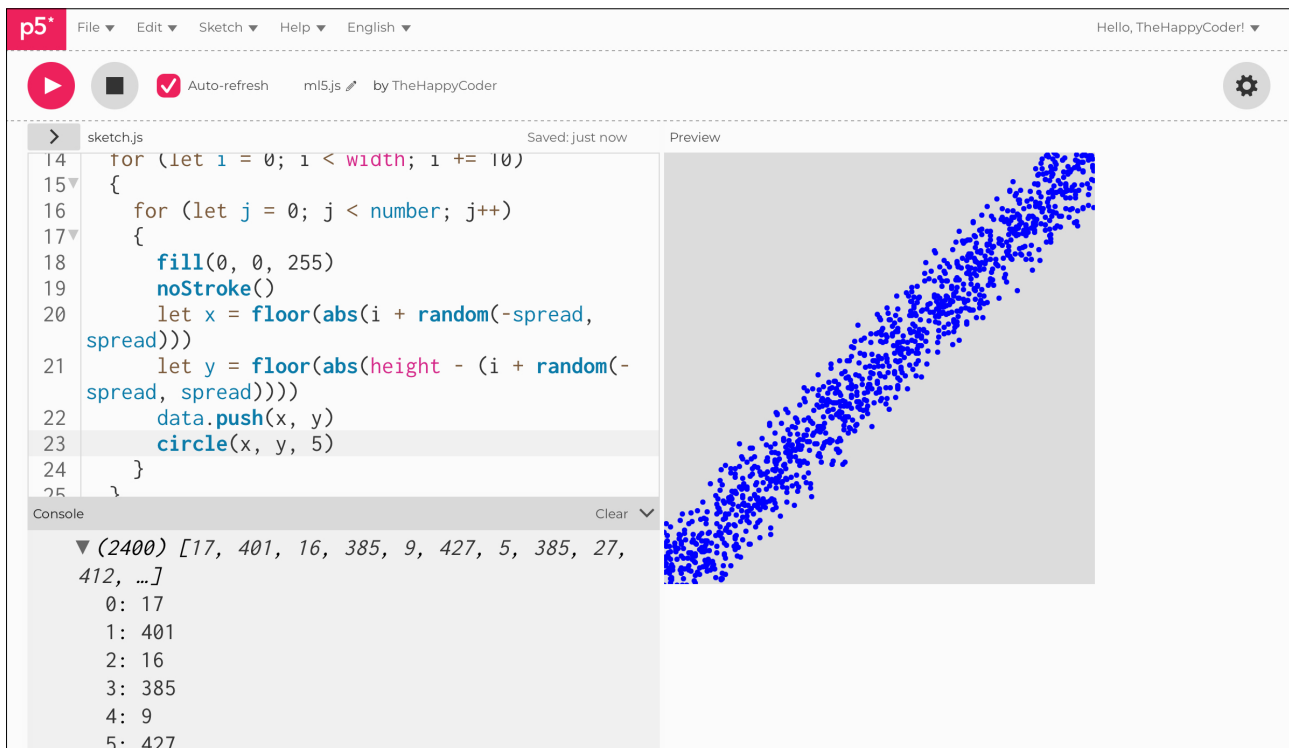
Notes

Recap: The `data` is an empty array that is filled with the `x` and `y` values from the nested loop using the `push()` function. One of the benefits of the `console.log()` function is that we can see the size of the `data` array. In this case, it is `2400` elements, or `1200` pairs of co-ordinates.

Challenge

Try `console.log(data.length)`

Figure A13.1





Sketch A13.2 we need a button to press

We are going to add a button so that we can click on it and save a **JSON** file with all the data in it. First, let's create the button.

```
const number = 30
const spread = 30
let data = []
let button

function setup()
{
  createCanvas(400, 400)
  background(220)
  button = createButton('save data')
  button.style('font-size', '20px')
  button.style('background-color', color('darkred'))
  button.style('color', color('yellow'))
  trainingData()
}

function trainingData()
{
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      noStroke()
      let x = floor(abs(i + random(-spread, spread)))
      let y = floor(abs(height - (i + random(-spread, spread))))
      data.push(x, y)
      circle(x, y, 5)
    }
  }
  console.log(data)
}
```

Notes

We can create a simple button, but I have included some styling to make it a bit more interesting. At this point, however, nothing happens when you click the button; that part is yet to come.

Challenge

Try different colours and sizes of font

Code Explanation

<code>let button</code>	The button variable
<code>button = createButton('save data')</code>	Creating the button and giving it some text
<code>button.style('font-size', '20px')</code>	Increasing the size of the font
<code>button.style('background-color', color('darkred'))</code>	Give the button a background colour
<code>button.style('color', color('yellow'))</code>	Give the text some colour



Sketch A13.3 adding a function

When we press the button, we want it to do something. We use a `mousePressed()` function, which leads to the `saveData()` function when the button is pressed. The data is then saved as a **JSON** file called `trainingData.json` using the `save()` function.

! You may encounter a brown error message. For the moment it is not critical.

```
const number = 30
const spread = 30
let data = []
let button

function setup()
{
  createCanvas(400, 400)
  background(220)
  button = createButton('save data')
  button.style('font-size', '20px')
  button.style('background-color', color('darkred'))
  button.style('color', color('yellow'))
  trainingData()
}

function trainingData()
{
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      noStroke()
      let x = floor(abs(i + random(-spread, spread)))
      let y = floor(abs(height - (i + random(-spread, spread))))
      data.push(x, y)
      circle(x, y, 5)
    }
  }
}

button.mousePressed(saveData)
```

```

    console.log(data)
  }

function saveData()
{
    console.log('button pressed')
    save(data, 'trainingData.json', true)
}

```

Notes

When you click on the button, you should get a request to download the file. It will be different for different machines and browsers. The reason for the boolean true in the file arguments is whether to trim unneeded whitespace. You can open the file to see what's inside it. In some instances, it might just download without a prompt.

Code Explanation

button.mousePressed(saveData)	When the button is pressed it activates the function saveData()
save(data, 'trainingData.json', true)	The save() function needs the first argument to be the data, the second is the name of the file to be saved and the third is a boolean optimiser.

Figure A13.3a

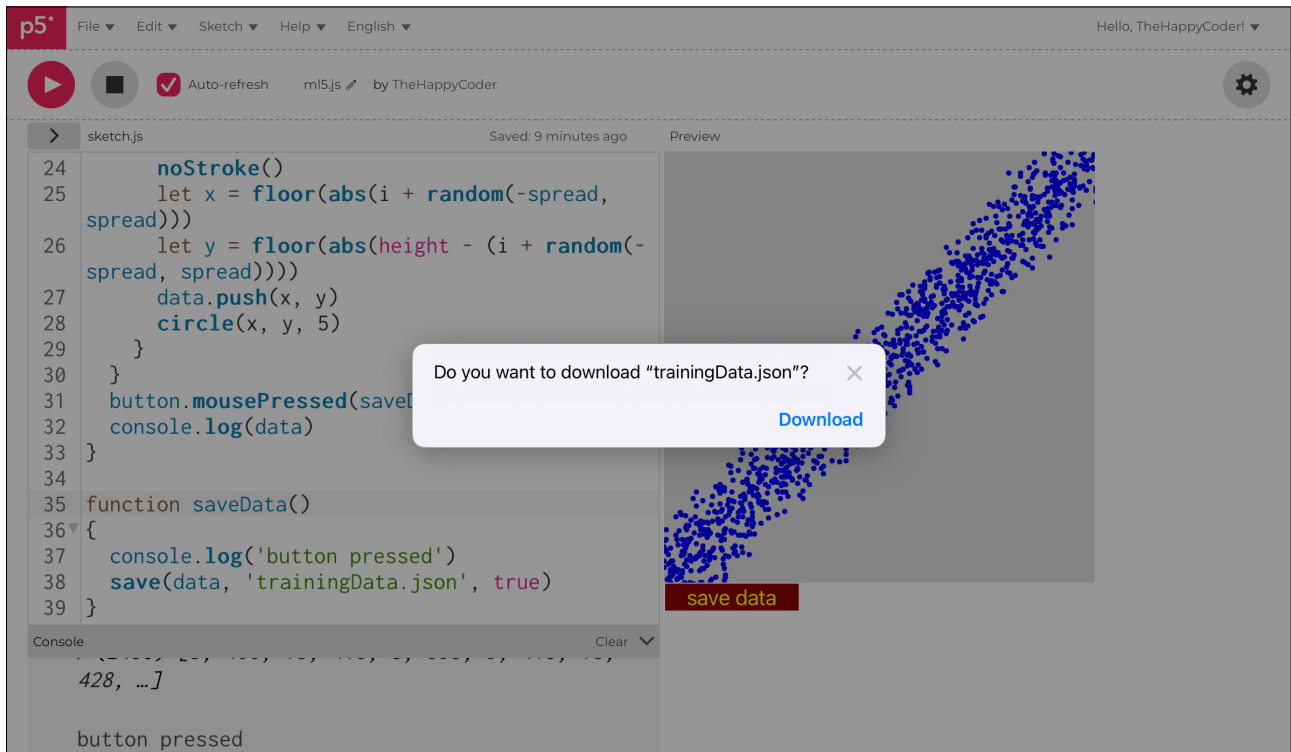


Figure A13.3b having a quick peak inside the trainingData.json file

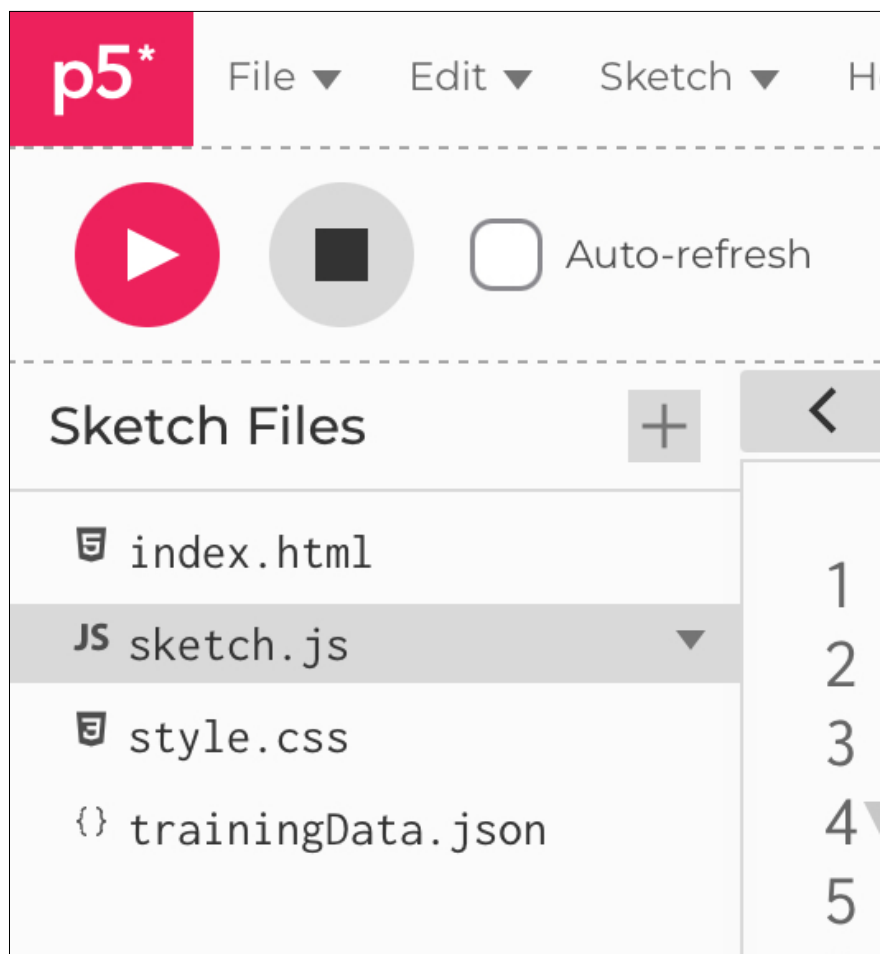
14,422,14,383,20,390,0,392,20,382,4,415,7,396,15,405,6,411,6,425,20,420,22,394,6,397,26,419,19,424,10,415,3,406,3,424,12,381,13,373,15,384,17,395,29,396,26,376,5,371,24,403,24,422,22,408,20,411,26,383,19,400,4,379,10,379,3,375,28,430,33,377,13,399,1,369,35,414,1,369,18,376,2,368,17,388,22,418,398,8,390,0,396,11,401,2,370,13,360,37,375,33,384,20,3,60,18,407,29,370,10,409,1,382,9,400,7,381,0,390,1,461,3,394,1,360,1,373,21,384,48,368,32,389,3,380,44,361,9,359,15,395,39,369,2,370,22,408,15,370,2,370,3,350,5,394,28,376,21,383,38,351,2,376,11,403,13,376,41,383,44,365,36,44,2,363,63,408,7,365,14,398,16,347,32,369,17,392,30,384,35,379,12,355,25,342,22,395,53,342,11,348,9,372,18,385,20,347,40,368,37,39,1,20,399,39,383,17,375,50,362,20,364,1,393,53,371,3,383,0,374,36,371,53,367,29,364,59,354,32,341,3,372,23,339,49,356,32,357,19,356,29,347,34,344,35,368,57,362,55,364,20,361,42,388,34,368,55,349,42,369,49,333,66,40,336,2,384,44,388,35,369,25,356,4,356,47,379,27,365,22,362,35,382,56,361,40,388,30,334,45,368,63,363,42,300,31,375,65,361,75,379,3,334,59,351,72,332,41,345,49,351,47,368,48,328,67,335,40,347,41,366,38,355,46,352,59,359,22,341,69,382,76,331,71,352,35,367,358,34,343,74,372,33,351,60,341,31,323,53,34,6,34,357,69,346,56,338,35,340,360,39,351,57,358,75,367,60,363,37,324,53,315,38,329,73,353,59,328,41,334,43,357,52,320,43,356,40,338,72,345,79,344,78,353,31,364,51,340,7,5,329,42,324,38,556,40,346,60,333,77,330,51,334,58,354,80,335,48,363,37,348,94,332,79,559,66,333,73,349,98,68,66,303,9,350,49,338,77,337,362,130,355,41,350,53,307,81,328,70,353,48,303,82,357,48,337,44,349,83,359,43,315,62,340,99,334,55,335,59,347,81,316,94,323,98,335,20,291,92,303,53,293,94,338,80,335,90,348,56,305,109,304,100,336,76,344,63,315,73,342,73,331,54,332,54,330,42,298,107,304,68,344,59,315,61,292,63,306,86,317,64,337,90,297,95,324,83,333,64,316,80,295,905,326,114,294,66,325,89,319,99,291,82,318,64,335,97,324,60,321,112,299,106,296,64,298,108,308,62,316,104,323,116,315,97,280,109,305,111,332,71,298,73,280,83,305,115,329,66,300,86,289,92,324,105,308,88,299,209,82,97,19,309,79,302,120,301,11,204,116,215,127,314,73,123,270,114,311,126,308,33,277,116,276,91,292,85,314,98,297,96,274,106,283,95,281,115,302,127,319,117,277,116,329,119,300,118,298,106,299,291,113,303,129,281,128,278,101,275,117,303,90,290,122,316,112,311,122,267,116,292,106,268,84,299,124,310,136,261,95,280,104,308,121,274,90,310,200,279,107,315,109,286,128,301,86,309,106,311,115,270,132,269,102,293,127,269,138,278,710,294,103,295,81,261,126,283,80,286,86,315,119,310,90,270,124,260,125,288,106,1291,104,302,126,268,137,29,3,141,286,111,287,95,283,90,309,94,258,135,287,117,259,146,301,114,258,102,298,105,310,273,115,299,119,258,100,340,134,250,104,304,132,256,104,284,138,270,127,254,110,255,133,254,109,285,155,128,114,266,124,295,136,289,129,250,108,274,31,270,140,284,103,242,142,287,136,271,151,276,100,271,149,267,152,91,271,126,74,270,157,271,144,252,149,26,1,106,248,155,252,126,292,155,282,159,286,128,101,277,107,276,143,283,128,268,163,286,168,449,153,235,167,275,130,235,127,262,111,250,123,286,111,254,145,257,134,248,127,230,128,275,119,288,162,277,161,281,117,246,140,252,127,283,113,289,155,285,150,247,125,247,133,244,161,259,124,281,110,289,180,261,337,236,153,229,164,225,128,223,128,261,155,221,147,258,160,220,158,223,159,263,137,254,120,261,386,270,136,246,77,226,148,249,140,257,140,275,132,235,154,255,173,255,138,281,161,253,166,264,157,259,158,247,138,273,149,249,150,230,173,238,173,258,188,268,133,261,166,233,173,242,174,248,130,230,182,227,135,254,183,252,173,266,165,220,130,230,187,243,189,218,183,237,175,266,127,267,148,136,158,263,140,210,140,249,188,268,150,268,137,230,150,258,184,233,173,249,177,116,149,257,169,265,153,224,154,236,148,231,166,258,153,245,142,204,194,255,149,216,173,238,154,256,180,242,196,237,181,248,163,200,174,206,158,209,180,215,198,243,162,207,162,223,179,253,182,39,167,238,168,248,152,222,161,244,183,227,169,226,150,255,140,240,178,214,205,202,179,244,156,243,163,223,180,242,22,158,213,181,194,184,213,175,205,160,199,171,202,201,194,201,245,189,225,165,229



Part #2: loading the data

We need to upload the file to the web editor before we can access it in the sketch. The steps are detailed in the unit called: [A Quick Exploration of p5.js](#). If you are unfamiliar with uploading files, images, videos, or music, then please refer to that section of the unit; it is not difficult. Needless to say, this is what you should have if you open the sidebar of files.

Figure 1: loading the data





Sketch A13.4 loading the file

! Starting a completely new sketch.

Firstly, we need to load the data. The `loadJSON()` function does exactly that. We have to do this asynchronously. This is why we have the keywords `async` and `await`. It is a better way of preloading the data. In version 1.11.x of p5.js we used the `preload()` function, but that will not work with version 2.1.x. Async and await will work with version 1.11.x however.

```
let data

async function setup()
{
  data = await loadJSON('trainingData.json')
  createCanvas(400, 400)
  console.log(data)
}

function draw()
{
  background(220)
}
```

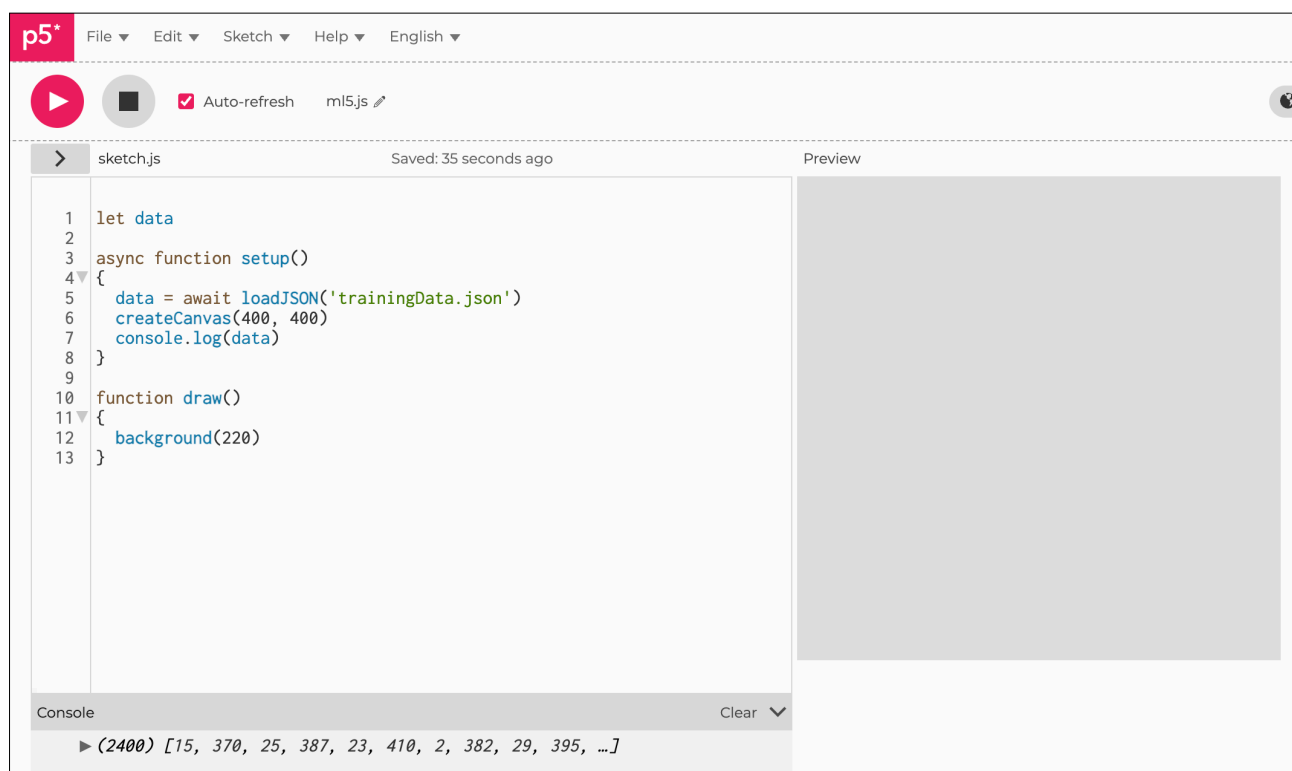
Notes

We `console.log(data)` just to make sure that it has loaded. You should get an object array in the console, but nothing on the canvas (yet).

Code Explanation

<code>async function setup()</code>	The <code>setup()</code> function needs to complete a task before anything else happens
<code>data = await loadJSON('trainingData.json')</code>	In <code>setup()</code> nothing else will happen until all the data is loaded

Figure A13.4





Sketch A13.5 checking the size

We have a small loop that goes through all the data points in the `data` array. We have a counter called `num` which adds `1` each time for each data point. When it has gone through the whole dataset (`data` array), we call `noLoop()` which stops the `draw()` function looping. We console log to see what `num` finally comes up with; this will give us the size of the `data` array.

! Remove the console log in the `setup()` function; we don't need it anymore.

```
let data
let num = 0

async function setup()
{
  data = await loadJSON('trainingData.json')
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  for (let datapoints in data)
  {
    num++
  }
  noLoop()
  console.log(num)
}
```

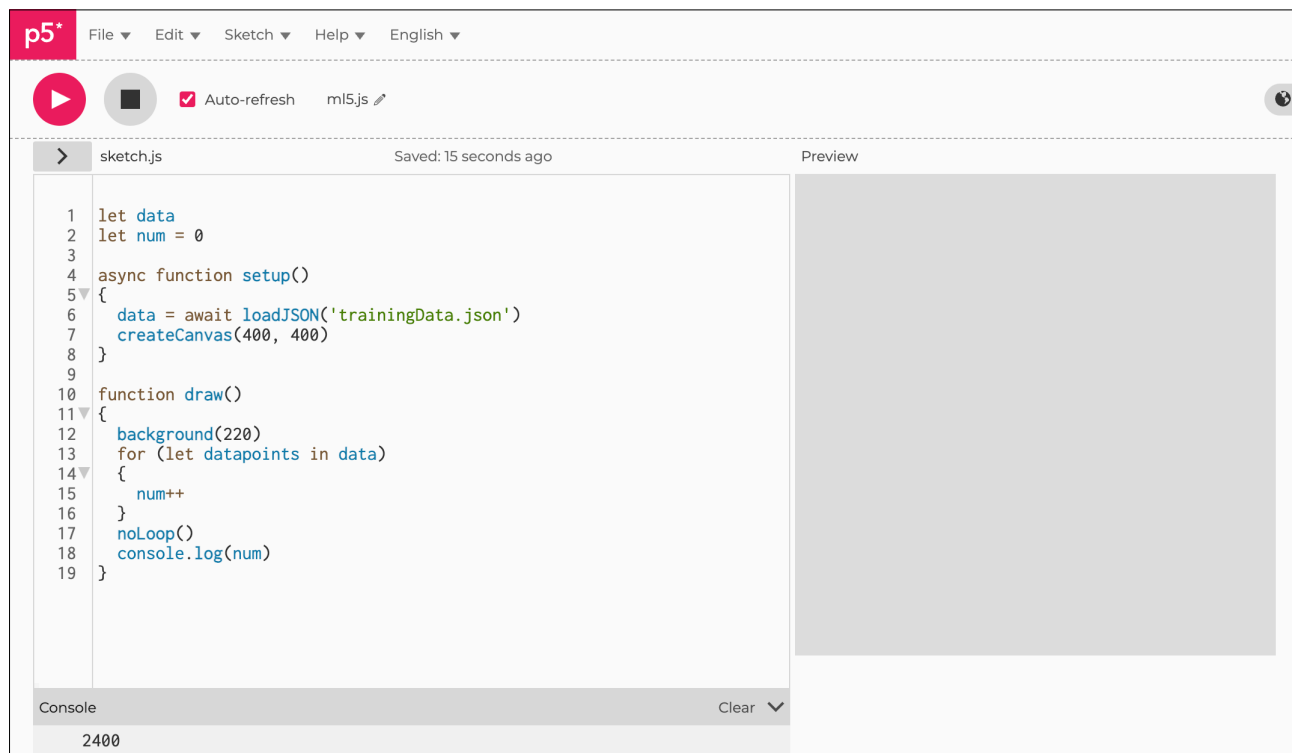
Notes

As expected, the size of the array is **2400** data points. This means we can use that number to draw all the data points.

Code Explanation

<code>let num = 0</code>	Declare and initialise a counter variable
<code>noLoop()</code>	Stops a loop if there is one running
<code>for (let datapoints in data)</code>	A for loop that iterates through all the elements in an array and copies them to a new array.

Figure A13.5





Sketch A13.6 drawing the points

We jump every two elements in the array, as they are **pairs** of elements (**x**, **y**). Hence, we use the index for **x** as **data[i]** and for **y** as **data[i+1]**.

! Remove the **console.log(num)**, we don't need it anymore.

```
let data
let num = 0

async function setup()
{
  data = await loadJSON('trainingData.json')
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  for (let datapoints in data)
  {
    num++
  }
  noLoop()

  for (let i = 0; i < num; i += 2)
  {
    fill(0, 0, 255)
    noStroke()
    circle(data[i], data[i + 1], 5)
  }
}
```

Notes

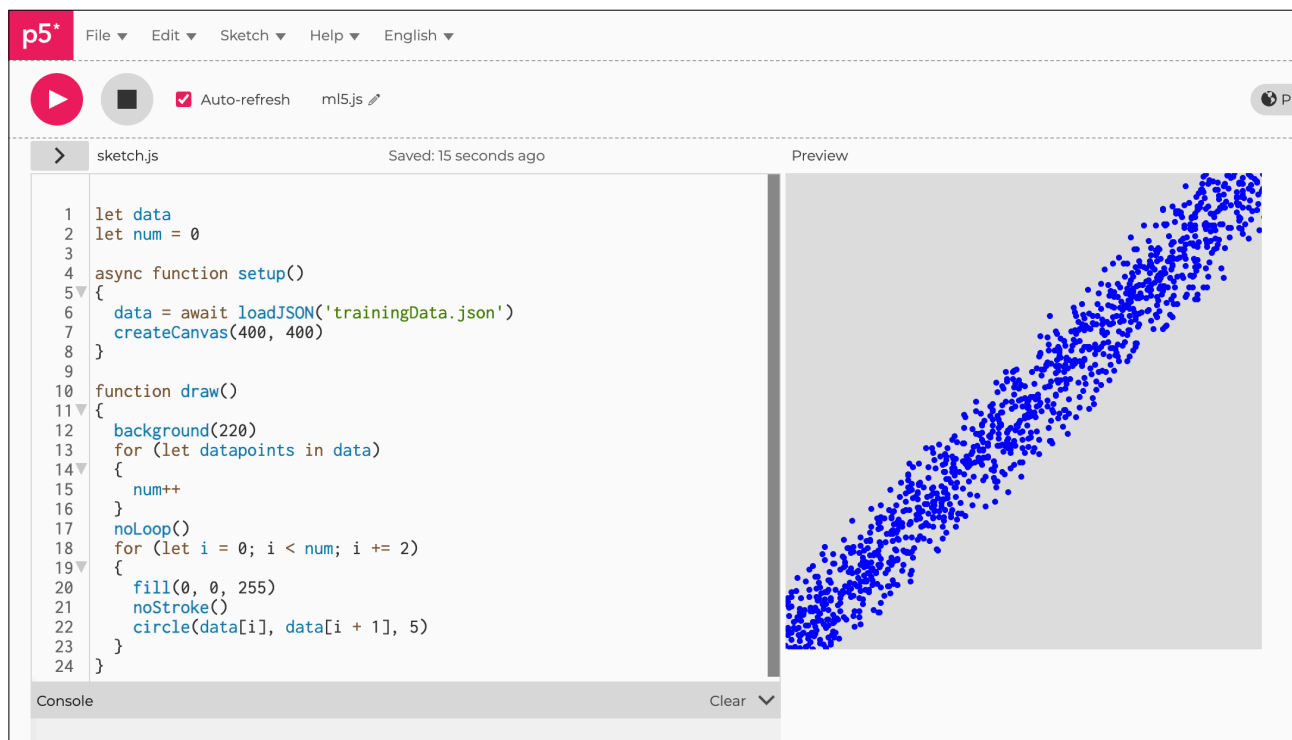
There are **2400** data points, but only **1200** pairs of data points. We get the same result as when we saved the data in the first instance.

! Keep the file in the sketch for later; we will use it with a model.

Code Explanation

<code>for (let i = 0; i < num; i += 2)</code>	We work through the size of the array by placing the variable <code>index[i]</code> every second element <code>i += 2</code>
<code>circle(data[i], data[i + 1], 5)</code>	This is where <code>i</code> is the x index data point and <code>i + 1</code> is the y data point

Figure A13.6

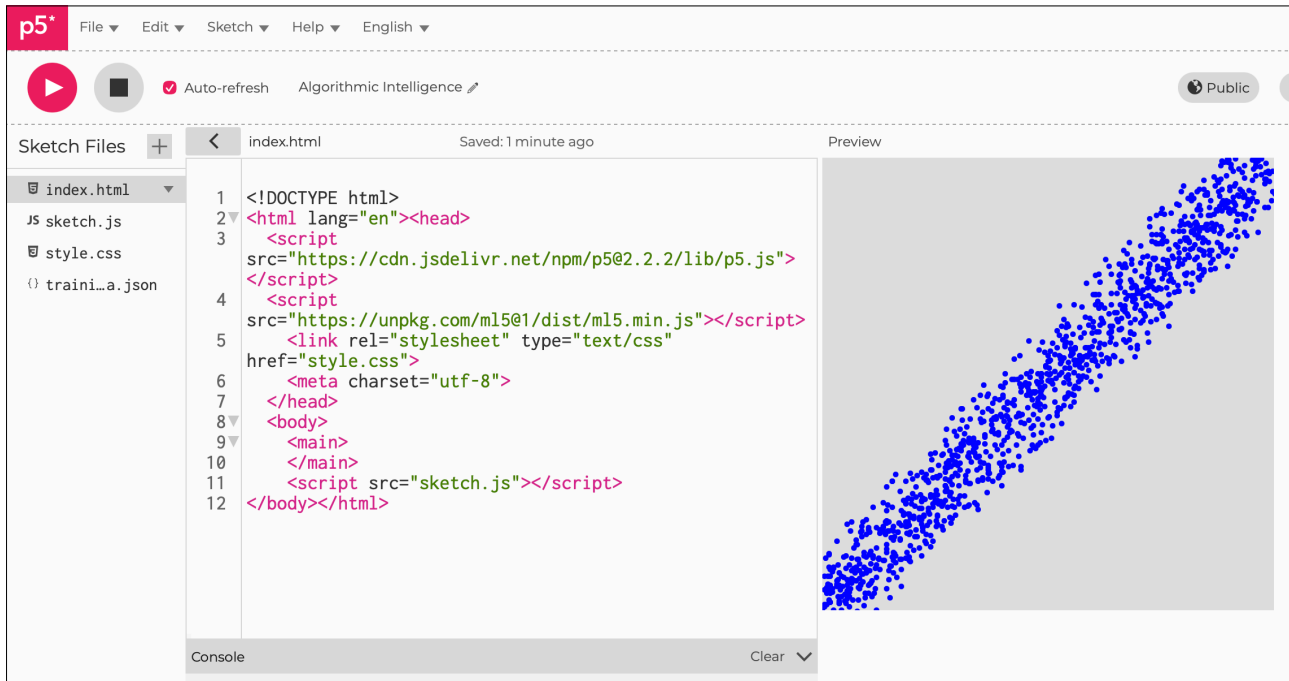




Part #3: saving the model

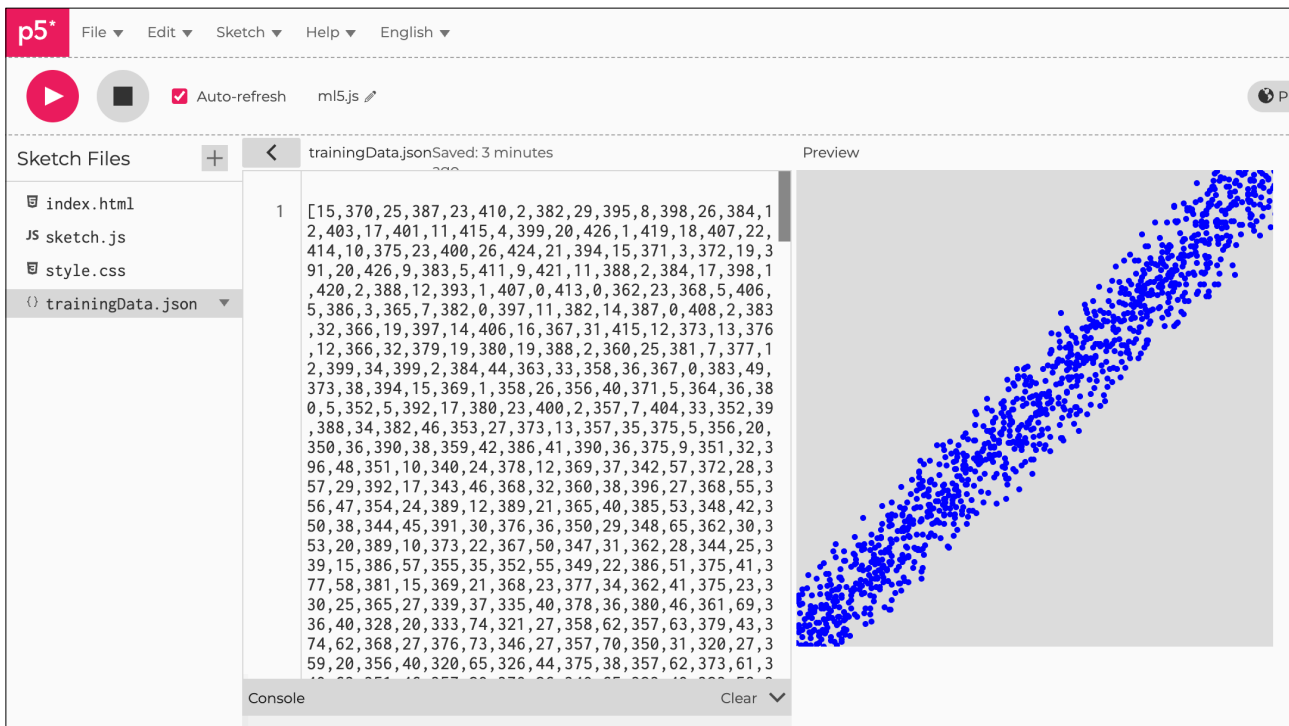
Make sure you have the ml5.js line of code in the index.html file, see Fig.2.

Figure 2: ml5.js in the index.html file



Also, make sure you have uploaded the file `trainingData.json`, see Fig.3.

Figure 3: Training data file





Sketch A13.7 we start again

This is our starting sketch; it isn't exactly the same as where we left off because we will be making major changes and additions. We won't be generating the data but loading it instead. So you will need to write the code below as is, then we can add in everything else as we go along.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: false
}

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
}

function finishedTraining()
{
  if (counter < 400)
  {
```

```
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  finishedTraining()
}
```

Notes

! Be aware, if you run the sketch at this stage, you will get an error.



Sketch A13.8 loading the data

We now use the `async/await` function to load the data into the sketch. We use the `for()` loop to get the number of data points (`num`) from the `data` array.

! Error message alert.

```
let nn
let counter = 0
let data
let num = 0
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: false
}

async function setup()
{
  data = await loadJSON('trainingData.json')
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
  for (let datapoints in data)
  {
    num++
  }
}
```

```
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  finishedTraining()
}
```

Notes

This gives us the data points. Now we want to add those data points to the neural network (and draw them).



Sketch A13.9 adding the data and drawing

We will now add the data to the neural network to predict the line and also draw the data points (circles).

```
let nn
let counter = 0
let data
let num = 0
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: false
}

async function setup()
{
  data = await loadJSON('trainingData.json')
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
  for (let datapoints in data)
  {
    num++
  }

  for (let i = 0; i < num; i += 2)
  {
```

```
    fill(0, 0, 255)
    noStroke()
    circle(data[i], data[i + 1], 5)
    nn.addData([data[i]], [data[i + 1]])
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  finishedTraining()
}
```

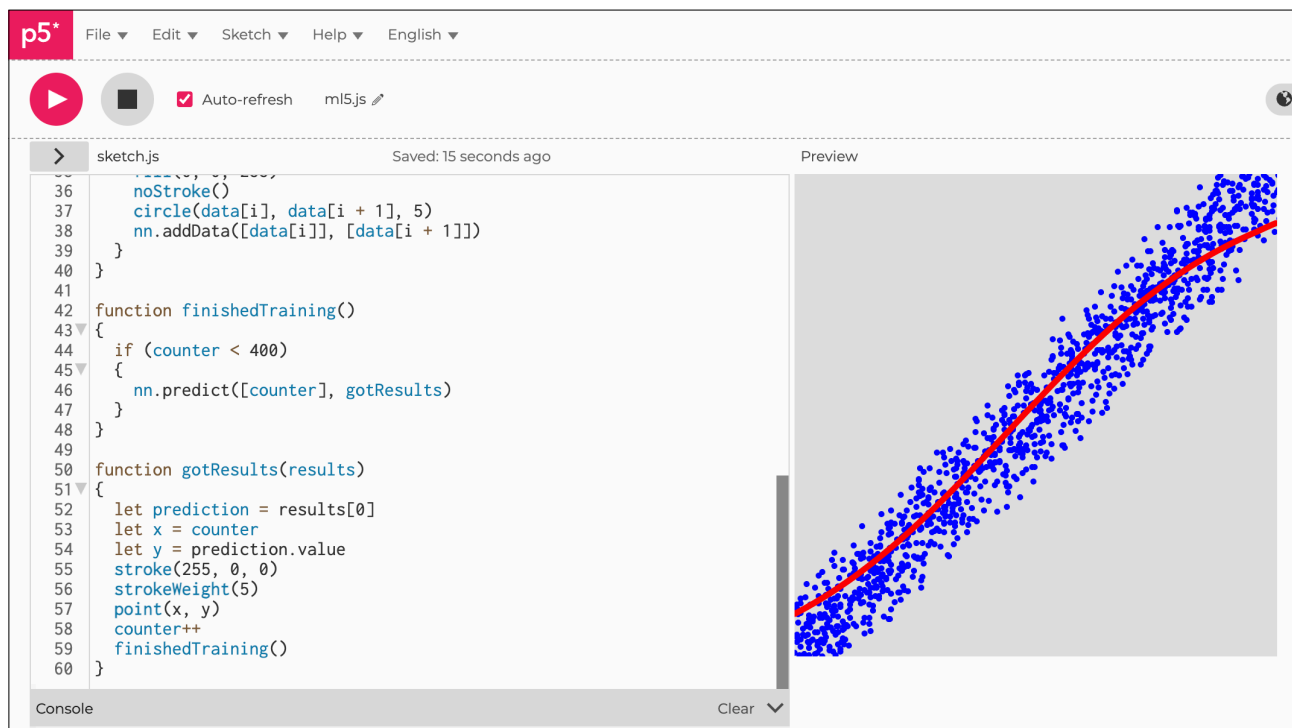
Notes

What I get is not a very straight line at all, but this is really to illustrate the process. The next section is about saving the model, at least one we think is good enough.

Code Explanation

<code>circle(data[i], data[i + 1], 5)</code>	The first element is the x coordinate and the second is the y coordinates.
<code>nn.addData([data[i]], [data[i + 1]])</code>	Adding the x and y coordinates to the neural network

Figure A13.9





Sketch A13.10 creating another button

We want a button to click on when we want to save the model. The button variable is declared, creating the button and also styling it with a bit of colour. We incorporate the button into the `gotResults()` function and call the function `saveModel()` to do just that. When the model is trained, we click on the button and the `save()` function acts on the neural network (`nn`). It should download three files, and those three files are:

`model.json`
`model.weights.bin`
`model_meta.json`

```
let nn
let counter = 0
let data
let num = 0
let button
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: false
}

async function setup()
{
  data = await loadJSON('trainingData.json')
  createCanvas(400, 400)
  button = createButton('save model')
  button.style('font-size', '20px')
  button.style('background-color', color('darkblue'))
  button.style('color', color('yellow'))
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512
  }
  nn.train(trainingOptions, finishedTraining)
```

```

}

function trainingData()
{
  background(220)
  for (let datapoints in data)
  {
    num++
  }
  for (let i = 0; i < num; i += 2)
  {
    fill(0, 0, 255)
    noStroke()
    circle(data[i], data[i + 1], 5)
    nn.addData([data[i]], [data[i + 1]])
  }
}

function finishedTraining()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  finishedTraining()
  button.mousePressed(saveModel)
}

```

```
function saveModel()
{
  nn.save()
}
```

Notes

If you want to give your saved model a different name, then you can (the default is model). You can also have a callback function if you want:

`nn.save('differentName', callback).`

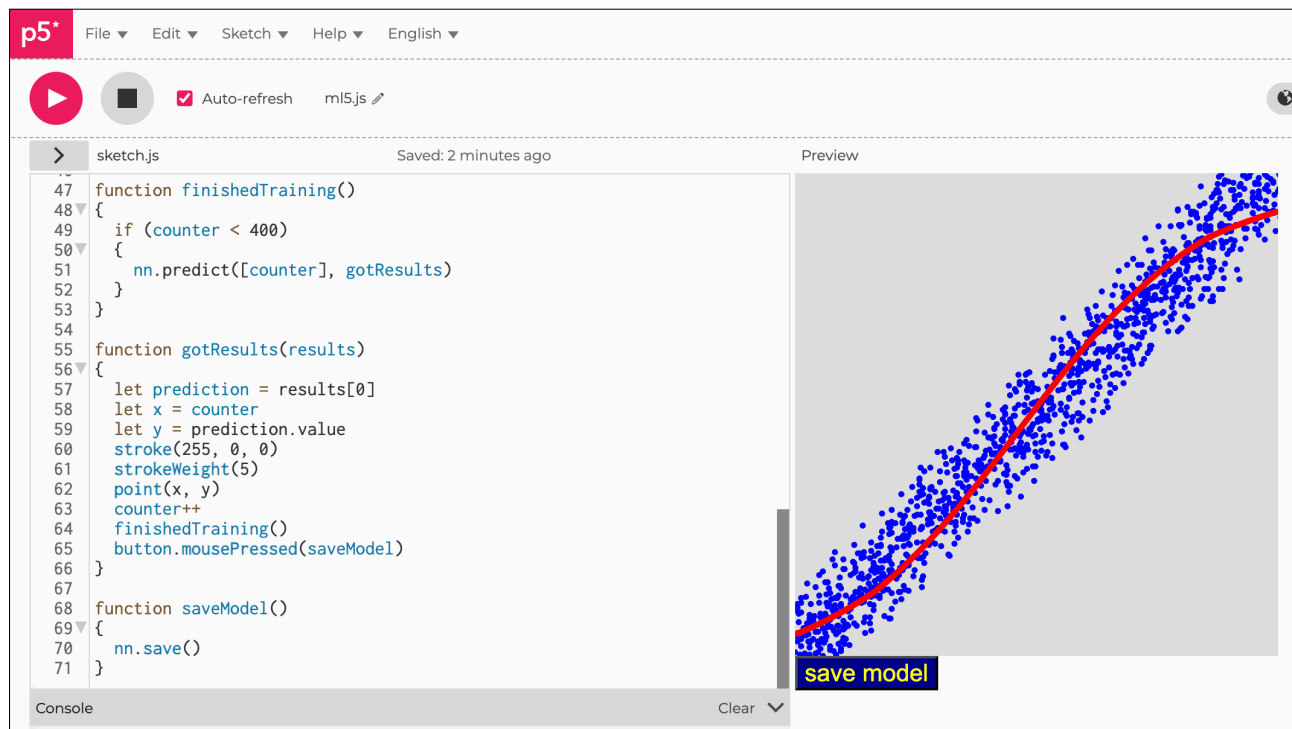
It is imperative that you use **Chrome** as your browser, I found that Safari would not allow me to download all the files no matter how hard I tried, you might be more successful, but my recommendation is use **Chrome**.

Code Explanation

`nn.save()`

This saves the nn model. It gives it the default name 'model', you can specify another name.

Figure A13.10





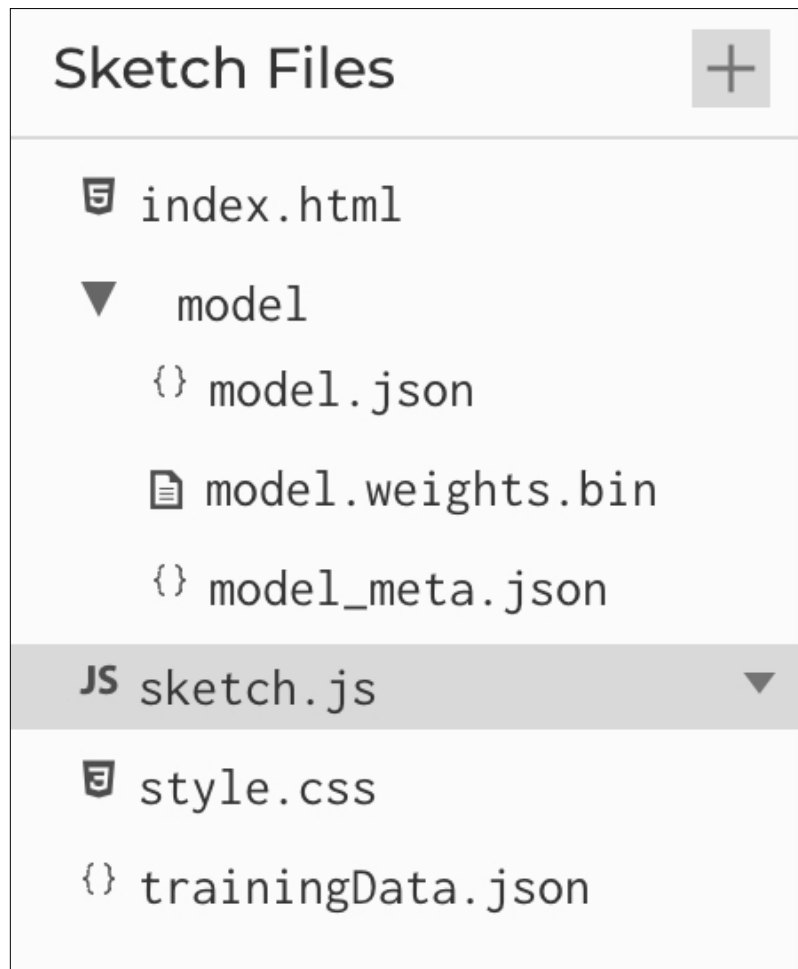
Part #4: loading the model

We have saved the model, so it is now ready for deployment. We are going to see how well it really performs against straight-line data, but remember it was trained on data that had some variance to it, so it won't be exact like the examples previous units.

! To store the saved model files, we have to put them into the same folder (this is important).

Firstly, create that folder; I have called it **model**. Now upload (or drag and drop) the three files into that folder; you can do it one at a time or all three at once. You should have a folder called **model**, and inside that folder, you should have your three files, see Fig. 4 below. Check you have all three models (I had to upload one of them separately for some reason).

Figure 4: model folder





Sketch A13.11 loading the model

! We are starting almost a completely new sketch.

We are now going to load the model into our sketch. What we are going to do is a straight predict on some inputs (x values). But first, we load the model with the `load()` function. New relevant stuff highlighted.

```
let nn
let counter = 0
let data

const modelInfo = {
  model: "model/model.json",
  metadata: "model/model_meta.json",
  weights: "model/model.weights.bin"
}

function setup()
{
  createCanvas(400, 400)
  background(220)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork()
  nn.load(modelInfo)
}
```

Notes

We load the model with the `load()` function, giving it the folder and clear path details. This is very important; they are all case-sensitive. The three files do need to be in the same folder, and they do need the name of the folder followed by a forward slash and then the name of the file. Each one is prefixed by `model`, `metadata`, and `weights`.

Code Explanation

<code>nn.load(modelInfo)</code>	Loading the model with the details of the model reference
---------------------------------	---



Sketch A13.12 a callback

We want to make sure that it is loaded before we do the predictions. We simply use **async/await** for this.

```
let nn
let modelLoaded
let counter = 0
let data

const modelInfo = {
  model: "model/model.json",
  metadata: "model/model_meta.json",
  weights: "model/model.weights.bin"
}

async function setup()
{
  createCanvas(400, 400)
  background(220)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork()
  modelLoaded = await nn.load(modelInfo)
}
```

Notes

We are nearly there. We have loaded the model and created a callback function. We have yet to predict and draw the line.



Sketch A13.13 predicting

We create a function called `predicting()`. This function checks to see if the counter is less than `400`. This uses the counter as the `x` input, and `y` is the predicted output value. We draw the red line as before.

```
let nn
let modelLoaded
let counter = 0
let data

const modelInfo = {
  model: "model/model.json",
  metadata: "model/model_meta.json",
  weights: "model/model.weights.bin"
}

async function setup()
{
  createCanvas(400, 400)
  background(220)
  ml5.setBackend("webgpu")
  nn = ml5.neuralNetwork()
  modelLoaded = await nn.load(modelInfo)
  predicting()
}

function predicting()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

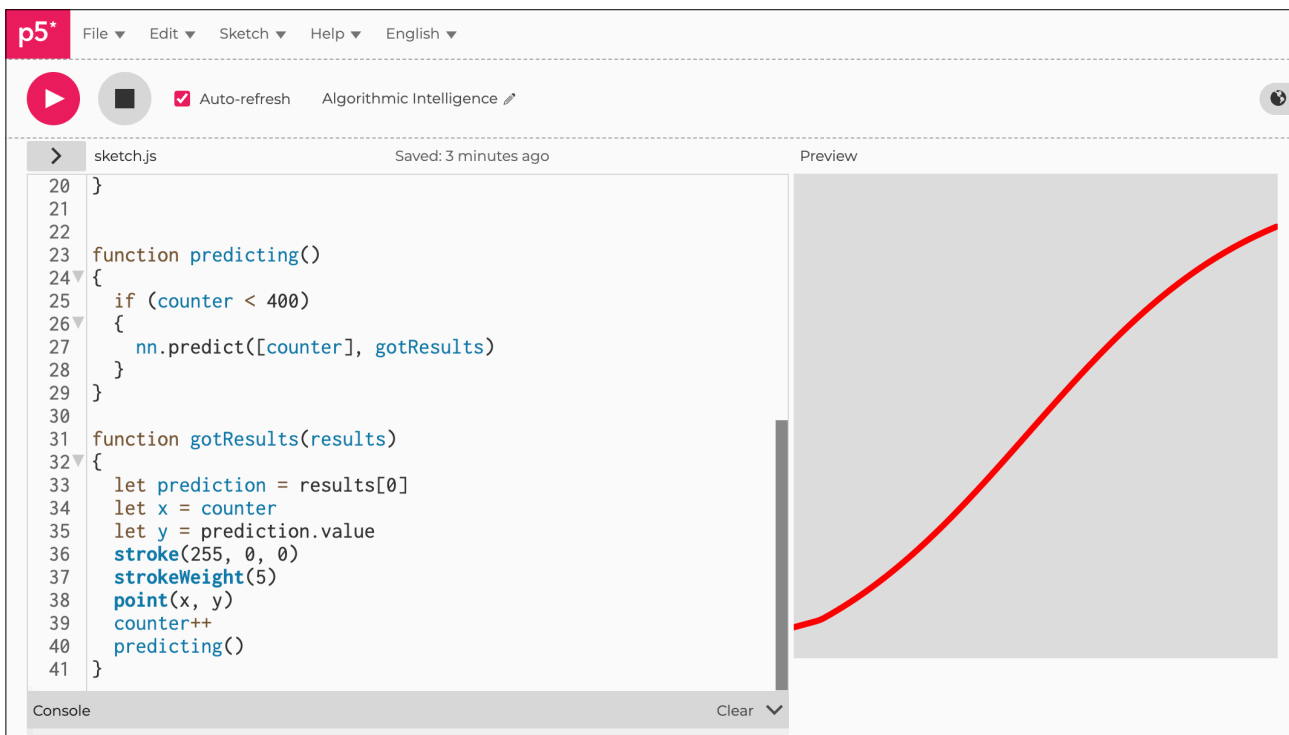
function gotResults(results)
{
  let prediction = results[0]
  let x = counter
```

```
    let y = prediction.value
    stroke(255, 0, 0)
    strokeWeight(5)
    point(x, y)
    counter++
    predicting()
  }
```

Notes

We can see the predictions. It should draw the same line as before. It is a trained model.

Figure A13.13





Sketch A13.14 comparing

We will now draw what the line should look like as a comparison to the true relationship between the x input values (400) and the corresponding y values ($y = x$ straight line equation).

```
let nn
let modelLoaded
let counter = 0
let data

const modelInfo = {
  model: "model/model.json",
  metadata: "model/model_meta.json",
  weights: "model/model.weights.bin"
}

async function setup()
{
  createCanvas(400, 400)
  background(220)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork()
  modelLoaded = await nn.load(modelInfo)
  predicting()
}

function predicting()
{
  if (counter < 400)
  {
    nn.predict([counter], gotResults)
  }
}

function draw()
{
  for (let i = 0; i < width; i++)
  {
```

```
    data = [i, height - i]
    fill(0, 0, 255)
    noStroke()
    circle(data[0], data[1], 5)
  }
}
```

```
function gotResults(results)
{
  let prediction = results[0]
  let x = counter
  let y = prediction.value
  stroke(255, 0, 0)
  strokeWeight(5)
  point(x, y)
  counter++
  predicting()
}
```

Notes

You can see it is reasonably close to the blue line. Not bad but not perfect. We haven't used many of the other **hyperparameters** we could alter, so there is plenty of room for improvement.

Challenges

1. Try the line but with more of the hyperparameters.
2. Try the sine wave

Figure A13.14





Cluster example Module A Unit #9

Using the process outlined above I saved the data of the four coloured circle clusters (A, B, C, and D). Then trained the model on that data, saved the model. Used the model to classify ever pixel on the canvas rather than use the mouse to drag across the canvas.

```
let nn
let counterX = 0
let counterY = 0

const modelInfo = {
  model: "model/model.json",
  metadata: "model/model_meta.json",
  weights: "model/model.weights.bin"
}

async function setup()
{
  createCanvas(400, 400)
  background(220)
  ml5.setBackend("webgpu")
  nn = ml5.neuralNetwork()
  modelLoaded = await nn.load(modelInfo)
  predicting()
}

function predicting()
{
  let inputs = {
    x: counterX,
    y: counterY
  }
  nn.classify(inputs, gotResults)
}

function gotResults(results)
{

```

```
if (counterX > width)
{
    counterX = 0
    counterY++
}
if (counterY > height)
{
    counterY = height
    counterX = width
}
if (results[0].label === 'A')
{
    stroke(0, 0, 255)
}
if (results[0].label === 'B')
{
    stroke(255, 0, 0)
}
if (results[0].label === 'C')
{
    stroke(0, 255, 0)
}
if (results[0].label === 'D')
{
    stroke(255, 255, 0)
}
point(counterX, counterY)
counterX++
predicting()
}
```

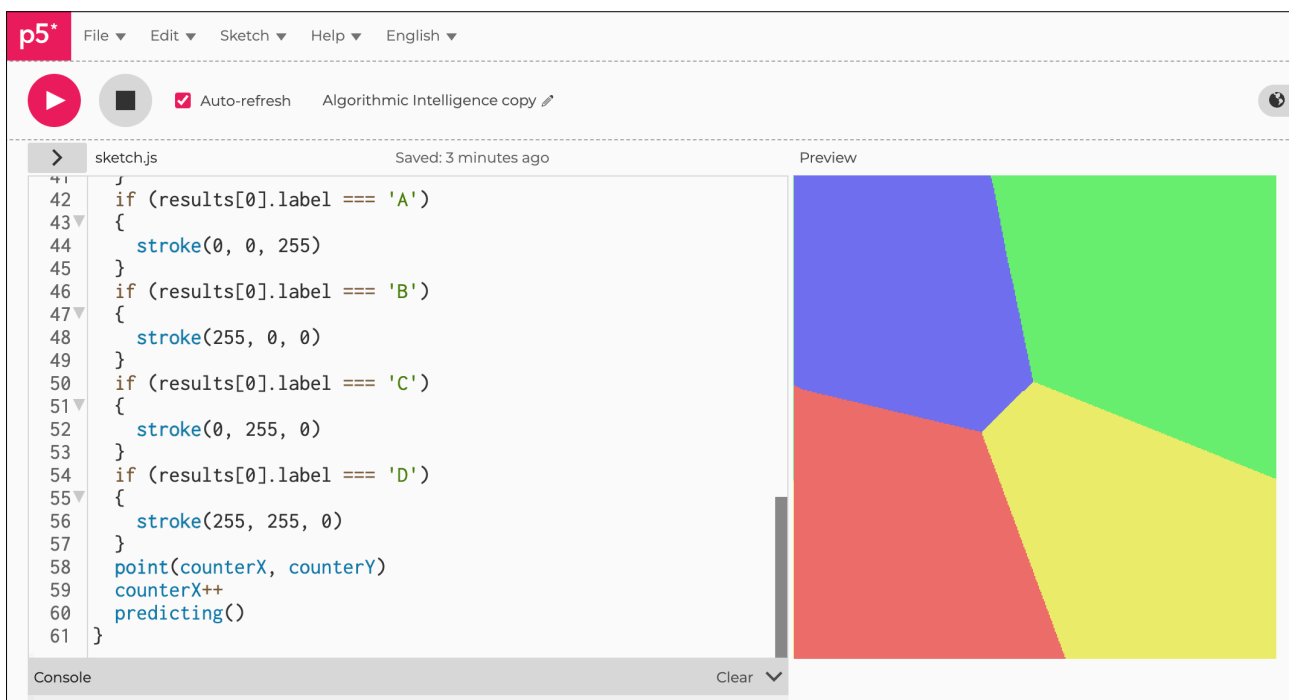

Notes

This was a bit of a challenge and after many failed attempts, we have success.

Challenges

1. Add in the data points that were saved (overlay the circles)
2. Apply this process for Module A Unit #10

Figure 5: module A Unit #9 example





Additional notes

Although we know the length of the dataset, it isn't always obvious, so here are three options for collecting that bit of information when looping through the dataset. We can get the length of a .json file with the following code snippets and button styling.

An alternative way of getting length of an array (more concise).

```
let num = Object.keys(data).length
```

The method we adopted in the workbook to get the length of an array.

```
let num = 0
for (let key in data)
{
  num++
}
```

... or

```
let num = Object.entries(data).length
```

Here is the code for changing the font colour and background of a button.

```
let button

function setup()
{
  createCanvas(400, 400)
  button = createButton("Click Me")
  button.position(190, 200)
  button.style('background-color', color(255, 0, 0))
  button.style('color', color(255))
}
```