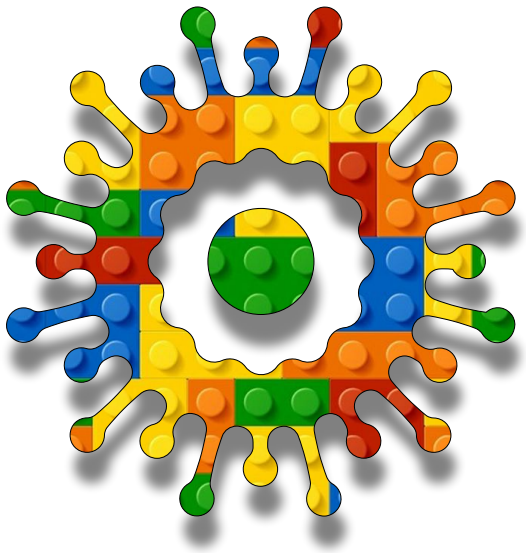
Algorithmic Intelligence Module A Unit #5 linear regression



Content

Module A Unit #5 linear regression with ml5.js

In simple terms, we are going to use the data points for a straight line. We will use this dataset to train the model. It will then predict the relationship between x and y of a simple straight line (linear). We will be drawing the prediction so we have a visual result to see how well it has performed.

This may seem like a rather simple exercise, but it will serve as a demonstration of how a machine learning neural network will learn, demonstrating its usefulness and also its limitations. To make it a bit more realistic and challenging, we are going to introduce a bit of variance; what this means is adding some randomness to the data.

This is because real-world data is very rarely so neat and tidy. Machine learning (AI) comes into its own when patterns aren't necessarily so obvious. This becomes even more so when there is a lot of data, with many variables, and the usual strategies or algorithms would struggle.

Here we have two variables. The `x` and the `y` co-ordinates. The equation for a line is `y = mx + b`. The `m` is the slope of the line, and the `b` is where it intercepts the y-axis. We artificially create the data and add in the variance. This is `synthetic data`; it isn't real data seen in the world, but it will serve the purpose.

We train the model on this data and get the model to predict a line through the data. Its prediction is the generalisation (or best guess) of the data. The beauty of this exercise is that we can play around with a number of hyperparameters, such as the number of hidden layers, the number of nodes (in each layer), the activation function, the learning rate, the optimiser, the batch size, and of course, the epochs.

Backend Stuff

To speed up, or even make it work at all, depending on your machine or browser, you may need to have one of the following lines of code:

```
ml5.setBackend("webgl")
ml5.setBackend("cpu")
```

By default, we are going to add `webgl` to make sure that it works across all browsers. You can add `cpu` instead; it might work very much faster if you do, but, however, it might affect the programme depending on your machine and browser. Other alternatives are `gpu` or `webgpu`. Just experiment at a later date to get a better performance.

Introducing ml5.js

In the unit a gentle overview of ml5.js I showed you how to add the line of code into the index.html file. This is important; otherwise, there is no neural network to train the model. So, if you have not read through that section, it is important that you do so, or you can duplicate the template I have provided on my website by clicking on the button, and then come back to this unit.

The index.html file

The line of code needs to be added to the `index.html` file before you do anything else.

② if you swap to version 2 you will have slightly different index.html, it will show you the p5.js library as 2.1.x but there will not be any sound library. As far as I know it might be incorporated into the new version.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.11.0/p5.js"></
script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.11.0/addons/
p5.sound.min.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 A5.1 starting sketch

This is our starting sketch.

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

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

Sketch A5.2 drawing the points on a line

We create a `for()` loop to draw each data point as a blue circle. We space them out by 10 pixels.

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

function draw()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    fill(0, 0, 255)
    circle(i, height - i, 5)
  }
}
```

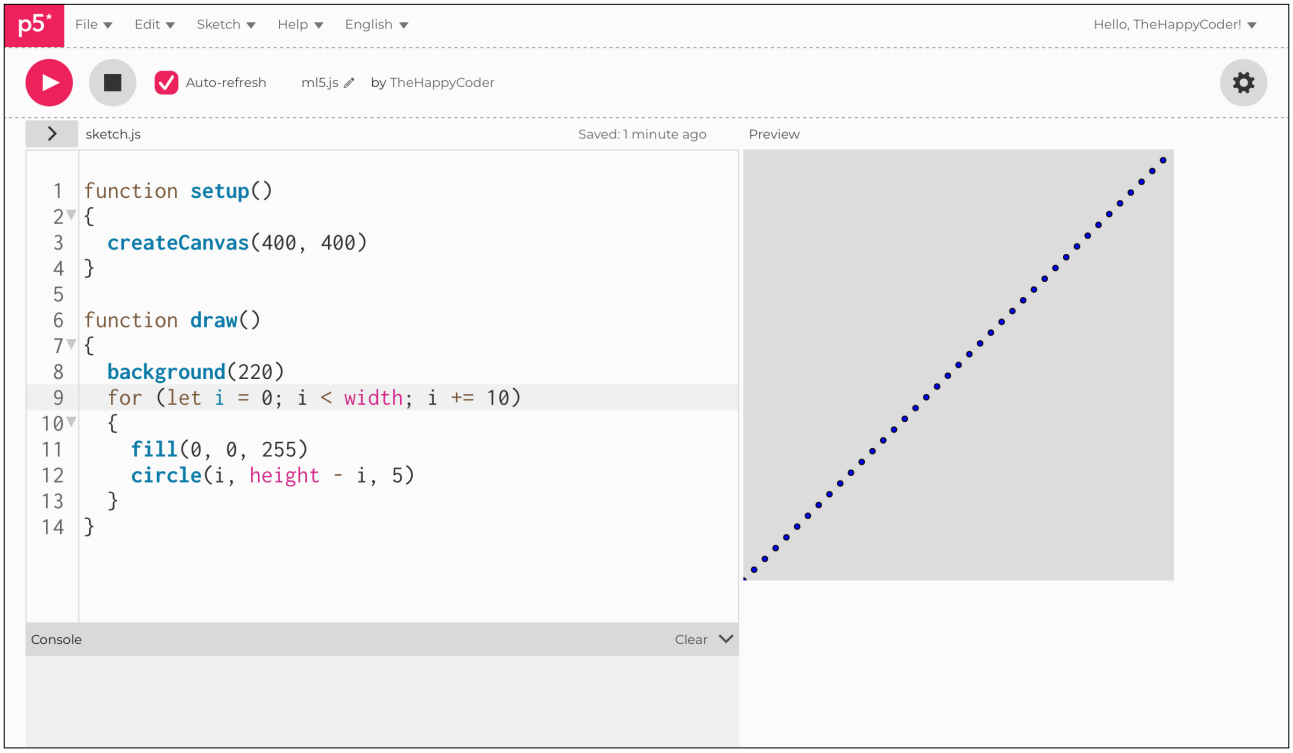
🗒️ Notes

For each value of `i`, this will be the `x` co-ordinate of the circle; the `height-i` is the `y` co-ordinate. Simply, this is a straight line where `y = x`, and the slope `m` is effectively `1` and the intercept `b` is `0`.

🛠️ Code Explanation

`i += 10`	Adding 10 each iteration of the for loop, this increases the value of i by 10

Figure A5.2


p5*
File
Edit
Sketch
Help
English
Hello, TheHappyCoder!
Auto-refresh
ml5.js
by TheHappyCoder
sketch.js
Saved: 1 minute ago
Preview
function setup()
{
createCanvas(400, 400)
}
function draw()
{
background(220)
for (let i = 0; i < width; i += 10)
{
fill(0, 0, 255)
circle(i, height - i, 5)
}
}
Console
Clear

Sketch A5.2 initial sketch with data points

We want to draw 30 circles, one on top of the other. Hence, we introduce a constant (const) variable called number. In the next part, we will spread those 30 circles randomly around the line with some variance (spread).

```
const number = 30

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

function draw()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      circle(i, height - i, 5)
    }
  }
}
```

🗒️ Notes

Now we are drawing 30 circles for each data point. We use j as the variable for the nested loop.

Sketch A5.3 sketch revised

Now we are going to spread the data points using the variable `spread`. You will notice that it draws the circles repeatedly in random points.

```
const number = 30
const spread = 30


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


function draw()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      circle(i + random(-spread, spread), height - i + random(-spread, spread), 5)
    }
  }
}
```

🗒️ Notes

We don't want the circles being drawn continuously; we will address the issue in the next sketch.

🛠️ Code Explanation

`random(-spread, spread)`	Gives us a random value between -30 and +30
`i + random(-spread, spread)`	This means that for every i (x) value we move it above or below the real value.

Sketch A5.4 revising the data

To stop it looping through, we change the name of the draw() function to trainingData(). This means it is no longer a continuous loop. We move background(220) into the setup() function and call the trainingData() function there. Now it draws the data points just once every time you run the sketch, but not continually as before.

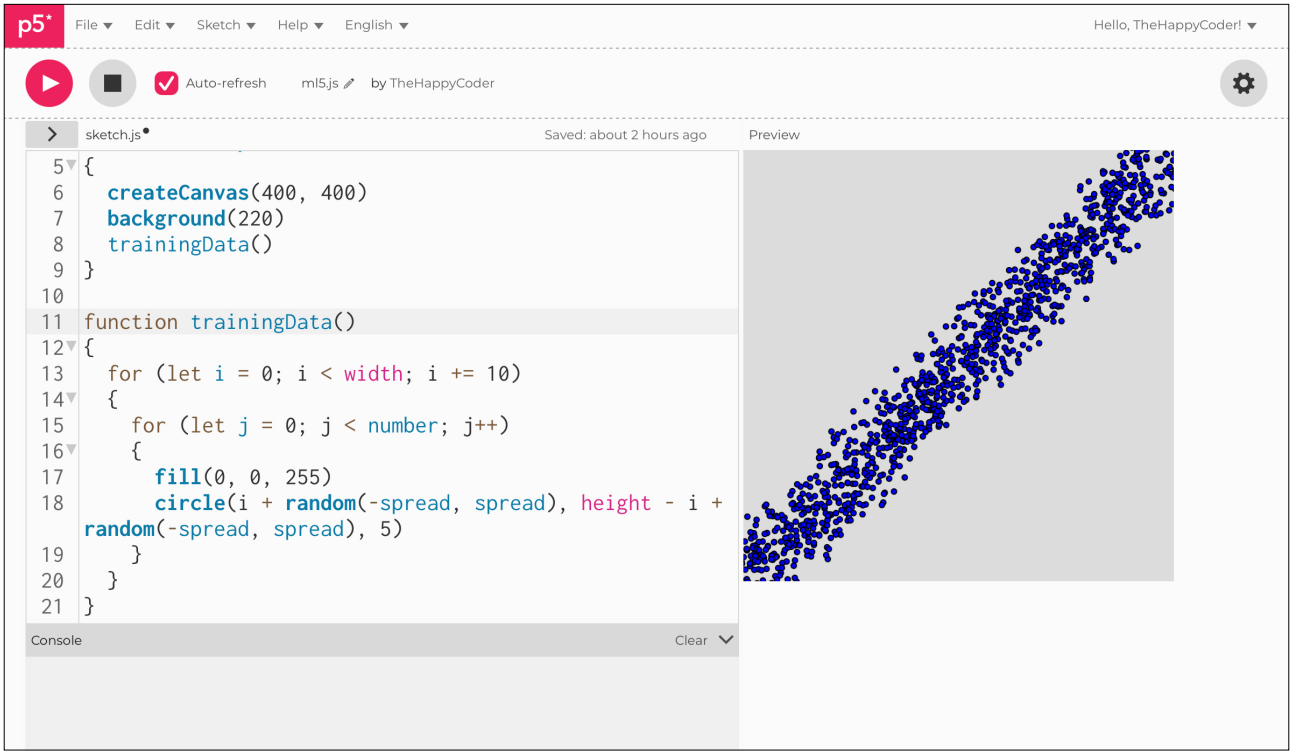
```
const number = 30
const spread = 30

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

function trainingData()
{
  // background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      circle(i + random(-spread, spread), height - i + random(-spread, spread), 5)
    }
  }
}
```

Notes

This may sound complicated, but it is just a little bit of refactoring. The training data is called just once in setup(). This will become evident later as we don't want to be creating new data as we train.

! Remove the background(220) from the trainingData() function.

Figure A5.4


p5*
File
Edit
Sketch
Help
English
Hello, TheHappyCoder!
Auto-refresh
ml5.js
by TheHappyCoder
sketch.js
Saved: about 2 hours ago
Preview
{
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)
circle(i + random(-spread, spread), height - i + random(-spread, spread), 5)
}
}
}
Console
Clear

Sketch A5.5 adding the model

We are going to call our neural network `nn`. This may not sound very creative, but it makes sense. We then call the `ml5.js` library, and specifically, we are using the neural network. This is because we are building our own model rather than a pre-trained one.

```
let nn
const number = 30
const spread = 30

function setup()
{
  createCanvas(400, 400)
  background(220)
  nn = ml5.neuralNetwork()
  trainingData()
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
      circle(i + random(-spread, spread), height - i + random(-spread,
spread), 5)
    }
  }
}
```

Notes

Nothing is meant to happen; all we have done is call the neural network library ml5.js. This is why we need it in the index.html file.

Code Explanation

`let nn`	Naming the neural network model
`nn = ml5.neuralNetwork()`	Calling or assigning the ml5 neural network to the nn model name

Sketch A5.5 adding the model

We have initialised the model and now we have to give it some `options`. This is information the model needs. First off, we have to decide what kind of task we want it to do; in this case, it is a `regression` task because we want it to output a value `y` that varies based on the `x` input.

The second option is something called `debug:` what this does is draw the loss as a graph/chart so we can see the loss values visually. You don't have to use this, but it is handy to see what progress the model is making as it is training. To see the graph, you set `debug: true`; alternatively, set it to `false`.

We have the `options` object as a `const` so that we don't change any of the parameters. We then call the options inside ml5. As mentioned in the introduction, we have added `webgl` to the backend to help with the training process.

```
let nn
const number = 30
const spread = 30
const options = {
  task: 'regression',
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  background(220)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      fill(0, 0, 255)
```

```
      circle(i + random(-spread, spread), height - i + random(-spread,
spread), 5)
    }
  }
}
```

🗒️ Notes

Adding this code won't make any difference yet. If you get any errors, make sure you have typed everything in correctly and also that you have the ml5.js line of code in the index.html correctly written.

🌻 Challenge

If your machine will accept it, then try: `ml5.setBackend("cpu")` for faster training.

🛠️ Code Explanation

`const options`	This creates an object of options we can use with ml5.js
`ml5.setBackend("webgl")`	Using the rendering that is most appropriate
`nn = ml5.neuralNetwork(options)`	Adding the options to the neural network

Sketch A5.6 adding the data

What the model needs is the data; therefore, we need to change a number of things. We need to collect all those data points into the `data` variable. So the variable `data` now cycles through all the `x` and `y` values. The data is now an array of those `x` and `y` values. We then draw the circle to the `data.x`, which is `data[0]`, and `data.y` is `data[1]`.

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

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

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-spread, spread))]
      fill(0, 0, 255)
      circle(data[0], data[1], 5)
    }
  }
}
```

🗒️ Notes

We still draw the data points as before, but the code is looking quite different now. Try to follow what the code is doing.

🛠️ Code Explanation

`i + floor(random(-spread, spread))`	We use floor so that we get whole numbers (integers) not floats.
`circle(data[0], data[1], 5)`	The circle data[0] is the x value, the data[1] the y value.

Sketch A5.6 adding the data

We are going to add this data to the neural network. To do this, we use the `addData()` function. After the data has been added, we need to normalise the data with the `normalizeData()` function. This transforms the data to between `-1` and `+1`. The `console.log()` will let us know when that has been completed.

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

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  console.log('done')
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-
spread, spread))]
      fill(0, 0, 255)
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
```

```
  }
}
```

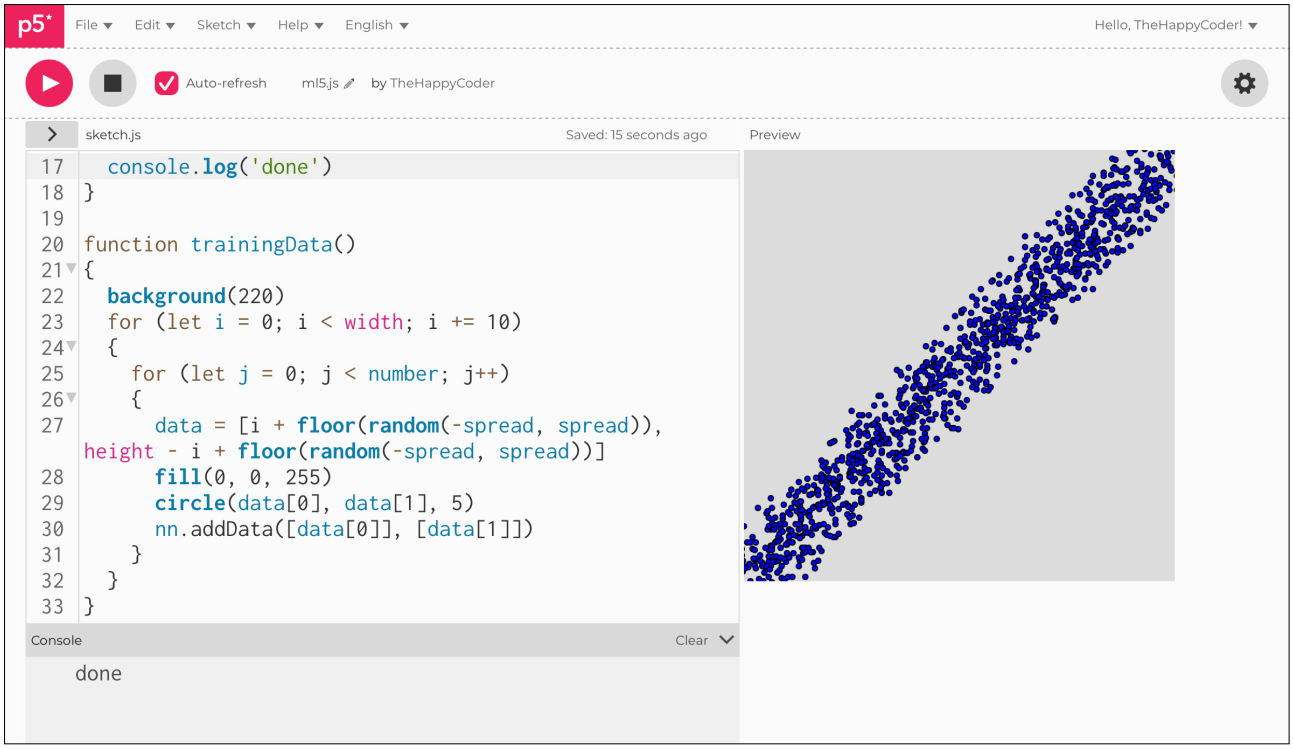
🗒️ Notes

We have passed the data into the model; next, we need to train the model on that data.

🛠️ Code Explanation

`nn.normalizeData()`	This normalises the data
`console.log('done')`	Let us know when it is all done
`nn.addData([data[0]], [data[1]])`	This adds the data to the nn model

Figure A5.6



Sketch A5.7 training on the data

To train the model, we use the `train()` function. We are going to train the model on the data we have normalised. We add a callback function in the `train()` function called `finishedTraining()` so that when the training is complete, it will let us know. Here, the loss graph kicks into action, and you will notice that it stops training after about ten epochs.

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

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  nn.train(finishedTraining)
  console.log('done')
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-
spread, spread))]
      fill(0, 0, 255)
      noStroke()
```

```
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}

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

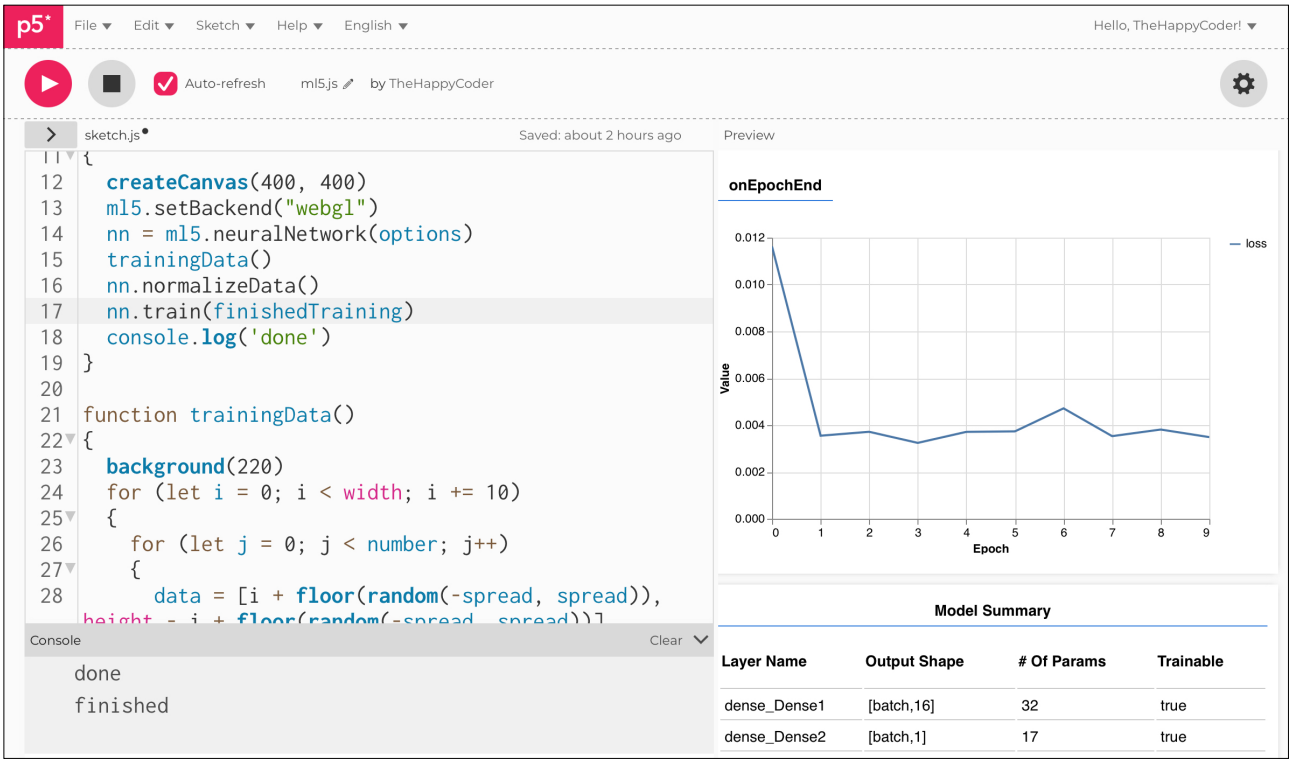
Notes

When the loss graph has finished, you can remove it by clicking on the Hide button, so that you can see the canvas (if your screen is small). In the console, we get done and finished after the training is completed. The default settings are for 10 epochs, and there are 16 nodes (neurons) in the hidden layer. It automatically works out the number of nodes for the input and output. Notice it makes reference to Dense 1 and Dense 2. This is the hidden layer and the output layer. The input layer is not considered a layer. Dense means fully connected.

Code Explanation

`nn.train(finishedTraining)`	Trains the model and when finished calls the function finishedTraining()

Figure A5.7



Sketch A5.8 predicting the result

We need to predict the result after the training is complete. We do this using the `predict()` function. This will be very telling as it is going to draw an approximation of the data. In theory, it should be a straight line where `y = x` drawn from the bottom left corner to the top right corner on the canvas, but we will see.

❗ Remove the `console.log()`s, they were there just to check it worked. We will set `debug:` to `false` so you can see the line being drawn (better still, just remove it completely).

In the `finishedTraining()` function, we use the `counter` as an array; this is the `x` value. The callback function `gotResults()` will draw the predicted result for `y` for each `x` input. We only want `400` because that is the number of pixels in the width of the canvas.

```
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()
  nn.train(finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
```

```
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-
spread, spread))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[1]])
    }
  }
}


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

🗒️ Notes

If you run this, you will get an error message because we haven't created the `gotResults()` function just yet. We will need to advance the counter in the `gotResults()` function by one on each iteration (see next sketch).

🛠️ Code Explanation

`nn.predict([counter], gotResults)`	This takes the x value (counter), predicts the result and passes to the callback function gotResults()

Sketch A5.9 the callback function

Next, we add the `gotResults()` callback function. The prediction is the first element in the array, which will be a value of `y` for a particular input of `x` (from the `counter`). We draw a point at that value and move onto the next. We add one to the `counter` and return to the `finishedTraining()` function for the next prediction.

```
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()
  nn.train(finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-
spread, spread))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
```

```
      nn.addData([data[0]], [data[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

The `results` is an argument that is an array of objects, actually one object at a time, for each `x` (`counter`) value. What you may get is a wonky line at the start and finish. This is most likely because we have some negative values when we create the variance of the data points.

🌻 Challenges

1. Use console.log(results) to see what it is returning (see second image below)
2. We could fill an array with values and work through each element one at a time

🛠️ Code Explanation

`let prediction = results[0]`	The prediction variable takes all the data in the first object in the results
`let x = counter`	Gives the counter value to the x co-ordinate of the line
`let y = prediction.value`	This takes the value part of the object element. This is the actual y value
`counter++`	Adding one to the counter
`finishedTraining()`	Then finally return to the finishedTraining() function to get the next value based on the next counter (x) value

Figure A5.9a

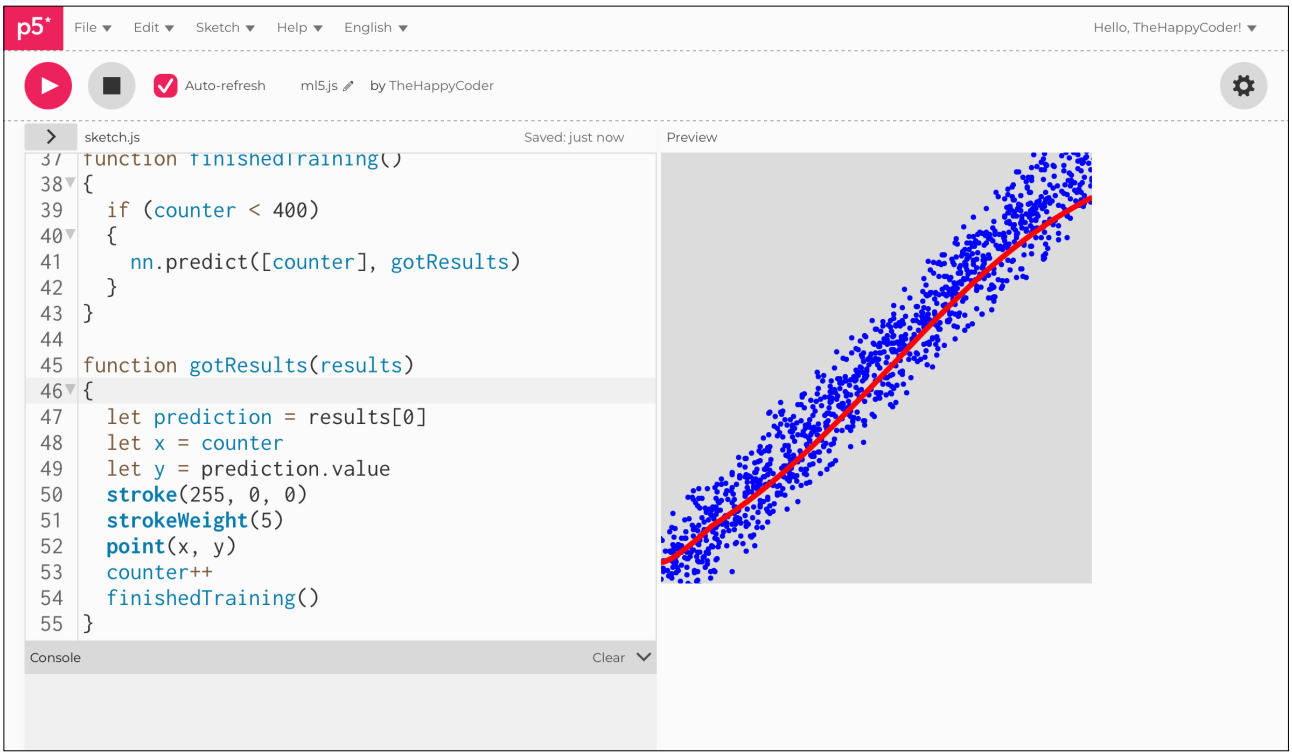
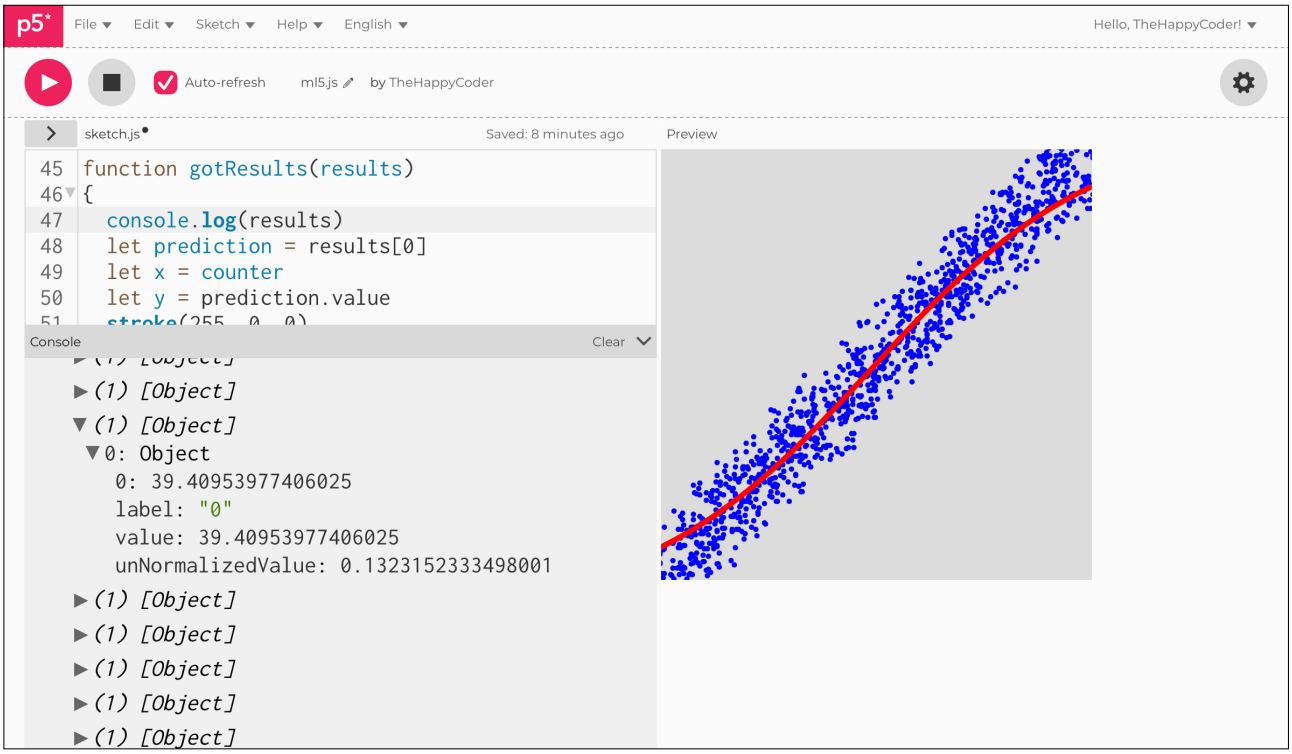


Figure A5.9b: with `console.log(results)` and the `value` is the actual predicted value for the canvas



Sketch A5.10 batch size

The `batch size` is how much data we send through in one go. We could send it through one at a time or all of it in one go. A large `batch size` is usually faster but potentially less accurate. Whereas a smaller `batch size` takes longer but may give overall better results. We are going to give the `train()` function some options, which is where we specify the `batch size`. Our dataset is around `2400` data points, but first we will try a `batch size` of `1` and see what happens.

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

function setup()
{
  createCanvas(400, 400)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 1
  }
  nn.train(trainingOptions, finishedTraining)
}

function trainingData()
{
  background(220)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
```

```
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-
spread, spread))]
      fill(0, 0, 255)
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[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

The results speak for themselves. We need to consider a larger `batchSize` than 1. We could go to the other extreme (all the dataset) and see what happens then.

🌻 Challenge

Try other values of batch size

🛠️ Code Explanation

`const trainingOptions = {`	Create a permanent object for our training options
`batchSize: 1`	Specify the batch size
`nn.train(trainingOptions, finishedTraining)`	Add the training options to the train() function

Figure A5.10a Our loss chart looks woeful

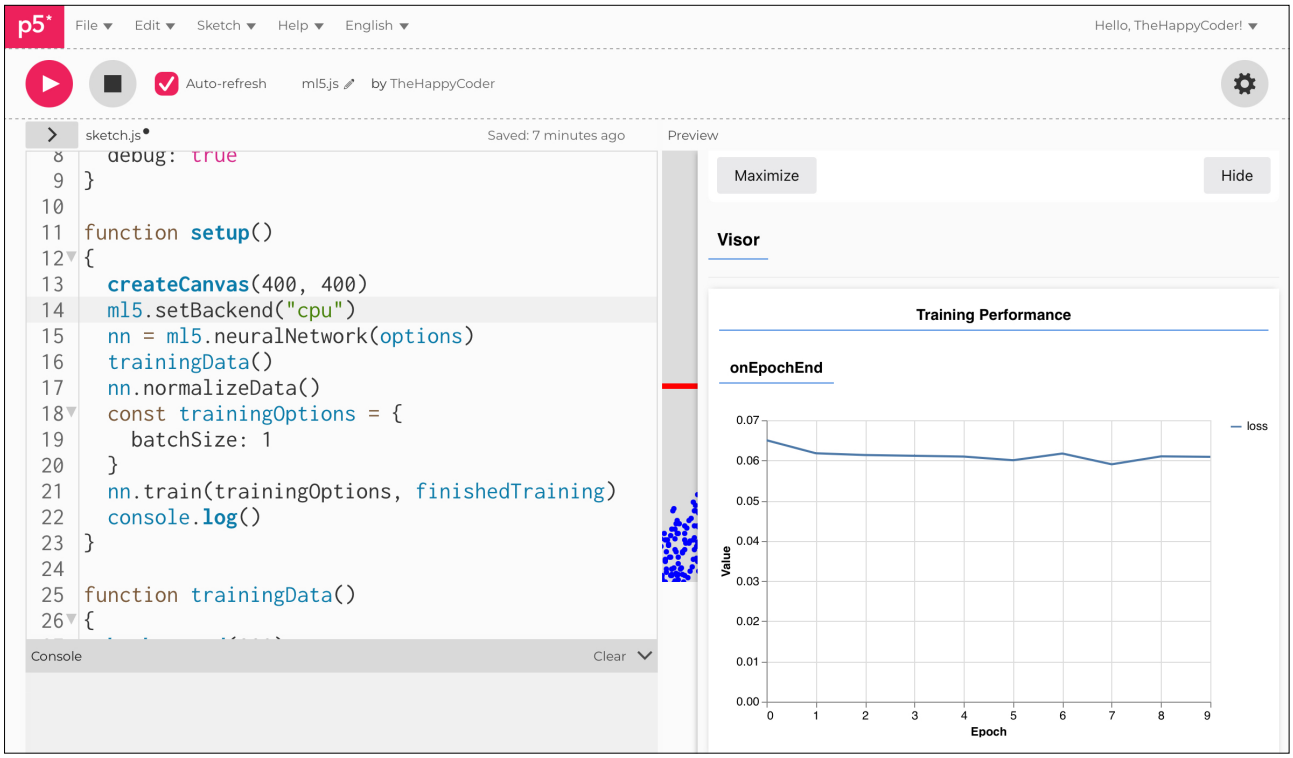
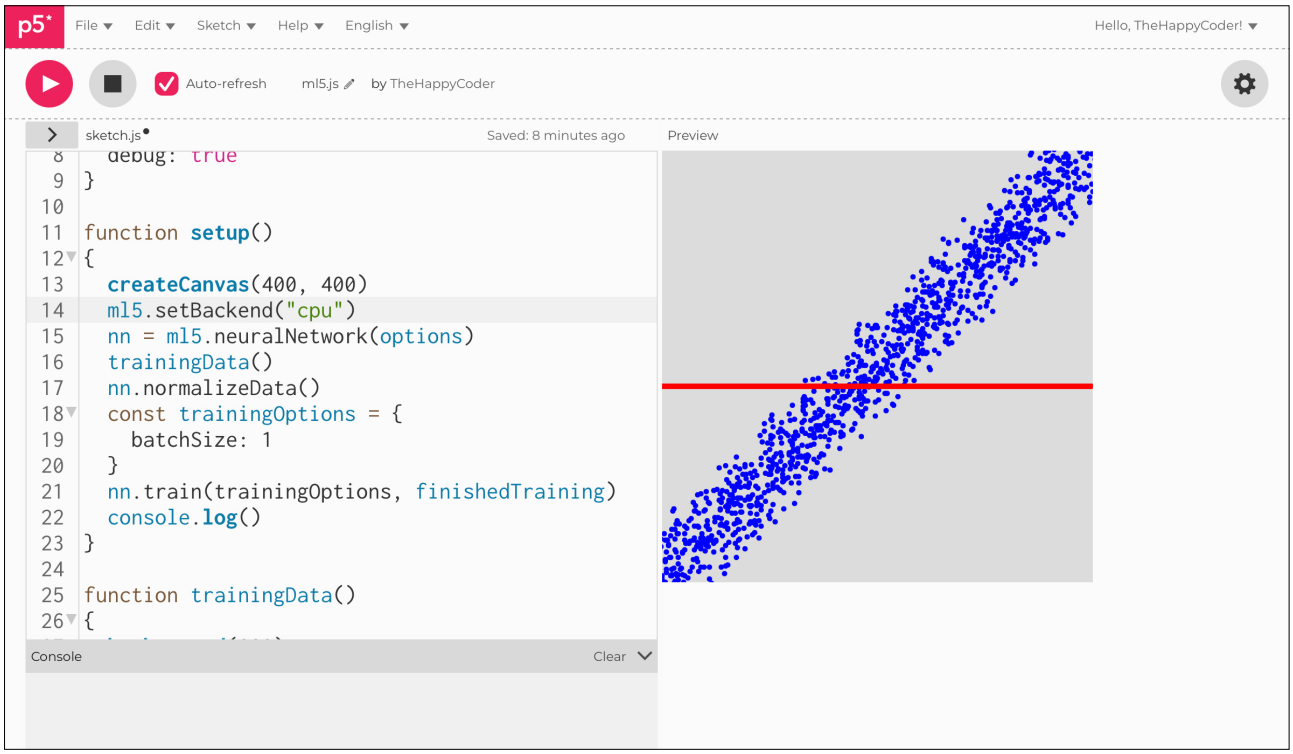


Figure A5.10b the results look woeful also



Sketch A5.11 increasing the batch size

Batch sizes are often in the order 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and so on. You can have any batch size you want, though. You often notice the difference instantly.

❗ Keep this code for the next unit

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

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)
  for (let i = 0; i < width; i += 10)
  {
    for (let j = 0; j < number; j++)
    {
      data = [i + floor(random(-spread, spread)), height - i + floor(random(-
spread, spread))]
      fill(0, 0, 255)
```

```
      noStroke()
      circle(data[0], data[1], 5)
      nn.addData([data[0]], [data[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

We get a better loss curve and better results, still a bit skewed at the ends. The results are far more consistent.

❗ Remember to keep this code for: `module A unit #3 sine wave regression`

🌻 Challenge

Try other batch sizes

Figure A5.11a loss much better

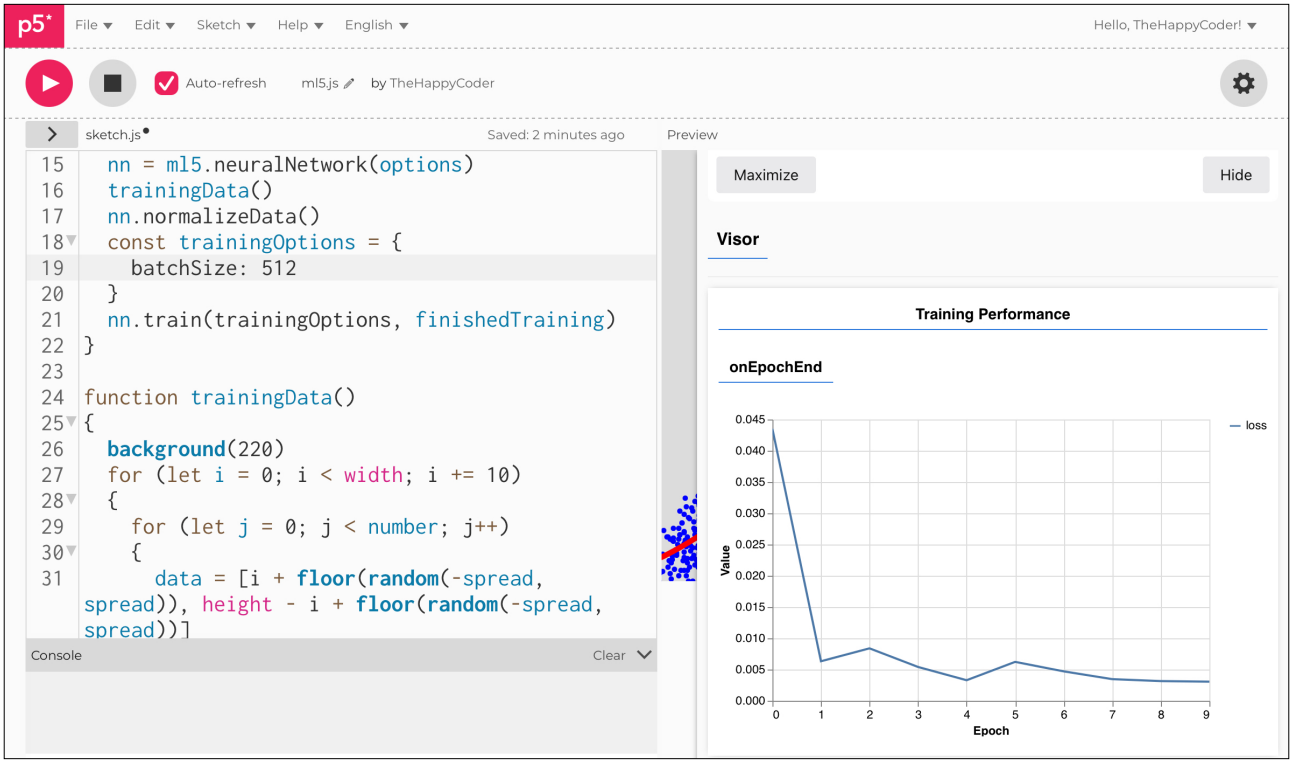


Figure A5.11b and better results