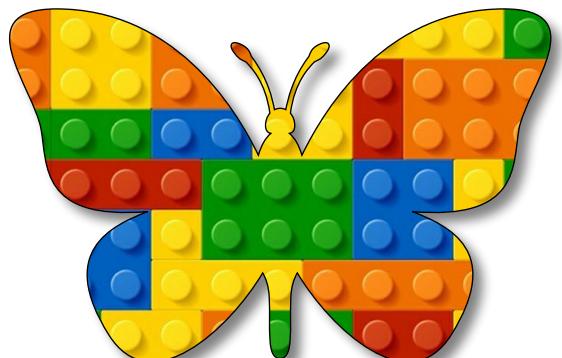


Artificial Intelligence

Module A

Unit #3

sine wave regression





Module A Unit #3 sine wave regression

Introduction to sine wave regression

The index.html file

Sketch A3.1 sine wave sketch

Sketch A3.2 epochs

Sketch A3.3 layers

Sketch A3.4 more hidden layers

Sketch A3.5 more nodes

Sketch A3.6 learning rate

Sketch A3.7 always a bit of trial and error

Sketch A3.8 changing the activation function



Introduction to sine wave regression with ml5.js

In the previous unit we trained a model to predict a straight line, and had a peek at the batch size hyperparameter. This time we are going to make things a bit more challenging. We are going to upgrade from a straight line to a sine wave.

This will give us a chance to look at some more hyperparameters and their impact on the training of the model. We will use them to try to make better predictions of a sine wave.

The hyperparameters we will be looking at will be

- [_] Number of hidden layers
- [_] Number of nodes in each layer
- [_] Activation functions
- [_] Epochs
- [_] Learning Rate



The index.html

Make sure that you have the ml5.js line of code in your index.html file

```
<!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 A3.1 sine wave sketch

We are starting with the sketch from linear regression. Then we'll modify the data because instead of a line we want a sine wave. In p5.js the default angle units are radians, but we are going to work in degrees which is a little more intuitive. To make this change, we use the function `angleMode()` in `setup()`. Everything else in the sketch remains the same.

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

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  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)), 200 + (50 * sin(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

For every **x** input we get the sine of that input as our **y** output. We multiply by **50** to increase the amplitude (otherwise a very flat sine wave) and move it by **200** pixels so we have it in the centre of the canvas, plus the usual variance, **random(spread)**. The results aren't as bad as you might expect as it uses default settings for most of the hyperparameters except for the batch size.

Code Explanation

angleMode(DEGREES)	This changes the default to degrees (notice all capitals)
sin(i)	Returns the sine of i

Figure A3.1a loss chart

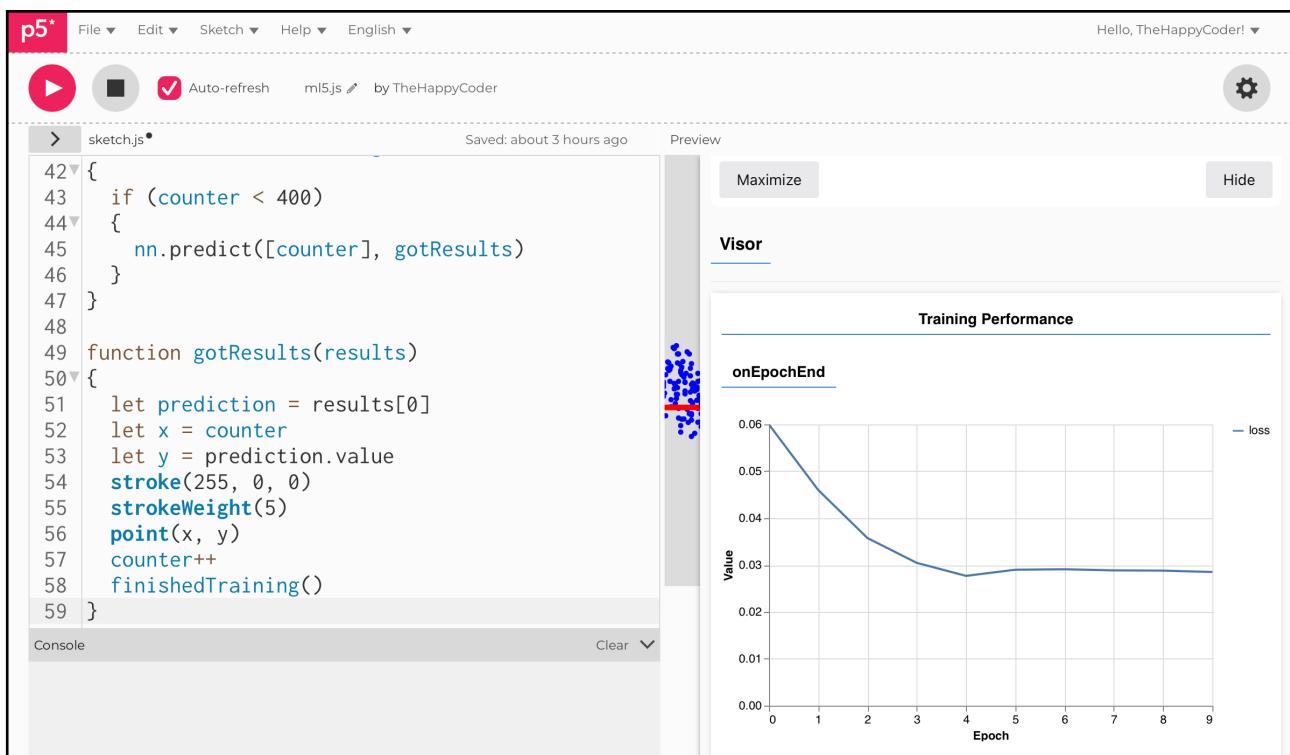
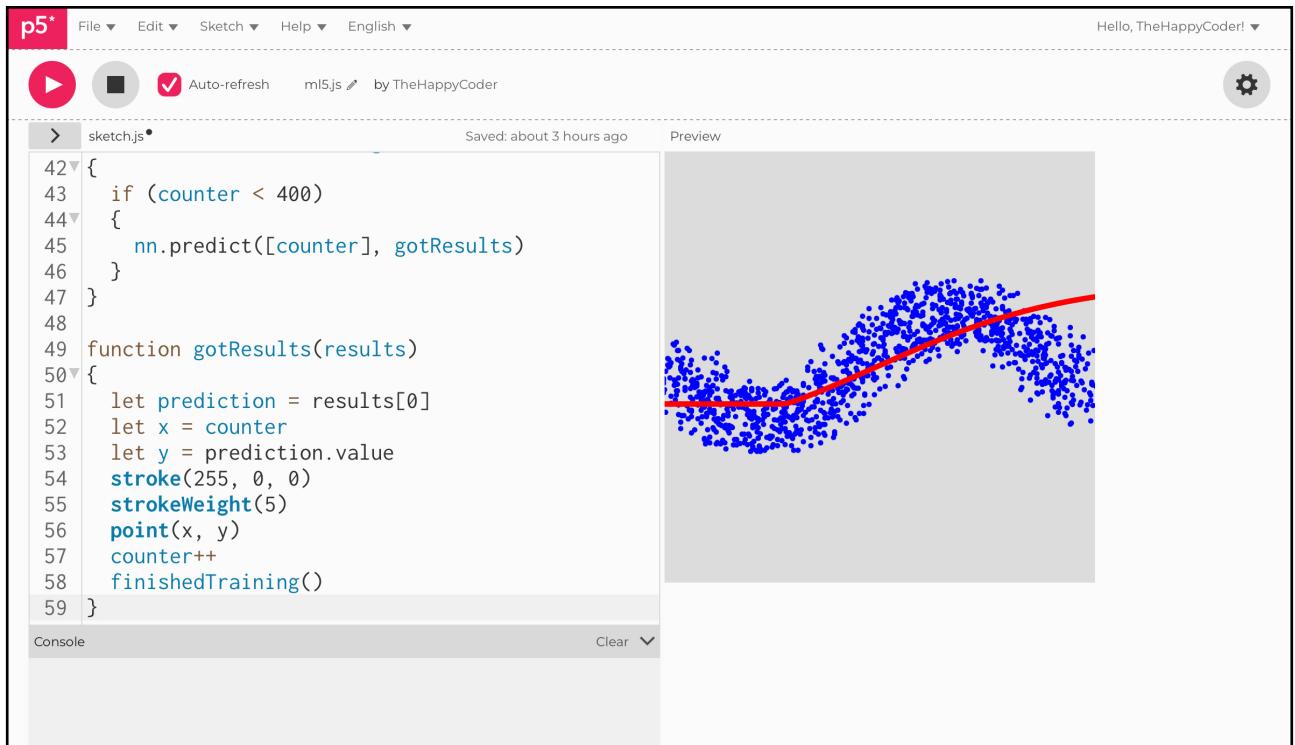


Figure A3.1b predicted results



The screenshot shows the p5.js code editor interface. The top bar includes the p5 logo, file navigation, and a preview button labeled "Preview". The code editor window contains a script named "sketch.js" with the following content:

```
sketch.js
42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
```

```
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()
}
```

The preview area displays a scatter plot of blue points forming a curve, with a thick red line drawn through it, representing a predicted model fit.



Sketch A3.2 epochs

Our first **hyperparameter** change will be the number of **epochs**. Each epoch is one complete dataset. Currently we are only running **10 epochs** and it looks as if the loss might be still going down after that. Let's extend it to **100 epochs** and see where it levels off (or stops learning). We can introduce this **hyperparameter** into the training options just like we did with the batch sizes.

! You will need to put a **comma (,**) after the **512**.

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

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512,
    epochs:100
  }
  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)), 200 + (50 * sin(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 changes were not a great improvement. You may get slightly different results depending on how the weights are initialised in the model but it seems to level out after around 10 epochs.

Challenge

Try again with different epoch settings.

Code Explanation

epochs:100	Defines how many epochs if not using the default value
------------	--

Figure A3.2

The screenshot shows the p5.js IDE interface. On the left, the code editor displays a sketch named 'sketch.js' containing JavaScript code for setting up a regression task using ml5.js. The code includes creating a canvas, setting angle mode to degrees, initializing ml5's neural network with specific options, and defining training data and training options. It also includes a function to train the neural network. On the right, the 'Visor' panel is open, showing a 'Training Performance' visualization. This visualization features a scatter plot of blue dots representing data points and a line graph showing the 'loss' value over 100 epochs. The loss starts at approximately 0.085 and drops sharply to around 0.025 by epoch 10, remaining relatively stable with minor fluctuations until epoch 100.

```
> sketch.js
1 const options = {
2   task: 'regression',
3   debug: true
4 }
5
6 function setup()
7 {
8   createCanvas(400, 400)
9   angleMode(DEGREES)
10  ml5.setBackend("webgl")
11  nn = ml5.neuralNetwork(options)
12  trainingData()
13  nn.normalizeData()
14  const trainingOptions = {
15    batchSize: 512,
16    epochs: 100
17  }
18  nn.train(trainingOptions, finishedTraining)
19}
20
21
22
23
```

Maximize Hide

Visor

Training Performance

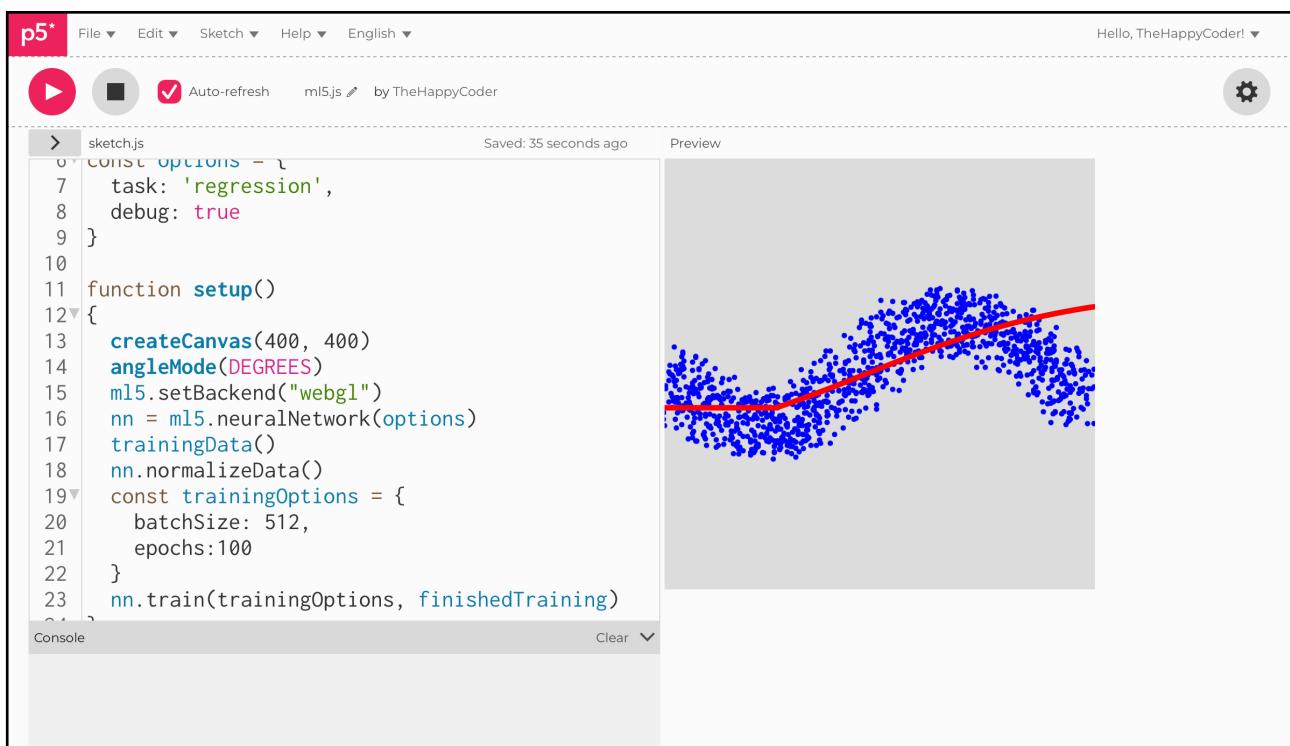
onEpochEnd

Value

loss

Epoch

Figure A3.2



The screenshot shows the p5.js IDE interface. The code editor on the left contains a script named 'sketch.js' with the following content:

```
sketch.js
1 const options = {
2   task: 'regression',
3   debug: true
4 }
5
6 function setup()
7 {
8   createCanvas(400, 400)
9   angleMode(DEGREES)
10  ml5.setBackend("webgl")
11  nn = ml5.neuralNetwork(options)
12  trainingData()
13  nn.normalizeData()
14  const trainingOptions = {
15    batchSize: 512,
16    epochs: 100
17  }
18  nn.train(trainingOptions, finishedTraining)
19}
20
21 function finishedTraining(error) {
22  if (error) {
23    console.error(error)
24  } else {
25    // Training finished successfully
26  }
27}
```

The preview window on the right displays a scatter plot of blue data points forming a curved pattern. A red line represents a linear regression fit through the data.



Sketch A3.3 layers

We can add more layers, more accurately we can add more hidden layers. In those layers we can specify the number of nodes and the activation function we want to use. The default settings are shown below:

Input Layer

One input node

Output Layer

One output node

Hidden Layer

By default we have one hidden layer with 16 nodes

Dense

Means all the nodes in one layer are fully connected to the node in the previous layer.

Activation Functions

In the hidden layer they are the **ReLU** function and for the output layer it is the **sigmoid** function

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  layers: [
    {
      type: 'dense',
      units: 16,
      activation: 'relu'
```

```

    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512,
    epochs:100
  }
  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)), 200 + (50 *
sin(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 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

As you can see it has made no difference, which is really as expected. If you get a better (or worse) result just remember that it is starting each time with different (random) weights as well as random data.

Challenge

You might want to start changing the number of nodes in the hidden layer

Code Explanation

layers: [...]	Adding layers in the training option
type: 'dense'	Fully connected nodes between each layer
units: 16	16 nodes (neurons) in the hidden layer
activation: 'relu'	Hidden layer activation function
activation: 'sigmoid'	Output layer activation function

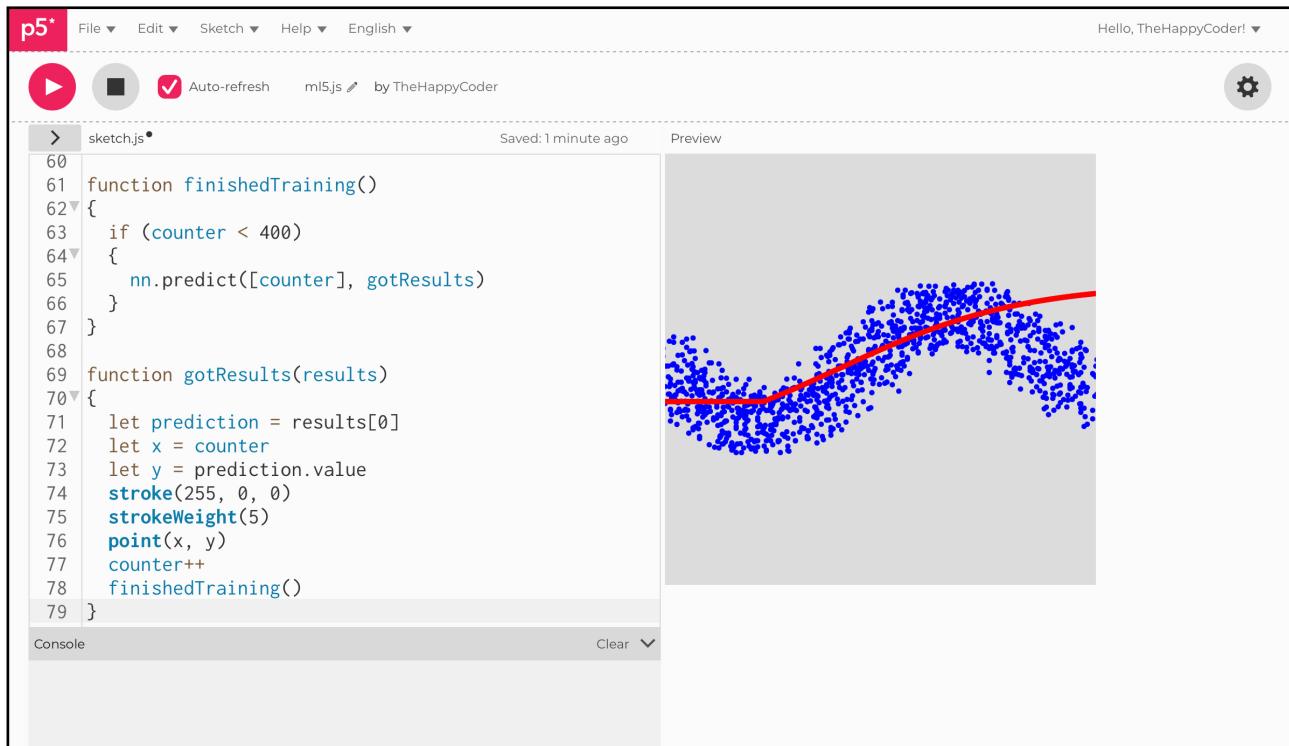
Figure A3.3a

The screenshot shows the p5.js IDE interface. On the left, the code editor displays `sketch.js` with the following content:

```
60
61 function finishedTraining()
62 {
63   if (counter < 400)
64   {
65     nn.predict([counter], gotResults)
66   }
67 }
68
69 function gotResults(results)
70 {
71   let prediction = results[0]
72   let x = counter
73   let y = prediction.value
74   stroke(255, 0, 0)
75   strokeWeight(5)
76   point(x, y)
77   counter++
78   finishedTraining()
79 }
```

The right side of the interface features a "Visor" panel titled "Training Performance". It contains a line graph titled "onEpochEnd" showing the "loss" value over 100 epochs. The y-axis is labeled "Value" and ranges from 0.00 to 0.08. The x-axis is labeled "Epoch" and ranges from 0 to 100. The loss starts at approximately 0.08 and drops sharply to around 0.03 by epoch 10, remaining relatively stable with minor fluctuations until epoch 100.

Figure A3.3b



The screenshot shows the p5.js IDE interface. The top bar includes 'File ▾', 'Edit ▾', 'Sketch ▾', 'Help ▾', 'English ▾', 'Hello, TheHappyCoder! ▾', and a gear icon. Below the menu is a toolbar with a play button, a square, a refresh icon with 'Auto-refresh' checked, and 'ml5.js' by 'TheHappyCoder'. The code editor on the left displays 'sketch.js' with the following code:

```
60
61 function finishedTraining()
62{
63  if (counter < 400)
64  {
65    nn.predict([counter], gotResults)
66  }
67}
68
69 function gotResults(results)
70{
71  let prediction = results[0]
72  let x = counter
73  let y = prediction.value
74  stroke(255, 0, 0)
75  strokeWeight(5)
76  point(x, y)
77  counter++
78  finishedTraining()
79}
```

The preview window on the right shows a scatter plot of blue points forming a curve, with a red line representing a linear regression fit.



Sketch A3.4 more hidden layers

Added two more identical hidden layers

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  layers: [
    {
      type: 'dense',
      units: 16,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 16,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 16,
      activation: 'relu'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
```

```

}

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512,
    epochs: 100
  }
  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)), 200 + (50 * sin(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

You can see in my example the beginnings of a sine wave but far from what we might want.

Figure A3.4a

The screenshot shows the p5.js IDE interface. On the left, the code editor displays `sketch.js` with the following content:

```
63
64 function finishedTraining()
65 {
66   if (counter < 400)
67   {
68     nn.predict([counter], gotResults)
69   }
70 }
71
72 function gotResults(results)
73 {
74   let prediction = results[0]
75   let x = counter
76   let y = prediction.value
77   stroke(255, 0, 0)
78   strokeWeight(5)
79   point(x, y)
80   counter++
81   finishedTraining()
82 }
```

The right side of the interface features a "Visor" panel titled "Training Performance". It contains a line graph titled "onEpochEnd" showing the "loss" value over 100 epochs. The y-axis is labeled "Value" and ranges from 0.00 to 0.08. The x-axis is labeled "Epoch" and ranges from 0 to 100. The loss starts at approximately 0.075, drops sharply to around 0.025 by epoch 10, and then fluctuates between 0.02 and 0.025 for the remainder of the training.

Figure A3.4b

The screenshot shows the p5.js IDE interface. The top bar includes the p5 logo, File, Edit, Sketch, Help, English, and a user profile. The main area has tabs for sketch.js and ml5.js, with sketch.js selected. The code editor contains the following script:

```
sketch.js
63
64 function finishedTraining()
65 {
66   if (counter < 400)
67   {
68     nn.predict([counter], gotResults)
69   }
70 }
71
72 function gotResults(results)
73 {
74   let prediction = results[0]
75   let x = counter
76   let y = prediction.value
77   stroke(255, 0, 0)
78   strokeWeight(5)
79   point(x, y)
80   counter++
81   finishedTraining()
82 }
```

The preview window shows a scatter plot of blue points forming a bell-shaped curve. A red line with a stroke weight of 5 is drawn across the middle of the curve, representing the predicted output of the neural network. The status bar at the bottom indicates "Saved: 35 seconds ago".



Sketch A3.5 more nodes

Increasing the number of nodes, they don't have to be equal for each hidden layer, it is worth trying mixing it up a bit to see what happens.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  layers: [
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
}
```

```

    debug: true
}

function setup()
{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512,
    epochs: 100
  }
  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)), 200 + (50 * sin(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

You can see the pattern emerging but still less than satisfactory.

Challenge

Think about more/fewer nodes, more/fewer layers

Figure A3.5a

The screenshot shows the p5.js IDE interface. On the left, the code editor displays `sketch.js` with the following content:

```
64
65 function finishedTraining()
66 {
67   if (counter < 400)
68   {
69     nn.predict([counter], gotResults)
70   }
71 }
72
73 function gotResults(results)
74 {
75   let prediction = results[0]
76   let x = counter
77   let y = prediction.value
78   stroke(255, 0, 0)
79   strokeWeight(5)
80   point(x, y)
81   counter++
82   finishedTraining()
83 }
```

The right side of the interface shows a preview window with a scatter plot titled "Visor". Below it is a chart titled "Training Performance" showing "onEpochEnd" data. The chart has "Value" on the y-axis (ranging from 0.00 to 0.18) and "Epoch" on the x-axis (ranging from 0 to 100). A blue line represents the "loss" metric, which starts at approximately 0.18 and quickly drops to around 0.02, remaining relatively stable thereafter.

Figure A3.5b

The screenshot shows the p5.js code editor interface. The top bar includes the p5 logo, file navigation, and a preview button. The preview window shows a scatter plot of blue points forming a parabolic shape, with a red line representing a fitted curve. The code editor displays the following JavaScript code:

```
sketch.js
64
65 function finishedTraining()
66 {
67   if (counter < 400)
68   {
69     nn.predict([counter], gotResults)
70   }
71 }
72
73 function gotResults(results)
74 {
75   let prediction = results[0]
76   let x = counter
77   let y = prediction.value
78   stroke(255, 0, 0)
79   strokeWeight(5)
80   point(x, y)
81   counter++
82   finishedTraining()
83 }
```

The code uses a neural network (nn) to predict values for a counter variable, drawing a red line through the resulting points.



Sketch A3.6 learning rate

Another **hyperparameter** we can change is the **learning rate**. The **learningRate** are the size of steps the algorithm takes to find the lowest point in the calculations. If the steps are too big they can miss the global minimum, too small and it can never find the minimum and gets stuck on a local minimum. We are keeping the number of layers and nodes the same as the previous sketch.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  learningRate: 0.1,
  layers: [
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 64,
      activation: 'relu'
    },
    {
  }
```

```

        type: 'dense',
        activation: 'sigmoid'
    }
],
debug: true
}

function setup()
{
    createCanvas(400, 400)
    angleMode(DEGREES)
    ml5.setBackend("webgl")
    nn = ml5.neuralNetwork(options)
    trainingData()
    nn.normalizeData()
    const trainingOptions = {
        batchSize: 512,
        epochs:100
    }
    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)), 200 + (50 * sin(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

Not a significant difference but the loss function is very jumpy

Challenge

I could fiddle with these values endlessly and that is why Machine Learning is an art as much as a science. I will let you play around perhaps trying learning rates of **1** or **0.001**, more nodes but fewer hidden layers.

Code Explanation

learningRate: 0.1

We can specify the learning rate, the steps (or jumps) in the training process

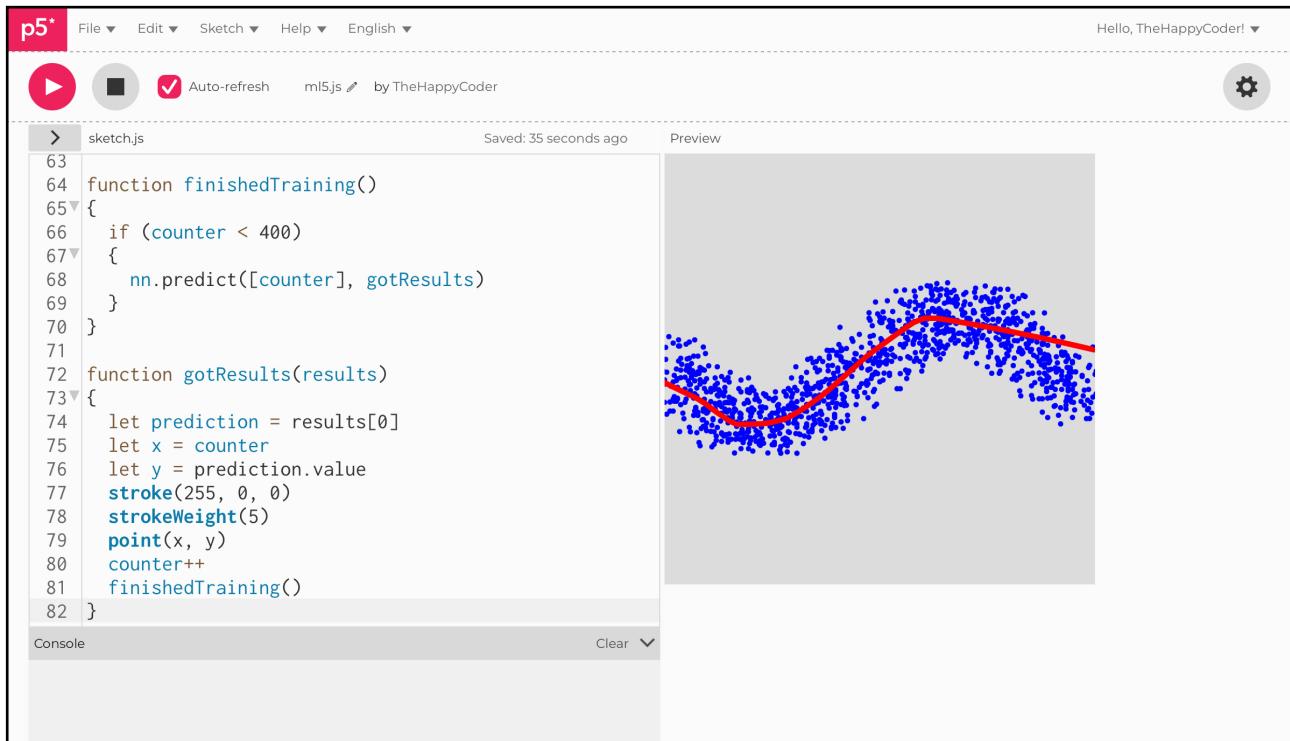
Figure A3.6a

The screenshot shows the p5.js IDE interface. The top bar includes 'File ▾', 'Edit ▾', 'Sketch ▾', 'Help ▾', 'English ▾', and a user profile 'Hello, TheHappyCoder! ▾'. The main area has tabs for 'sketch.js' and 'ml5.js' by 'TheHappyCoder'. The code editor contains the following script:

```
sketch.js
63
64 function finishedTraining()
65{
66  if (counter < 400)
67  {
68    nn.predict([counter], gotResults)
69  }
70}
71
72 function gotResults(results)
73{
74  let prediction = results[0]
75  let x = counter
76  let y = prediction.value
77  stroke(255, 0, 0)
78  strokeWeight(5)
79  point(x, y)
80  counter++
81  finishedTraining()
82}
```

The right side of the interface features a 'Visor' panel titled 'Training Performance' with a chart titled 'onEpochEnd'. The chart plots 'Value' (y-axis, 0.00 to 0.08) against 'Epoch' (x-axis, 0 to 100). A single blue line represents the 'loss' metric, which starts at approximately 0.08 and rapidly decreases to stabilize around 0.02 after epoch 20.

Figure A3.6b



The screenshot shows the p5.js IDE interface. The top bar includes the p5 logo, file navigation (File, Edit, Sketch, Help, English), and a user profile (Hello, TheHappyCoder!). Below the menu is a toolbar with play, stop, and refresh buttons, and an auto-refresh checkbox. The main area has tabs for 'sketch.js' and 'Preview'. The code editor contains the following JavaScript code:

```
63
64 function finishedTraining()
65{
66  if (counter < 400)
67  {
68    nn.predict([counter], gotResults)
69  }
70}
71
72 function gotResults(results)
73{
74  let prediction = results[0]
75  let x = counter
76  let y = prediction.value
77  stroke(255, 0, 0)
78  strokeWeight(5)
79  point(x, y)
80  counter++
81  finishedTraining()
82}
```

The preview window shows a scatter plot of blue points forming a curve, with a thick red line representing a fitted curve.



Sketch A3.7 always a bit of trial and error

I have increased the number of nodes but reduced the number of layers, as well as reducing the learning rate to **0.01**.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  learningRate: 0.01,
  layers: [
    {
      type: 'dense',
      units: 256,
      activation: 'relu'
    },
    {
      type: 'dense',
      units: 256,
      activation: 'relu'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}

function setup()
```

```

{
  createCanvas(400, 400)
  angleMode(DEGREES)
  ml5.setBackend("webgl")
  nn = ml5.neuralNetwork(options)
  trainingData()
  nn.normalizeData()
  const trainingOptions = {
    batchSize: 512,
    epochs: 100
  }
  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)), 200 + (50 * sin(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()
}
```

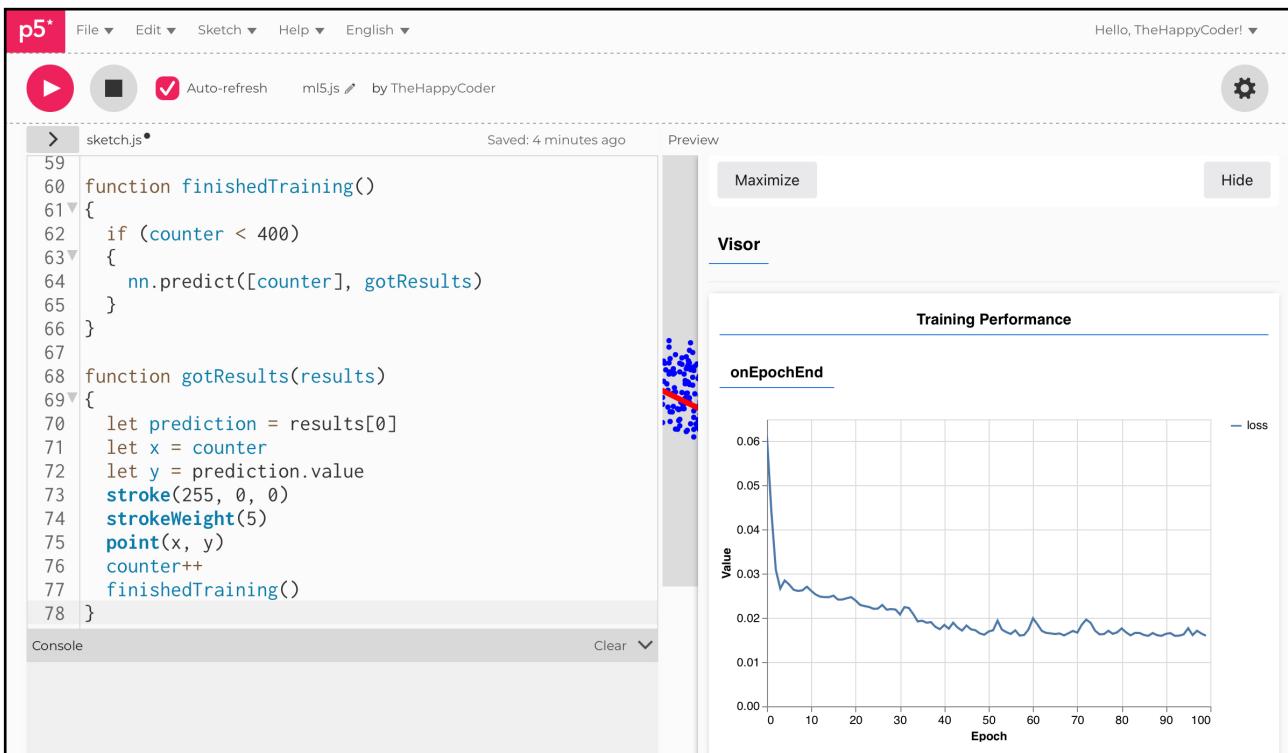
Notes

Starting to see some improvements. Bear in mind however that every time you run the code we are using different data (not saved or uploaded) but this is for ease of use and demonstration purposes.

Challenge

Play around with the number of **nodes** and **learning rate**, consider saving the data in the first place and then experimenting with the hyperparameters.

Figure A3.7a

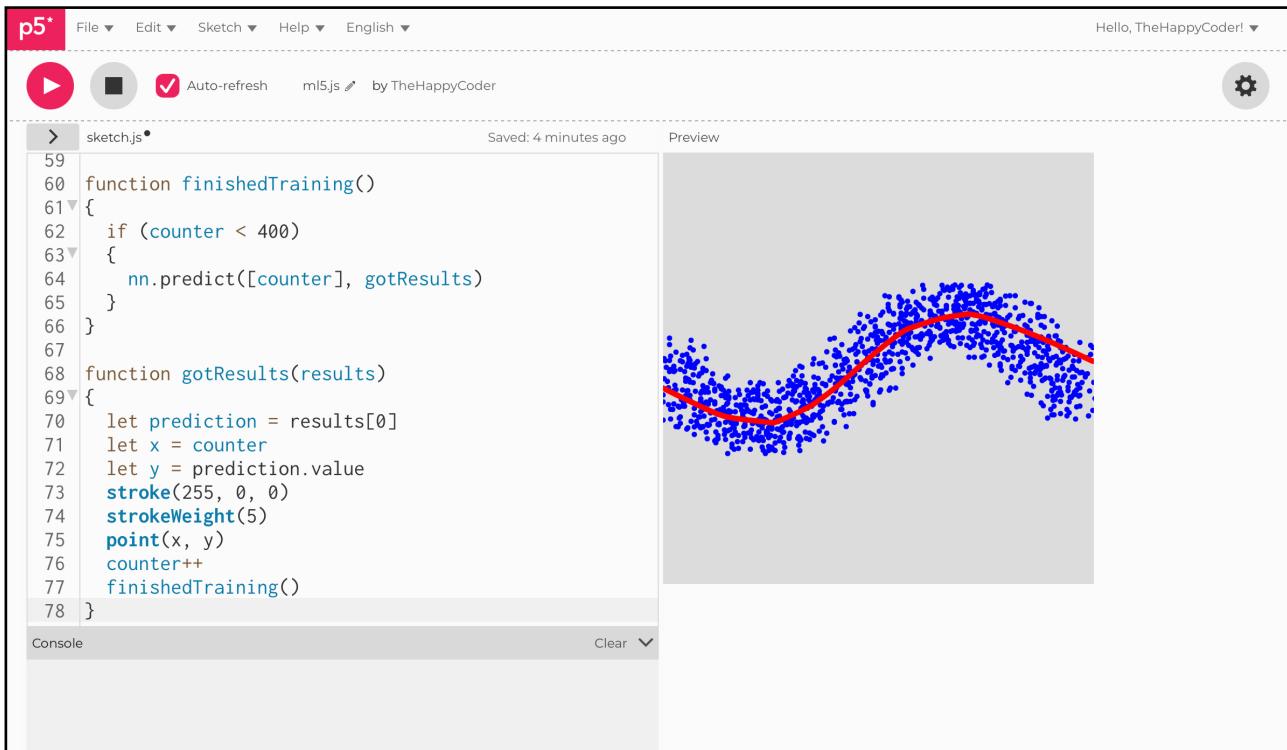


The screenshot shows the p5.js IDE interface. On the left, the code editor displays `sketch.js` with the following content:

```
59
60 function finishedTraining()
61 {
62   if (counter < 400)
63   {
64     nn.predict([counter], gotResults)
65   }
66 }
67
68 function gotResults(results)
69 {
70   let prediction = results[0]
71   let x = counter
72   let y = prediction.value
73   stroke(255, 0, 0)
74   strokeWeight(5)
75   point(x, y)
76   counter++
77   finishedTraining()
78 }
```

The right side of the interface features a "Visor" panel titled "Training Performance". It contains a chart titled "onEpochEnd" showing the "loss" value over 100 epochs. The y-axis is labeled "Value" and ranges from 0.00 to 0.06. The x-axis is labeled "Epoch" and ranges from 0 to 100. The loss starts at approximately 0.06 and drops rapidly, stabilizing around 0.015 after epoch 20.

Figure A3.7b



The screenshot shows the p5.js IDE interface. The top bar includes the p5 logo, file navigation (File, Edit, Sketch, Help, English), and a user profile (Hello, TheHappyCoder!). Below the bar, there are controls for play/pause, auto-refresh (checked), and preview. The left panel displays the code for 'sketch.js':

```
59
60 function finishedTraining()
61 {
62   if (counter < 400)
63   {
64     nn.predict([counter], gotResults)
65   }
66 }
67
68 function gotResults(results)
69 {
70   let prediction = results[0]
71   let x = counter
72   let y = prediction.value
73   stroke(255, 0, 0)
74   strokeWeight(5)
75   point(x, y)
76   counter++
77   finishedTraining()
78 }
```

The right panel shows a 'Preview' window containing a scatter plot of blue points forming a curve, with a red line representing a fitted model.



Activation functions

The final hyperparameter we are going to look at is the activation functions. The default for a regression task is hidden layer: **ReLU** and output layer: **Sigmoid**

Just to save on space I will not change any other lines of code except the options layers, so the rest of the code remains the same and is omitted for brevity.



Sketch A3.8 changing the activation function

Using the same **hyperparameters**, we will switch the **ReLU** to **sigmoid**.

```
let nn
let counter = 0
let data
const number = 30
const spread = 30
const options = {
  task: 'regression',
  learningRate: 0.01,
  layers: [
    {
      type: 'dense',
      units: 256,
      activation: 'sigmoid'
    },
    {
      type: 'dense',
      units: 256,
      activation: 'sigmoid'
    },
    {
      type: 'dense',
      activation: 'sigmoid'
    }
  ],
  debug: true
}

function setup()
{
```

```

createCanvas(400, 400)
angleMode(DEGREES)
ml5.setBackend("webgl")
nn = ml5.neuralNetwork(options)
trainingData()
nn.normalizeData()
const trainingOptions = {
  batchSize: 512,
  epochs: 100
}
nn.train(trainingOptions, finishedTraining)
console.log()
}

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)), 200 + (50 * sin(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

Oh, dear! **ReLU** gives consistently better results. It is more efficient and is the industry standard. You could see if you can get better results with fewer/more layers and nodes, I will leave that with you to experiment.

Challenge

Use **tanh** instead of **sigmoid**

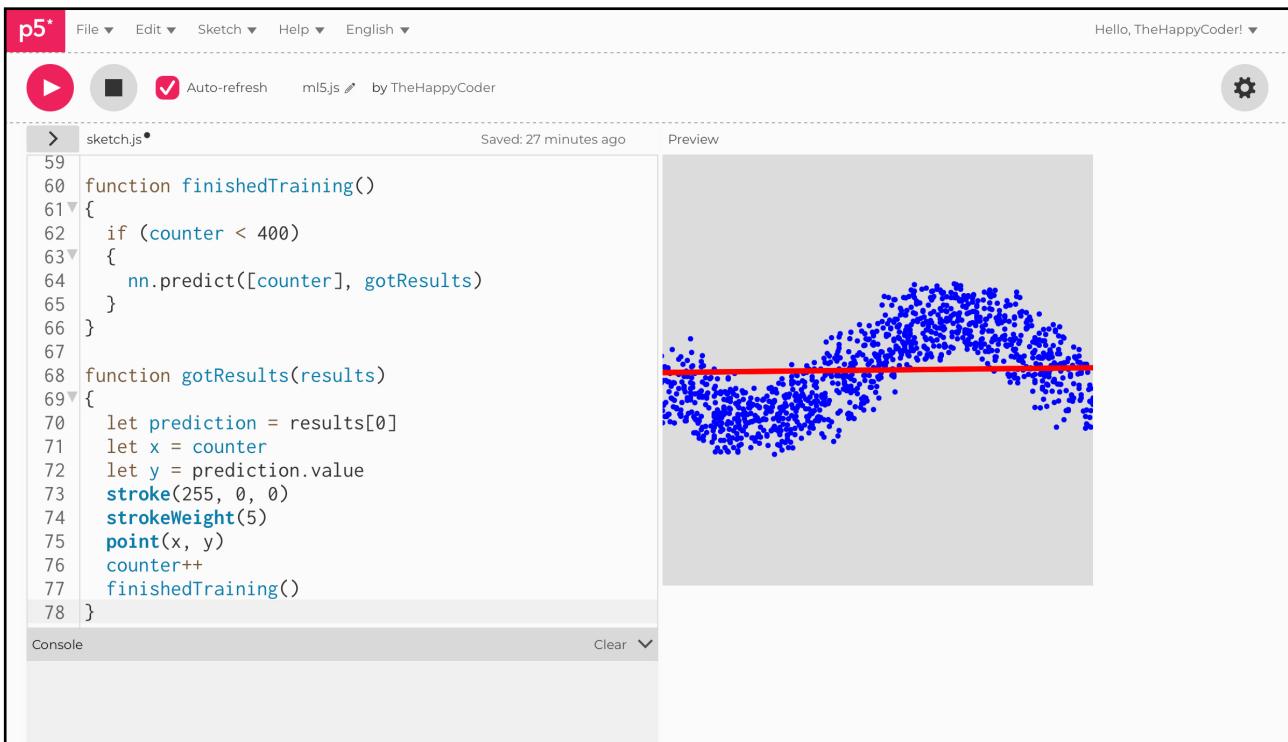
Figure A3.8a

The screenshot shows the p5.js IDE interface. On the left, the code editor displays `sketch.js` with the following content:

```
59
60 function finishedTraining()
61 {
62   if (counter < 400)
63   {
64     nn.predict([counter], gotResults)
65   }
66 }
67
68 function gotResults(results)
69 {
70   let prediction = results[0]
71   let x = counter
72   let y = prediction.value
73   stroke(255, 0, 0)
74   strokeWeight(5)
75   point(x, y)
76   counter++
77   finishedTraining()
78 }
```

The right side of the interface features a "Visor" panel titled "Training Performance". It contains a line graph titled "onEpochEnd" showing the "loss" value over "Epoch". The x-axis ranges from 0 to 100, and the y-axis ranges from 0.00 to 0.35. The graph shows a sharp initial drop from approximately 0.30 to 0.08, followed by a plateau around 0.08 until epoch 50, where it drops sharply again to stabilize around 0.06.

Figure A3.8b



The screenshot shows the p5.js IDE interface. The top bar includes the p5 logo, file navigation (File, Edit, Sketch, Help, English), and a user profile (Hello, TheHappyCoder!). The main area has tabs for 'sketch.js' (selected) and 'ml5.js' by TheHappyCoder. The code editor contains the following JavaScript code:

```
59
60 function finishedTraining()
61 {
62   if (counter < 400)
63   {
64     nn.predict([counter], gotResults)
65   }
66 }
67
68 function gotResults(results)
69 {
70   let prediction = results[0]
71   let x = counter
72   let y = prediction.value
73   stroke(255, 0, 0)
74   strokeWeight(5)
75   point(x, y)
76   counter++
77   finishedTraining()
78 }
```

The preview window shows a scatter plot of blue points forming a curve, with a single red horizontal line drawn through it, representing a linear regression fit.



Final challenges

We have looked at the following **hyperparameters**:

- [_] batch size
- [_] epochs
- [_] hidden layers
- [_] nodes
- [_] activation functions
- [_] learning rate

I would strongly recommend playing with all of these and seeing how they impact on the model's learning. Remember that you will never get a perfect result. Also try drawing the sine wave as a single line rather than a scatter graph approach as we did with the linear regression.



Summary

You will probably start to realise that throwing more nodes and hidden layers at the dataset doesn't necessarily improve the outcome, you may also realise having a smaller learning rate, smaller batch size may give you some slightly improved outcome but you may have to wait a considerably longer time. Remember in the real world time and energy is money.