An AI / neural network...in vanilla JS!

#ai#javascript#beginners#webdev

Have you ever tried to actually build a neural network? No, neither have I...until today!

In this article we will cover a few things I learned and 2 demos of some very simple neural networks, written in vanilla JS.

Introduction

I was reading the @supabase_io 'AI Content Storm' articles earlier today.

And one thing dawned on me. I get neural networks...except I don't actually get them at all!

Like, I get the concept of a neuron. But how does the maths work?

In particular how do you use 'back propagation' to train a neural network? How do bias and weightings work? What or who is a sigmoid? etc.

Now, the sensible thing to do would have been to read a load of articles, grab a library and play with that.

But I am not sensible.

So instead I read a load of articles...and then decided to build my first neural network.

But that wasn't hard enough, so I decided to do it in JavaScript (as everyone seems to use Python...). Oh and I decided I would do it without any libraries. Oh and I wanted to build a visualiser in it too.

There is something wrong with me...I seem to thrive on pain.

Anyway, I did it, and here is what I learned.

Note: this is not a tutorial

Look, I want to be clear, this is not a tutorial!

This is just me sharing a few things that I found interesting while learning and my **first** neural network.

Note that there is emphasis on **first**, so please don't take this as anything other than something interesting to look at and play with.

I also do my best to explain each section and what it does, but as with everything, you get better at explaining it the more proficient you are with something...so some of my explanations may be a little 'off'!

Anyway, with all that out of the way, let's get on with it!

If you want to skip straight to the final demo then go right ahead!

First steps

Ok first thing is first, what is the most basic neural network I can build?

Well after a bit of reading I found out that a neural network can be as simple as some input neurons and some output neurons.

Each input neuron is connected to an output neuron and then we can add weightings to each of these connections.

With that in mind I had to come up with a problem to solve that was simple to follow, yet complex enough to ensure my network was working.

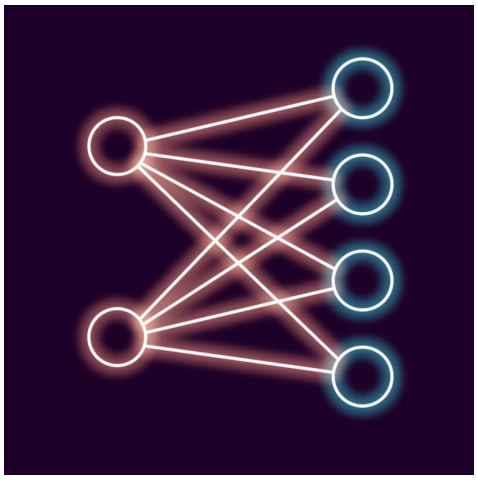
I decided upon a neural network that takes the X and Y coordinates of a point on a graph and then depending on whether they are positive or negative assigns a 'team' (colour) to them.

So that gives us 2 inputs (X and Y position) and then 4 outputs:

1. X ⩾ 0 and Y ⩾ 0
2. X < 0 and Y ⩾ 0
3. X ⩾ 0 and Y < 0
4. X < 0 and Y < 0

Due to how simple the requirements are here, we can get away without some 'hidden' neurons (that is something I will cover later) and keep things super simple!

So essentially we have to build a neural network that looks something like this:



The circles on the left are our inputs (X and Y positions) and the circles on the right are our outputs we discussed earlier.

Our first neurons

OK so now we can actually get started.

Now I didn't actually build a neuron first. In fact I actually built a visualiser first, as it was the easiest way to see if things were working, but I will cover that later.

So let's build a neuron (or more specifically, a few neurons and their connections).

Luckily, neurons are actually quite simple! (or should I say, they can be quite simple...they get more complex in Large Language Models (LLMs) etc.)

Simple neurons have a bias (think of that like an internal weighting, a number we will add to our final calculation to weight each neuron) and are connected to other neurons with weightings between each connection.

Now, in retrospect, adding the connections to each neuron individually may have been a better idea, but instead I decided to add each layer of neurons and each layer of connections as separate items as it made it easier to understand.

So the code to build my first neural network looked like this:

```
class NeuralNetwork {
    constructor(inputLen, outputLen) {
        this.inputLen = inputLen;
        this.outputLen = outputLen;
        this.weights = Array.from({ length: this.outputLen }, () =>
            Array.from({ length: this.inputLen }, () => Math.random())
        );
        this.bias = Array(this.outputLen).fill(0);
    }}
const neuralNetwork = new NeuralNetwork(2, 4);
```

Ok, I skipped a few steps, so let's briefly cover each part.

`this.inputLen = inputLen;` and `this.outputLen = outputLen;` are just so we can reference the number of inputs and outputs.

`this.weights = [...]` is the connections. Now it may look a little intimidating, but here is what we are doing:

> create an array of output neurons (`outputLen`)
>
> add an array of length `inputLen` to each of the array entries and populate it with some random values between `0` and `1` to start us off.

An example of the output of that code would looks like this:

```
this.weights = [
  [0.7583747881712366,0.4306037998314902],
  [0.40553698492617807,0.4419651593960727],
  [0.852978801662627,0.9762509253699836],
```

[0.8701610553353811,0.5583309725764114]]

And they essential represent the following:

[input 1 to output 1, input 2 to output 1],
[input 1 to output 2, input 2 to output 2],
[input 1 to output 3, input 2 to output 3],
[input 1 to output 4, input 2 to output 4],

Then we also have this.bias.

This is for each of the neurons in the output layer. It is what we use to add onto the output value later to make some neurons stronger and some weaker.

It is just an array of 4 zeros to start us off as we don't want an initial biases!

Now, although this is a neural network, it is completely useless.

We have no way of actually using it...and if we did use it the results it produces would be completely random!

So we need to solve these problems!

Using our network!

The first thing that we need to do is to actually take some inputs, run them through our network and gather the outputs.

Here is what I came up with:

```
propagate(inputs) {
  const output = new Array(this.outputLen);
  for (let i = 0; i < this.outputLen; i++) {
    output[i] = 0;
    for (let j = 0; j < this.inputLen; j++) {
      output[i] += this.weights[i][j] * inputs[j];
    }
    output[i] += this.bias[i];
    output[i] = this.sigmoid(output[i]);
  }
  return output;}
sigmoid(x) {
  return 1 / (1 + Math.exp(-x));}
```

Now there are two interesting things here.

Sigmoid

First of all one interesting thing is our sigmoid function. All that this does is transform a value we enter (say 12) into a value between 0 and 1 along an 's-shaped' curve.

This is our way of normalising values from extremes to something more uniform **and always positive.**

After further reading there are other options here on how we change a value to between 0 and 1, but I have not explored them fully yet (for example ReLU.

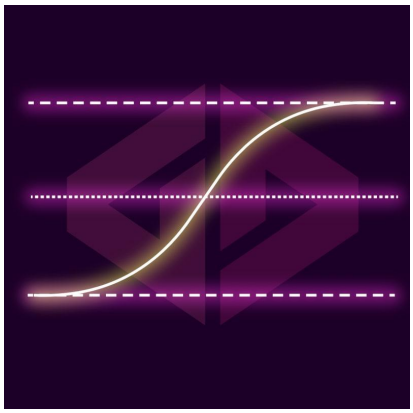
I am sure there are some very good explanations why this is needed, but in my monkey brain this is just the way of keeping values between 0 and 1 so that the multiplication stays within a certain bounds and the values are ‘flattened‘.

That way you don’t get ‘runaway‘ paths between neurons that are overly strong.

For example, imagine you had a connection with a weight of 16 and one with a weight of 1, using our sigmoid function we can reduce that from a difference of 16 times to a difference of about 35% ($\text{sigmoid}(1)$ is 0.73 and $\text{sigmoid}(16)$ is 0.99 after running through our function).

It also means that negative values are turned positive.

So running values though our sigmoid function means that negative numbers get transformed to a value between 0 and 0.5, a value of 0 becomes exactly 0.5 and a value greater than 0 becomes a value between 0.5 and 1.



If you think about it, that makes sense as the second we start multiplying negative and positive numbers together we can massively change our output.

For example, if we have a single negative neuron in a path of 100 and the rest are positive, this changes a strong value to a weak one and is likely to cause issues.

Anyway, as I read more and experiment more I am sure I will understand this part better!

Did I need biases?

The second interesting thing is the `output[i] += this.bias[i];`.

Well, in this neural network, all 4 outputs are equally important and we have no hidden neurons, so I later removed this to simplify the code!

Ironically though, on our more complex neural network, I needed to re-add the biases on the output neurons, due to how the network back propagation was working. Otherwise one output neuron activated all the time.

What I could not work out is whether this was a necessary step, or whether I made a mistake with my neural network and this was compensating for it.

Yet again, I want to remind you, I am still learning and only just grasp the basics, so I have no idea which it is!

We are nearly there

The rest of the code above is reasonably straight forward. We are just multiplying each input by a weighting associated with each output (and adding our unnecessary bias!).

In fact, we can run this now, but out results will be atrocious! Let's fix that!

Time to train!

Ok last important part of a neural network, training it!

Now as this article is getting long, I will just cover the main points of the following training code (which took me nearly an hour to write by the way...I told you I am a noob at this!)

```
train(inputs, target) {
    const output = this.propagate(inputs);
    const errors = new Array(this.outputLen);

    for (let i = 0; i < this.outputLen; i++) {
        errors[i] = target[i] - output[i];
        for (let j = 0; j < this.inputLen; j++) {
            this.weights[i][j] +=
                this.learningRate *
                errors[i] *
                output[i] *
                (1 - output[i]) *
                inputs[j];
        }
        this.bias[i] += this.learningRate * errors[i];
    }
}
```

'Why did it take so long?' I hear you ask! Well, it was getting my head around all the bits that needed multiplying together in order to update each of the weights.

Also this.learningRate took a little getting used to. It is simply a reduction in the rate that we change the weightings so that we don't 'over-shoot' our target values for each weighting, but tuning it to a reasonable value takes experience...I didn't have experience and set it way too low, so my code appeared broken!

After a bit of fiddling, I settled on a value of 0.1 (instead of 0.01

Right, so we have a training function. But bear in mind that this training function only does one pass of training.

We need to train our network a load of times, with each time it trains hopefully making it more accurate.

We will cover that in a second, but I want to share a quick side point / thing I learned.

Training data adjustment

I know we haven't even covered the final training data, but this was an interesting point I learned that fits here (as it explains why it took me so long to write this training function).

Originally I was generating hundreds of different training X and Y coordinates, all randomised.

But then I got much better results by generating just 4 static training points after some further reading:

```
const trainingData = [
    { x: -0.5, y: -0.5, label: 'blue' },
    { x: 0.5, y: -0.5, label: 'red' },
    { x: -0.5, y: 0.5, label: 'green' },
    { x: 0.5, y: 0.5, label: 'purple' }];
```

It makes sense, once you get it!

We want to ‘pull‘ values closer to a target, the above values are the exact ‘centre point‘ of each of our areas.

So our error rates will always be consistent for a given distance.

This means our neural network learns more quickly as our error rates are larger depending on whether they are further from X or further from Y.

I could explain that better, but that is beyond the scope of this article. Hopefully if you have a think about it, then it will also ‘click‘ for you like it did for me!

Ironically I went back to the more randomised data set for the bigger model as I wanted to really test my understanding of learning rates, over-training etc.

We have a functioning and useful neural network!

Now that is, in effect, our entire neural network.

There is one thing we need to do though.

Our training function needs to run it a load of times!

So we need one last function to do that, which takes our training data and runs our training function a few hundred times:

```
function train() {
  for (let i = 0; i < 10000; i++) {
    const data =
          trainingData[Math.floor(Math.random() * trainingData.length)];
    neuralNetwork.train([data.x, data.y], encode(data.label));
```

```
}
console.log('Training complete');}
```

Goldilocks iterations

Notice that we train our network 10,000 times in the for loop.

10,000 iterations was plenty to train this particular neural network. But for the more complex one we will cover in a bit, I needed more iterations (and to turn down the learning rate).

This is one of the interesting parts of machine learning, you need to train a neural network enough (which is hard to get right), but if you train it too much you get 'over fitting' happening and actually start getting worse results. So it needs to be perfectly balanced for the best results!

Anyway, that was a lot, we are finally at our first demo!

Simple vanilla JS neural network demo

It is a little messy, but our neural network and all of the training part is in the first 67 lines of the CodePen below.

The remaining lines of code actually run our network (neuralNetwork.propagate([x, y]); roughly line 85) and then output the points and their predicted colours onto a <canvas|.

encode and decode are purely to take our output neurons, find which one has the highest activation and then map that to a colour for our visualisation.

And here is the last thing to understand. Our output neurons will all have a value. A neural network doesn't just output 1, 0, 0, 0.

Instead it will output a ‘certainty‘ or guess for each of the output neurons. So we will get something like 0.92,0.76, 0.55, 0.87 as our output.

So that is why we have our decode function, which finds the highest outputting neuron and takes that as our final guess!

```
// this line finds the max value of all of our output neurons and then returns its index so we can use that to classify our X and Y coordinates.const maxIndex = output.indexOf(Math.max(...output));
```

Usage and actual demo

To use the example you have 3 buttons:

> **Train** - to train our neural network as it starts untrained and randomised.
>
> **Classify Points** - this is to run our neural network. It will plot the points on the graph and assign them a colour. I advise running this before and after training.
>
> **reset** - this will create a new neural network that is untrained. Great for testing out the classification of points before and after training.

Also note that each of the areas is coloured according to what colour should show there. It really let’s you see how far from successful a randomised and untrained neural network is (reset and then classify points to test)!

Have a play! Here is the full code for our simple demo, it is a working neural network:

```
class NeuralNetwork {
  constructor(inputLen, outputLen) {
```

```
        this.inputLen = inputLen;
        this.outputLen = outputLen;
        this.weights = Array.from({ length: this.outputLen }, () =>
            Array.from({ length: this.inputLen }, () => Math.random())
        );
        //this.bias = Array(this.outputLen).fill(0);
        this.learningRate = 0.1;
        this.points = [];
        console.log(this.weights, this.sigmoid(1), this.sigmoid(16), this.sigmoid(-2),
this.sigmoid(0));
    }

    propagate(inputs) {
        const output = new Array(this.outputLen);
        for (let i = 0; i < this.outputLen; i++) {
            output[i] = 0;
            for (let j = 0; j < this.inputLen; j++) {
                output[i] += this.weights[i][j] * inputs[j];
            }
            //output[i] += this.bias[i];
            //console.log('bias', i, this.bias[i]);
            output[i] = this.sigmoid(output[i]);
        }
        return output;
    }
    sigmoid(x) {
        return 1 / (1 + Math.exp(-x));
    }
    train(inputs, target) {
        const output = this.propagate(inputs);
        const errors = new Array(this.outputLen);

        for (let i = 0; i < this.outputLen; i++) {
            errors[i] = target[i] - output[i];
            for (let j = 0; j < this.inputLen; j++) {
                this.weights[i][j] +=
                    this.learningRate *
                    errors[i] *
                    output[i] *
```

```
                (1 - output[i]) *
                inputs[j];
            }
            //this.bias[i] += this.learningRate * errors[i];
        }
    }
}

const trainingData = [
    { x: -0.5, y: -0.5, label: 'blue' },
    { x: 0.5, y: -0.5, label: 'red' },
    { x: -0.5, y: 0.5, label: 'green' },
    { x: 0.5, y: 0.5, label: 'purple' }
];

function train() {
    for (let i = 0; i < 10000; i++) {
        const data =
            trainingData[Math.floor(Math.random() * trainingData.length)];
        neuralNetwork.train([data.x, data.y], encode(data.label));
    }
    console.log('Training complete');
}

function reset() {
    neuralNetwork = new NeuralNetwork(2, 4);
}


const canvas = document.getElementById('graph');
const ctx = canvas.getContext('2d');
const pointRadius = 5; // Radius of the points


let neuralNetwork = new NeuralNetwork(2, 4);


function classifyPoints() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawAxes();
```

```
    this.points = [];
    for (let i = 0; i < 100; i++) {
        const x = Math.random() * 2 - 1; // Random x-coordinate between -1 and 1
        const y = Math.random() * 2 - 1; // Random y-coordinate between -1 and 1
        const output = neuralNetwork.propagate([x, y]);
        const predictedLabel = decode(output);
        drawPoint(x, y, predictedLabel);
        points.push({ x, y, predictedLabel });
    }
    //console.log(points);
}
function encode(label) {
    const encoding = {
        blue: [1, 0, 0, 0],
        red: [0, 1, 0, 0],
        green: [0, 0, 1, 0],
        purple: [0, 0, 0, 1]
    };
    return encoding[label];
}

function decode(output) {
    const labels = ['blue', 'red', 'green', 'purple'];
    const maxIndex = output.indexOf(Math.max(...output));
    return labels[maxIndex];
}


function drawPoint(x, y, color) {
    ctx.beginPath();
    ctx.arc(
        ((x + 1) * canvas.width) / 2,
        canvas.height - ((y + 1) * canvas.height) / 2,
        pointRadius,
        0,
        2 * Math.PI
    );
    ctx.fillStyle = color;
    ctx.fill();
```

```
    ctx.closePath();
}

function drawAxes() {

    const canvasHalfWidth = canvas.width / 2;
    const canvasHalfHeight = canvas.height / 2;

    ctx.beginPath();
    ctx.fillStyle = `rgba(0,255,0,0.2)`;
    ctx.fillRect(0, 0, canvasHalfWidth, canvasHalfHeight);
    ctx.fillStyle = `rgba(225,0,255,0.2)`;
    ctx.fillRect(canvasHalfWidth, 0, canvasHalfWidth, canvasHalfHeight);
    ctx.fillStyle = `rgba(0,0,255,0.2)`;
    ctx.fillRect(0, canvasHalfHeight, canvasHalfWidth, canvasHalfHeight);
    ctx.fillStyle = `rgba(255,0,0,0.2)`;
    ctx.fillRect(canvasHalfWidth, canvasHalfHeight, canvasHalfWidth, canvasHalfHeight);

    ctx.moveTo(0, canvas.height / 2);
    ctx.lineTo(canvas.width, canvas.height / 2); // X-axis
    ctx.moveTo(canvas.width / 2, 0);
    ctx.lineTo(canvas.width / 2, canvas.height); // Y-axis
    ctx.strokeStyle = 'black';
    ctx.stroke();
    ctx.closePath();
}
```

End of our most basic neural network

So there we have our most basic neural network!

It functions well for our needs and we managed to learn a little bit about back propagation (our train function in the main class) and weightings and biases.

But it is very limited. If we want to do anything more advanced in the future, we need to add some hidden neurons!

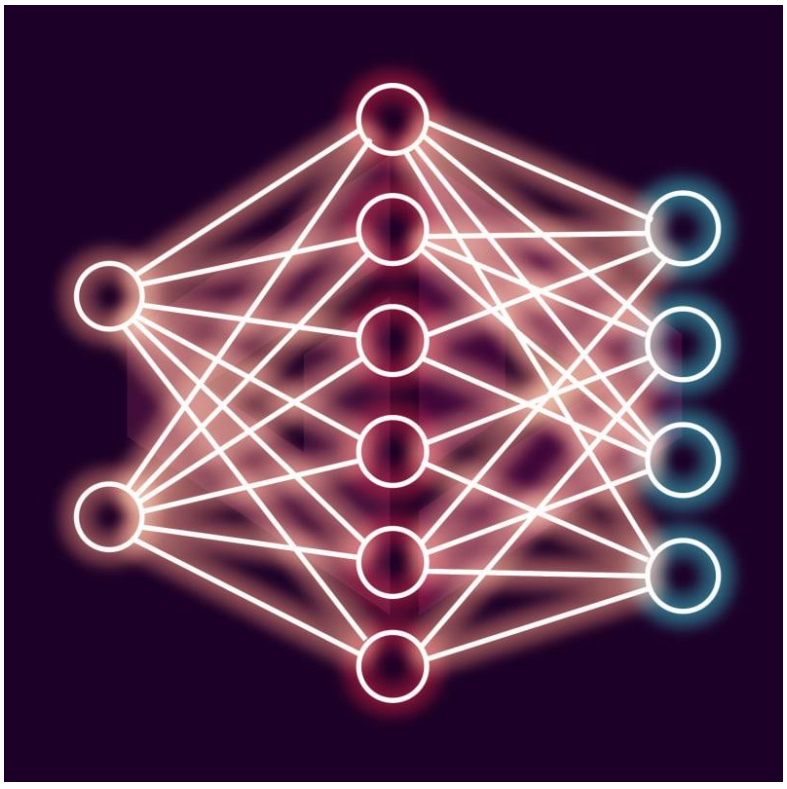
Version 2 - hidden neurons

OK, so why hidden neurons? What purpose do they serve?

In more complex examples they serve as a way to take the inputs and add more dimensions to the way they are categorised.

We are still using 2 input neurons and 4 output neurons, but this time we have added an additional layer in the middle (which we can change and adjust the number of neurons in).

So our neural network looks something like this:



As neural networks need to handle more inputs and do more complex calculations, additional neurons in hidden layers allow them to better categorise inputs and provide better results.

Hidden layers can also be different ‘depths‘.

So let’s say we have 2 input neurons. We could connect them to 6 ‘hidden‘ neurons, and then connect them to out 4 output neurons.

But we could also connect out 6 neurons in our first layer to a second layer of hidden neurons. This second layer could have 8 neurons, which then connect to our 4 output neurons.

But that is a lot to follow, and this was for me to learn the basics, so I chose to add a single hidden layer. This also meant I could keep each connection layer as a separate array, which is just easier to understand at this stage!

So what is new?

Not much has changed, just we have more connections and a few more neurons!

You can think of it as adding 2 of our original neural networks in series, just that the output of the first one now acts as the input for the second one.

While the code may be a lot more convoluted, our neural network follows the same principles.

Here is the code:

```
class NeuralNetwork {
  constructor(inputSize, hiddenSize, outputSize) {
    this.inputSize = inputSize;
    this.hiddenSize = hiddenSize;
    this.outputSize = outputSize;
    this.weightsInputToHidden = Array.from({ length: hiddenSize }, () =>
          Array.from({ length: inputSize }, () => Math.random() * 2 - 1)
        );
    this.biasHidden = Array(hiddenSize).fill(0);
    this.weightsHiddenToOutput = Array.from({ length: outputSize }, () =>
      Array.from({ length: hiddenSize }, () => Math.random() * 2 - 1)
    );
    this.biasOutput = Array(outputSize).fill(0);
    this.learningRate = document.querySelector('#learningRate').value; //
Adjusted learning rate
    this.hiddenLayer = new Array(this.hiddenSize);
  }

  feedForward(inputs) {
    for (let i = 0; i < this.hiddenSize; i++) {
      this.hiddenLayer[i] = 0;
      for (let j = 0; j < this.inputSize; j++) {
        this.hiddenLayer[i] +=
        this.weightsInputToHidden[i][j] * inputs[j];
      }
      this.hiddenLayer[i] += this.biasHidden[i];
      this.hiddenLayer[i] = sigmoid(this.hiddenLayer[i]);
    }
```

```
  const output = new Array(this.outputSize);
  for (let i = 0; i < this.outputSize; i++) {
    output[i] = 0;
    for (let j = 0; j < this.hiddenSize; j++) {
      output[i] +=
      this.weightsHiddenToOutput[i][j] * this.hiddenLayer[j];
    }
    output[i] += this.biasOutput[i];
    output[i] = sigmoid(output[i]);
  }
  return output;
}

train(inputs, target) {
  for (let i = 0; i < this.hiddenSize; i++) {
    this.hiddenLayer[i] = 0;
    for (let j = 0; j < this.inputSize; j++) {
      this.hiddenLayer[i] +=
      this.weightsInputToHidden[i][j] * inputs[j];
    }
    this.hiddenLayer[i] += this.biasHidden[i];
    this.hiddenLayer[i] = sigmoid(this.hiddenLayer[i]);
  }

  const output = new Array(this.outputSize);
  for (let i = 0; i < this.outputSize; i++) {
    output[i] = 0;
    for (let j = 0; j < this.hiddenSize; j++) {
      output[i] += this.weightsHiddenToOutput[i][j] * this.hiddenLayer[j];
    }
    output[i] += this.biasOutput[i];
    output[i] = sigmoid(output[i]);
  }
```

```
  const errorsOutput = new Array(this.outputSize);
  const errorsHidden = new Array(this.hiddenSize);

  for (let i = 0; i < this.outputSize; i++) {
    errorsOutput[i] = target[i] - output[i];
    for (let j = 0; j < this.hiddenSize; j++) {
      this.weightsHiddenToOutput[i][j] +=
        this.learningRate *
        errorsOutput[i] *
        output[i] *
        (1 - output[i]) *
        this.hiddenLayer[j];
      }
      this.biasOutput[i] += this.learningRate * errorsOutput[i];
    }

    for (let i = 0; i < this.hiddenSize; i++) {
      errorsHidden[i] = 0;
      for (let j = 0; j < this.outputSize; j++) {
        errorsHidden[i] += this.weightsHiddenToOutput[j][i] * errorsOutput[j];
      }
      this.biasHidden[i] += this.learningRate * errorsHidden[i];
      for (let j = 0; j < this.inputSize; j++) {
        this.weightsInputToHidden[i][j] +=
        this.learningRate *
        errorsHidden[i] *
        this.hiddenLayer[i] *
        (1 - this.hiddenLayer[i]) *
        inputs[j];
      }
    }
  }}
```

Now, don't be intimidated, I have just copied a few loops with slightly different target sets of data to manipulate.

We have added an extra set of biases (for our hidden layer) and an extra set of connections: our input layer to our hidden layer and then our hidden layer now connects to our output layer.

Finally our train function has a few extra loops just to back propagate through each of the steps.

And the only other change worth mentioning is that we now have a third input parameter (in the middle), for the number of hidden neurons.

Ugly, but it seems to work

Look, I want to say it one more time, this was me learning as I go and so the code reflects that.

There is a lot of repetition here and it is not very extensible.

However, as far as I can tell, it works.

With that being said, although it works, it appears to perform worse than our original, much simpler neural network.

It either means I made a mistake (likely), or it is that I haven't 'dialled in' the correct training settings.

Speaking of which...

Adding some variables to play with

As this was more complex I have ‘bodged‘ in some quick settings.

Now we can update:

Training Data Size - the number of different random points we generate

Training iterations - how many times we select a random data point from the training set and feed that into our train function on the neural network.

Learning Rate - our multiplier for how quickly we should adjust based on errors.

hidden nodes (**more than 2!**) - adjusting how many hidden nodes there are in the second layer (requires you to initialise the network again otherwise it will break!)

points to classify - the number of points to pass to our trained neural network and plot on the graph.

This means we can play with values far more quickly to see what effect they have on our neural network and it’s accuracy!

One last thing

Oh and I added a button for a visualisation of what the neural network looks like.

By all means press ‘Visualize Neurons and Weights‘, but it isn’t finished. I also have no immediate intention to finish it

as I want to completely redesign my approach to building a neural network so it is more extensible.

Conclusion

It was really fun building a neural network in vanilla JS.

I have no seen many people doing it like this, so I hope it is useful to you or at least somebody!

I learned a lot about biases, back propagation (the key to neural networks) etc.

Obviously this example and the things learned here are only 1% of machine learning. But the core principles are the same for a tiny, unoptimized neural network like mine and a gigantic multi-billion parameter model.

This example was like the 'hello world' of Machine Learning (ML) and Neural Networks.

Next I really want to try and build a much larger neural network that is better structured and easier to extend, to see if we can do some Optical Character Recognition (OCR). You can think of that as the 'ToDo List' of ML and Neural Networks!