A ~~Neural Network~~Neuron from scratch in JavaScript

Ian Channing
https://github.com/ianchanning/neural-network-js

February 12, 2019

My background (biases)

Maths and Computer Science at Imperial in London

JavaScript / React for Imec

All AI knowledge from online courses (In Andrew Ng we trust)

The beginning

> *Implementing it myself from scratch was the most important*
> — *Andrej Karpathy talking to Andrew Ng [2] (2018)*

Aim: generate data, visualize it, label it and train a neuron to classify it.

Inspired / blatantly copied from

Funfunfunction NN playlist [3]

... but it's missing maths

deeplearning.ai week 2 [4]

... but code isn't open, filling in blanks

Neural Networks & Deep Learning course [5]

... but no code

Get ourselves setup

Install VS Code (optional) https://code.visualstudio.com

Download & extract the zip
https://github.com/ianchanning/neural-network-js

Run `npm install` (totally optional)

Open `index.html`

Open Browser tools (F12)

Start the coding

In `index.html`:

```
<script src="tutorial/neural-network.skeleton.js"></script>
<script>
  nn();
</script>
```

In `tutorial/neural-network.skeleton.js`:

Wrap code inside a function to avoid evil global scope [9]

```
function nn() {
  // all your var are belong to us
}
```

Skeleton

The outline of what we're going to produce

```
// data
function generator() {}

// SVG chart elements
function chart() {}

// perceptron / neuron
function neuron() {}

// generator + neuron + chart
function build() {}

// draw the chart to root `<div>`
function draw() {}
```

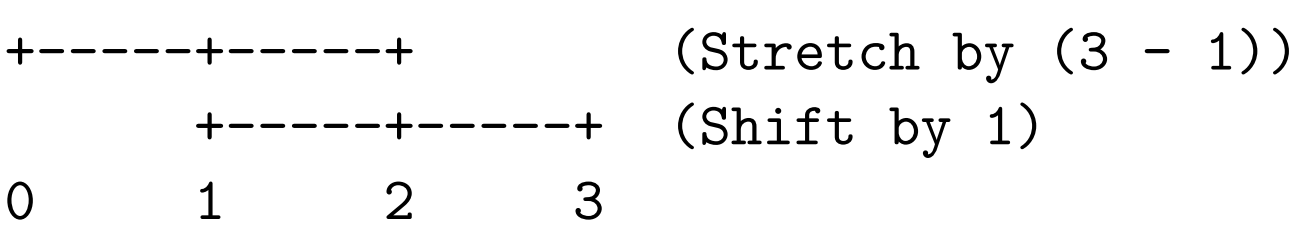
I want it to display random values

Generate random test and training sets

```
function rand(min, max) {
  return Math.random() * (max - min) + min;
}
rand(1,3);
rand(0,400); // x1, x2 range for our graph
```

Stretch (*) and shift (+)

```
rand(0,1) -->  rand(1,3)
```



Slight digression (humour me)

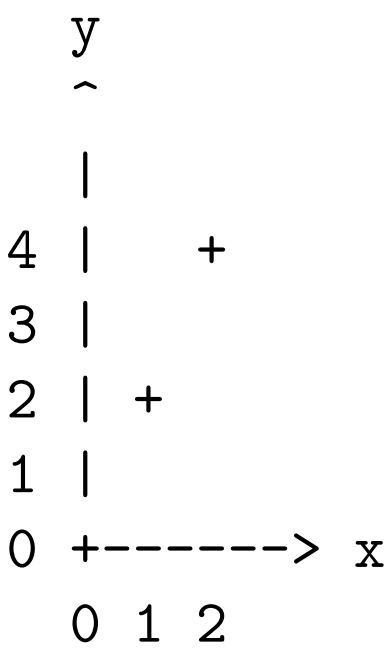
Code <3 Maths

JavaScript's `map` functions in maths

Reduce the gap between maths and code

Let's draw a graph

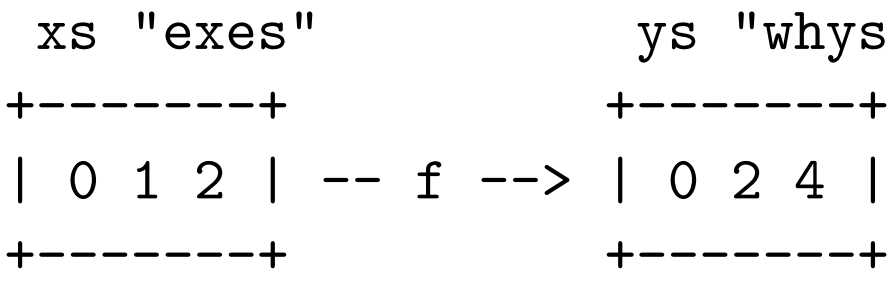
$$y = f(x) = 2x$$



Mathsy definitions

What's the mathsy name for:

I've got one 'set' and I want to go to another 'set' using f?

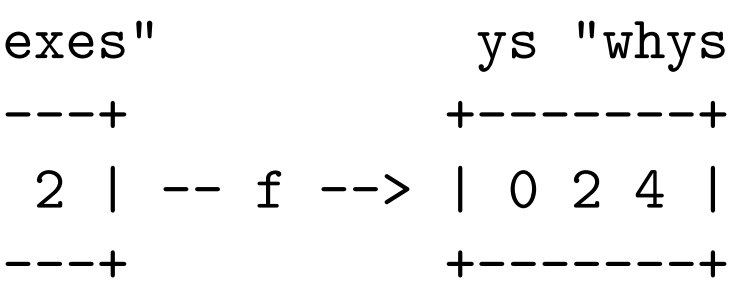


(This is actually University level maths - Set Theory)

Mathsy definitions

What's the mathsy name for:

I've got one 'set' and I want to go to another 'set' using f?



(This is actually University level maths - Set Theory)

Mapping! `f` 'maps' `0,1,2` on to `0,2,4`

$f(x)$ in JavaScript

$$y = f(x) = 2x$$

```
function f(x) {return 2 * x;}
var xs = [0,1,2];
var ys = xs.map(f); // [0,2,4]
```

`map` is awesome. Kill all loops!

What's the point?

Our graph will be made up of `[x1, x2]` points.

One random point in JavaScript:

```
var point = [rand(0, 400), rand(0, 400)];
```

I want to generate a set of random test values

Perhaps I should use a for loop? (never!)

Generate an empty array and use that to generate our new set.

```
function points(length) {
  return Array(length)
    .fill(0)
    .map(function(i) {
      return [rand(0, 400), rand(0, 400)];
    });
}
```

Mapping `[0,0,0] ---> [[x1,x2],[x1,x2],[x1,x2]]` (demo?)

Make `rand` and `points` available, functions are passed as values

```
return {rand, points};
```

I want to display these test values

Gonna need a graph mate, how does that SVG work again?

Should've read CSS-Trick's excellent guide on SVG Charts [10]

```
<svg
  version="1.1"
  xmlns="http://www.w3.org/2000/svg"
  height="400"
  width="400"
>
  <circle cx="90" cy="192" r="4"></circle>
</svg>
```

Brain shift required: `(0,0)` is top left

Putting that in JavaScript

```
function chart(height, width) {
  // <name xmlns="..."></name>
  function element(name) {
    var ns = "http://www.w3.org/2000/svg";
    return document.createElementNS(ns, name);
  }
  // <svg ...></svg>
  function svg() {
    // JS note: svg() can access element()
    // var s is private to svg()
    var s = element("svg");
    s.setAttribute("height", height);
    s.setAttribute("width", width);
    return s;
  }
}
```

I want to draw the circle

```
// centre is a point [x1,x2]
// <circle cx="0" cy="0" r="4" fill="blue"></circle>
function circle(centre, radius, colour) {
  var c = element("circle");
  c.setAttribute("cx", centre[0]);
  c.setAttribute("cy", centre[1]);
  c.setAttribute("r", radius);
  c.setAttribute("fill", colour);
  return c;
}
```

Make `svg` and `circle` available, functions are passed as values

```
return {svg, circle};
```

I want to draw the test values as circles on a graph

I smell a `map`. I want to map my test values onto the graph.

```
function build(generator, chart) {
  var svg = chart.svg();
  generator.points(100).map(function(point) {
    svg.appendChild(chart.circle(point, 4, "black"));
  });
  return svg;
}
```

Add this to `draw()`:

```
var svg = build(generator(), chart(400, 400));
document.getElementById("root").appendChild(svg);
```

And... we've got a visualization of our data

I want to colour the circles red or blue

In `build()`, rather than black circles we can draw random red or blue circles.

```
var colours = ["red", "blue"];
...
var team = Math.round(Math.random());
svg.appendChild(chart.circle(point, 4, colours[team]));
```

I want to separate these circles with a line

Time to racially discriminate our happy circles ...err "linearly separate" them.

We need a wall!

Add this to `chart()`:

```
// start, end are points [x1,x2]
// <line x1="0" y1="0" x2="10" y2="10" fill="blue"></line>
function line(start, end, colour) {
  var l = element("line");
  l.setAttribute("x1", start[0]);
  l.setAttribute("y1", start[1]);
  l.setAttribute("x2", end[0]);
  l.setAttribute("y2", end[1]);
  l.setAttribute("stroke", colour);
  return l;
}
return {svg, circle, line};
```

Build the wall! Build the wall!

Add this to `build()`:

```
svg.appendChild(
  chart.line([0, 0], [400, 400], "black")
);
```

I want to make the colour depend on which side of the line

One side are the blues, and the other side are the reds. Go blues!

Top half is for the blues, the reds get everything else.

Now as the all-seeing-being we know how to label them. Reminder: SVG coordinates have (0,0) in the top left.

In our `generator()`:

```
// which side of the wall
function team(point) {
  return (point[0] > point[1]) ? 1 : 0;
}
```

and in `build()` set the team dynamically:

```
var team = generator.team(point);
svg.appendChild(chart.circle(point, 4, colours[team]));
```

I want to label my random examples

Get our own slave labour / Amazon Mechanical Turk [11] to label data for us.

```
var labelledPoint = {
  point: [0, 1],
  actual: ???
};
```

I want to say whether my examples are red or blue

In `generator()`:

```
// points is a set of [x1,x2] points
function labeller(points) {
  return points.map(function(point) {
    return {
      point: point,
      actual: team(point)
    };
  });
}

// labelled training data
function examples(length) {
  return labeller(points(length));
}

return {rand, points, team, examples};
```

I want to make a guess based on x1, x2 whether a circle is red or blue

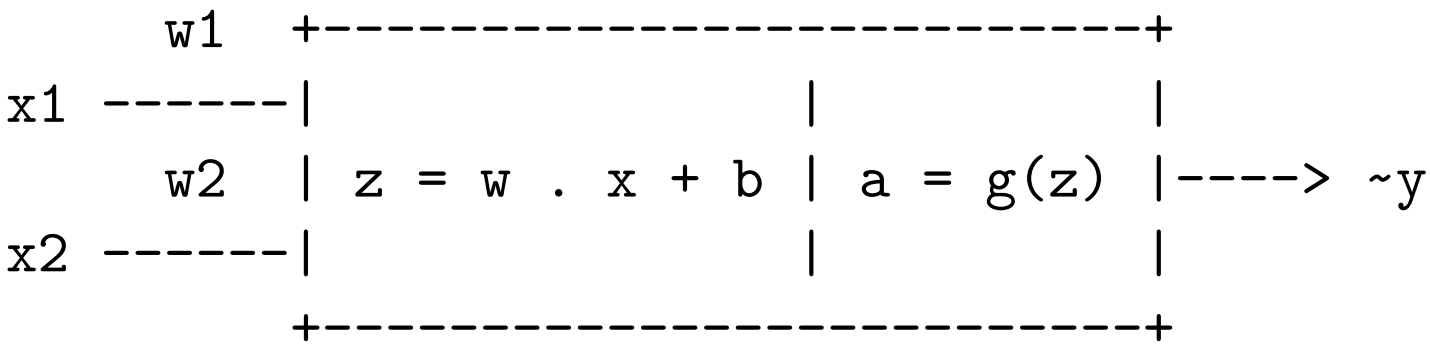
Time for the good stuff

Don't confuse x, y with x1, x2

A neural network of one neuron

> *An Englishman, even if he is alone, forms an orderly queue of one*
> — *George Mikes*

Neurons act independently so can scale up process to a network



`w . x` is the dot product / weighted sum

`b` is the bias

`g` is our 'activation' function

`~y` is our approx/guess of `y`, usually called $\hat{y}$ 'y hat'

Perceptron or neuron?

Originally called a perceptron [6]

Changed to a neuron with the sigmoid activation function - (there's probably a better definition)

Mathematical concepts different, but coding concepts similar

For us:

1. Fully code perceptron
2. Iterate to a neuron (if we get time)

I want to combine my inputs into one value

Combine inputs into one value

More important inputs have a bigger impact

Pathways in the brain become stronger the more they are used (see Inner Game of Tennis)

Weighted sum / dot product (1 row x 1 column)

I want to multiply 1 row matrix x 1 column matrix

Total the inputs using vector dot product / weighted sum

```
w . x = [w1 w2]|x1|
               |x2|
```

A vector in Python is a list, in JavaScript an array

Add this to `neuron()`:

```
// 2-D dot product only, [w1,w2], [x1,x2]
function dot(w, x) {return w[0] * x[0] + w[1] * x[1];}
```

We can scale the dot product to as many elements as we want

I want to describe a perceptron firing

Perceptron 'fires' when inputs reach a threshold

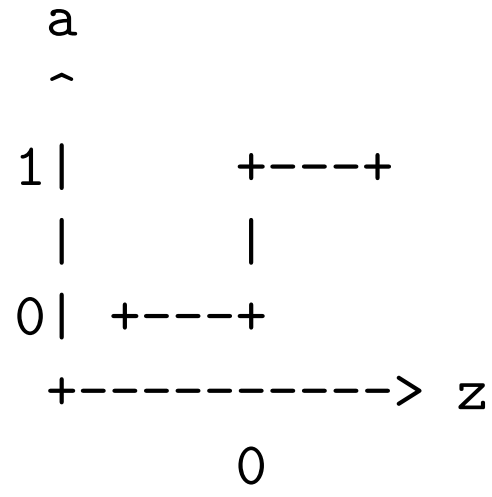
```
                | 0 if w . x <= threshold
activation =    |
                | 1 if w . x > threshold
```

Subtract threshold from both sides and call it 'bias'

```
bias = -threshold
```

```
                | 0 if w . x + bias <= 0
activation =    |
                | 1 if w . x + bias > 0
```

A bit confusing, let's see some code



N.B. Our line goes through zero so we don't need bias

if then, else...

Add this to `neuron()`:

```
// z = dot(w, x) + bias (but no bias here)
// a = g(z)
function activation(z) {return (z <= 0) ? 0 : 1;}
```

Easiest function you can write $\rightarrow$ basis for all AI

Someone somewhere is having a laugh

I want to start somewhere

Initialise our weights to either 0 or small random values.

Add `weights` to `generator()` and return

```
// experiment with (0,0) or rand(0,400), or rand(0,1)
var weights = [rand(-1,1), rand(-1,1)];

return {rand, points, team, examples, weights};
```

I want to my a first guess

Make a prediction from our weights

In `neuron()`:

```
// y = g(w . x + b)
function prediction(w, x) {
  return activation(dot(w,x));
}
```

I want to display my predictions

Instead of our known `team` use our `prediction`.

In `build()` replace:

```
// var team = generator.team(point);
var team = neuron.prediction(generator.weights, point);
```

Add `neuron` to `build()`:

```
function build(generator, chart, neuron) {
  // ...
}
```

Create & pass neuron in `draw()`:

```
myNeuron = neuron();
var svg = build(myGenerator, myChart, myNeuron);
```

I want to get a better feel for what the weights mean

Change the initial `weights` to some random values and show the weights we're using.

In `draw()` add this at the end:

```
drawP("intial w: "+myGenerator.weights.join());
```

Which weights give the best predictions?

I want to specify how I can improve

Define a loss/error function: a function we want to *minimise*

How different our prediction was from the actual value

```
function loss(y, prediction) {
  return y - prediction;
}
```

I want to adjust the weights to improve my guess

Feed the loss back into the weights

```
function adjust(w, x, loss, i) {
  return w[i] + loss * x[i];
}
```

I want to combine these into a training step

One small step for one example

https://en.wikipedia.org/wiki/Perceptron#Steps

```
// step 1: initialise weights w
// step 2: for each example x with actual y
function step(w, x, y) {
  // step 2a: calculation actual output
  var yhat = prediction(w, x);
  // step 2b: update the weights for each x[i]
  var l = loss(y, yhat);
  return [
    adjust(w, x, l, 0),
    adjust(w, x, l, 1)
  ];
}
```

I want to do a single step of training

We can look at how the weights change step by step but I think it's overkill

In `neuron()`, return the `step` function and then apply it in `build()`

```
example = generator.examples(1);
var weights = neuron.step(
  generator.weights,
  example[0].point,
  example[0].actual
);
generator.points(100).map(function(point) {
  var team = neuron.prediction(weights, point);
```

One last digression

More maths in JavaScript

The `reduce` function

$2 + 2 + 2$

$$y = \sum f(x) = \sum 2x$$

```
 x   2x Running total
 1   2  2
 1   2  4
+1  +2  6
--  --
 3   6
```

```
function sum(t, x) { return t + f(x); }
var xs = [1,1,1];
var y  = xs.reduce(sum, 0); // 6
```

(demo?)

I want to train using all examples

`reduce` the examples down into a single set of trained weights

Add this to `neuron()` and include it in the return value

```
// intial weights w
// labelled examples
function train(w, examples) {
  // wrapper function to work with reduce
  function trainExample(w, example) {
    return step(w, example.point, example.actual);
  }
  // repeatedly updates w and returns the trained w
  return examples.reduce(trainExample, w);
}

return {prediction, train}
```

I want to replace the guess with trained weights

In `build()` add:

```
var weights = neuron.train(
  generator.weights,
  generator.examples(100) // how many?
);
```

Then replace:

```
// var team = neuron.prediction(generator.weights, point);
var team = neuron.prediction(weights, point);
```

Have we gotten any better at guessing?

I want to see what the trained weights are

Draw another paragraph and put it in a function as we're repeating the steps

Return the trained weights from `build()`:

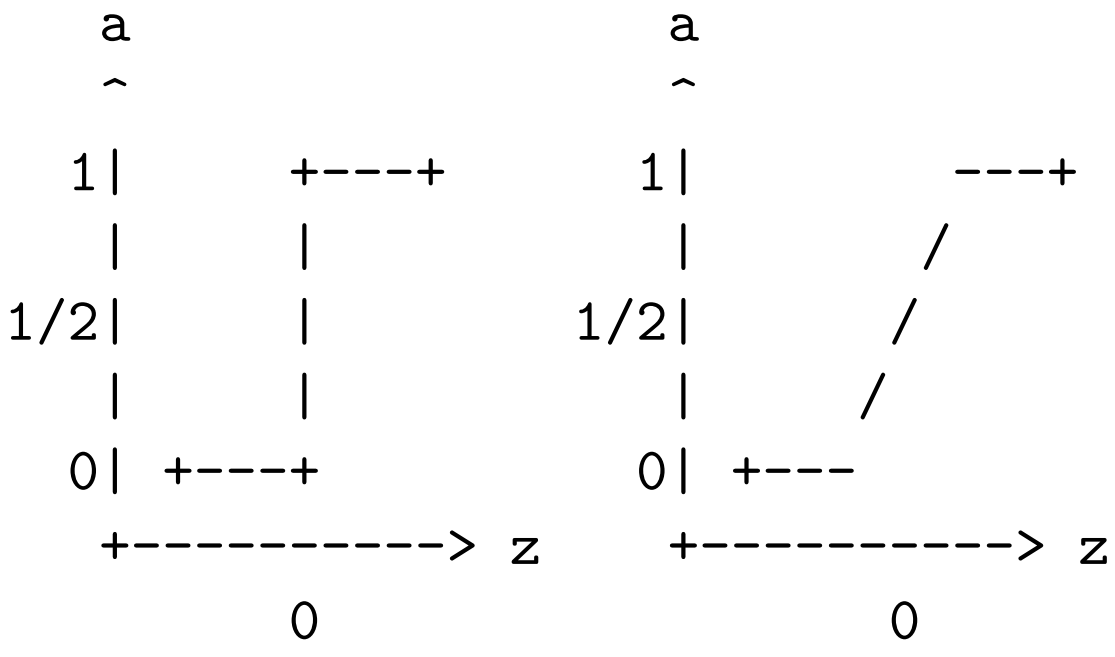
```
return {svg, weights};
```

Then in `draw()` replace:

```
var myBuild = build(myGenerator, myChart, myNeuron);
document.getElementById("root").appendChild(myBuild.svg);
drawP("initial w: "+myGenerator.weights.join());
drawP("trained w: "+myBuild.weights.join());
```

Sigmoid neuron

Smooth curved perceptron



Todo ...

I want to specify the cost function

Let's meet the the cross entropy cost function.

The bit we use is the derivative for back-propagation in eqn (61)

$dC/dW_j = 1/n * \sum_x x_j(g(z) - y)$

To be continued...

I want to explain why we get bias/over-fitting

Here we loop around our examples just once. But for more complex problems we loop over the same examples thousands of times. When you say the same word a thousand times over you start to notice tiny details about the word that aren't relevant. e.g. conscience, that's actually con-science but that's totally irrelevant. Neural Networks have no other ideas about the world except for the examples we give them.

In summary

Generated random set of training and test data that we displayed on a graph for testing

Split the points on the graph using an abritrary line (why? back propagation needs linear separation)

Used a perceptron with an activation function and back propagation algorithm

Trained this perceptron to adjust two weights that then colour the test points depending on which side of the line

This is one step of gradient descent

'Improved' this with a sigmoid activation function and it's differentiated back propagation.

The end

What I cannot create, I do not understand.
— Richard Feynman [8] (1988)

...

Young man, in mathematics you don't understand things. You just get used to them.
— John Von Neumann

...

What you really want is to feel every element (and the connections between them) in your bones.
— Michael Nielsen (2019)