A ~~Neural Network~~Perceptron 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 plan

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

1. Generate data
2. Visualize it
3. Label it
4. Train a perceptron 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`:

```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]

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

Skeleton

The outline of what we're going to produce

```javascript
// 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() {}
```

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
```
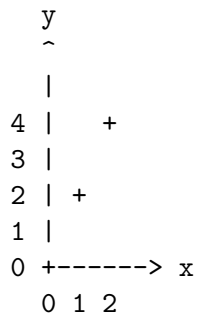
Slight digression (humour me)

Code <3 Maths

JavaScript's `map` functions in maths

Reduce the gap between maths and code

Let's draw a graph

$y = f(x) = 2x$

```
  y
  ^
  |
4 |     +
3 |
2 | +
1 |
0 +------> x
  0 1 2
```

Mathsy definitions

What's the mathsy name for:

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

```
 xs "exes"              ys "whys"
+-------+             +-------+
| 0 1 2 | -- f --> | 0 2 4 |
+-------+             +-------+
```

(This is actually University level maths - Set Theory)

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

$f(x)$ in JavaScript

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

```javascript
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:

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

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() {
      return [rand(0, 400), rand(0, 400)];
    });
}
return { points };
```

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

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

```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;
  }
}
```

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 };
```

Draw the test values as circles on a graph

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

And… we've got a visualization of our data

Colour the circles red or blue

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

```
var colours = ["red", "blue"];
generator.points(100).map(function(point) {
  var team = Math.round(Math.random());
  svg.appendChild(chart.circle(point, 4, colours[team]));
```

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"));
```

Make the colour depend on which side of the line

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

Now as the mighty dictator 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;
}
return { points, team };
```

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

```
var team = generator.team(point);
```

Label my random examples

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

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

Say whether my examples are red or blue

In generator():

```javascript
// 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 { points, team, examples };
```

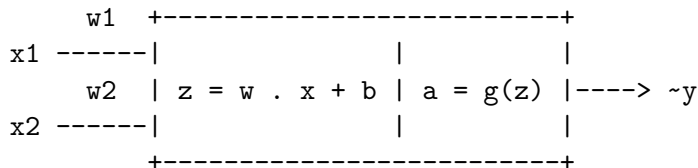# Make a guess based on x1, x2 whether a circle is red or blue

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

```
     w1  +------------------------+
 x1 ------|                        |          |
     w2  | z = w . x + b | a = g(z) |----> ~y
 x2 ------|                        |          |
         +------------------------+
```

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

`b` is the bias

`g` is our 'activation' function

`a / ~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. Focus on the perceptron
2. Discuss all elements leading to gradient descent

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)

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()`:

```javascript
// 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

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
```
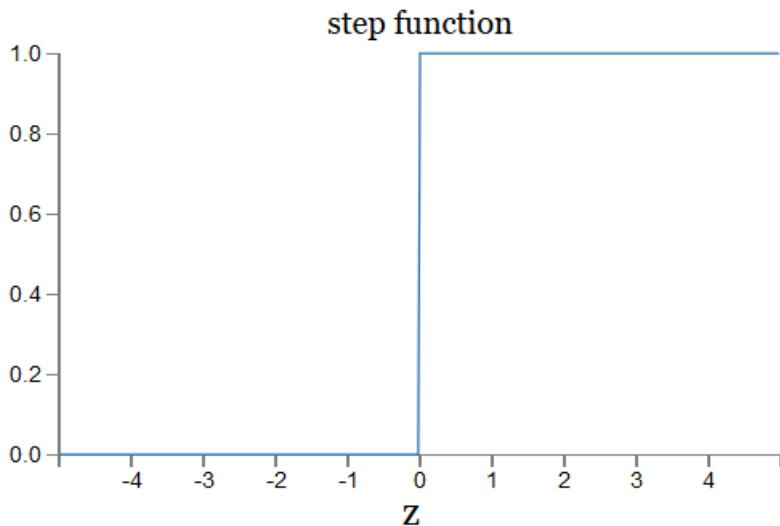
Activation graph

Figure 1: step (Neural Networks and Deep Learning [13])

Activation code

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

Add this to neuron():

```
// z = dot(w, x) + bias
// a = g(z)
function activation(z) {
  return z <= 0 ? 0 : 1;
}
```

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

Someone is having a laugh / it's genius $E = mc^2$

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

Make a first guess

Make a prediction from our weights

In `neuron()`:

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

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()`:

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

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?

Adjust the weights to improve my guess

Feed the difference back into the weights

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

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

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 a = prediction(w, x);
  // step 2b: update the weights for each x[i]
  var ydiff = diff(y, a);
  return [adjust(w, x, ydiff, 0), adjust(w, x, ydiff, 1)];
}
```

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()`

```
var example = generator.examples(1);
var weights = neuron.step(
  generator.weights, // w
  example[0].point, // x
  example[0].actual // y
);
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
```

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

(demo?)

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 };
```

Replace the guess with trained weights

In `build()` replace single step with:

```
var weights = neuron.train(
  generator.weights,
  generator.examples(400) // how many?
);
```

Have we gotten any better at guessing?

We can expand the count of examples and get a pretty perfect answer

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());
```

Re-use the examples

Used all the examples, but only once. Like revising with looking at your notes just once.

So it's a good idea to re-use them but… **bias**

Box with no senses + same examples over and over again

→ start noticing things that aren't relevant

A metric for how well the weights worked

First consider how well an individual example worked

A metric for how far a prediction is from the actual

Compare the Euclidean distance [12] between actual and predicted

y and a are both numbers on the number line, so we compare a
one-dimensional distance - the absolute difference

```
        +--+
-2 -1  0  1  2  3  4
// sqrt((y - a) ** 2) = abs(y - a)
function loss(w, example) {
  var predict = prediction(w, example.point);
  return Math.abs(example.actual - predict);
}
```

Average the metric across all examples

$$\frac{1}{m}\sum_{j}^{m}|y_j - g(z_j)|$$

Sum using the `reduce` function and then divide by the length

```
// the average of all loss functions
function cost(w, examples) {
  function sum(total, example) {
    // console.log({w, loss: loss(w, example)});
    return total + loss(w, example);
  }
  return (1 / examples.length) * examples.reduce(sum, 0);
}
```

Minimise this metric until it's good enough

Our `train` did one iteration through the examples

Now repeatedly iterate through until we reach a threshold

```javascript
// threshold when cost is low enough
// how many iterations (epochs)
function gradientDescent(w, examples, threshold, epochs) {
  var c = cost(w, examples);
  if (epochs < 0 || c < threshold) {
    return w;
  }
  return gradientDescent(train(w, examples), examples, thre
}

return { prediction, train, gradientDescent };
```

N.B. recursion - alternative to a `while` loop

Build it and give this process a name

In build(), replace the weights with:

```
var weights = neuron.gradientDescent(
  generator.weights,
  generator.examples(400),
  0.0001, // threshold
  100 // epochs
);
```

Once we have a function we can differentiate, this is called Gradient Descent.

We can't differentiate the activation function, but once we use the sigmoid function the differentiated variables become almost exactly the same

For perceptrons this is just called an iteration

Add some logging to the gradientDescent function to see progress

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

Improved this by applying multiple steps of 'gradient descent'

THE END (almost)
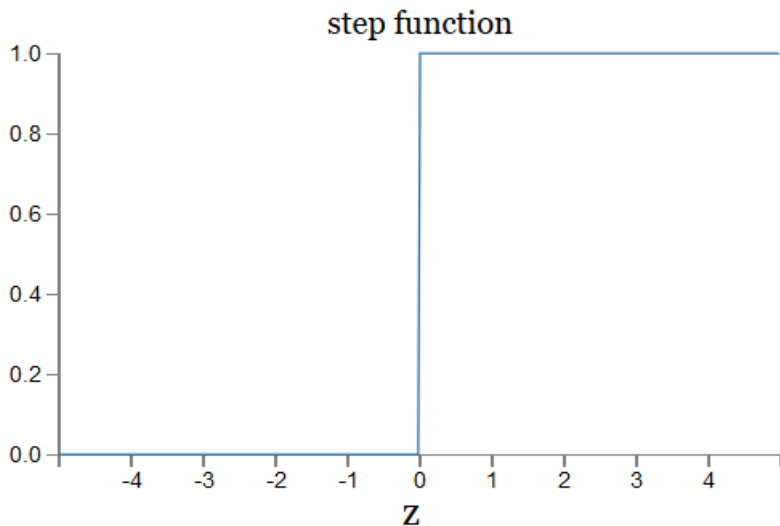
Perceptron (not differentiable)

Figure 2: step (Neural Networks and Deep Learning [13])
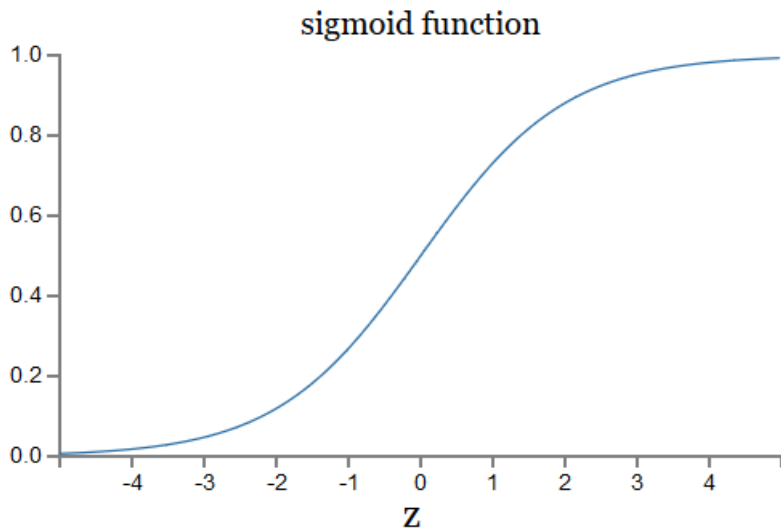
Sigmoid neuron (differentiable)

Figure 3: sigmoid (Neural Networks and Deep Learning [13])

Sigmoid in code

A new activation and new loss function

More complex, but they're still functions

$g(z) = 1/(1 + e^{-z})$

```
function activation(z) {
  return 1 / (1 + Math.exp(-z));
}
```

$L = -(y \log a + (1 - y) \log(1 - a))$

```
function loss(y, a) {
  return -(y * Math.log(a) + (1 - y) * Math.log(1 - a));
}
```

Gradient descent for sigmoid

Potential for a workshop - implement the following code

```
var J,
  dw1,
  dw2,
  db = 0;
for (var i = 0; i < m; i++) {
  var x_i = examples[i].point;
  var y_i = examples[i].actual;
  var z_i = dot(w, x_i) + b;
  var a_i = activation(z_i);
  J += loss(y_i, a_i);
  var dz_i = a_i - y_i; // diff function
  dw1 += x_i[0] * dz_i; // adjust function
  dw2 += x_i[1] * dz_i;
  db += dz_i;
}
```

Gradient descent part 2

```
J /= m;
dw1 /= m;
dw2 /= m;
db /= m; // averages
w1 = w1 - alpha * dw1; // learning rate alpha
w2 = w2 - alpha * dw2;
b = b - alpha * db;
```

The end

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

...

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) [7]