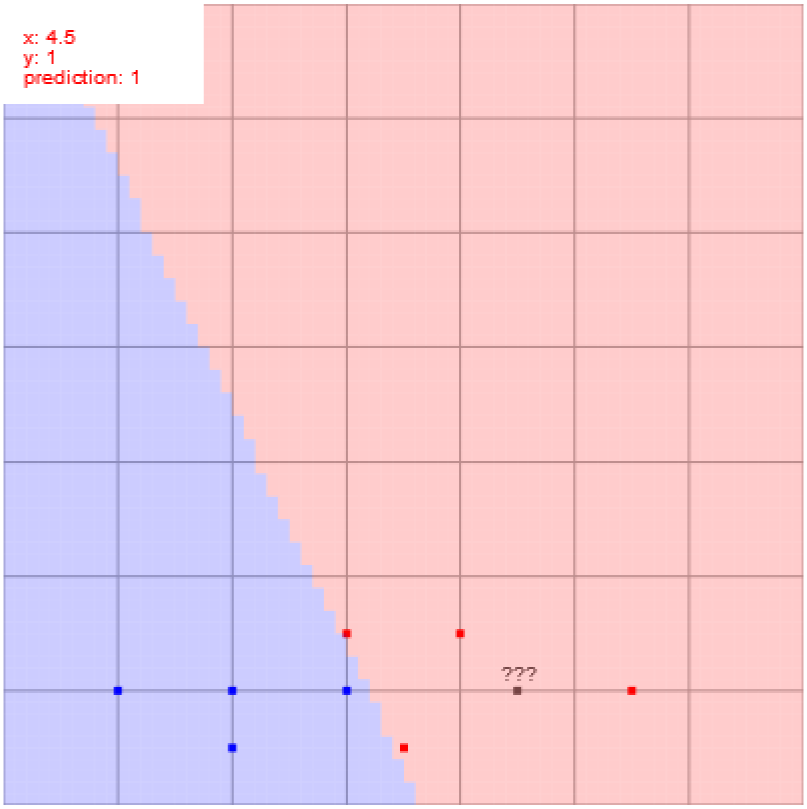
Made by Brandon Amezcua Mora – REVIEW OF giant_neural_network series of videos on Youtube

CODE EXPLANATION – JavaScript Algorithm

Part 0 – Defining or data



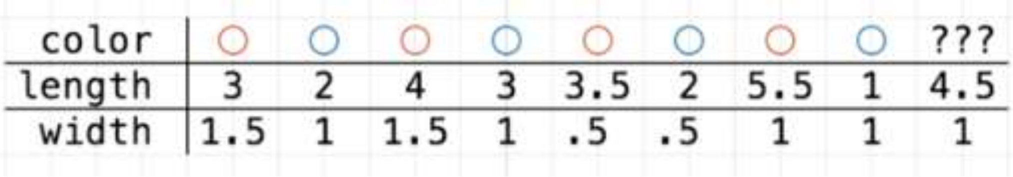
We first initialize our data so it can be represented easily understandable to the computer

This is one of the most important steps on using ML techniques. As our data sets are just numerically small in size and range. Although, our dataset has 3 types of features can simply represent them in a 2-dimensional axis plane, and a color.

Here’s our current dataset. 2-types of flowers. Where length of a petal is represented by the x-axis and width of the petals by y-axis. Note that for the last feature: color, we just used their respective colors: Blue and Red.

But as means to be of use to the computer, it needs to be a value it can hold, and that value is a normalization of a binary classification, in other words we represent this color as “points” by a value, let’s say 0 for blue, and 1 for red. Then we can represent them on a 3-dimensional space. where blue points are in the surface of the plane (0) and red points above a place that is above this plane one unit (+1). Let’s show then how this table of values has changed from a table understandable to us, to vectors that are simply represented by arrays in the computer.

Note that we want to predict a point which is not known, either is blue (0) or red (1). This point is not being used in this definition. All other points of know color are our true data values which will be used as training data, while our last value is a value to predict. If you knew the last data, you could improve this model, as it can adjust again its weights and therefore the point will be part of the training data.



color	🔴	🔵	🔴	🔵	🔴	🔵	🔴	🔵	???
length	3	2	4	3	3.5	2	5.5	1	4.5
width	1.5	1	1.5	1	.5	.5	1	1	1

Our objective it's to make adjustable weights that can predict our unknown values as means of making the task of classifying easily, this may be a trivial task for you, but there's no other implementation as means of manually engineering a model from scratch, neural networks have the advantage of coming with this unknown models automatically on the get-go. This is also very considerable, once you come to realize the amounts of data a computer is able to process in a short amount of time. This brute force approach it's a huge step towards many advances in technology and science (like the first image of a black hole, or the detection of tumors in medicine, as for our entertainment, customizable social media and streaming services with content based on our taste, perception: listening, seeing, speaking….of machines) and it is possible that it's a key technology in the coming challenges we are about to encounter (¿Thinking?).

JAVASCRIPT CODE - PART 1 – INCOMPLETE, CHECK LAST PAGES FOR THE FULL VERSION

```
//training set. [length, width, color(0=blue and 1=red)]
    var dataB1 = [1, 1, 0];
    var dataB2 = [2, 1,   0];
    var dataB3 = [2, .5, 0];
    var dataB4 = [3,   1, 0];

    var dataR1 = [3, 1.5, 1];
    var dataR2 = [3.5,   .5, 1];
    var dataR3 = [4, 1.5, 1];
    var dataR4 = [5.5,   1,   1];
// Unknown test value
    var dataU = [4.5,  1, "it should be 1"];
// Matrix: vector of vectors
     var all_points = [dataB1, dataB2, dataB3, dataB4, dataR1, dataR2,
dataR3, dataR4];
```

PART 1 – Defining the first functions for the training

As this algorithm is making a task know as, binary classification, where 2 features are exclusive each other (blue/red), it is convenient to use a proper activation function. This means some kind of "artificial" spark from the artificial neuron, as natural neurons make their responses by chaotic… and asynchronous activation, and for this purpose its commonly used an activation function called the sigmoid function. This function takes all real numbers, let it be every possible positive and negative numbers, i.e. -455, -210, 333, or 1, etc. And transforms them in either a negative number close to 0 or a positive number close 1, never reaching this numbers, except in their respective infinity. Remember e means exponential, and such has a value of 2.7183 approx. Then if $e^{-(-5)} = e^{5} = 148.41$

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

$$\text{sigmoid}(0) = \frac{1}{1 + e^{0}}$$

$$\text{sigmoid}(0) = 0.5$$

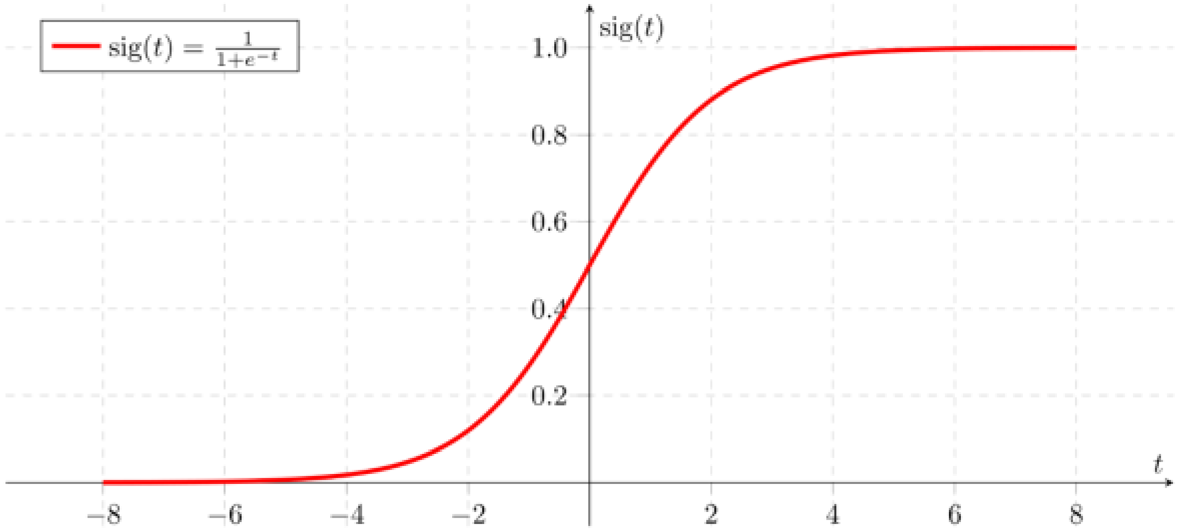
$$\text{sigmoid}(-5) = \frac{1}{1 + e^{-(-5)}}$$

$$\text{sigmoid}(-5) = \frac{1}{149.41}$$

$$\text{sigmoid}(-5) = 0.0067$$

Definition of our Sigmoid function on JavaScript – function sigmoid(x)

```
function sigmoid(x) {
  return 1/(1+Math.exp(-x));
}
```



JAVASCRIPT CODE - PART 1

```
// training
    function train() {
      let w1 = Math.random()*.2-.1;
      let w2 = Math.random()*.2-.1;
      let b = Math.random()*.2-.1;
      let learning_rate = 0.2;
      for (let iter = 0; iter < 50000; iter++) {
        // pick a random point
        let random_idx = Math.floor(Math.random() * all_points.length);
        let point = all_points[random_idx];
        let target = point[2]; // target stored in 3rd coord of points

        // feed forward
        let z = w1 * point[0] + w2 * point[1] + b;
        let pred = sigmoid(z);

        // now we compare the model prediction with the target
        let cost = (pred - target) ** 2;

        // now we find the slope of the cost w.r.t. each parameter (w1, w2, b)
        // bring derivative through square function
        let dcost_dpred = 2 * (pred - target);

        // bring derivative through sigmoid
        // derivative of sigmoid can be written using more sigmoids! d/dz
sigmoid(z) = sigmoid(z)*(1-sigmoid(z))
        let dpred_dz = sigmoid(z) * (1-sigmoid(z));

        // I think you forgot these in your slope calculation?
        let dz_dw1 = point[0];
        let dz_dw2 = point[1];
        let dz_db = 1;

        // now we can get the partial derivatives using the chain rule
        // notice the pattern? We're bringing how the cost changes through each
function, first through the square, then through the sigmoid
        // and finally whatever is multiplying our parameter of interest becomes
the last part
        let dcost_dw1 = dcost_dpred * dpred_dz * dz_dw1;
        let dcost_dw2 = dcost_dpred * dpred_dz * dz_dw2;
        let dcost_db =  dcost_dpred * dpred_dz * dz_db;

        // now we update our parameters!
        w1 -= learning_rate * dcost_dw1;
        w2 -= learning_rate * dcost_dw2;
        b -= learning_rate * dcost_db;
      }

      return {w1: w1, w2: w2, b: b};
    }
```

We will divide this function into 3 segments which are: Part 1.1 Weights initialization (Neural Net definition – Artificial brain creation), Part 1.2 - Training iteration and Forward and backward (Artificial neuron synapsis) propagation algorithms, and finally, Part 1.3 - Weight (Spark potential in the brain) adjustment. That will improve over time from a random set of values to a good model of the data…

1.1 - Weights initialization.

JAVASCRIPT CODE - PART 1.1

```
      let w1 = Math.random()*.2-.1;
      let w2 = Math.random()*.2-.1;
       let b = Math.random()*.2-.1;
```

This is the most self-explanatory to understand part of the function and the shortest one. w1, w2 are weights for our neural network. And b is a term latter treated, it is a constant known as bias. As we can see in the image 3. The weights are values for a hidden layer in the neural network.

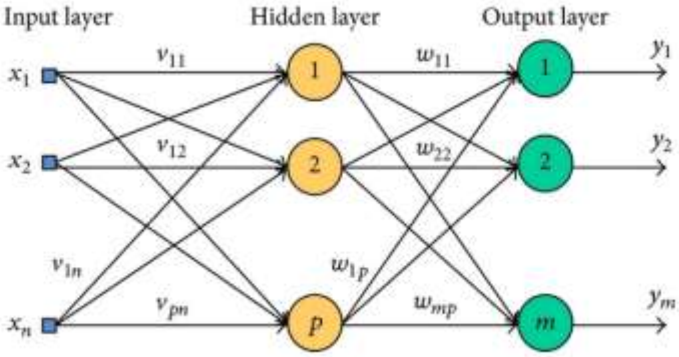


Figura 1 Imagen de red neural con una capa de datos de entrada (sentidos o sensores), una capa escondida o "hidden" (neuronas) y una capa de salida (estimulo respuesta en acción o modelado de la información – nuevas conexiones)

Since we have 2 features (length and width) and 1 single output (color) since we are doing a binary classification task. The neural network has 2 neurons as an input layer, every neuron connects to a hidden layer neuron. We could have more hidden layers or neurons (columns or rows) if we want, and I'm not completely sure if less hidden neurons would make a bad prediction model, but the rumor says that the number of layers does affect the better performance of the neural networks, a subdiscipline of machine learning called Deep Learning takes this approach. Since matrix multiplication is a task highly demanding in resources and time for computers, but since machines are getting more powerful this is able to do more efficiently as every passing day takes less time and way easier to manage lots of data manipulation and operations. This was the reason this kind of approaches of AI became at a low pace just about 60 years ago as 2020, making really serious strikes for their time, they (neural networks) were debunked and left behind since in an experiment while trying to predict the function outcomes of all the values in logic functions, the perceptron (single hidden layer neuron) cannot come to the solution of the XOR logic gate, becoming (The XOR Problem in Neural Networks.), turns out, when you add a second layer to a single layer neuron, you are able to get the function prediction just right...

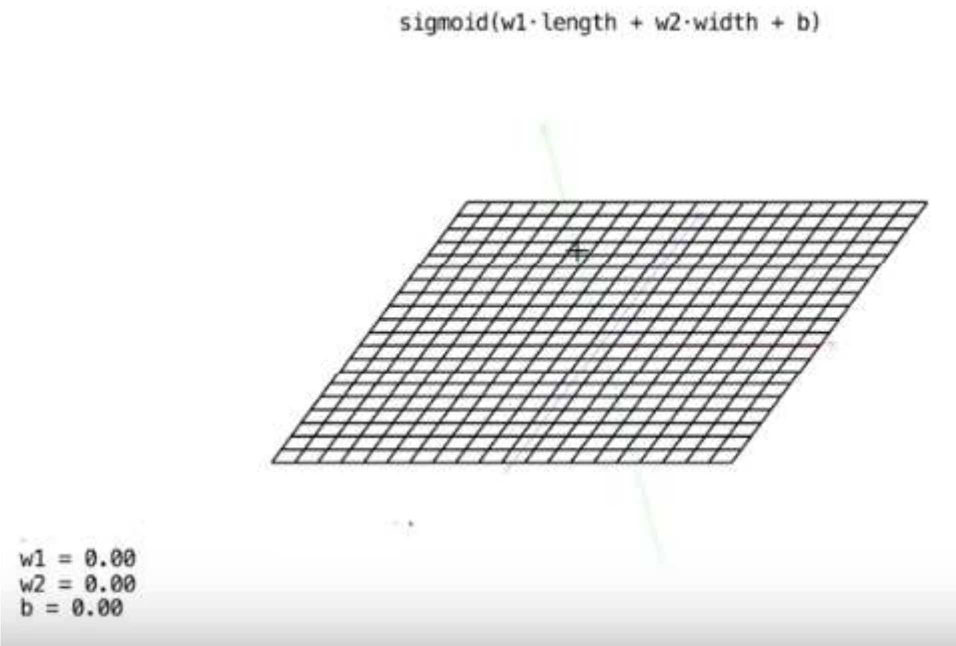
1.1.1. BIAS (b parameter)

Since a random number in weights can trigger all neurons, we need to make sure that with each iteration and model optimization (weight adjusting) we are triggering more or less features in consideration. As means of simplicity, the middle terms must overcome a threshold… So it makes an accuracy prediction with enough certainty. So to speak. “I think that biases are almost always helpful. In effect, a bias value allows you to shift the activation function to the left or right, which may be critical for successful learning.” Also, there's another reason to it, more on that later.

```
let b = Math.random()*.2-.1;
```

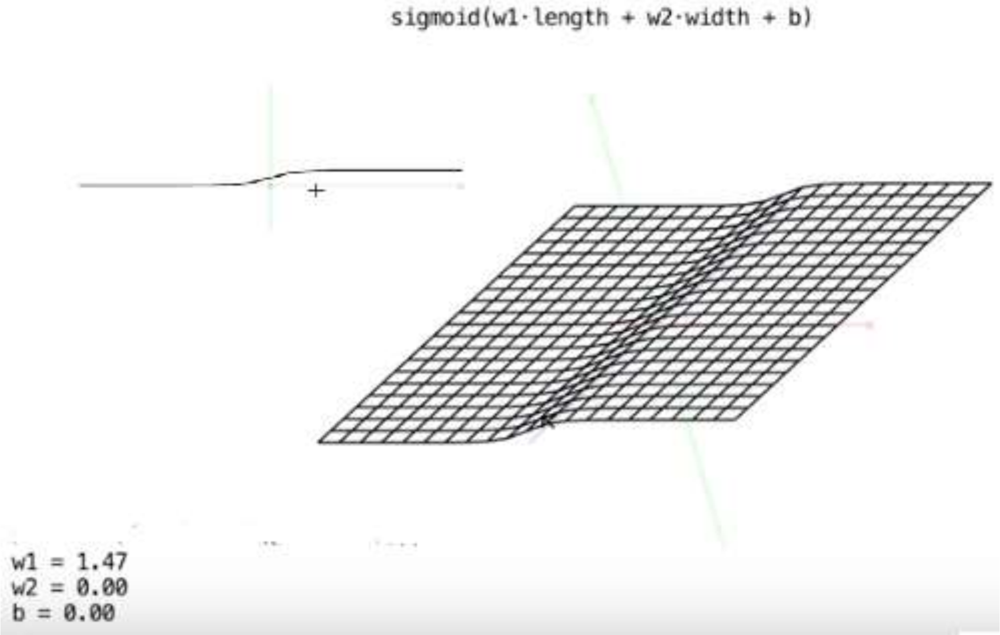
1.1.2. Dimension in features (Bidimensional space)

Imagine our grid world, where we have 2-dimensions, width and length. We are plotting our activations function $\frac{1}{1+e^{-x}}$ in this bidimensional space. As shown in figure 4. The argument $x = w_1 \cdot lenght + w2 \cdot width + d$ of the sigmoid will be a key concept as we progress.



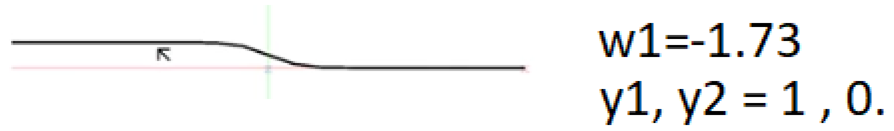
w1 = 0.00
w2 = 0.00
b = 0.00

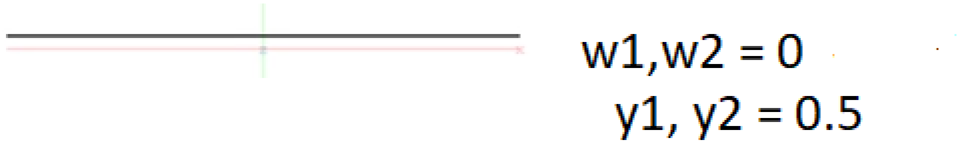
Each axis on the plane it’s a feature (width, length of petal) or a parameter in the Sigmoid function, currently, seems there are no changes in the plane, lets change some of the weights and see the result in this plane. We can see in Figure 5 that a change in the weights makes the plane have a familiar shape.



w1 = 1.47
w2 = 0.00
b = 0.00

Same goes if we change some other parameter, we get this result.





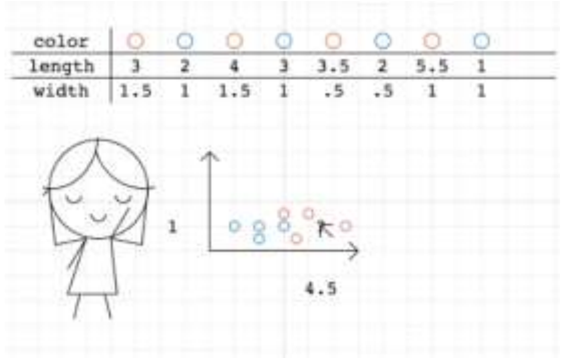
Now let’s go back at our parameter x, as we said earlier this parameter is key in understanding what is going on inside the neural network. And we have to explain another function for doing so.

1.1.3. COST FUNCTION OR EVALUTATION (OR THE ERROR) FUNCTION.

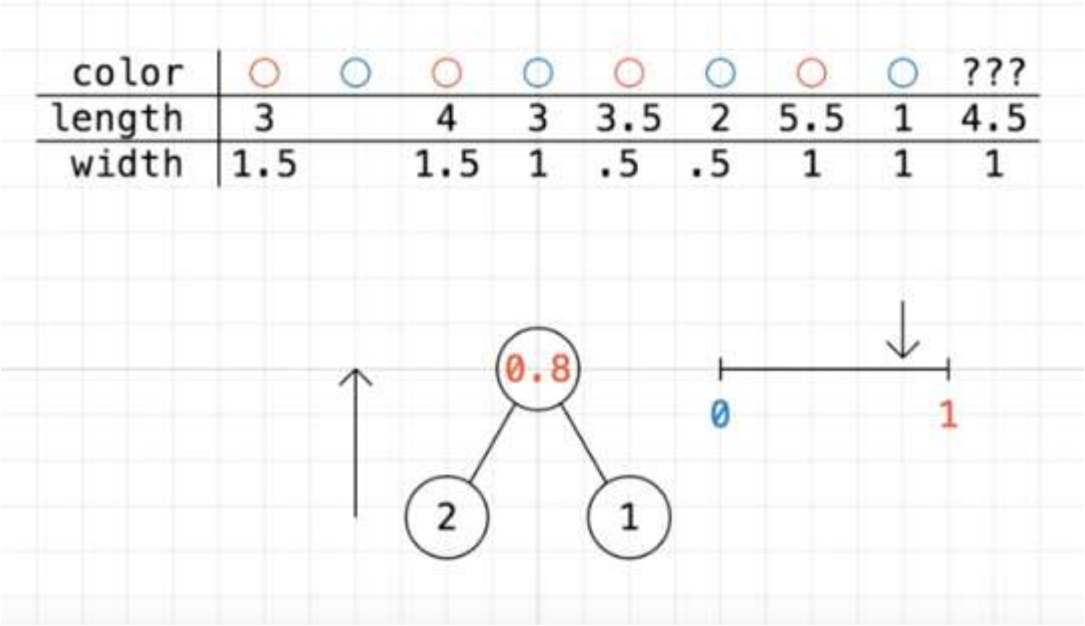
Once a farmer of flowers makes use of the data and tries to separate a line between the plotted data, she can guess the unknown flower is a red flower

She has done the task, all by her self and taking her time to make the plot. But also if it can make a computer make this task to, in an automated way, when data is really big to process or graph

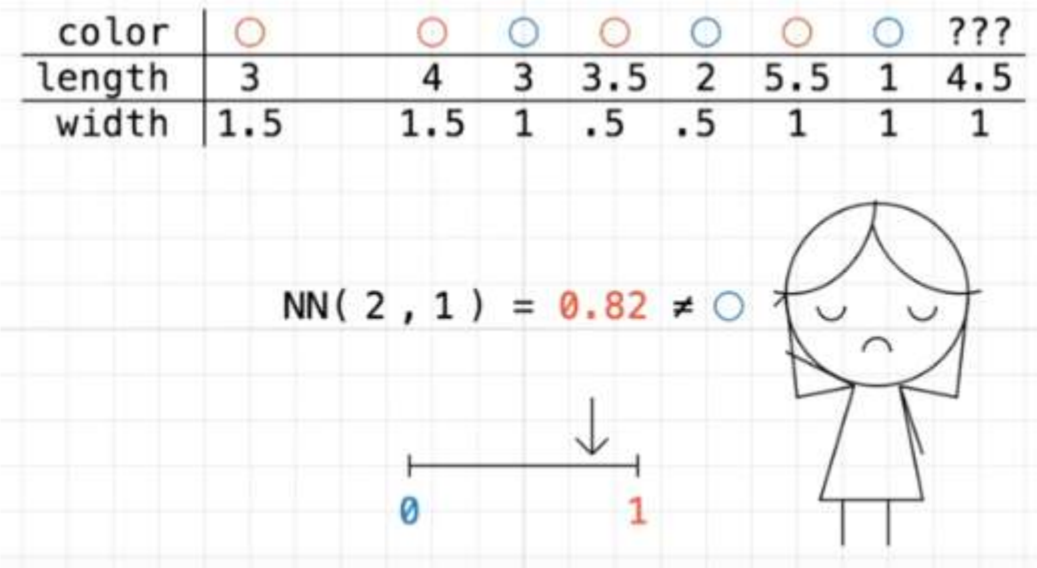
When she feeds the data to the random neurons in the artificial neural network in the computer. She will evaluate the performance of the guesses of the computer.



color	○	○	○	○	○	○	○	○
length	3	2	4	3	3.5	2	5.5	1
width	1.5	1	1.5	1	.5	.5	1	1

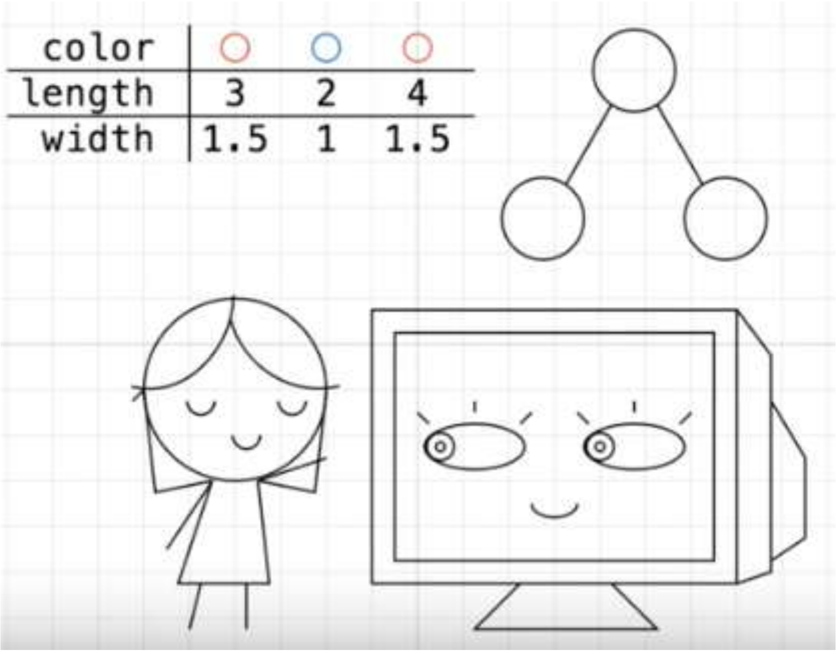


color	○	○	○	○	○	○	○	○	???
length	3		4	3	3.5	2	5.5	1	4.5
width	1.5		1.5	1	.5	.5	1	1	1

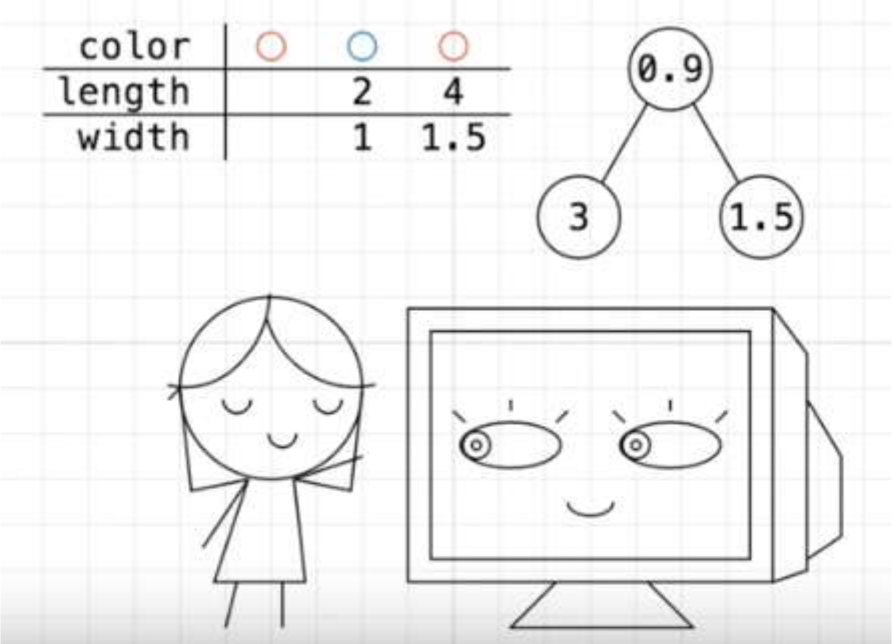


color	○	○	○	○	○	○	○	???
length	3	4	3	3.5	2	5.5	1	4.5
width	1.5	1.5	1	.5	.5	1	1	1

And makes sense it gets it wrong since the neural network had random initializing numbers. If we could change these weights inside the computer maybe we could get the computer to rightly guess the numbers

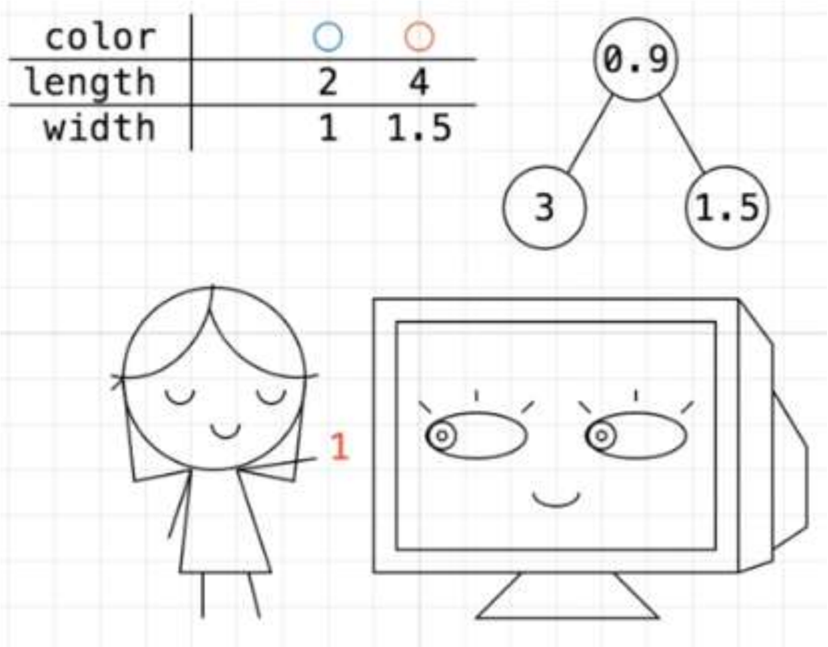


color	○	○	○
length	3	2	4
width	1.5	1	1.5



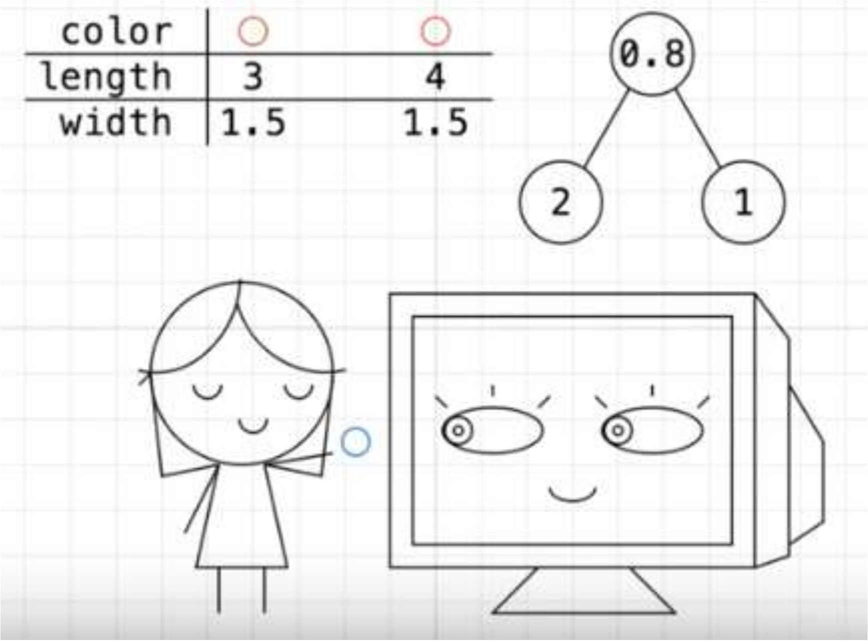
color	○	○	○
length		2	4
width		1	1.5

For the first guess, the computer got it right!



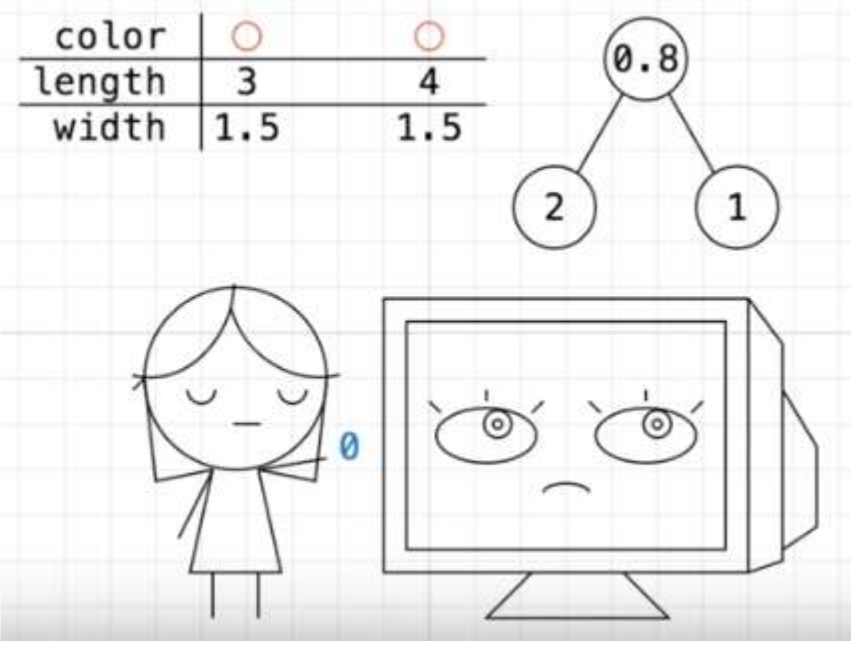
color	○	○
length	2	4
width	1	1.5

Great job computer! Even though your guesses are completely random, you could guess that this flower is red. Although we need it to get it as close to 1 as possible.

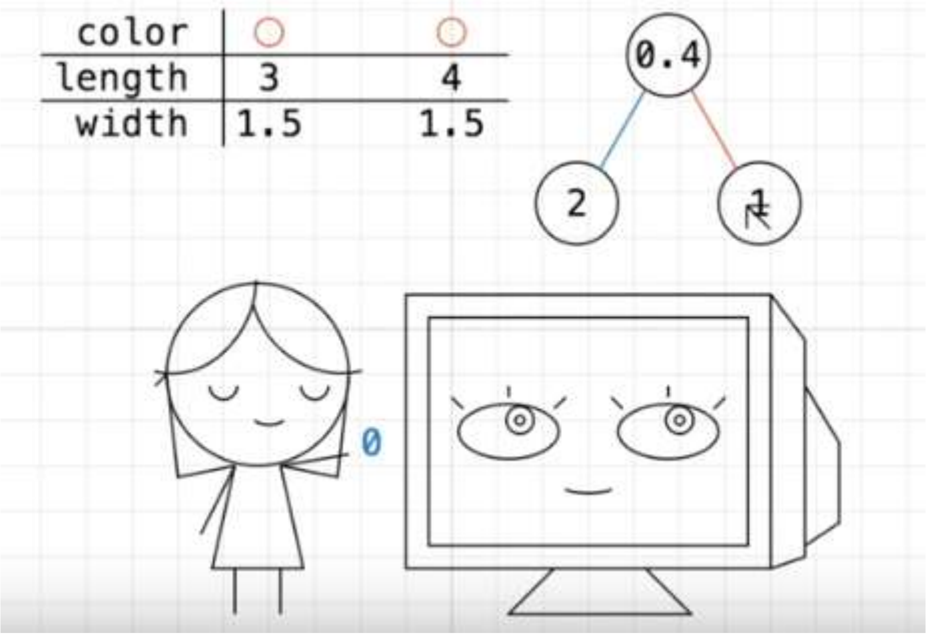


color	○	○
length	3	4
width	1.5	1.5

For the second guess, things don’t seem so good…



color	○	○
length	3	4
width	1.5	1.5



color	○	○
length	3	4
width	1.5	1.5

The right way of changing its weights is by training the neural network with more examples, and the more of our examples or data, the best performances we seem able to get.

But we need an automated way to do the function of the evaluation that makes the farmer, a mathematical function that can make this evaluation for her. Taking the inputs of data and the output, some sort of feedback between those.

Let’s say you are a Data Scientist, and your job is to make plots of a dataset and find some possible approximations or hidden functions. If you recall the plot from our red and blue flowers, seems to be some sort of patter in the way we cluster the points inside our heads. That is a model we try to construct in the computer. If you could come to the representation of this model, what would you use between a line and a circle?

We have come to a solution in the field of statistic, since we manage truths we are not making guesses without clue, the data leave traces of possible outcomes, and the more the data, the better the model it predicts, we use a function called square medium error to make an estimation of the variance of the error.

In any kind of automatic or self-driven process, it is common the measure of error, the difference between the outputs and the inputs. We want to compare the expected output value with the obtained output and if the difference in this error makes a measure for how much the system needs to adjust his parameters so this difference be equal to zero. This make cause a robot to stay balanced even in the presence of disturbances or a computer to self-program once certain condition make it change its state of equilibrium…

The field of Statistics has come with a right evaluation function, an estimator (of a procedure for estimating an unobserved quantity), that consist in the average of the squares of the errors—that is, the average squared difference between the estimated values and what is estimated. This formula it’s called then mean squared error (MSE) or mean squared deviation (MSD) of an estimator…

The MSE is a measure of the quality of an estimator—its always positive, and values closer to zero are better.

Let it be our function for the evaluation of the error or cost function be this definition:

$$cost\ function = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

Where:

$$Y_i = vector\ of\ observed\ values\ of\ the\ variable\ being\ predicted$$

$$\hat{Y}_i = vector\ of\ predicted\ values$$

$$n = number\ of\ iterations$$

1.1.4. LINEAR REPRESENTATION

If you recall the value, we pass into our sigmoid function

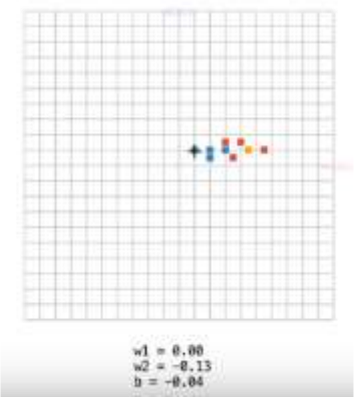
$$x = w_1 \cdot y + w_2 \cdot z + b$$

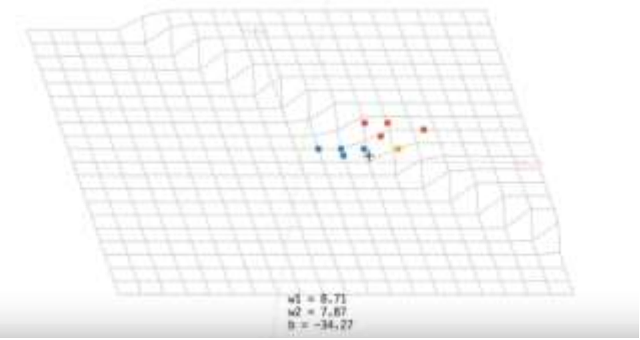
It’s a simple linear equation, and it’s no surprise results way to familiar. Let it be our weights a single vector and the axis a single plane function…

If we graph the points in the plane we had before in figure, we see it’s the formula of a plane, more specifically

$$x = \overrightarrow{W} \cdot \overrightarrow{yz} + b$$

It’s even more familiar to the function of the slope of a curve or the formula of a straight line...





We adjust the weights and bias so the data can it can be separated by a line, diferent values or this model, more especifically by different altitudes in the graph (0, blue) and (1, red). and if our activation function (Sigmoid) is the form of the plane bisected in half, follows a patter linear that fits with our data. The function that correlates both of our features, seems to be linear.

Since this line fits himself once the evaluation value reaches zero, we must come with a measure of which weight values changes and how to change them.

THE SLOPE OF THE COST FUNCTION.

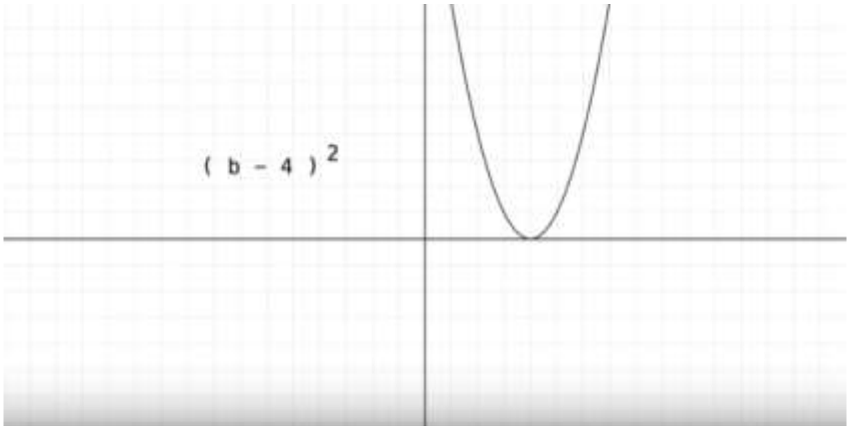
If you recall our cost function for measuring the quality of our estimations, would look like this

$$cost\ function = \frac{1}{n}(prediction - target)^2$$

Lets supose that the taget value of desired output its 4, and for the sake of simplicity lets make our weights equal to zero…

$$cost = (b - 4)^2$$

Suppose then that we graph this cost function in a 2-dimensional axis, where our x axis represents the bias and the vertical axis represents the cost function.

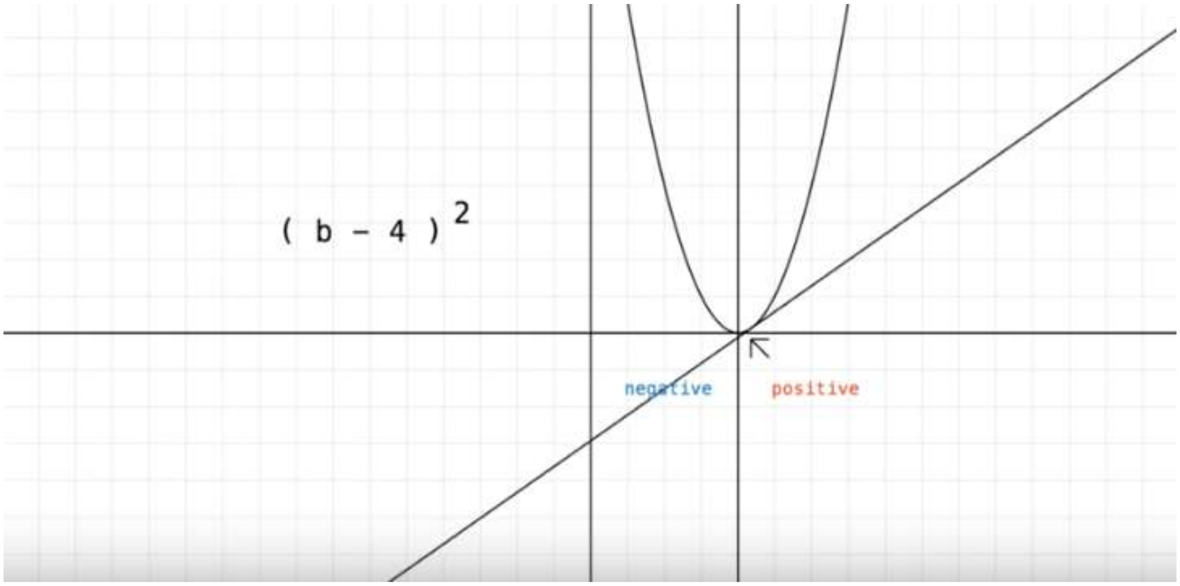


Since the cost function cannot be ever negative, reaches the zero error, the correct interpretation of the guess value at the lowest point in the graph, also called the global minima.

1.1.5. LEARNING RATE

How to move from a random point (b, cost(b)) in the cost function to a global minimum. What can we use to get to the desired value of cost function or error evaluation to zero?

Look the value of the slope in the figures.



The slope is the change in position of the infinitesimal junction between two points… in other words, if we move a straight line in the curve in a single point, the line will be either positive if its oriented to the right or negative if its oriented to the left, this characteristic can be used as means to know the value of position in a random point from this parabolic function and how close are to the global minimum.

The answer lies in the definition of a slope equal to zero. Which it’s known in mathematics as, its derivative.

Since we use a linear mean to move around the activation function in the plane as means to divide our data in a correct model. The adjustments of this weights will finally make its purpose as the evaluation function reaches zero. As these weights are modified, they are learning, once they are less and less errors on the evaluation, we say out system is ready to make a good guess.

Let us then obtain the rate at which the evaluation will decrease its learning, also known as…

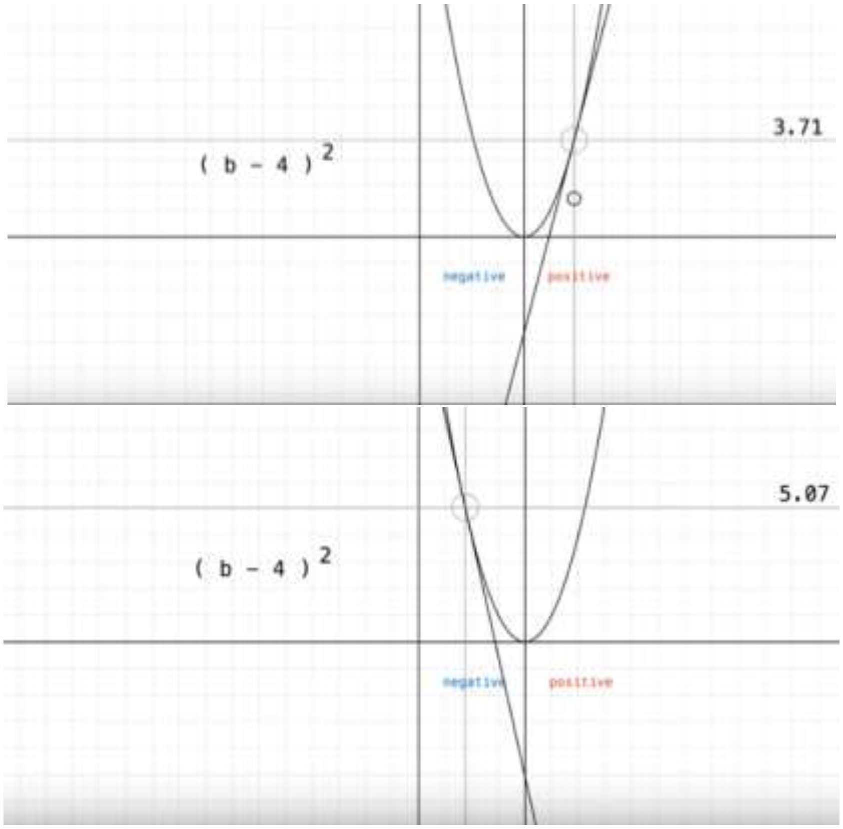
LEARNING RATE (OR ALPHA CONSTANT) & LINEAR REGRESSION

```
      let learning_rate = 0.2;
```

The learning rate can be seen as the way linear regression works, linear regression it’s the brute force way to reach a minimum point in a multidimensional plane by obtaining calculations of the curve at the current point, specifically, the partial derivative, in the case of 2 or more features or the derivative in the case of a single feature. This kind of approach its computationally tractable since it’s a derivative of single values on current and previous data points. You can imagine this, as punching a ball or marble down the parabola, if falls slowly, won’t be oscillating from one side of the curve to the other. See the next figure. You should be careful with this parameter, since it can overshoot and will never reach a local minimum, it must be small enough to make it to the down position easily and fast enough, but not so small it takes a lot of time to reach the minimum so slow that it even takes more iterations or steps that its currently running or to be practical in the task at hand.

Parabola Slope: Straight-line

$$\text{cost}(b) = (b-4)^{2} \nwarrow \quad \frac{d}{db}\text{cost}(b) = 2(b-4)$$



Part 1.2 - Training iteration and Forward and backward propagation algorithms

JAVASCRIPT CODE - PART 1.2.1 – FORWARD PROPAGATION

The next code its self-explanatory once we have talked about in the previous section

```
for (let iter = 0; iter < 50000; iter++) {
        // pick a random point
        let random_idx = Math.floor(Math.random() * all_points.length);
        let point = all_points[random_idx];
        let target = point[2]; // target stored in 3rd coord of points

        // feed forward
        let z = w1 * point[0] + w2 * point[1] + b;
        let pred = sigmoid(z);

        // now we compare the model prediction with the target
         let cost = (pred - target) ** 2;
```

The first sentence is the loop required to make the iteration for the process of training, it has been chosen to be made out of 50,000 iterations. Note the indentation that makes the next sentences part of the loop of 50,000 epocs or iterations, this means will repeat that number of times.

The next is a comment and 3 declaration of local variables named respectively random_idx, point and target. The random index choses a random float point decimal number (Math.random()) from zero to one, to make it of the length number of the vector of data values (all_points.lenght()), the condition for these (Math.random() * all_points.length) numbers is that they have to be integers, if not, they will become one by applying the function Math.floor().

This random integer will be used to choose a random point value from our data representation, as show in the local variable value let point = all_points[random_idx]; This point is used again to obtain their respective target value or desired output. Color value (0 or 1) that its located in the third position of the point vector, remember that in an array, we start the first position of the values from zero to n-1. i.e. if random index its 3, corresponds to the point in our dataset matrix chosen is point[3] which it's not the third array in the vector but the forth. `var dataB4 = [3,   1, 0]; the target[2] = dataB4[2] = 0`. This corresponds to the blue color.

The feed-forward propagation of the network it's represented by the formula of the plane:

let z = w1 * point[0] + w2 * point[1] + b;

Note that since we only have 2 weights in the hidden network and 2 parameters, width and length, we have a plane formula. The multiplication of each one, makes the neuron sparkling or value of potential between connections that give form to the unknown model. It's amazing how simple yet such conclusions… These values will continue to repeat and sum themselves if there were more hidden layers between weights. So for this case, we have finished and its time for the activation function to pass the sparks to the next set of neurons

And the activation function:

let pred = sigmoid(z);

Whose parameter value is the previous feed-forward propagation from the input values to the next layer of weights, or in case being the last, the output values. Pred in this case is our output.

Now we compare the output value with the cost function or evaluation of error.

let cost = (pred - target) ** 2;

As we mention earlier, this is the square mean error, fitted to value quality of prediction …

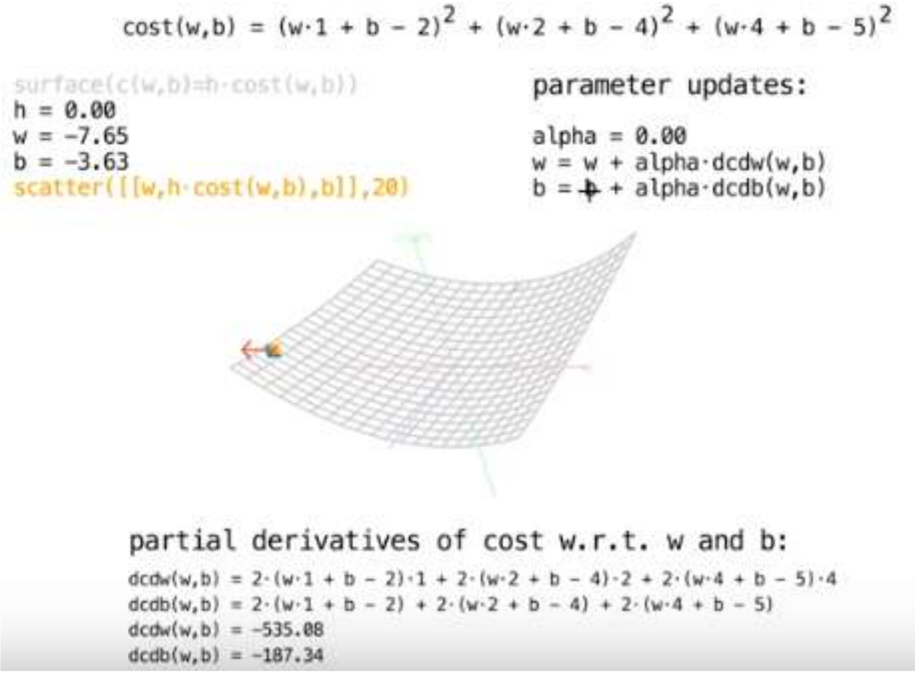
JAVASCRIPT CODE - PART 1.2 – BACKWARD PROPAGATION

```
       // now we find the slope of the cost w.r.t. each parameter (w1, w2, b)
        // bring derivative through square function
        let dcost_dpred = 2 * (pred - target);

        // bring derivative through sigmoid
        // derivative of sigmoid can be written using more sigmoids! d/dz
sigmoid(z) = sigmoid(z)*(1-sigmoid(z))
        let dpred_dz = sigmoid(z) * (1-sigmoid(z));

        // I think you forgot these in your slope calculation?
        let dz_dw1 = point[0];
        let dz_dw2 = point[1];
        let dz_db = 1;


        // now we can get the partial derivatives using the chain rule
        // notice the pattern? We're bringing how the cost changes through each function,
first through the square, then through the sigmoid
```

```
        // and finally whatever is multiplying our parameter of interest becomes the last
part
        let dcost_dw1 = dcost_dpred * dpred_dz * dz_dw1;
        let dcost_dw2 = dcost_dpred * dpred_dz * dz_dw2;
        let dcost_db =  dcost_dpred * dpred_dz * dz_db;

        // now we update our parameters!
        w1 -= learning_rate * dcost_dw1;
        w2 -= learning_rate * dcost_dw2;
        b -= learning_rate * dcost_db;
      }

      return {w1: w1, w2: w2, b: b};
    }
```

BACK PROPAGATION

The magic happens, the impulses have gone to thought the neural network, but they must change in order to reduce the error in the cost function 2-dimensional plot, making big changes and small changes as is cost function is squared and time passes.



```
cost(w,b) = (w·1 + b - 2)^2 + (w·2 + b - 4)^2 + (w·4 + b - 5)^2

surface(c(w,b)=h·cost(w,b))
h = 0.00
w = -7.65
b = -3.63
scatter([[w,h·cost(w,b),b]],20)

parameter updates:

alpha = 0.00
w = w + alpha·dcdw(w,b)
b = b + alpha·dcdb(w,b)

partial derivatives of cost w.r.t. w and b:
dcdw(w,b) = 2·(w·1 + b - 2)·1 + 2·(w·2 + b - 4)·2 + 2·(w·4 + b - 5)·4
dcdb(w,b) = 2·(w·1 + b - 2) + 2·(w·2 + b - 4) + 2·(w·4 + b - 5)
dcdw(w,b) = -535.08
dcdb(w,b) = -187.34
```

The only direction data we have is the random point in space, and the partial derivatives of the cost function respect to their weights and bias. Let’s obtain their respective partial derivatives in the program

$$x = w_1 \cdot y + w_2 \cdot z + b \quad (Feed - Foward)$$

$$pred = sigmoid(x) = \frac{1}{1 + e^{-x}} \; (Activation\ function)$$

$$cost(w, b) = \frac{1}{n}(pred - target)^2$$

If you do enough iterations each value is less effective than the first ones so…

$$cost(w, b) = (pred - target)^2$$

We have in our code the next formulas to obtain the partial derivative of the cost function, these expressions follow the chain rule for derivatives:

$$\frac{d\ cost(w_1, w_2, b)}{dw_1} = \frac{d\ cost(w_1,\ w_2, b)}{pred} \cdot \frac{d\ pred}{dz} \cdot \frac{d\ z}{d\ w_1}$$

$$\frac{d\ cost(w_1, w_2, b)}{dw_2} = \frac{d\ cost(w_1,\ w_2, b)}{pred} \cdot \frac{d\ pred}{dz} \cdot \frac{d\ z}{d\ w_2}$$

$$\frac{d\ cost(w_1, w_2, b)}{d\ b} = \frac{d\ cost(w_1,\ w_2, b)}{pred} \cdot \frac{d\ pred}{dz} \cdot \frac{d\ z}{d\ b}$$

We can calculate some expressions due to the simplified cost function

$$\frac{d\ cost(w_1,\ w_2, b)}{pred} = 2(pred - target)(1)$$

```
         // now we find the slope of the cost w.r.t. each parameter (w1, w2, b)
        // bring derivative through square function
        let dcost_dpred = 2 * (pred - target);
```

$$\frac{d\ pred}{dz} = \frac{d}{dz} sigmoid(x) = (1 - sigmoid(z))$$

```
         // bring derivative through sigmoid
        // derivative of sigmoid can be written using more sigmoids! d/dz
sigmoid(z) = sigmoid(z)*(1-sigmoid(z))
```

$$z = w_1 \cdot x + w_2 \cdot y + b$$

$$\frac{d\ z}{d\ w_1} = x = with\ of\ the\ current\ point$$

$$\frac{d\ z}{d\ w_2} = y = lenght\ of\ the\ current\ point$$

$$\frac{d\ z}{d\ b} = 1$$

```
        // Current point of the random point
        let dz_dw1 = point[0];
        let dz_dw2 = point[1];
        let dz_db = 1;
```

Part 1.3 - Weight (Spark potential in the brain) adjustment.

We make updates backpropagating the information from the cost or error evaluation function through all the weights by subtracting each of the derivatives (orientation of the point to each plane axis) by a small percentage. This magnitude will either be positive or negative, depending on the value of the current slope. But it will be gradually proportional to our chosen learning rate also known as alpha constant (α).

```
        // now we update our parameters!
        w1 -= learning_rate * dcost_dw1;
        w2 -= learning_rate * dcost_dw2;
        b -= learning_rate * dcost_db;
      }

      return {w1: w1, w2: w2, b: b};
```

We repeat this process until all weights and biases are well adjusted to be near as possible the global minima. After all iteration, we can say that the system has learned the correct interpretation of data and his respective classification model by predictions. If you save the values for w1, w2 and b values in a file, they will perform as good as the last iteration has been trained for flower identification.

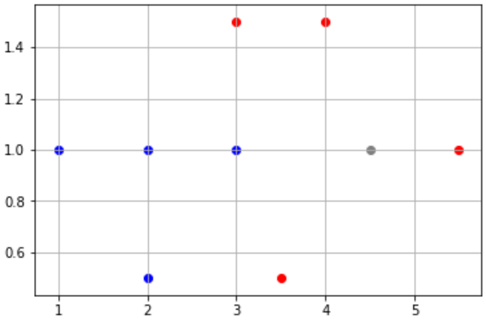
FINAL JAVASCRIPT CODE - COMPLETE

The remaining JavaScript code makes a web API that let us see the canvas module for the plotting of these points along with their respective model classification. This includes as well a Python implementation, if you are not able to do/run it in a web development environment.

```
//training set. [length, width, color(0=blue and 1=red)]
    var dataB1 = [1, 1, 0];
    var dataB2 = [2, 1,   0];
    var dataB3 = [2, .5, 0];
    var dataB4 = [3,   1, 0];

    var dataR1 = [3, 1.5, 1];
    var dataR2 = [3.5,   .5, 1];
    var dataR3 = [4, 1.5, 1];
    var dataR4 = [5.5,   1,   1];

    //unknown type (data we want to find)
    var dataU = [4.5,  1, "it should be 1"];

    var all_points = [dataB1, dataB2, dataB3, dataB4, dataR1, dataR2, dataR3, dataR4];

    function sigmoid(x) {
      return 1/(1+Math.exp(-x));
    }

    // training
    function train() {
      let w1 = Math.random()*.2-.1;
      let w2 = Math.random()*.2-.1;
      let b = Math.random()*.2-.1;
      let learning_rate = 0.2;
      for (let iter = 0; iter < 50000; iter++) {
        // pick a random point
        let random_idx = Math.floor(Math.random() * all_points.length);
        let point = all_points[random_idx];
        let target = point[2]; // target stored in 3rd coord of points

        // feed forward
        let z = w1 * point[0] + w2 * point[1] + b;
        let pred = sigmoid(z);

        // now we compare the model prediction with the target
        let cost = (pred - target) ** 2;

        // now we find the slope of the cost w.r.t. each parameter (w1, w2, b)
        // bring derivative through square function
        let dcost_dpred = 2 * (pred - target);

        // bring derivative through sigmoid
        // derivative of sigmoid can be written using more sigmoids! d/dz sigmoid(z) =
sigmoid(z)*(1-sigmoid(z))
        let dpred_dz = sigmoid(z) * (1-sigmoid(z));

        // I think you forgot these in your slope calculation?
        let dz_dw1 = point[0];
        let dz_dw2 = point[1];
        let dz_db = 1;

        // now we can get the partial derivatives using the chain rule
        // notice the pattern? We're bringing how the cost changes through each function,
first through the square, then through the sigmoid
        // and finally whatever is multiplying our parameter of interest becomes the last
part
        let dcost_dw1 = dcost_dpred * dpred_dz * dz_dw1;
        let dcost_dw2 = dcost_dpred * dpred_dz * dz_dw2;
        let dcost_db =  dcost_dpred * dpred_dz * dz_db;

        // now we update our parameters!
        w1 -= learning_rate * dcost_dw1;
        w2 -= learning_rate * dcost_dw2;
        b -= learning_rate * dcost_db;
      }

      return {w1: w1, w2: w2, b: b};
    }

    let canvas = document.createElement("canvas");
    canvas.width = 400;
    canvas.height = 400;
    document.body.appendChild(canvas);
    let ctx = canvas.getContext("2d");
    ctx.font = "Helvetica";

    // map points from graph coordinates to the screen
    let graph_size = {width: 7, height: 7};
    function to_screen(x, y) {
      return {x: (x/graph_size.width)*canvas.width, y: -(y/graph_size.height)*canvas.height
+ canvas.height};
    }

    // map points from screen coordinates to the graph
    function to_graph(x, y) {
      return {x: x/canvas.width*graph_size.width, y: graph_size.height -
y/canvas.height*graph_size.height};
    }

    // draw the graph's grid lines
    function draw_grid() {
      ctx.strokeStyle = "#AAAAAA";
      for (let j = 0; j <= graph_size.width; j++) {

        // x lines
        ctx.beginPath();
        let p = to_screen(j, 0);
        ctx.moveTo(p.x, p.y);
        p = to_screen(j, graph_size.height);
        ctx.lineTo(p.x, p.y);
        ctx.stroke();

        // y lines
        ctx.beginPath();
        p = to_screen(0, j);
        ctx.moveTo(p.x, p.y);
        p = to_screen(graph_size.width, j);
        ctx.lineTo(p.x, p.y);
        ctx.stroke();
      }
    }

    // draw points
    function draw_points() {
        // unknown
        let p = to_screen(dataU[0], dataU[1]);
        ctx.fillStyle = "#555555";
        ctx.fillText("???", p.x-8, p.y-5);
        ctx.fillRect(p.x-2, p.y-2, 4, 4);

        // draw points
        ctx.fillStyle = "#0000FF";
        for (let j = 0; j < all_points.length; j++) {
          let point = all_points[j];
          if (point[2] == 0) {
            ctx.fillStyle = "#0000FF";
          } else {
            ctx.fillStyle = "#FF0000";
          }
          p = to_screen(point[0], point[1]);
          ctx.fillRect(p.x-2, p.y-2, 4, 4);
        }
    }

    // visualize model output on grid of points
    function visualize_params(params) {
      ctx.save();
      ctx.globalAlpha = 0.2;
      let step_size = .1;
      let box_size = canvas.width/(graph_size.width/step_size);

      for (let xx = 0; xx < graph_size.width; xx += step_size) {
        for (let yy = 0; yy < graph_size.height; yy += step_size) {
          let model_out = sigmoid( xx * params.w1 + yy * params.w2 + params.b );
          if (model_out < .5) {
            // blue
            ctx.fillStyle = "#0000FF";
          } else {
            // red
            ctx.fillStyle = "#FF0000";
          }
          let p = to_screen(xx, yy);
          ctx.fillRect(p.x, p.y, box_size, box_size);
        }
      }
      ctx.restore();
    }

    // find parameters
    var params = train();

    // visualize model output
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    draw_grid();
    draw_points();
    visualize_params(params);

    // say what the model would say for a given mouse position
    window.onmousemove = function(evt) {
      ctx.clearRect(0, 0, 100, 50);

      let p = {x: 10, y: 20};

      let mouse = {x: evt.offsetX, y: evt.offsetY};
      let mouse_graph = to_graph(mouse.x, mouse.y);

      ctx.fillText("x: " + Math.round(mouse_graph.x*100)/100, p.x, p.y);
      ctx.fillText("y: " + Math.round(mouse_graph.y*100)/100, p.x, p.y + 10);
      // model output
      let model_out = sigmoid( mouse_graph.x * params.w1 + mouse_graph.y * params.w2 +
params.b );
      model_out = Math.round(model_out*100)/100;
      ctx.fillText("prediction: " + model_out, p.x, p.y + 20);
    }
```

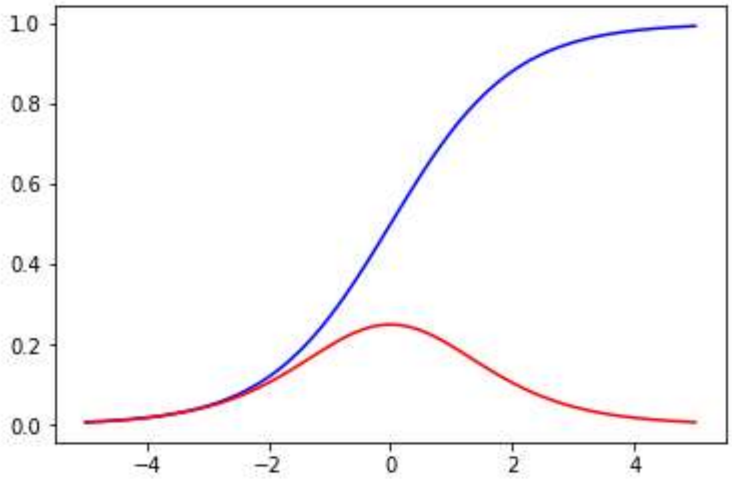
FINAL PYTHON CODE - COMPLETE

```
from matplotlib import pyplot as plt
import numpy as np

# each point is length, width, type (0, 1)

data = [[3,   1.5, 1],
        [2,   1,   0],
        [4,   1.5, 1],
        [3,   1,   0],
        [3.5, .5,  1],
        [2,   .5,  0],
        [5.5,  1,  1],
        [1,    1,  0]]

mystery_flower = [4.5, 1]

# scatter plot them
def vis_data():
    plt.grid()

    for i in range(len(data)):
        c = 'r'
        if data[i][2] == 0:
            c = 'b'
        plt.scatter([data[i][0]], [data[i][1]], c=c)

    plt.scatter([mystery_flower[0]], [mystery_flower[1]], c='gray')

## vis_data()
```

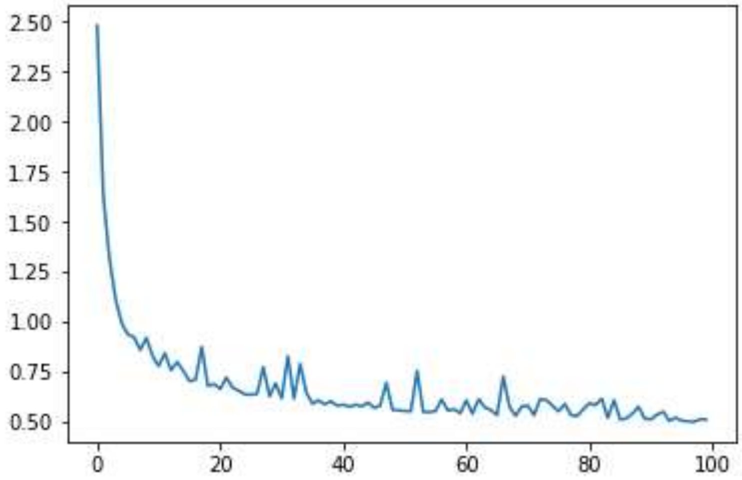


```
# network
#       o  flower type
#      / \  w1, w2, b
#     o   o  length, width


# activation function

def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_p(x):
    return sigmoid(x) * (1-sigmoid(x))
```

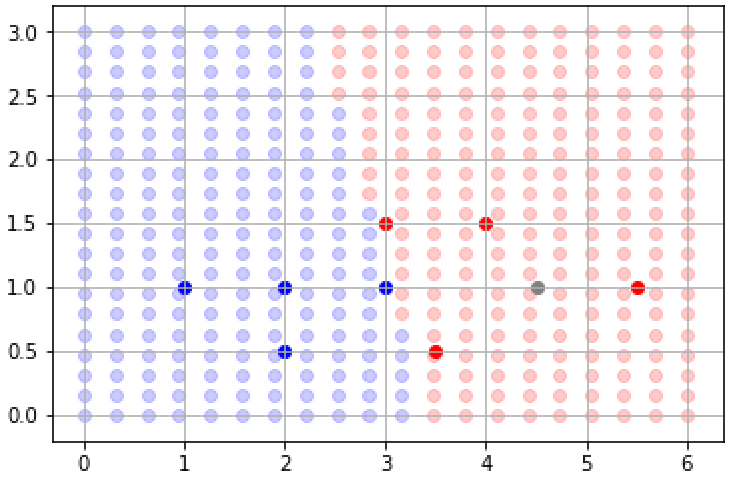
```
X = np.linspace(-5, 5, 100)

plt.plot(X, sigmoid(X), c="b") # sigmoid in blue
fig = plt.plot(X, sigmoid_p(X), c="r") # sigmoid_p in red
```



```
# train

def train():
    #random init of weights
    w1 = np.random.randn()
    w2 = np.random.randn()
    b = np.random.randn()

    iterations = 10000
    learning_rate = 0.1
    costs = [] # keep costs during training, see if they go down

    for i in range(iterations):
        # get a random point
        ri = np.random.randint(len(data))
        point = data[ri]

        z = point[0] * w1 + point[1] * w2 + b
        pred = sigmoid(z) # networks prediction

        target = point[2]

        # cost for current random point
        cost = np.square(pred - target)

        # print the cost over all data points every 1k iters
        if i % 100 == 0:
            c = 0
            for j in range(len(data)):
                p = data[j]
                p_pred = sigmoid(w1 * p[0] + w2 * p[1] + b)
                c += np.square(p_pred - p[2])
            costs.append(c)

        dcost_dpred = 2 * (pred - target)
        dpred_dz = sigmoid_p(z)

        dz_dw1 = point[0]
        dz_dw2 = point[1]
        dz_db = 1

        dcost_dz = dcost_dpred * dpred_dz

        dcost_dw1 = dcost_dz * dz_dw1
        dcost_dw2 = dcost_dz * dz_dw2
        dcost_db = dcost_dz * dz_db

        w1 = w1 - learning_rate * dcost_dw1
        w2 = w2 - learning_rate * dcost_dw2
        b = b - learning_rate * dcost_db

    return costs, w1, w2, b

costs, w1, w2, b = train()

fig = plt.plot(costs)
```



```
# predict what the myster flower is!

z = w1 * mystery_flower[0] + w2 * mystery_flower[1] + b
pred = sigmoid(z)

print(pred)
print("close to 0 -> blue, close to 1 -> red")

# check out the networks predictions in the x,y plane
for x in np.linspace(0, 6, 20):
    for y in np.linspace(0, 3, 20):
        pred = sigmoid(w1 * x + w2 * y + b)
        c = 'b'
        if pred > .5:
            c = 'r'
        plt.scatter([x],[y],c=c, alpha=.2)

# plot points over network predictions
# you should see a split, with half the predictions blue
# and the other half red.. nicely predicting each data point!
vis_data()
```


3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6