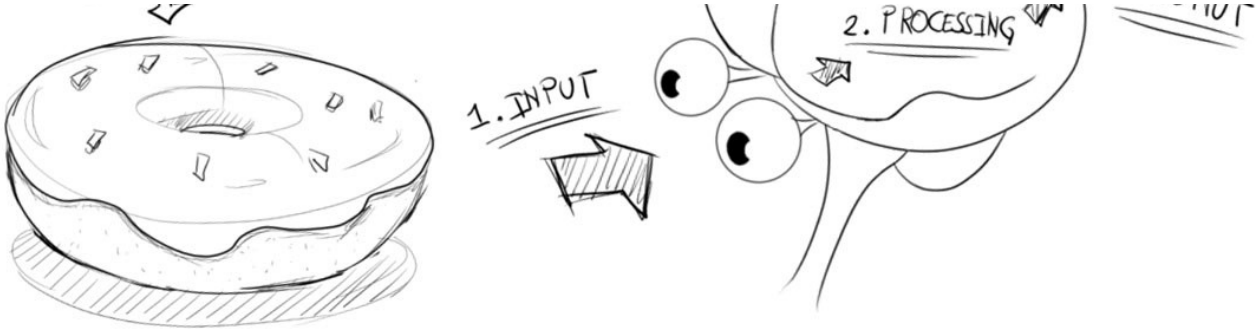
Keno
Leon
FRONT END / WEB DEVELOPER — DESIGNER: www.k3no.com
Apr 9, 2017 · 12 min read

Making a Simple Neural Network

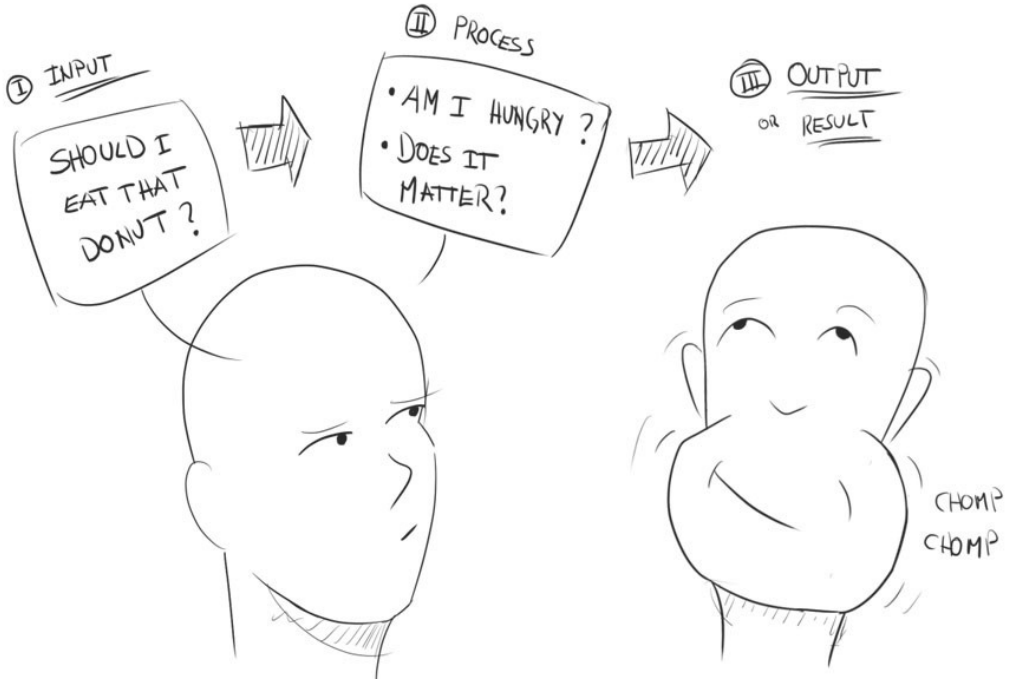
What are we making ? We'll try making a simple & minimal **Neural Network** which we will e**xplain** and **train** to **identify something**, there will be little to no history or math (tons of that stuff out there), instead I will try (*and possibly fail*) to explain it to both you and I mostly with doodles and code,let us begin.

A lot of Neural Network terms both originate and make somehow more sense from a biological perspective, so let's start from the very top :

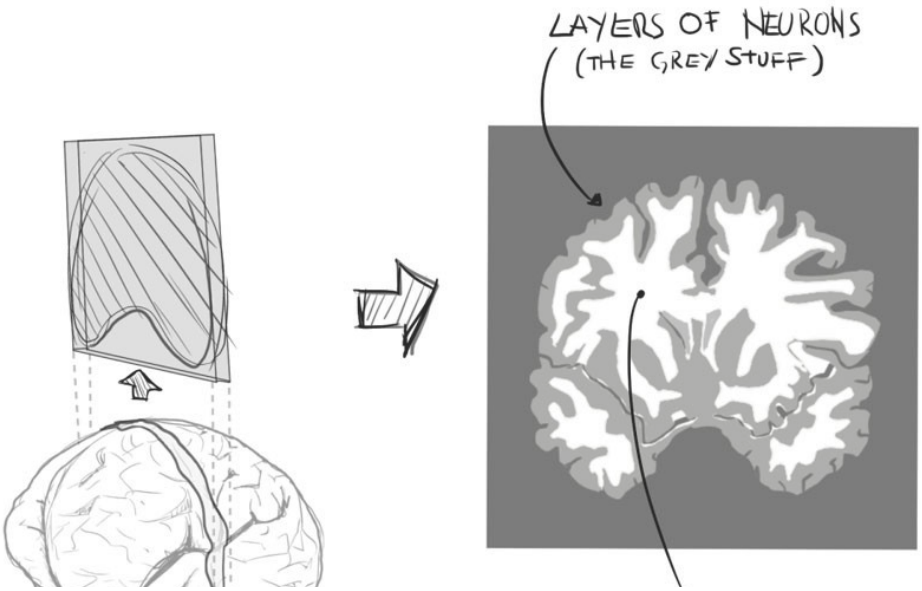
The brain is complex, but in general it can be divided into a few basic parts and ***operations:***



Stimuli can also be ***internal*** (like a percept or idea)***:***

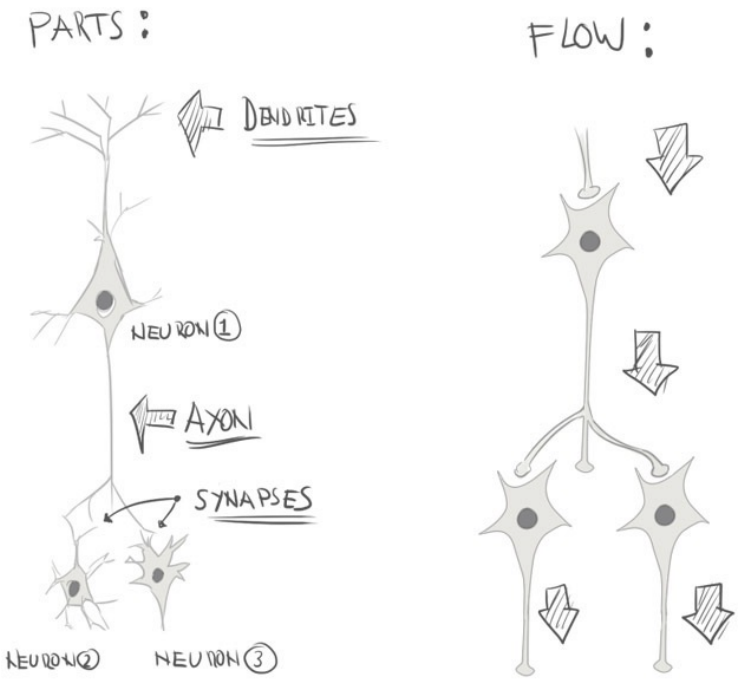


Let’s look at some basic and simplified brain ***parts:***



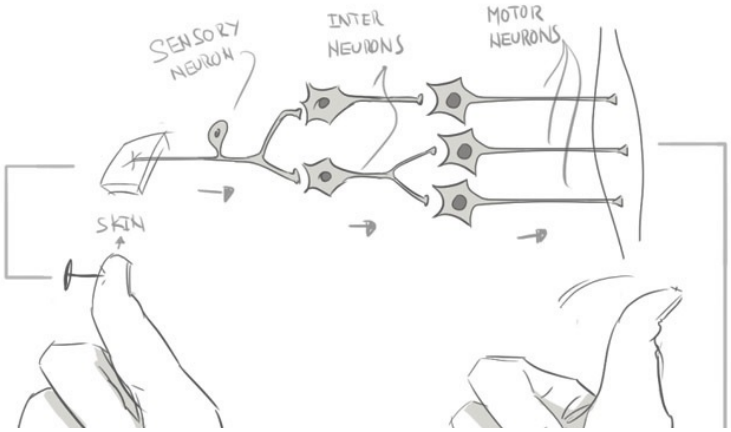
The brain is surprisingly mostly cabling.

The ***Neuron*** is the basic unit of computation in the brain, it receives and integrates chemical signals from other neurons and depending on a number of factors it either does nothing or generates an electrical signal or *Action Potential* which in turn signals other ***connected*** neurons via synapses:

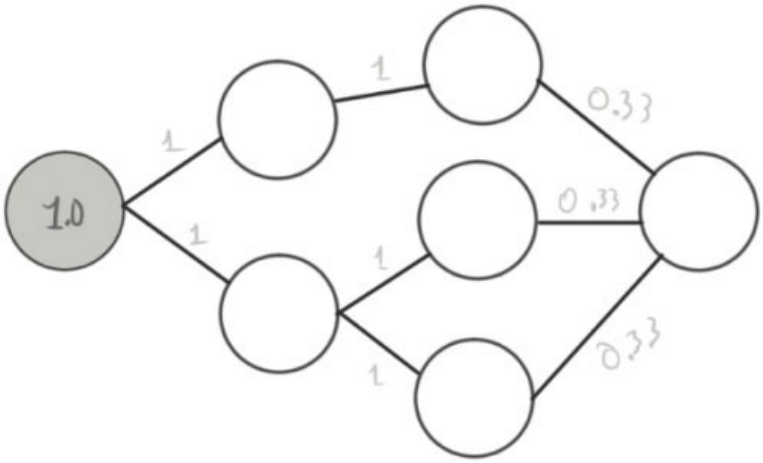


Dreams,memories,ideas,self regulated movement, reflexes and everything you think or do is all generated through this process: millions, maybe even billions of neurons firing at different rates and making connections which in turn create different subsystems all running in parallel and creating a biological ***Neural Network.***

This is of course both a generalization and a simplification,but now we can describe a minimal biological neural network:

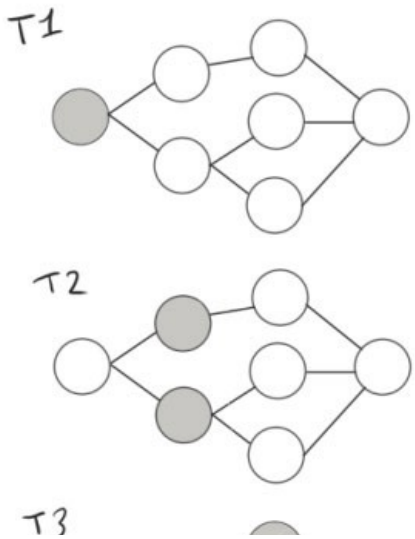


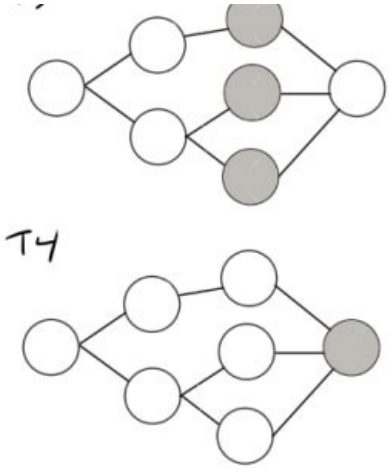
And describe it in a formal way with a ***graph :***



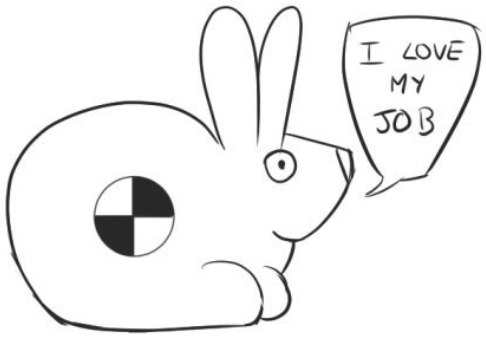
A little extra explanation is in order, the circles represent neurons, and the lines ***connections*** in between them, to keep things simple at this stage, the connections represent a ***forward*** movement of information from left to right. The first neuron is currently firing and is shaded to indicate it. It is also given a number (***1*** when it is firing, and ***0*** when it is not). The numbers in between neurons indicate the ***weight*** of the connection.

The above graph represents a moment in time of the network, a more accurate depiction would be divided into time segments:



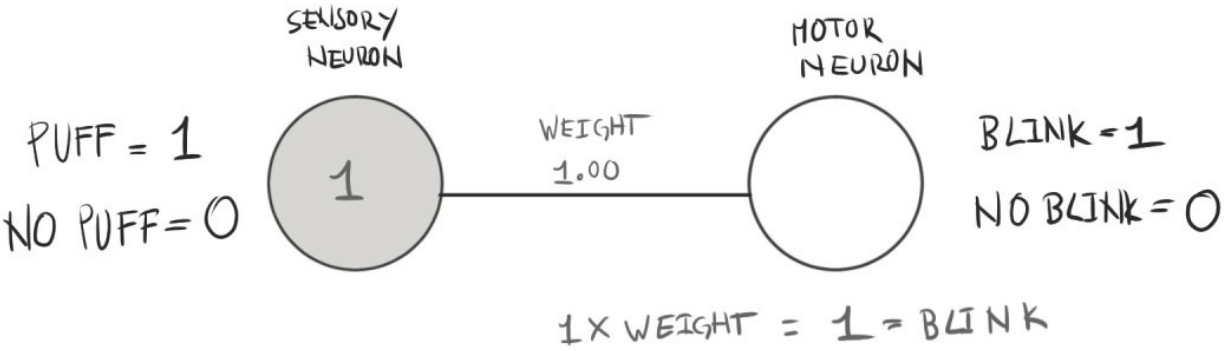


Before making our neural network, we need to understand how weights affect neurons and how neurons learn, let’s start with a bunny *(a test bunny)* and a classical conditioning experiment.



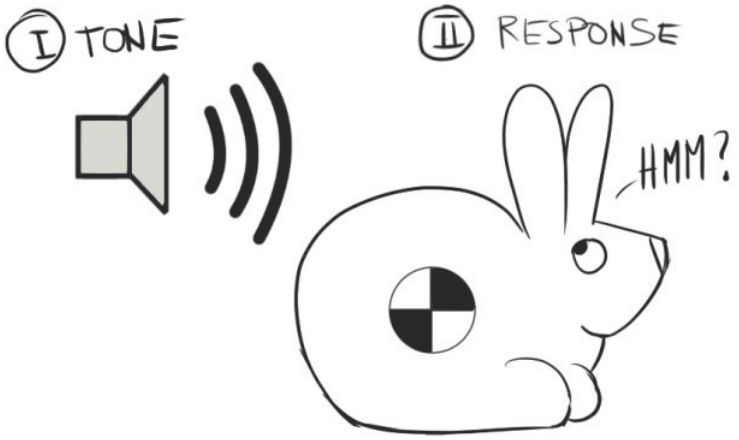
When subjected to a harmless puff of air, bunnies like humans usually blink:

We can model this behavior with a simple graph:

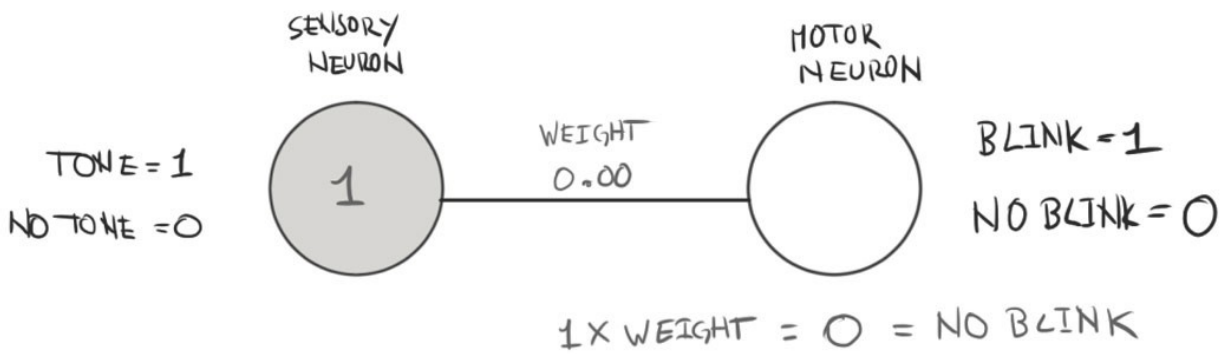


Like in our previous graph, this one displays just the moment when the bunny is sensing the puff of air, we are thus ***encoding*** the puff into a Boolean value. Additionally, we are now calculating if the second neuron fires or not based on the weight value, if the weight is 1 and the sensory neuron is firing we blink,anything below 1 we don't or in other words our second neuron has a ***threshold*** of 1.

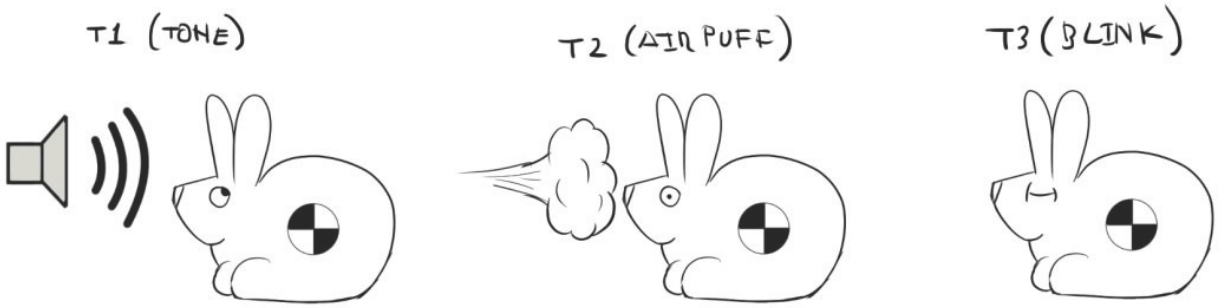
Let's introduce another element, a harmless auditory tone:



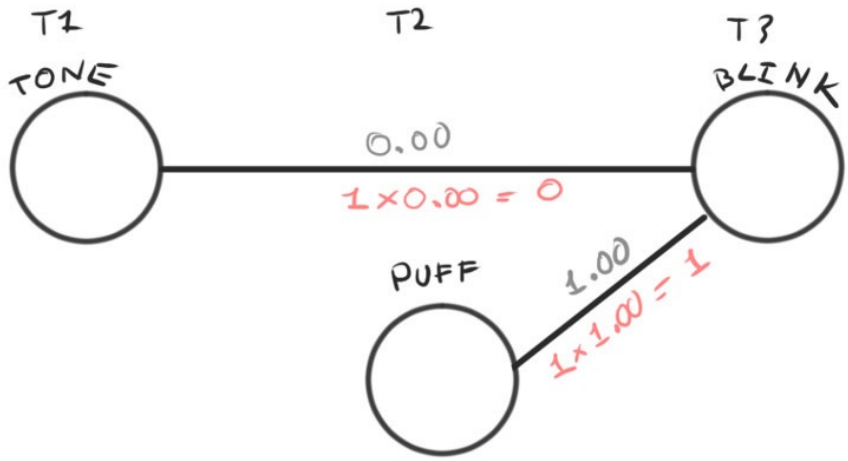
We can model the bunnies lack of interest in the following way:



The only difference is that the weight is now **Zero,** so there won't be a blinking bunny, at least not yet, let's train our bunny to blink on command by mixing stimuli (the tone and the air puff):



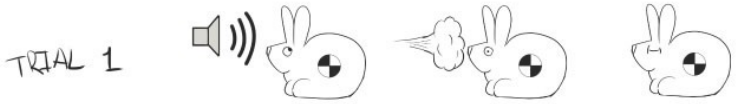
Importantly each of this events happens at a different time ***epoch,*** or moments in time, the graph would look like this :

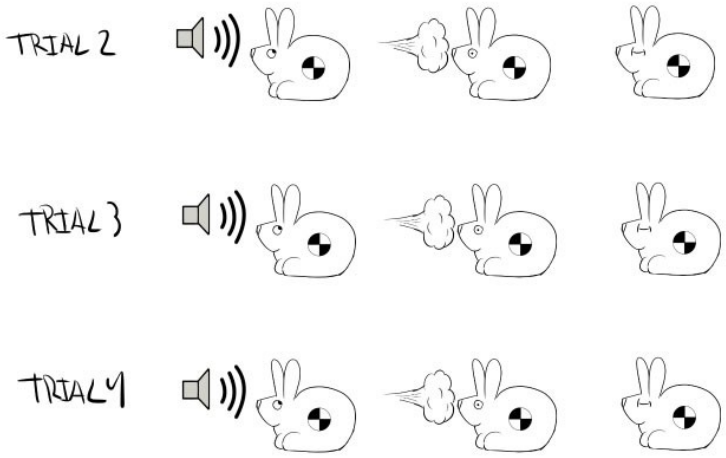


The tone does nothing, but the air puff still elicits a blink response, the important thing to note here is that we are indicating this through the weights multiplied by the stimuli (in red).

Learning while a complex behavior,can be simplified for the time being as the incremental change in weight between connected neurons in a neural network through time.

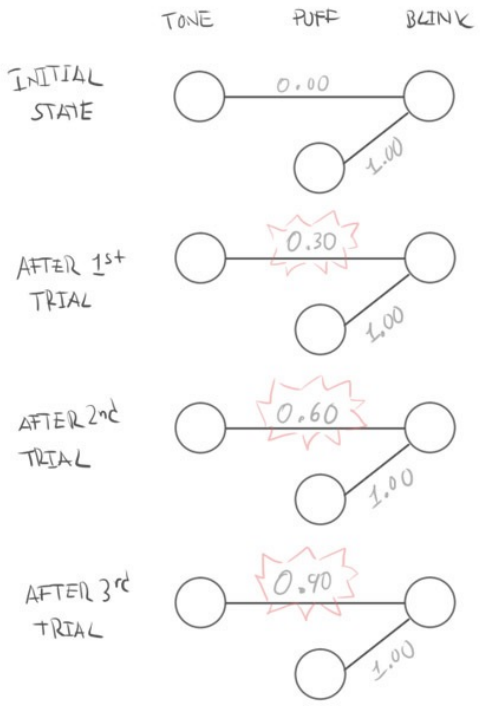
In order to train our bunny, we would have to repeat the process:





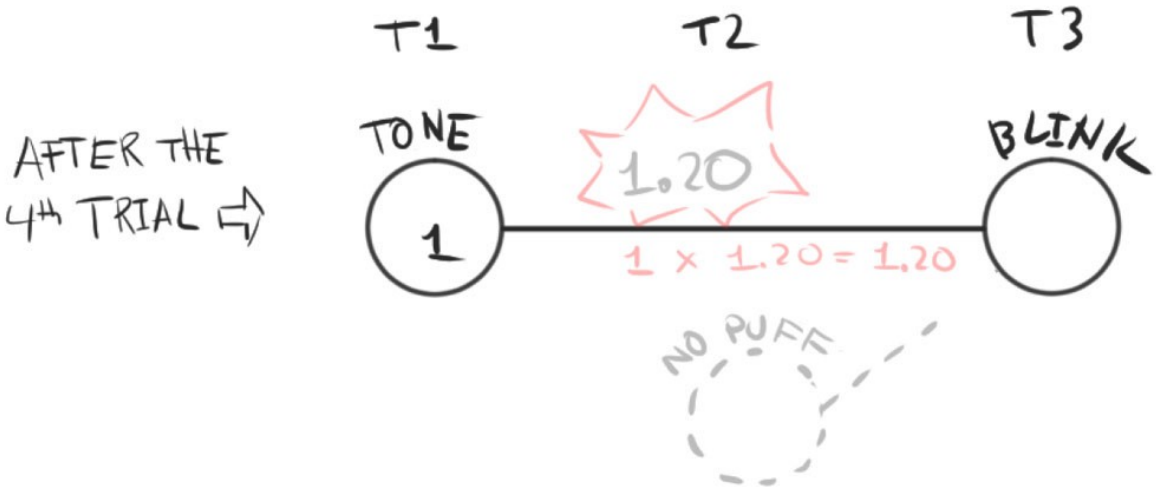
The bunny trials ?

The graph for the first 3 trials and the initial state would look like this:

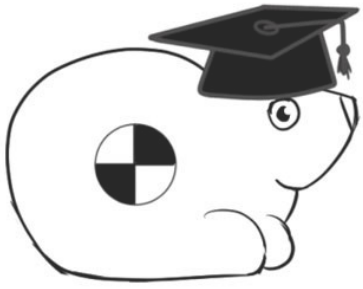


Note that the weights for the tone go up after each trial (*in re*d), this amount is arbitrary at this point—*I chose 0.30, but it could be anything, even a negative number.* Up til after the 3rd trial we have the same behavior from the bunny, but after the 4th trial, something new and amazing happens… new behavior.

We have removed the air puff yet the bunny still blinks after hearing the tone ! The explanation to this new behavior can be found in our final graph:



We have now trained our bunny to recognize a tone and blink.

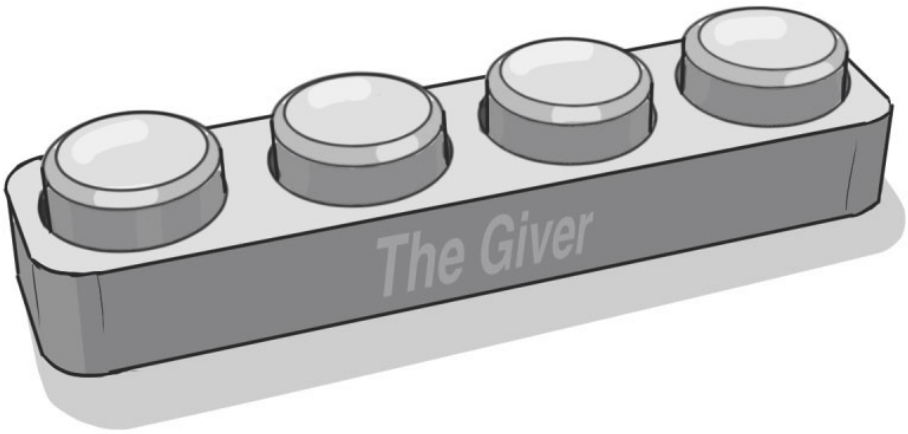


Real experiments of this kind can take about 60 trials over a few weeks or more to elicit a response.

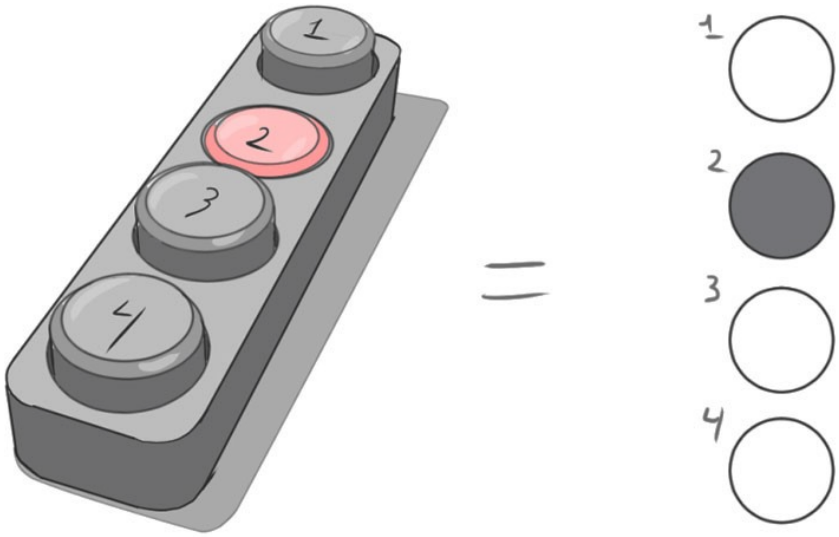
We now leave the biological world of brains and bunnies and will adapt some of what we have ***learned*** to make an *Artificial Neural Network*. The first thing we will do is try to define a simple problem.

Let’s say there is a 4 button machine that gives you food if you press the right button (*or perhaps energy if you are a robot*),the objective will be to ***learn*** which button provides

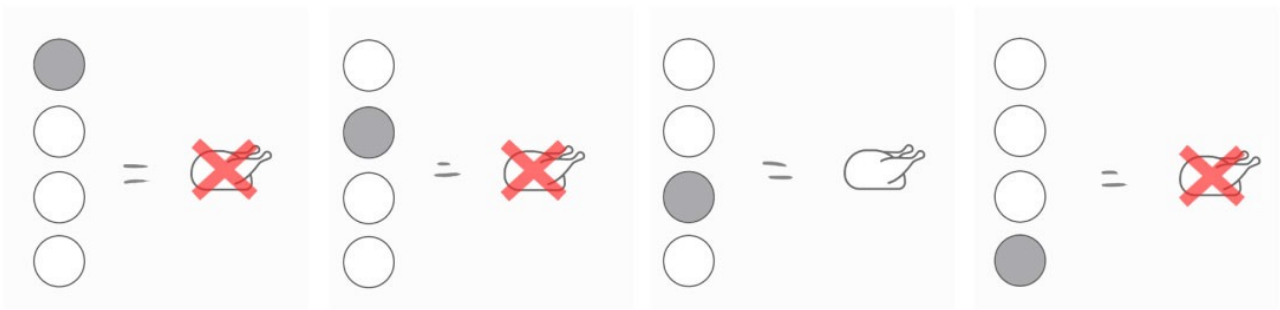
the goods:



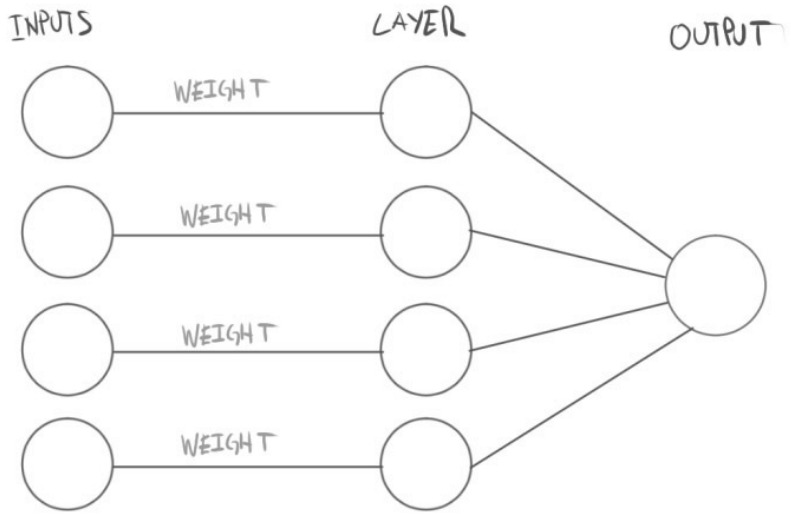
We can indicate when a button is pressed (*graphically*) in the following way:



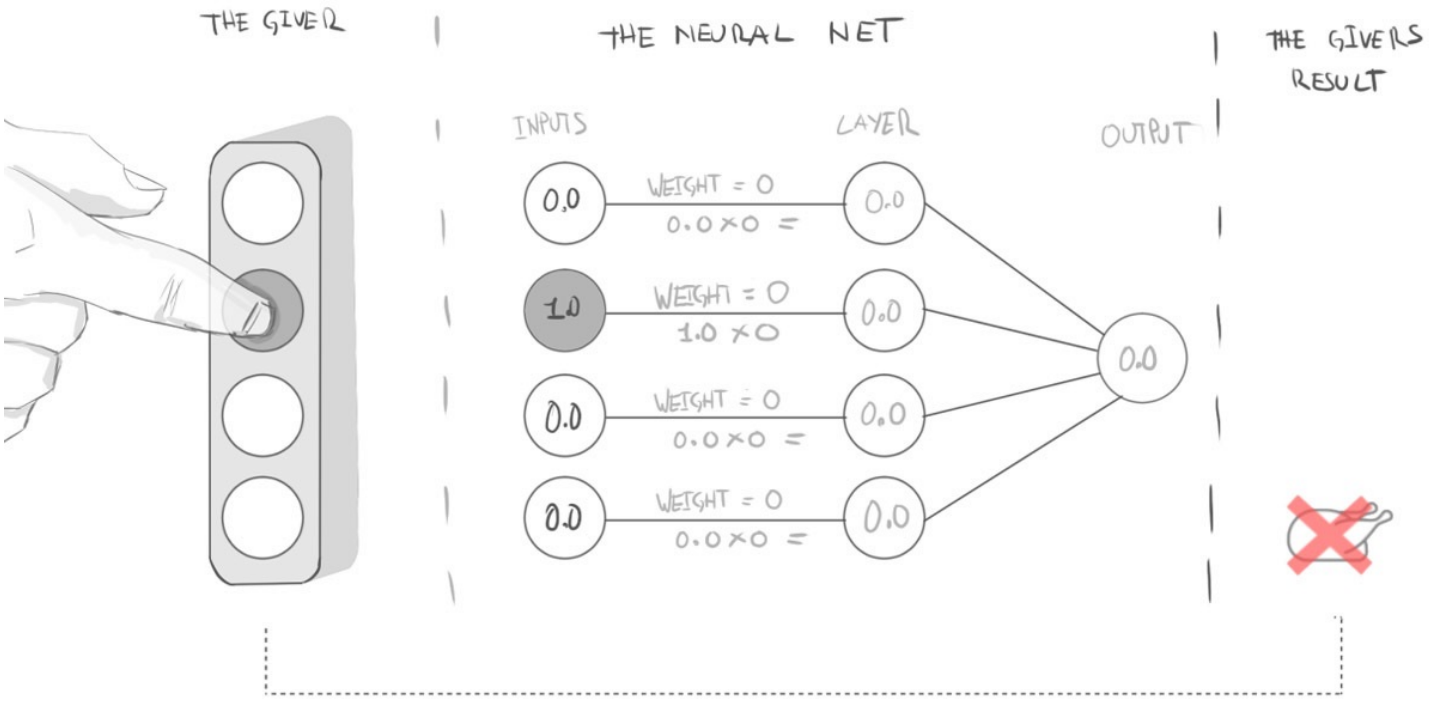
This problem is easy to digest in it’s entirety, so let’s see all the possible outcomes including the solution:



In order to create a neural network in code, we first need a model or graph we can match it with, here is one such arrangement suited for our current problem that loosely emulates it’s biological counterpart :



What this Neural Network does, is simply receive an input, which in this case would be the ***perception*** of which button was pressed,then modify said input by a weight, and finally return an output based on the addition of a layer. It sounds a little complicated, so let’s look at how our model would represent a button press:



Notice all the weights are zero, so the neural net is in a blank state yet fully connected, same as a newborn.

We are thus mapping an external event to the neural nets input layer and calculating an

output, this output might or might not match with reality, but for now we will ignore this fact and start writing this problem in a computer friendly way.

Let's start with inputs and weights (*I will be using Javascript*):

```
var inputs = [0,1,0,0];
var weights = [0,0,0,0];

// we can call these vectors for convenience.
```

The next step is then to create a function that takes these inputs & weights and calculates an output; this is such a function:

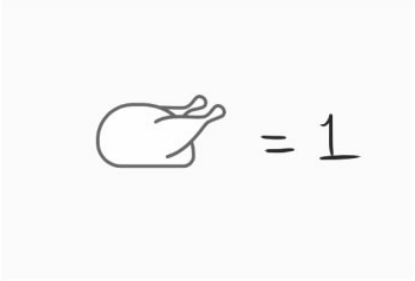
```
function evaluateNeuralNetwork(inputVector, weightVector){
  var result = 0;
  inputVector.forEach(function(inputValue, weightIndex) {
    layerValue = inputValue*weightVector[weightIndex];
     result += layerValue;
  });
  return (result.toFixed(2));
}

// Might look complex, but all it does is multiply weight/input pairs and adds the result.
```

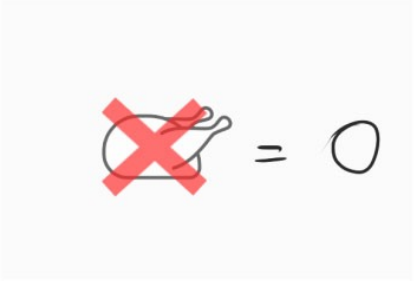
And as expected if we run the above we would get the same result as our graph or model…

```
evaluateNeuralNetwork(inputs, weights); // 0.00
```

Live example: Neural Net 001

The next step or upgrade to our neural net will be a way to check its own output or result against the reality of the situation, let's first encode this particular reality into a variable :



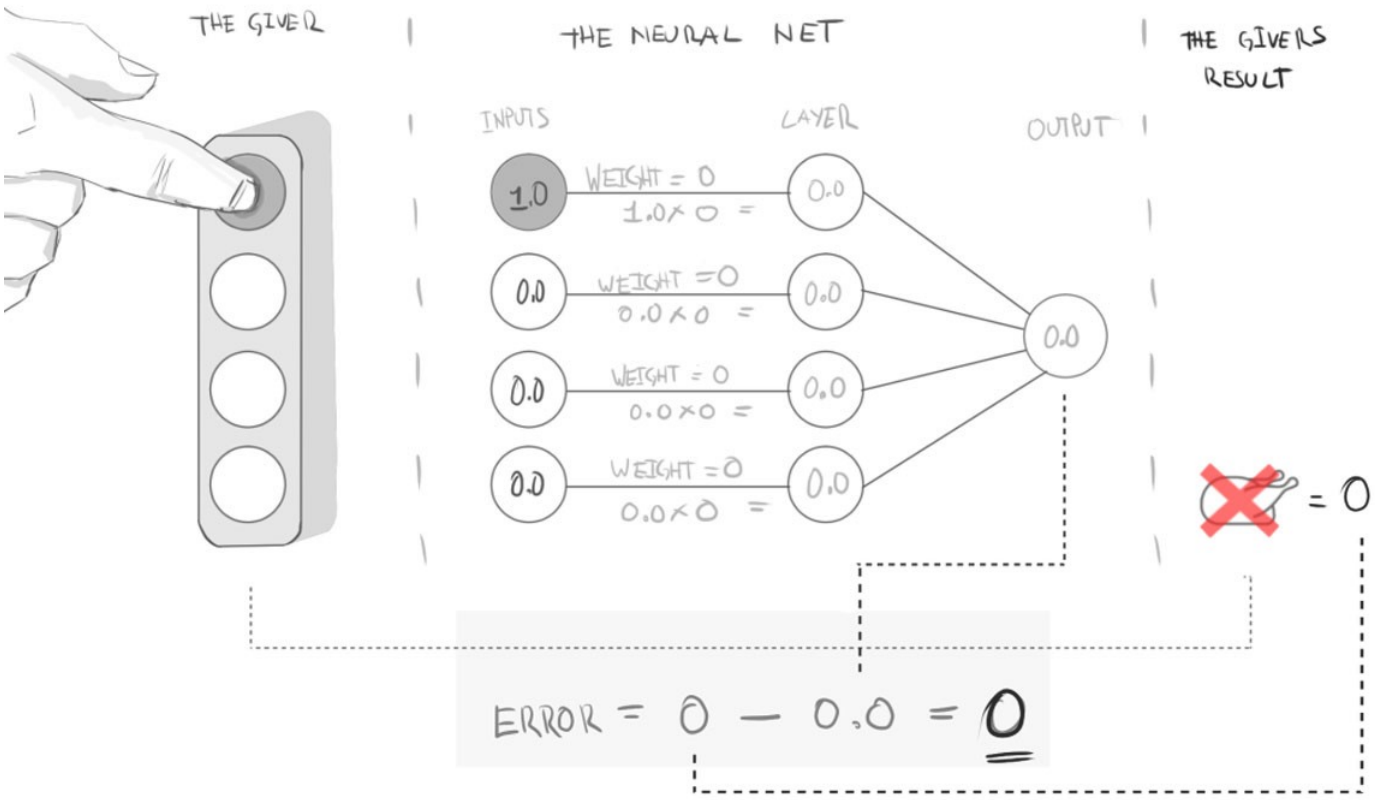


In order to detect a mismatch (and by how much) we will add an error function:

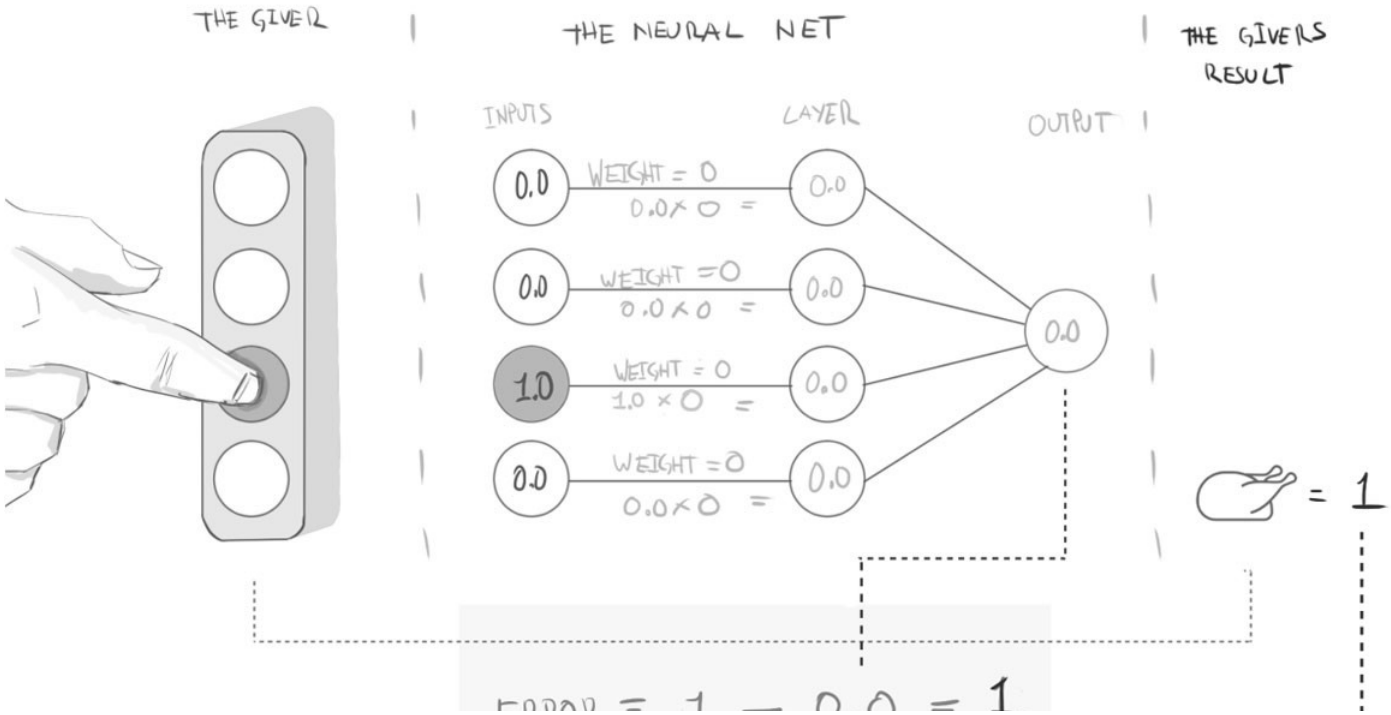
In order to detect a mismatch (and by how much) we will add an error function.

> *Error = Reality - Neural Net Output*

With this 2 new components, we can now judge our neural net:



And more importantly how about when reality provides a positive result ? (*chicken dinner in our example*).



So now we definitely know that our neural network model ***does not*** work (*and by how much*), great ! This is great because now we can use this error to guide our learning, it all makes a little more sense if we redefine our error function as follows :

```
Error = Desired Output - Neural Net Output
```

A subtle yet important difference that tacitly implies we will be using previously observed results to match against future performance (*and learning as we will soon see*), this also works in real life because reality is full of repeating patterns, so it probably pays as an evolutionary strategy (*most of the time)*.

Further modifying our code sample is just a matter of adding a new variable:

```
var input = [0,0,1,0];
var weights = [0,0,0,0];
var desiredResult = 1;
```

and a new function:

```
function evaluateNeuralNetError(desired,actual) {
  return (desired — actual);
}

// After evaluating both the Network and the Error we would get:
// "Neural Net output: 0.00 Error: 1"
```

Live example: Neural Net 002

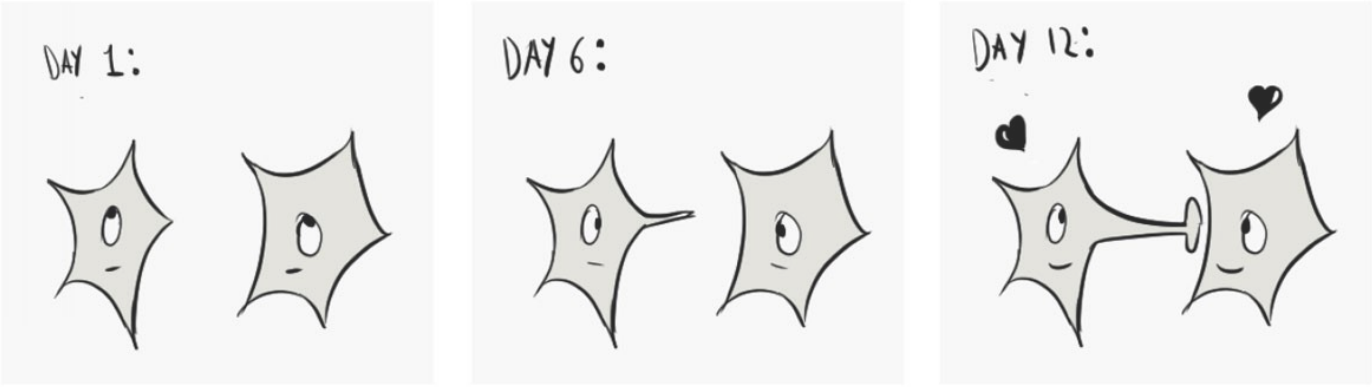
Friendly recap: *We started with a problem, then we made a simple model of it in the image of a biological network of neurons and now we have a way of measuring its performance against reality or a desired output. What we need now is a way to fix this mismatch, a process that in both computers and humans can be thought of as* ***learning****.*

So how does one teach a neural net ?

The basis of learning both for biological networks and artificial ones seems to be ***repetition*** and a ***learning algorithm***, so we will deal with each one separately, let's start with the learning algorithm.

In nature a learning algorithm can be thought of as a change in the physical and

In nature a learning algorithm can be thought of as a change in the physical and chemical characteristics of neurons after experience:



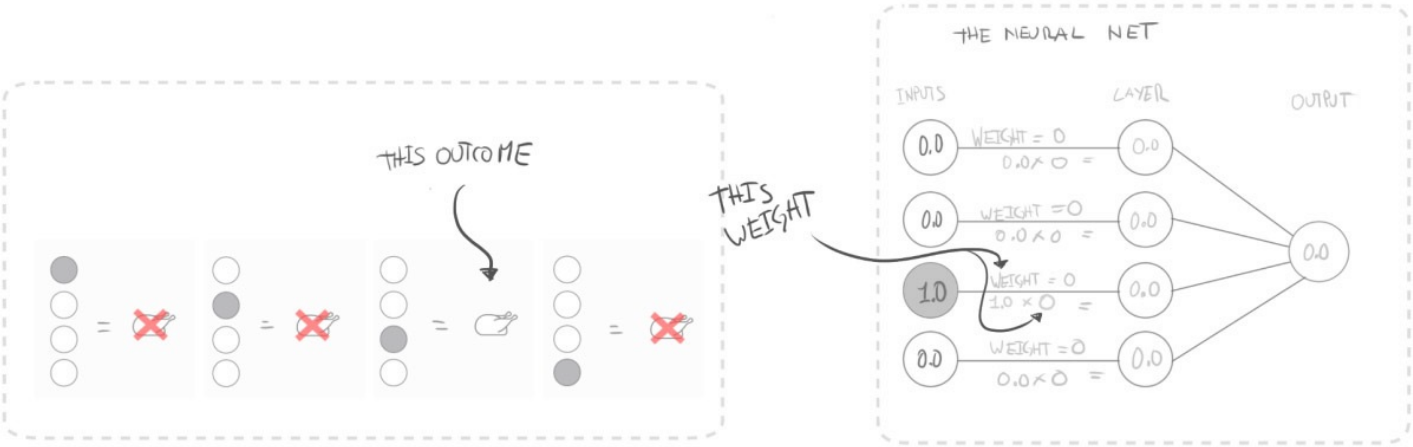
Dramatization of 2 neurons changing over time

In code and in our model a learning algorithm simply means we will be changing something over time ,to make our life easier, let's add a variable to symbolize by how much:

```
var learningRate = 0.20;
// bigger rate,bigger faster learnings : )
```

And what will we be changing ?

We will be changing the weights (*same as the bunny does !*) , specifically the weights of the outcome we want:



How one codes such an algorithm is a matter of choice, for simplicity sake I am just adding the learning rate to the weight, here it is as a function:

```
function learn(inputVector, weightVector) {
  weightVector.forEach(function(weight, index, weights) {
    if (inputVector[index] > 0) {
      weights[index] = weight + learningRate;
```

```
    weights[index] = weight + learningRate;
   }
  });
}
```

In use this learn function would simply add our learning rate to the ***active neuron's*** weight vector, before and after one learning round *(or trial)* this is the result:

```
// Original weight vector: [0,0,0,0]
// Neural Net output: 0.00 Error: 1

learn(input, weights);
```

```
// New Weight vector: [0,0.20,0,0]
// Neural Net output: 0.20 Error: 0.8
// If it is not apparently obvious, a Neural Net output closer to 1
// (chicken dinner) is what we want, so we are heading in the right
// direction
```

Live example: Neural Net 003

Ok, so now that we are heading in the right direction, the last piece we need to incorporate is ***repetition.***

There is really not much to it, in nature we just do stuff over and over again, in code we just specify a number of trials:

```
var trials = 6;
```

And apply our learn function to our neural net the number of times defined in our trials, a training function will do this:

```
function train(trials) {
 for (i = 0; i < trials; i++) {
  neuralNetResult = evaluateNeuralNetwork(input, weights);
  learn(input, weights);
 }
}
```

And our final readout:

```
Neural Net output: 0.00 Error: 1.00 Weight Vector: [0,0,0,0]
```

```
Neural Net output: 0.20 Error: 0.80 Weight Vector: [0,0,0.2,0]
```

```
Neural Net output: 0.40 Error: 0.60 Weight Vector: [0,0,0.4,0]
```

```
Neural Net output: 0.60 Error: 0.40 Weight Vector: [0,0,0.6,0]
```

```
Neural Net output: 0.80 Error: 0.20 Weight Vector: [0,0,0.8,0]
```

```
Neural Net output: 1.00 Error: 0.00 Weight Vector: [0,0,1,0]
```

```
Neural Net output: 1.00 Error: 0.00 Weight Vector: [0,0,1,0]
// Chicken Dinner !
```

Live example: Neural Net 004

We now have a weight vector that will only result in the output of 1 *(chicken dinner)* if the input vector corresponds to that of reality *(a push of the 3rd button down).*

So what good is this thing we just built ?

In our specific case, our Neural Net *(after being trained)* can recognize or discriminate in between inputs and tell you wich one will generate a desired output *(we would still need to code for the specific situation)* :

Hey kids, meet Mr. Neural Net !

Additionally, it is a scale model, a toy and a learning tool for both you and I, so we can further learn about machine learning,neural networks and artificial Intelligence.

Caveat emptor:

- No mechanism for storing the learnt weights is provided, so this neural net looses everything it knows upon refreshing or running the code again.
- It would take 6 succesful trials to fully train this neural net, if you consider a human or machine would push buttons at random… this could take a while.
- For important things biological networks have a learning rate of 1, so it would only take one successful trial.
- A learning algorithm that resembles biological neurons exists, it goes by the catchy name of *widroff-hoff rule or widroff-hoff learning*.
- The neurons treshold (*1 in our example*) and the effects of over learning (*with more trials the output would be greater than 1*), are glossed over, but they are important in

nature and a source of great and complex behaviour.

- So does negative weights.

Notes & further reading:

I've tried avoiding math and strict terms, but if you want to know, we just built a **perceptron** which is defined as an algorithm for supervised learning of binary classifiers, heavy stuff.

The biology of the brain is a big subject, in part because of it's inaccesability and in part because it's complexity, still there is a lot of information out there, possibly the best place to start is ***Neuroscience (Purves)*** and ***Cognitive Neuroscience (Gazzaniga)***.

I modified and adapted the bunny example from **Gateway to Memory (Gluck),** which also has an excellent introduction to graphs.

Another heavily consulted source was ***An Introduction to Neural Networks (Gurney),*** good for your general A.I. needs.

About the Author :

Eugenio N. Leon (Keno) is a Designer,Web Developer/programmer, Artist and Inventor, currently living in Mexico City, you can find bits and pieces of his work at www.k3no.com , he likes problems,getting paid to solve them and curiously is mostly vegetarian and hasn't touched chicken in ages.

Artificial Intelligence | Neural Networks | Machine Learning | Neuroscience | Tutorial