First neural network for beginners explained (with code)

Understand and create a Perceptron

Arthur Arnx · Follow
Published in Towards Data Science
9 min read · Jan 13, 2019

Listen Share

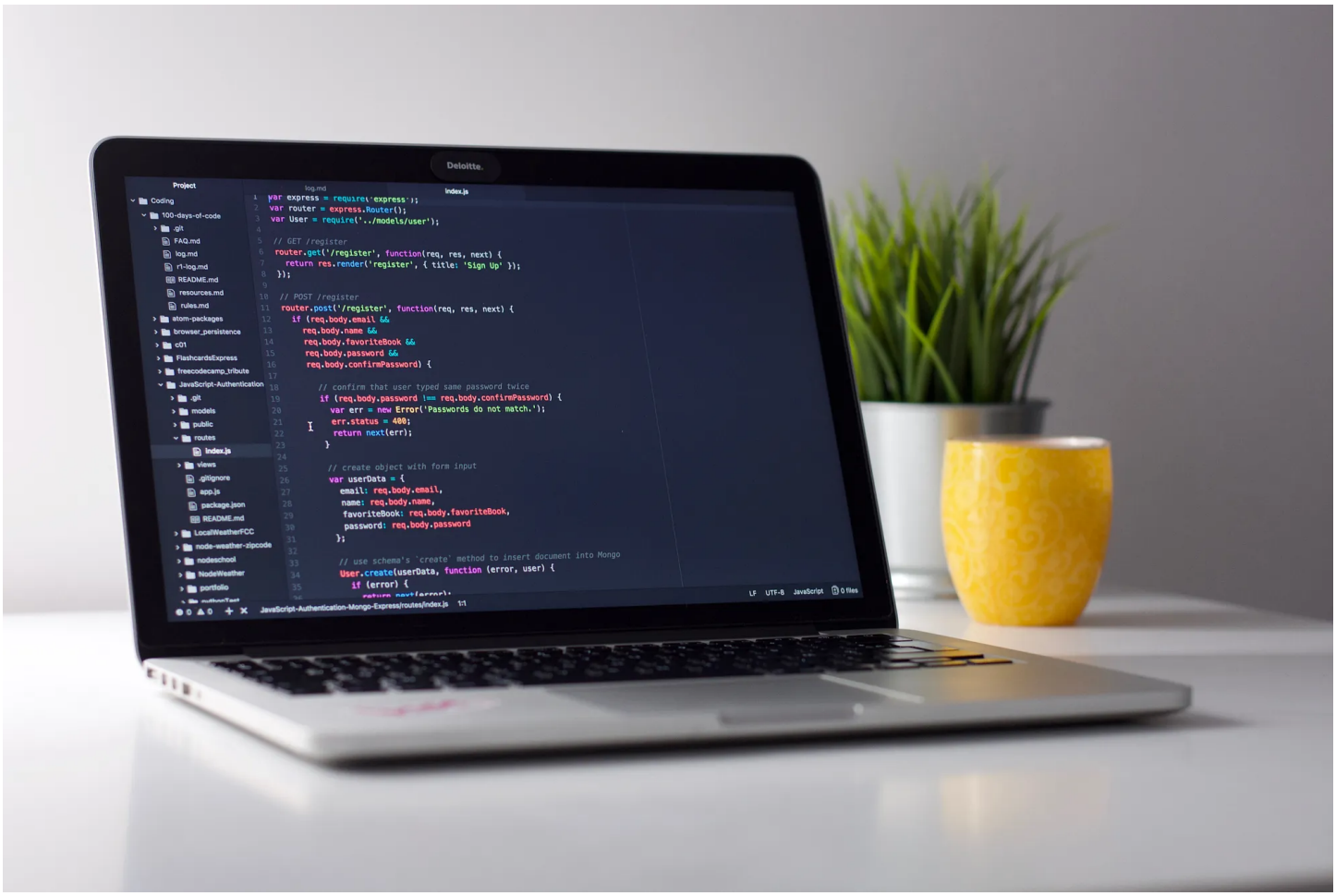

Photo by Clément H on Unsplash

So you want to create your first artificial neural network, or simply discover this subject, but have no idea where to begin ? Follow this quick guide to understand all the steps !

What is a neural network ?

Based on nature, neural networks are the usual representation we make of the brain : neurons interconnected to other neurons which forms a network. A simple information transits in a lot of them before becoming an actual thing, like “move the hand to pick up this pencil”.

The operation of a complete neural network is straightforward : one enter variables as inputs (for example an image if the neural network is supposed to tell what is on an image), and after some calculations, an output is returned (following the first example, giving an image of a cat should return the word “cat”).

Now, you should know that artificial neural network are usually put on columns, so that a neuron of the column *n* can only be connected to neurons from columns *n-1* and *n+1*. There are few types of networks that use a different architecture, but we will focus on the simplest for now.

So, we can represent an artificial neural network like that :

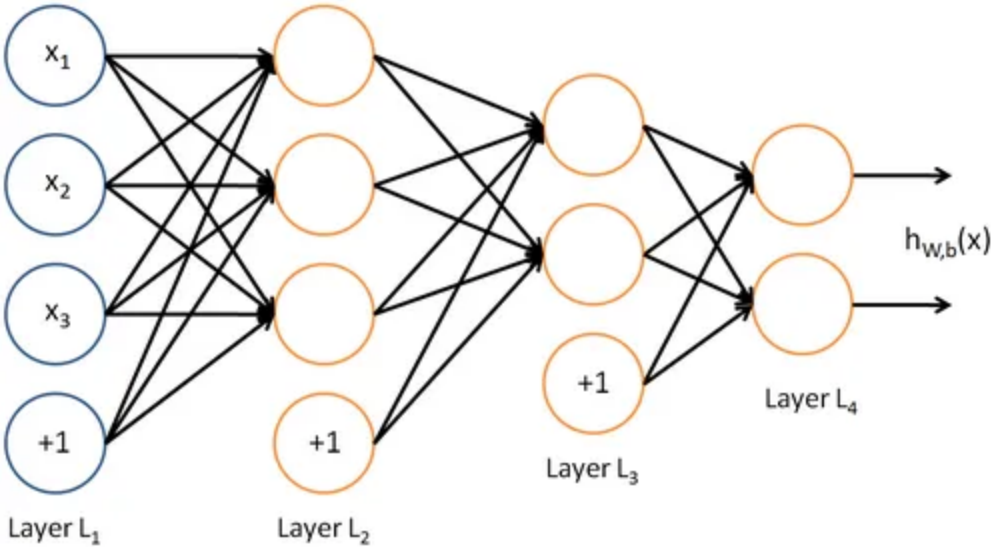

Figure 1 — Representation of a neural network

Neural networks can usually be read from left to right. Here, the first layer is the layer in which inputs are entered. There are 2 internals layers (called hidden layers) that do some math, and one last layer that contains all the possible outputs. Don't bother with the "+1"s at the bottom of every columns. It is something called "bias" and we'll talk about that later.

> *By the way, the term "deep learning" comes from neural networks that contains several hidden layers, also called "deep neural networks" . The Figure 1 can be considered as one.*

What does a neuron do ?

The operations done by each neurons are pretty simple :

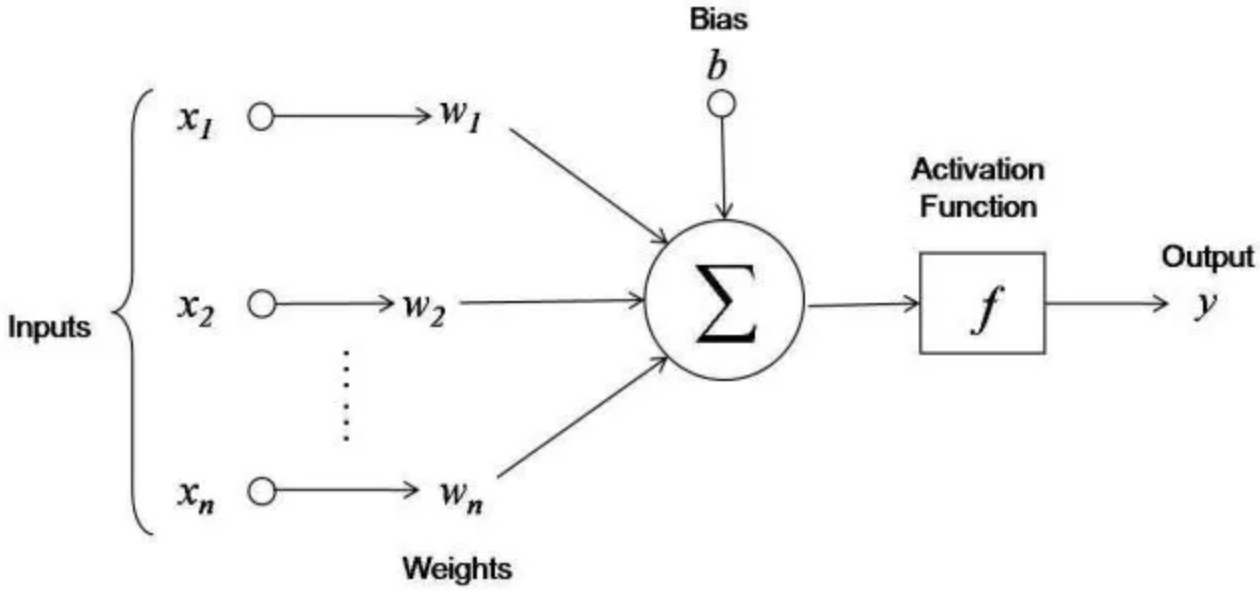

Figure 2 — Operations done by a neuron

First, it adds up the value of every neurons from the previous column it is connected to. On the Figure 2, there are 3 inputs (x1, x2, x3) coming to the neuron, so 3 neurons of the previous column are connected to our neuron.

This value is multiplied, before being added, by another variable called “weight” (w1, w2, w3) which determines the connection between the two neurons. Each connection of neurons has its own weight, and those are the only values that will be modified during the learning process.

Moreover, a bias value may be added to the total value calculated. It is not a value coming from a specific neuron and is chosen before the learning phase, but can be useful for the network.

After all those summations, the neuron finally applies a function called “activation function” to the obtained value.

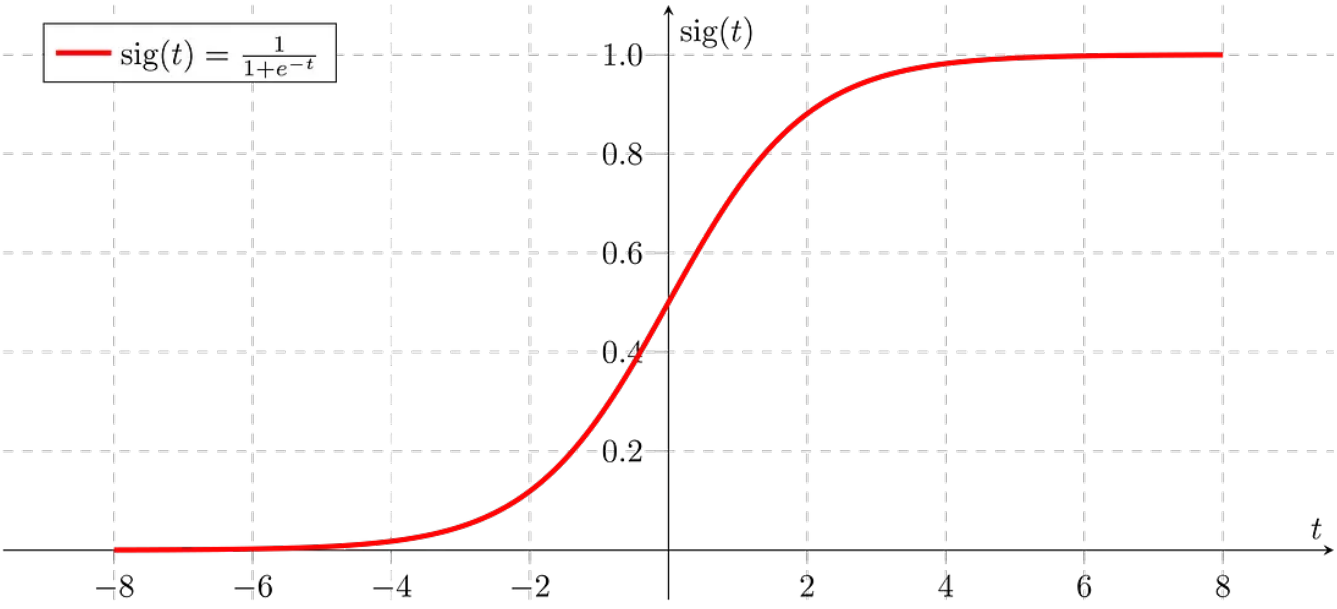

Figure 3 — Sigmoid function

The so-called activation function usually serves to turn the total value calculated before to a number between 0 and 1 (done for example by a sigmoid function shown by Figure 3). Other function exist and may change the limits of our function, but keeps the same aim of limiting the value.

That's all a neuron does ! Take all values from connected neurons multiplied by their respective weight, add them, and apply an activation function. Then, the neuron is ready to send its new value to other neurons.

After every neurons of a column did it, the neural network passes to the next column. In the end, the last values obtained should be one usable to determine the desired output.

Now that we understand what a neuron does, we could possibly create any network we want. However, there are other operations to implement to make a neural network learn.

How does a neural network learn ?

Yep, creating variables and making them interact with each other is great, but that is not enough to make the whole neural network learn by itself. We need to prepare a lot of data to give to our network. Those data include the inputs and the output expected from the neural network.

Let's take a look at how the learning process works :

First of all, remember that when an input is given to the neural network, it returns an output. On the first try, it can't get the right output by its own (except with luck) and that is why, during the learning phase, every inputs come with its label, explaining what output the neural network should have guessed. If the choice is the good one, actual parameters are kept and the next input is given. However, if the obtained output doesn't match the label, weights are changed. Those are the only variables that can be changed during the learning phase. This process may be imagined as multiple buttons, that are turned into different possibilities every times an input isn't guessed correctly.

To determine which weight is better to modify, a particular process, called "backpropagation" is done. We won't linger too much on that, since the neural network we will build doesn't use this exact process, but it consists on going back on the neural network and inspect every connection to check how the output would behave according to a change on the weight.

Finally, there is a last parameter to know to be able to control the way the neural network learns : the "learning rate". The name says it all, this new value determines on what speed the neural network will learn, or more specifically how it will modify a weight, little by little or by bigger steps. 1 is generally a good value for that parameter.

Perceptron

Okay, we know the basics, let's check about the neural network we will create. The one explained here is called a Perceptron and is the first neural network ever

created. It consists on 2 neurons in the inputs column and 1 neuron in the output column. This configuration allows to create a simple classifier to distinguish 2 groups. To better understand the possibilities and the limitations, let's see a quick example (which doesn't have much interest except to understand) :

Let's say you want your neural network to be able to return outputs according to the rules of the "inclusive or". Reminder :

- if A is true and B is true, then A or B is true.
- if A is true and B is false, then A or B is true.
- if A is false and B is true, then A or B is true.
- if A is false and B is false, then A or B is false.

If you replace the "true"s by 1 and the "false"s by 0 and put the 4 possibilities as points with coordinates on a plan, then you realize the two final groups "false" and "true" may be separated by a single line. This is what a Perceptron can do.

On the other hand, if we check the case of the "exclusive or" (in which the case "true or true" (the point (1,1)) is false), then we can see that a simple line cannot separate the two groups, and a Perceptron isn't able to deal with this problem.

So, the Perceptron is indeed not a very efficient neural network, but it is simple to create and may still be useful as a classifier.

Creating our own simple neural network

Let's create a neural network from scratch with Python (3.x in the example below).

```
import numpy, random, os
lr = 1 #learning rate
bias = 1 #value of bias
weights = [random.random(),random.random(),random.random()] #weights
generated in a list (3 weights in total for 2 neurons and the bias)
```

The beginning of the program just defines libraries and the values of the parameters, and creates a list which contains the values of the weights that will be modified (those are generated randomly).

```
def Perceptron(input1, input2, output) :
   outputP = input1*weights[0]+input2*weights[1]+bias*weights[2]
   if outputP > 0 : #activation function (here Heaviside)
      outputP = 1
   else :
      outputP = 0
   error = output - outputP
   weights[0] += error * input1 * lr
   weights[1] += error * input2 * lr
   weights[2] += error * bias * lr
```

Here we create a function which defines the work of the output neuron. It takes 3 parameters (the 2 values of the neurons and the expected output). "outputP" is the variable corresponding to the output given by the Perceptron. Then we calculate the error, used to modify the weights of every connections to the output neuron right after.

```
for i in range(50) :
   Perceptron(1,1,1) #True or true
   Perceptron(1,0,1) #True or false
   Perceptron(0,1,1) #False or true
```

```
Perceptron(0,0,0) #False or false
```

We create a loop that makes the neural network repeat every situation several times. This part is the learning phase. The number of iteration is chosen according to the precision we want. However, be aware that too much iterations could lead the network to over-fitting, which causes it to focus too much on the treated examples, so it couldn't get a right output on case it didn't see during its learning phase.

However, our case here is a bit special, since there are only 4 possibilities, and we give the neural network all of them during its learning phase. A Perceptron is supposed to give a correct output without having ever seen the case it is treating.

```
x = int(input())
y = int(input())
outputP = x*weights[0] + y*weights[1] + bias*weights[2]
if outputP > 0 : #activation function
   outputP = 1
else :
   outputP = 0
print(x, "or", y, "is : ", outputP)
```

Finally, we can ask the user to enter himself the values to check if the Perceptron is working. This is the testing phase.

The activation function Heaviside is interesting to use in this case, since it takes back all values to exactly 0 or 1, since we are looking for a false or true result. We could try with a sigmoid function and obtain a decimal number between 0 and 1, normally very close to one of those limits.

```
outputP = 1/(1+numpy.exp(-outputP)) #sigmoid function
```

We could also save the weights that the neural network just calculated in a file, to use it later without making another learning phase. It is done for way bigger project, in which that phase can last days or weeks.

That's it ! You've done your own complete neural network. You created it, made it learn, and checked its capacities. Your Perceptron can now be modified to use it on another problem. Just change the points given during the iterations, adjust the number of loop if your case is more complex, and just let your Perceptron do the classification.

Do you want to list 2 types of trees in the nearest forest and be able to determine if a new tree is type A or B ? Chose 2 features that can dissociate both types (for example height and width), and create some points for the Perceptron to place on the plan. Let it deduct a way to separate the 2 groups, and enter any new tree's point to know which type it is.

You could later expand your knowledge and see about bigger and deeper neural network, that are very powerful !

There are multiple aspects we didn't treat, or just enough for you to get the basics, so don't hesitate to go further. I would love to write about more complex neural networks so stay tuned !

> Thanks for reading !
>
> I hope this little guide was useful, if you have any question and/or suggestion, let me know in the comments.

Artificial Intelligence | Neural Networks | Perceptron | Guides And Tutorials

Follow

Published in Towards Data Science

773K Followers · Last published 8 hours ago

Your home for data science and AI. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

Follow

Written by Arthur Arnx

371 Followers · 3 Following

Software Engineer, Paris, France

Responses (19)

What are your thoughts?

Respond

Alex Schill
over 5 years ago

weights[0] += error * input1 * a