NEURAL NETWORKS FOR THE VERY CURIOUS

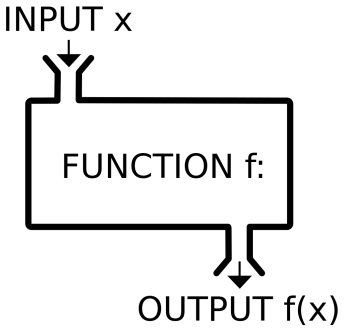
"Give it the compute, give it the data, and it will do amazing things. This stuff is like—it's like alchemy!"

~ **Ilya Sutskever**

1.

The standard way to introduce Neural Networks (NNs) is to make an analogy with neurons in the brain, which in my opinion is not the right approach. It's not how I personally think about them. So we will try to approach the concept from a mathematical point-of-view. Because that's where the beauty is.

I will keep math-y-ness light but I should introduce a very simple but useful thing: the concept of a function.



You can think of a function as an abstract machine, one which takes some input, and outputs something else. Whatever that happens inside the machine is not our concern as long as we have the inputs and outputs.

A function can be anything, it can be something as simple as a multiplicative operation: takes two inputs, multiplies them, and outputs the product. Again, how exactly it multiplies the numbers doesn't matter. A function can also be something as complex as a Cute Panda

Generator: you tell the machine you want another cute panda image (input), it does whatever alchemical magic it does, and outputs your desiderata.

A mathematically literate reader will point out that the definition is not complete and that there are some other conditions too. Explaining those intricacies is not the point here, understanding is. I will follow the same approach throughout this write-up: *understanding over intricacies*.

2.

A neural network can be thought of as a function — a *copy-cat* function, a mathematical charmeleon. You give the function you want it to copy-cat and it copy-cats it. In mathematical terms, a neural network is a Universal Approximator, it can approximate any function, no matter how complex, to an arbitrary precision. *Any* function. How cool is that?!

Very cool indeed, but the annoying caveat is that you need to have the function you want it to approximate, if you don't have that, then sorry bud. This may not seem like a very reasonable complaint at first, "we have this wonderful thing which can mimic any function, what else do you want, greedy man?!" But the complaint is actually *not* unreasonable, see if you already have a Cute Panda Generator, what use is summoning the approximation genie?

All AI research, or *most* rather, and at least at present, goes into getting around that caveat. And yes, getting around that caveat is possible, hurray! Likewise, most of the portion of this write-up will be devoted to explaining how exactly AI researchers get around that caveat.

But first let's understand how, at a concrete level, a neural network mimics a given function. Someone who has played around with Photoshop or something similar may know about *Bézier curves*. You should try playing with them here. There are two types of points in a Bézier curve: a pivot and an anchor. In the linked webpage, the black dots are anchors and the blue squares are pivots. You can approximate *any* curve, in theory, by changing positions of those

points and making new ones if need be. Anchor points are placed directly on the curve and pivots points are moved around to reach the desired level of approximation. Go and play around with them to get a feel.

Now, neural nets do NOT use Bézier curves for approximation, they use linear functions instead. And AI researchers, of course, don't configure neural nets by hand, they automate the process by using a technique called Gradient Descent. I mentioned Bézier curves because I find them useful for intuition. It is one thing to *know* that neural nets can approximate any function and another to *understand* how the process would look like under the hood.

3. [You can skip this section if you don't want to understand the math.]

So, what are these linear functions and the Gradient Distant, Distance, *whatever!*

Before understanding linear functions, we should learn how to represent a function using its mathematical notation. It will come in handy. The standard way is to represent a prototypical function as simply *f(x),* read as "eff of axe".
f is the name of the function, *x* represents one placeholder input. Do not confuse *f(x)* with multiplying *f* and *x*, *f(x)* represents one single thing, namely, a function named *f* with one single input represented by one placeholder variable *x*. If you want two inputs it can be *f(x, y)*, and so on.

A function which takes a single number as input and outputs its square can be written as: $f(x) = x^2$.

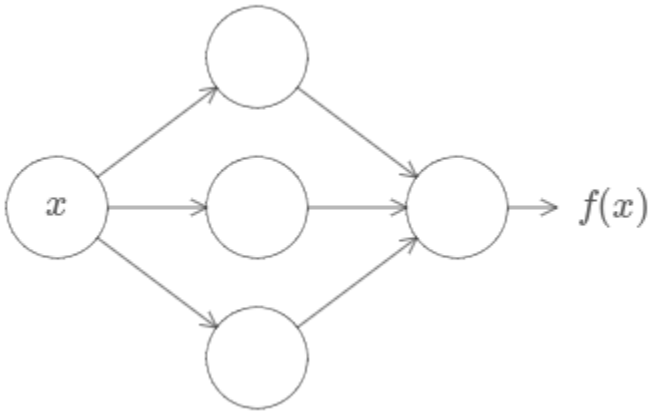
Furthermore, functions can be nested, this is called composition. Simply put, functions can take other functions as inputs. Like *f(g(h(x)))*. A better notation for writing these compositions is $f \circ g \circ h$

A linear function is nothing but a function of the form $f(x) = Ax + B$ and a neural network is nothing but a composition of several of these linear functions. Each of these linear functions is called a layer and each layer itself

consists of several other linear functions called neurons. In mathematical terms, in a layer $F(x) = Ax + B$, A and B are vectors which have weights and biases of their respective neurons as their elements. In a neuron $f(x) = wx + b$, w is called its weight and b is called its bias. These weights and biases are nothing but numbers, albeit important numbers. Just like a Bézier curve has pivot and anchor points whose positions determine how a curve is approximated, a neural network has weights and biases whose values determine how a function is approximated.

4.

Is everything getting a little too complicated? A visualization should help.



What you're seeing above is a very simple neural network, it has three layers: the first layer consists one just one neuron, the second has three, and the third has one. Recall that a neural network is nothing but a function — a machine which takes-in an input and gives-out an output. This machine has some inside machinery, which consists of layers. The layers by themselves are not very interesting parts of this machine, although these layers are made up of smaller parts called neurons, which *are* interesting. You can analogise neurons with gears in a mechanical machine, a gear has some properties like size and degree of lubrication, the gear of a neural net — a neuron — has similar properties called a weight and a bias. Changing these properties of neurons in a neural network changes its behavior.

5.

You may have occasionally come across some terms like “machine learning” or “deep learning”. *Learning*, huh? That kinda sounds cooler than Neural Nets, doesn’t it? Well, you know what... Neural Nets can learn!

You see, finding the *correct* values of these weights and biases in a Neural Net is a very hard task, so we don’t do it ourselves, we make the Neural Nets learn from their mistakes instead. How exactly do we do that? Let’s see!

At first, the NN will work incorrectly because it will have random values of weights and biases (which we put in). It can be made to “learn” the correct values over time by providing it with feedback in terms of how *exactly* incorrect it is. Mathematically, we will need to calculate its Error Function. Doing so is very simple, we just need to subtract its incorrect outputs from correct ones. Additionally, we square the Error Function so that the error is always positive and the minimum possible error is zero, it makes things easier to work with. Using mathematical notation we can represent the Error Function as:

$$E(x) = (K(x) - N(x))^2$$

Where $E(x)$ is the error at input x, $K(x)$ is the correct output given input x, and $N(x)$ is the NN’s (incorrect) output.

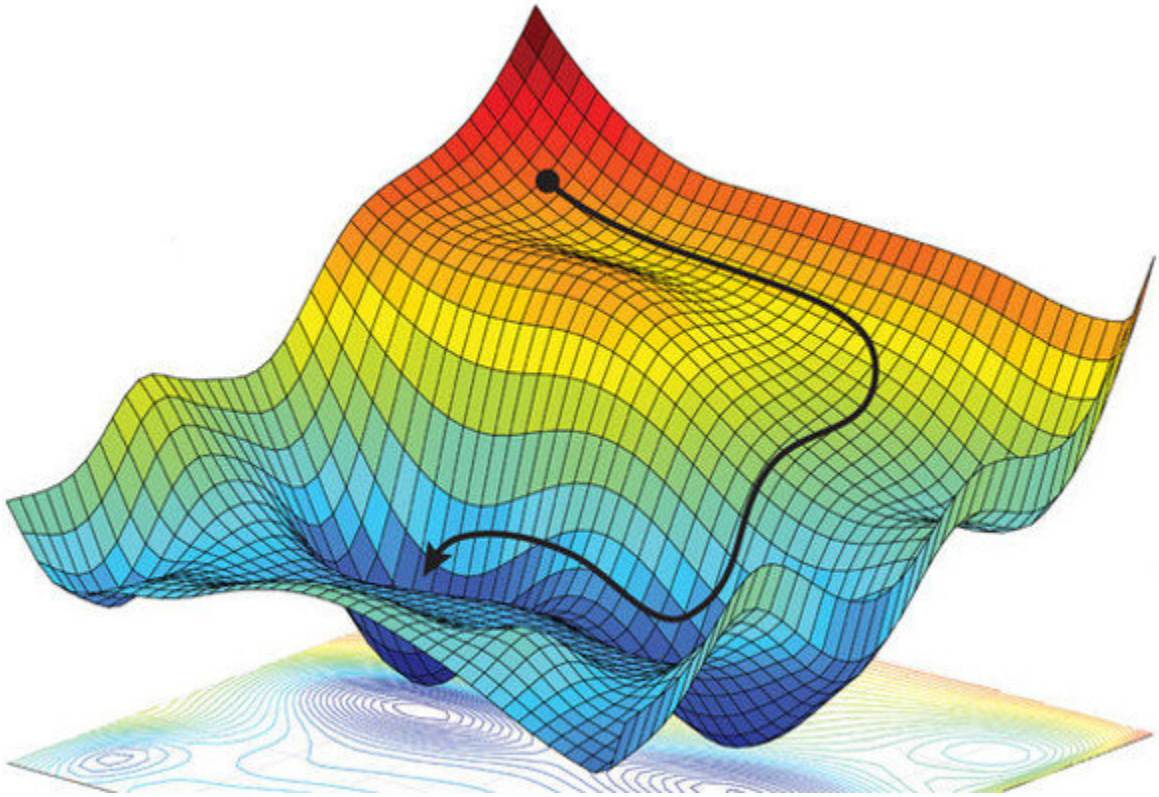
If you haven’t already guessed it, K is nothing but the function we are trying to approximate using NN, so of course by definition $K(x)$ will be the correct output given input x.

And to you make you recall again, N is nothing but a composition of linear functions, mathematically represent by $N(x) = F_1 \circ F_2 \circ F_3 \circ \dots \circ F_k\ (x)$, where each $F_n\ (x) = Ax + B$, and $A = [w_1, w_2, w_3, \dots, w_k]^T$ and $B = [b_1, b_2, b_3, \dots, b_k]^T$, w_n and b_n are of course weights and biases.

To learn is to make fewer mistakes. In other words, NN can learn weights and biases by minimizing the Error Function. The mathematically literate people in the crowd will jump upon hearing this because there's a standard tool in the mathematical toolbox for finding minima of functions — Calculus.

I will not go into the gory mathematical details of how you find the minima of the Error Function because finding the derivative of that beastly *N(x)* is a bit of a challenge. In fact so much so that Geoffrey Hinton won the Turing Prize for doing that. So erm... nah. Although, I *will* explain it in an intuitive way.

To dumb it down very very much, you plot the Error Function on a graph, like this:



Because the graph is of the Error Function, the higher points on the graph represent high values of error, and the lower ones lower values. What you do is follow the path which is steepest downhill and keep going until you reach the lowest point, at which Error Function will be minimum. Gradient *Descent*.

The coordinates of this lowest point will be the weights and biases of the Neural Networks. Hurray!