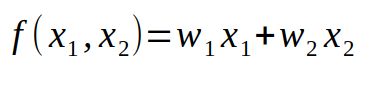
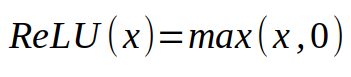
**How do we ‘train’ neural networks ?**

<https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73>

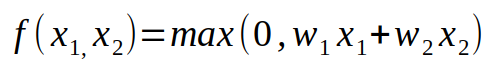
So what is a neuron ? It is a mathematical function. It takes several numbers as inputs(as many as you want). The neuron I drew above takes two numbers as input. Each input number we’re going to denote as*xₖ* where k stands for index of input. For each input *xₖ* neuron assigns another number *wₖ*. A vector consisting of these numbers *wₖ* is called Weights vector. These weights are what makes each neuron unique. They are fixed during testing, but during training these are the numbers we’re going to change in order to ‘tune’ our network. I’ll talk about it later in the post. As I said above, a neuron is a function. But what kind of function is that ? It’s a linear combination of weights and inputs with some kind of non-linear function on top of it. Let me explain further. Let’s look at the first part — linear part.



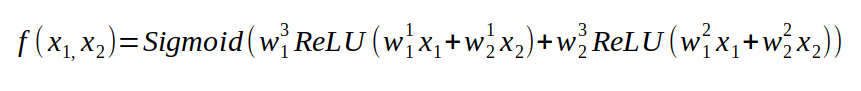
The formula above is what I mean by linear combination. We’re going to take inputs, multiply them by corresponding weights and sum everything together. The result of that is a number. The last part — is to apply some kind non-linear function on top of it. The most popular non-linearity that is used today is actually even easier than linear function called Rectified Linear Unit (*ReLU*). The formula is the following:



Our final formula for the neuron that is drown above is:



It’s time to define a neural network. A neural network is also a mathematical function. It is defined by a bunch of neurons connected to each other. And when I say connected, I mean that the output from one neuron is used as an input to other neurons.



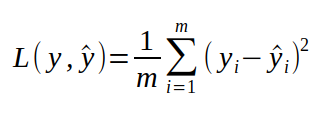
Function, defining out neural network. Superscript of w denoted to the index of the neuron. Subscript of w denotes the index of input.

 To accomplish that we’re going to apply a different activation function to our last neuron. We’re going to be using a [sigmoid](https://en.wikipedia.org/wiki/Sigmoid_function) activation. The only thing you need to know about this function is that it returns a number from 0 to 1, exactly what we want.

 It is actually not so important what formula this function has, what important is that we have complex non-linear function parametrized by some weights in a sense that we can change that function by changing the weights.

# III. Loss function

Loss function is a function that tells us, how good our neural network for a certain task. The intuitive way to do it is, take each training example, pass through the network to get the number, subtract it from the actual number we wanted to get and square it (because negative numbers are just as bad as positives).



Where y stands for the number we want to get from the network, y with a hat — the number we actually got by passing our example through the network, i — index of a training example.

 Let’s take again dogs-vs-cats for example. We have a dataset of pictures of dogs and cats labeled one if it is a dog, or zero if it is a cat. This label corresponds to y — it’s the number we want to get from network, when passing our image to it. To compute the loss function we would go over each training example in our dataset, compute *y*for that example, and then compute the function defined above. If the Loss function is big then our network doesn’t perform very well, we want as small number as possible. We can rewrite this formula, changing y to the actual function of our network to see deeper the connection of the loss function and the neural network.

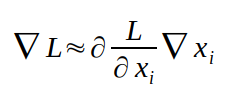
# IV. Training

When we start off with our neural network we initialize our weights randomly. Obviously, it won’t give you very good results. In the process of training, we want to start with a bad performing neural network and wind up with network with high accuracy. In terms of loss function, we want our loss function to much lower in the end of training. Improving the network is possible, because we can change its function by adjusting weights. We want to find another function that performs better than the initial one.

The problem of training is equivalent to the problem of minimizing the loss function. Why minimize loss instead of maximizing? Turns out loss is much easier function to optimize.

There are a lot of algorithms that optimize functions. These algorithms can gradient-based or not, in sense that they are not only using the information provided by the function, but also by its gradient. One of the simplest gradient-based algorithms — the one I’ll be covering in this post — is called Stochastic Gradient Descent. Let’s see how it works.

First, we need to remember what a derivative is with respect to some variable. Let’s take some easy function *f(x) = x*. If we remember the rules of calculus from high school we know, that the derivative of that is one at every value of x. What does it tell us ? The derivative is the rate of how fast our function is changing when we take infinitely small step in the positive direction. Mathematically it can be written as the following:



Which means: how much our function changes(left term) approximately equals to derivative of that function with respect to some variable *x* multiplied with how much we changed that variable. That approximation is going to be exact when we step we take is infinitely small and this is very important concept of the derivative. Going back to our simple function *f(x) = x,*we said that our derivative is 1, which means, that if take some step epsilon in the positive direction, the function outputs will change by 1 multiplied by our step epsilon which is just epsilon. It’s really easy to check that that’s rule. That’s actually not even an approximation, that’s exact. Why ? Because our derivative is the same for every value of *x.*That is not true for most functions. Let’s look at a slightly more complex function *f(x) = x².*

From rules of calculus we know, that the derivative of that functions is *2x.*It’s easy to check that now if we start at some value of *x*and make some step epsilon, then how much our function changed is not going to be exactly equal to the formula given above.

Now, gradient is vector of partial derivatives, whose elements contains derivatives with respect to some variable on which function is dependent. With simple functions we’ve considering so far, this vector only contains one element, because we’ve only been using function which take one input. With more complex functions(like our loss function), the gradient will contain derivatives with respect to each variable we want.

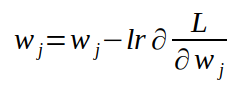
How can we use this information, provided for us by derivatives, in order to minimize some function ? Let’s go back to our function *f(x) = x².*Obviously, the minimum of that function is at point *x = 0*, but how would a computer know it ? Suppose, we start off with some random value of *x*and this value is 2. The derivative of the function in that in *x = 2*equals 4. Which means that is we take a step in positive direction our function will change proportionally to 4. So it will increase. Instead, we want to minimize our function, so we can take a step in opposite direction, negative, to be sure that our function will decrease, at least a little bit. How big of a step we can take ? Well, that’s the bad news. Our derivative only guarantees that the function will decrease if take infinitely small step. We can’t do that. Generally, you want to control how big of step you make with some kind of hyper-parameter. This hyper-parameter is called *learning rate*and I’ll talk about it later. Let’s now see what happens if we start at a point *x = -2.*The derivative is now equals -4, which means, that if take a small step in positive direction our function will change proportionally to -4, thus it will decrease. That’s exactly what we want.

When *x > 0*, our derivative greater than zero and we need to go in negative direction, when *x < 0*, the derivative less than zero, we need to go in positive direction. We always need to take a step in the direction which is opposite of derivative. Let’s apply the same idea to gradient.

Gradient is vector which points to some direction in space. It actually point to the direction of the steepest increase of the function. Since we want minimize our function, we’ll take a step in the opposite direction of gradient. Let’s apply our idea.

In neural network we think of inputs *x,*and outputs *y* as fixed numbers. The variable with respect to which we’re going to be taking our derivatives are weights *w,*since these are the values we want to change to improve our network.

If we compute the gradient of the loss function w.r.t our weights and take small steps in the opposite direction of gradient our loss will gradually decrease until it converges to some local minima. This algorithms is called Gradient Descent. The rule for updating weights on each iteration of Gradient Descent is the following:



For each weight subtract the derivative with respect to it, multiplied by learning rate.

lr in the notation above means learning rate. It’s there to control how big of a step we’re taking each iteration. It is the most important hyper-parameter to tune when training neural networks. If you choose learning that is too big, then you’ll make steps that are too large and will ‘jump over’ the minimum. Which means that your algorithms will diverge. If you choose learning rate that is too small, it might take too much time to converge to some local minima.

 In moder literature and very commonly, when people say Stochastic Gradient Descent(SGD) they actually refer to Mini-batch gradient descent. Most deep learning frameworks will let you choose batch size for SGD.