Nov 27, 2017 · 15 min read

How do we 'train' neural networks ?

I. Introduction

This is part 1 of my planned series on optimization algorithms used for 'training' in Machine Learning and Neural Networks in particular. In this post I cover Gradient Descent(GD) and its small variations. In the future I plan to write about some other popular algorithms such as:

1. SGD with momentum.
2. RMSprop.

3. Adam.
4. Genetic Algorithm.

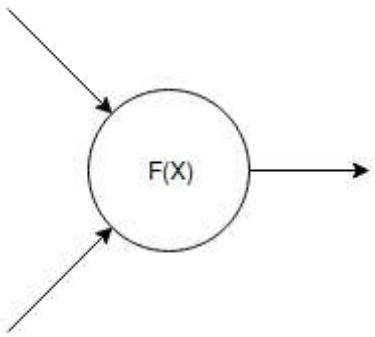
I'll put links above as finish writing about them.

Today I'll start off with very brief introduction of neural networks just enough to understand concepts I will be talking about. I'll explain what a Loss function is and what it means to 'train' neural network or any other Machine Learning model. I do not claim my explanation to be full, deep introduction to neural networks and, in fact, hope that you're already familiar with these concepts. If you want a better understanding of what is going on in neural networks I provide a list of resources to learn at the end of my post.

I will be explaining everything on the example of dogs vs. cats competition which ran several years ago on kaggle. In the competition we're faced with a task of recognizing whether a dog or a cat is appearing on a given image.

II. Let's define a Neural Network

Artificial Neural Networks(ANN) were inspired by what is actually going on in your brain. And while these analogies are pretty loose, ANNs have several similarities with their biological 'parent'. They consist of some number of neurons. So let's take a look at a single neuron.



Single Perceptron.

We're going to consider a little bit modified version of the simplest model of a neuron proposed by Frank Rosenblatt in 1957 called 'Perceptron'. All modifications I make are for the sake of simplicity, because I'm not going for a deep explanation of a neural nets. I'm only trying to give you an intuition of what is going on.

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 x_k where k stands for index of input. For each input x_k neuron assigns another number w_k. A vector consisting of these numbers w_k 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.

$$f(x_1, x_2) = w_1 x_1 + w_2 x_2$$

Linear combination of weight and inputs.

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:

$$ReLU(x) = max(x, 0)$$

Rectified Linear Unit formula.

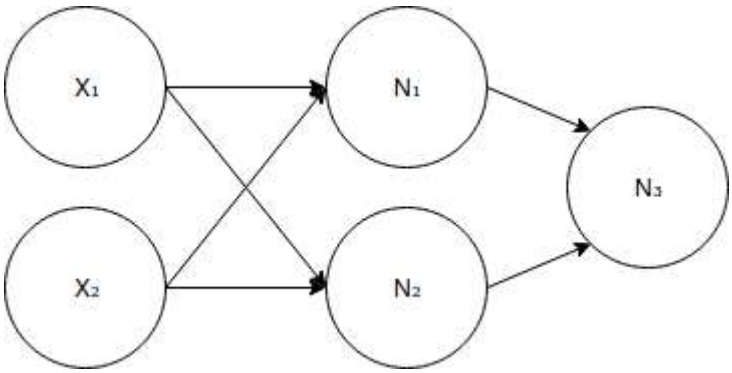
If our number is greater than zeros then we're going to take that number as is, if it's less than zero then we'll take zero instead. This non-linear function applied on top linear in neuron is called activation function. The reason we have to have some kind of non-linear function will be apparent later. To sum up, the neuron is a function that takes some fixed number of inputs and outputs a single number—its activation. Our final formula for the neuron that is drown above is:

$$f(x_1, x_2) = max(0, w_1 x_1 + w_2 x_2)$$

Neuron that takes two numbers as input.

Getting a little bit ahead of myself, If we take dogs vs. cats as an example, we're going to passing our images as inputs to neurons. You might be wondering how can we pass an image when I defined a neuron to be a function. You should remember, that the way we store images in a computer is by representing it as an array of numbers, each number indicating the brightness of a given pixel. So, the way to pass it to a neuron is to take that 2D array(or 3D in case of colored images), flatten it in a row to get a 1D vector and pass all those number to a neuron. Unfortunately, It makes our network dependent on image size in a way that we're only going to be able to process images of a given size defined by the network. Modern neural networks have found the to deal with this problem, but for now we're going to have that restriction.

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. Let's take a look at a very simple neural network and hope it will make it clearer.



Simple neural network.

The network, defined above, has 5 neuron. As you can see these neurons are stacked in 3 fully connected layers, that is each neuron from one layer is connected to each neuron from the following layer. How many layers you have in a network, how many neuron in each layer and how they are connected—all these choices define an *architecture* of the network. First layer, consisting of 2 neurons, is called an input layer. The neuron in this layer are not in fact neurons as I described them earlier, in a sense that they don't perform any

computations. They are only there to denote for the input of the network. The need for non-linearity comes from the fact, that we connect neurons together and the fact the linear function on top of linear function is itself a linear function. So, if didn't have non-linear function applied in each neuron, the neural network would be a linear function, thus not more powerful than a single neuron. The last thing to note, is that we usually want a number between 0 and 1 as an output from out neural network so that we treat is as a probability. For example, in dogs-vs-cats we could treat a number close to zero as a cat, and a number close to one as a dog. To accomplish that we're going to apply a different activation function to our last neuron. We're going to be using a sigmoid 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. Having said all that, we're ready to define a function corresponding to the network I drew above:

$$f(x_1, x_2) = Sigmoid\left(w_1^3 ReLU\left(w_1^1 x_1 + w_2^1 x_2\right) + w_2^3 ReLU\left(w_1^2 x_1 + w_2^2 x_2\right)\right)$$

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

As a result, we have some kind of function, that takes some numbers and outputs another number between 0 and 1. 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

The only thing left to define before I start talking about training is a 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).

$$L(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

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.

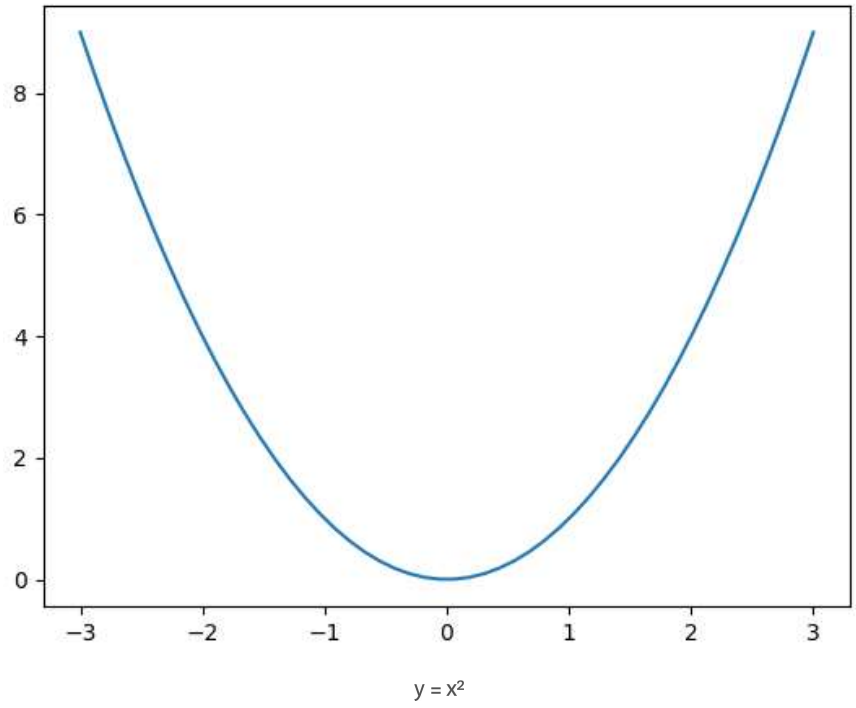
The problem of training is equivalent to the problem of minimizing the loss function. Why minimize loss instead of maximizing? Turns out loss is much easier function to optimize.

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:

$$\nabla L \approx \partial \frac{L}{\partial x_i} \nabla x_i$$

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 $f(x) = x^2$.



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 $f(x) = x^2$. 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.

Have noticed a pattern here ? 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:

$$w_j = w_j - lr \, \partial \frac{L}{\partial w_j}$$

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. People have developed some very good techniques for finding an optimal learning rate, but this goes beyond the scope of this post. Some of them are described in my other post 'Improving the way we work with learning rate'.

Unfortunately, we can't really apply this algorithm for training neural networks and the reason lies in our formula for the loss function.

As you can see from what we defined above, our formula is the average over the sum. From calculus we know, that derivative of the sum is the sum of the derivatives. Therefore, in order to compute the derivatives of the loss, we need to go through each example of our dataset. It would be very inefficient to do it every iteration of the Gradient Descent, because each iteration of the algorithm only improves our loss by some small step. To solve this problem, there's another algorithm, called Mini-batch Gradient Descent. The rule for updating the weights stays the same, but we're not going to be calculating the exact derivative. Instead, we'll approximate the derivate on some small mini-batch of the dataset and use that derivative to update the weights. Mini-batch isn't guaranteed to take steps in optimal direction. In fact, it usually won't. With gradient descent, if choose small enough learning rate, your loss is guaranteed to decrease every iteration. With mini-batch that's not true. Your loss will decrease over time, but it will fluctuate and be much more 'noisy'.

The size of the batch to use for estimating the derivatives is another hyper-parameter that you'll have to choose. Usually, you want as big batch size as your memory can handle. But I've rarely seen people use batch size much larger than around 100.

Extreme version of mini-batch gradient descent with batch size equals to 1 is called Stochastic Gradient Descent. 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.

That's it on Gradient Descent and its variations. Recently, more and more people have been using more advanced algorithms. Most of them are gradient-based and actually based on SGD with slight modifications. I'm planning on writing about those as well.

VII. Backpropagation

The only thing left to say about gradient-based algorithms is how we compute gradient. The most fast method for calculating would be analytically find the derivative for each neural network architecture. I guess, I shouldn't event say that this is an insane idea, when it comes to neural networks. The formula we defined above for a very simple neural network was hard enough to get stuck trying to find all the derivatives and we only had 6 parameters. Modern architectures have millions of them.

The second way to go about it, and, in fact, the easiest to implement, is to approximate the derivative with the following formula we know from calculus:

$$\partial \frac{f}{\partial x_i} = \frac{f(x_1, x_2, \dots, x_i + \nabla, \dots, x_n) - f(x_1, x_2, \dots, x_i, \dots, x_n)}{\nabla}$$

While it is super easy to implement, it's way too much computationally expensive to do so.

The final way to calculate the derivatives, which gives a good balance on how hard it is and how computationally expensive it is, is called backpropagation. Discussing this algorithm goes beyond the scope of this post, but if you want to learn more about it, go to last section of this post where I list number of recourses to learn more about neural networks.

VI. Why will it work ?

When I first learned about neural networks and how they worked, I understood all the equations, but I wasn't quite sure why they worked. The idea that we can take some function, then take some derivatives and end up with algorithm that can tell a dog from a cat on an image seemed a bit surreal to me. Why I can't give you a really good intuition on why neural nets work so well, there are some aspects you should note.

1. Any problem we solve with neural networks has to be expressed in some mathematical way. For dogs and cats it's the following: We need to find a functions, which takes all the numbers from an image and outputs a probability of it being a dog. You can actually define any classification problem this way.

2. It might not be clear, why there is such a function that can tell a dog from a cat on a given image. And the idea here is that, as long as you have some dataset with inputs and labels, there's always going to be a function that works really well on a given dataset. The problem is that function is going to be incredibly complex. And neural networks come to help. There's a "Universal approximation theorem", which says that neural network with just one hidden layer can approximate any function as good as you want. Now, It's not clear, why, even if we find approximation of that function, it's going to be working just as good on a new dataset, the one neural network didn't see during training. This is called a generalization problem and it's an open research problem. The research showed that SGD has a 'self-generalization' effect. But we still don't understand that problem really well.

VII. Where to learn more

Some of the resources I found really useful when learning about neural networks:

1. fast.ai courses provide two excellent courses on practical deep learning for coders and a fantastic course on computational linear algebra. It's a great place to start coding neural networks as quick as possible while learning more on theory of neural networks as you go deeper in the courses.

2. neuralnetworksanddeeplearning.com book is a great online book about basic. theory behind neural networks. The author explains the math you need to know in a very good way. He also provides and explain the code to implement the neural network from scratch without using any deep learning frameworks.

3. Andrew Ng's courses on deep learning course on coursera is excellent as well to learn more about neural networks, starting with very simple networks and going further to convolutional networks and more!

4. 3Blue1Brown youtube channel has some great videos to help you with understanding deep learning and linear algebra. They provide great visualizations and a very intuitive way to think about the math and neural networks.

5. Stanford CS231n class on Convolutional Neural Networks for visual recognition is a great place to learn more about deep learning and CNNs in particular.