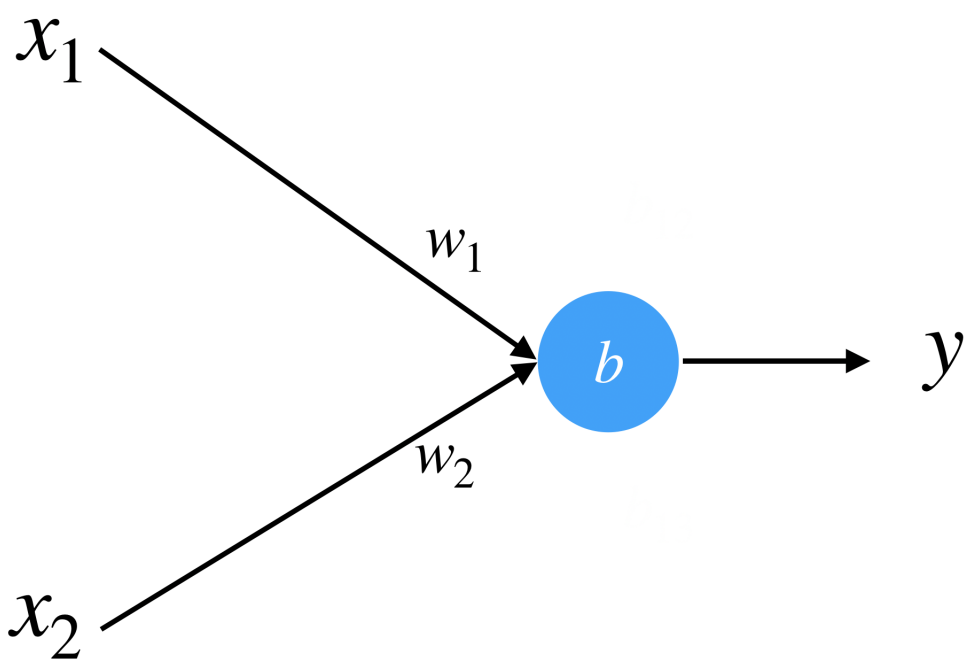
Neural Network Math / Training Process

June 2, 2020

Consider the following neural network. This is essentially a single neuron, but it demonstrates the concepts just as well as a more complicated one.



This network can be written as

$$\begin{aligned} f(x_1, x_2) &= x_1 w_1 + x_2 w_2 + b \\ y &= \tanh\left[f(x_1, x_2)\right] \end{aligned}$$

Here I have chosen to apply a hyperbolic tangent activation function to the output of the neuron. Like I said, this choice is arbitrary and it is generally a good idea to try different activation functions when you are experimenting with a dataset. The training process will start by evaluating the function (neural network), for all of the inputs that you have available (your dataset). The following definitions are helpful,

$$\begin{aligned} \mathbf{X} &= \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\} \\ \mathbf{x}_i &= \{x_{i1}, x_{i2}\} \\ \tilde{\mathbf{Y}} &= \{\tilde{\mathbf{y}}_1, \tilde{\mathbf{y}}_2, \ldots, \tilde{\mathbf{y}}_N\} \\ \tilde{\mathbf{y}}_i &= \{\tilde{y}_{i1}\} \\ \mathbf{P} &= \{p_1, p_2, \ldots, p_M\} = \{w_1, w_2, b\} \\ N &\equiv \text{the number of datapoints in your dataset} \\ M &\equiv \text{the number of parameters in your model} \end{aligned} \tag{1}$$

I have essentially defined the inputs and their corresponding known outputs as two dimensional arrays. It is completely equivalent to treat them as matrices, but I'm using the "array" terminology here, because we are dealing code. Here, I use the tilde to denote the actual, known output for each input. The output of the neural network is

$$\mathbf{y}_i = \{\tanh\left[f\left(\mathbf{x}_i\right)\right]\} \tag{2}$$

The curly brackets denote that the output for each set of inputs is an array with a single element. This is because, in general, a neural network can have multiple outputs associated with each input. In this particular case it seems excessive, but in general it is necessary. At each iteration of the training, eq. 2 will be evaluated for every input. I'll define the loss function for this network as the mean squared error (one of many options)

$$\sigma\left(\mathbf{Y},\, \tilde{\mathbf{Y}}\right) = \frac{1}{N}\sum_{i=1}\left(\tilde{y}_i - y\right)^2 \tag{3}$$

After eq. 2 is calculated, eq. 3 will be calculated. PyTorch will then use chain rule and its internal computer algebra system to calculate the following,

$$\begin{aligned} \mathbf{G} &= \{g_1,\, g_2,\, \ldots,\, g_M\} \\ g_i &= \frac{\partial \sigma}{\partial p_i} \end{aligned} \tag{4}$$

where each p_i is one of the parameters (weight or bias) that defines the model. In practical models, the number of elements in the array in eq. 1 will be from 10^3 to 10^5. The most simple training algorithm available is known as Stochastic Gradient Descent (SGD). At each iteration of the training process, the algorithm will adjust the parameters of the model based on the following rule

$$\begin{aligned} p_i^{j+1} &= p_i^j - \alpha g_i \\ \alpha &\equiv \text{learning rate} \end{aligned}$$

Here, the superscript on p denotes the step at which p takes that value. p^{j+1} would mean "the value of p at the next step", not "p to the power of $j+1$". In this way, the algorithm will make incremental steps towards local minima in the neighborhood of the initial conditions of the neural network. This is an example of the most simple optimization algorithm available. The most important thing to understand is that almost all of them make use of gradients to make decisions about how to modify the parameters of the model with each step.

This process has several major problems

1. If the learning rate is too high, it will overshoot local minima by taking steps that are too large.
2. If the learning rate is too low, it will only converge to local minima which may be very far from more optimal minima.
3. If the learning rate is too low it may take a very long time to converge to a local minima.
4. First order partial derivatives with respect to parameters are not a complete picture of the error surface of a function, see below.

The stochastic gradient descent method (much like almost all optimization methods) will modify every parameter by some amount at each step in the training process. By definition, the partial derivatives in eq. 4 are the rates of change of the error with respect to a change in a specific parameter **with all other parameters held constant**. Despite this, the SGD algorithm changes all of the parameters by at least some small amount at each step. In

principle, the following situation could occur, for a sufficiently complicated problem

$$\frac{\partial \sigma}{\partial p_1} = -1$$
$$\frac{\partial \sigma}{\partial p_2} = -1$$
$$\frac{\partial^2 \sigma}{\partial p_1 \partial p_2} = 1$$

In this case, the algorithm would take a step based on the first order gradients without knowing that a simultaneous step of both parameters will actually increase the error! A perfect algorithm would calculate all possible combinations of mixed partial derivatives. This isn't possible in practice though. For example, a network with 1000 parameter would have roughly 1000^{1000} possible combinations of mixed partial derivative (going up to arbitrary order). That number is so large that most calculators cant even compute it. In practice, most algorithms settle for only first order derivatives in order to save time. The moral of the story is that optimization algorithms are not perfect and sometimes they can fail outright.