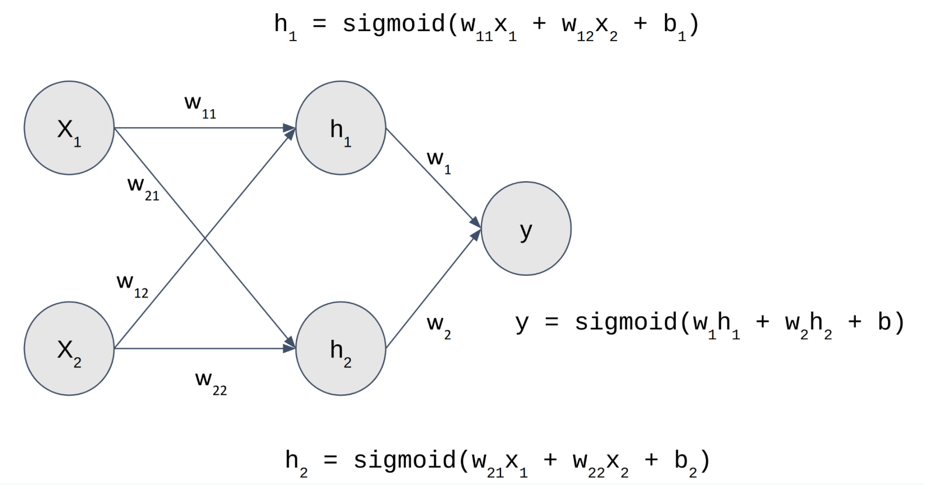
Lecture 15 - Gradient Descent and Backpropagation

Multilayer networks enable the classification of data distributions that are not manageable by single-layer networks. However, as data distributions grow more complex, the depth of the hidden layer or the number of neurons within it must also increase, complicating the identification of appropriate weights for each hidden layer.

The loss function, which has the weights and biases of the neural network as its independent variables, must be minimized to yield optimal predictions. The process of identifying these optimal weights and biases employs gradient descent.

Let's take a look at a neural network with a single hidden layer, in a simplified way.


$h_1 = \text{sigmoid}(w_{11}x_1 + w_{12}x_2 + b_1)$
X_1
X_2
w_{11}
w_{21}
w_{12}
w_{22}
h_1
h_2
w_1
w_2
y
$y = \text{sigmoid}(w_1h_1 + w_2h_2 + b)$
$h_2 = \text{sigmoid}(w_{21}x_1 + w_{22}x_2 + b_2)$

Let's take a closer look at the equation for the final output layer:

$$y = \text{sigmoid}(w_1h_1 + w_2h_2 + b)$$

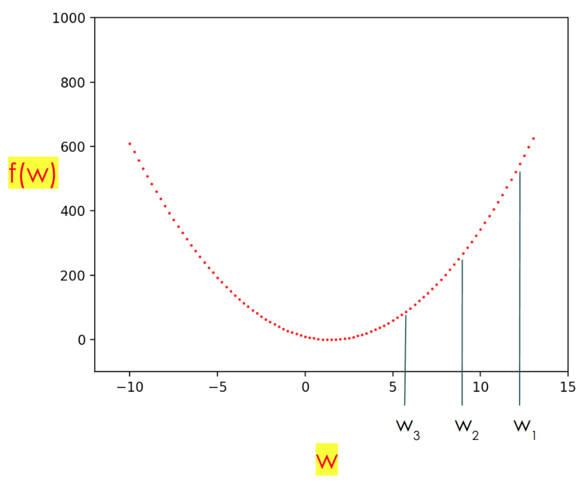
To calculate this y value, we first take x1 and x2 from the training data at the input layer. Using the randomly initialized w11, w21, w12, w22, b1, and b2, we compute h1 and h2. Now, with h1 and h2, we can compute the equation for the final output layer in a similar manner.

A key point to note is that the y value changes depending on the parameters, that is, the weights and biases. In other words, y is a function of weights and biases.

Next is the calculation of the loss function. The loss function is a function that calculates the difference between our predicted y and the y in the actual training data. The loss function can be simply expressed as follows.

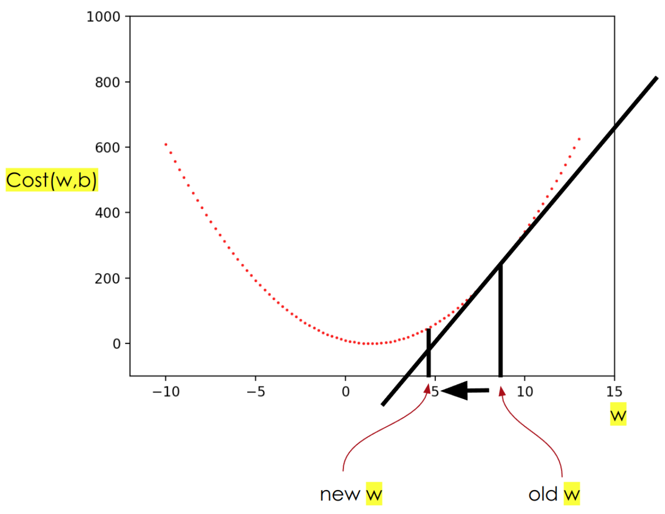
L(y’, y)

Given that the y value is provided in the training data, the loss function is still a function of the weights and biases. Therefore, let's simply plot the loss function as a graph.



(In this case, we've included the bias in w for now.) f(w) is the value of the loss function and w is the parameter. That is, different losses are calculated depending on w. Because our goal is to minimize the loss, we need to find the w that makes f(w) minimal. We will use a method called gradient descent to find the optimal w.

Gradient Descent Algrithm



The extent to which we move from the starting point (old w) is determined by the gradient of the loss function and the learning rate. To find out the gradient at the starting point, you need to differentiate the loss function. If the gradient is positive, w must move in the negative direction. Conversely, if the gradient is negative, w must move in the positive direction.

At this time, w moves by the size of the gradient, but you can move more or less using the learning rate. Therefore, the update formula for w is as follows.

$$w' = w - \alpha \cdot \frac{dL(y, y')}{dw}$$

Let's reiterate: our loss function is expressed in terms of 'w', and 'alpha' represents the learning rate, which is a hyperparameter that needs to be manually set. If the learning rate is set too high, although training will be faster, it could lead to over-optimization and consequently cause exploding gradients. On the other hand, if we set the learning rate too low in an effort to cautiously approach the optimum, training might become excessively slow.

Up until now, we've discussed how to update a single weight. However, in practice, neural networks may contain hundreds of thousands of weights, making it impractical to manually compute the derivative and adjust the loss function for each one. So, what's the solution?

Backpropagation

The backpropagation algorithm is a procedure where error is propagated from the output layer of a neural network back to the input layer, thus moving in the opposite direction to the feedforward process. Essentially, we want to quantify the impact each node (including weights and biases) in the hidden and input layers has on the value of the loss function. Put differently, we want to determine the derivative of the loss function with respect to each node. Knowing the derivative with respect to the weights specifically allows us to update them.

Before we dive into differentiating within the context of neural networks, let's take a moment to review some basic principles of differentiation. The derivative of the following expressions are as follows, correct?

$$f(x) = 3 \qquad \frac{\partial f}{\partial x} = 0$$

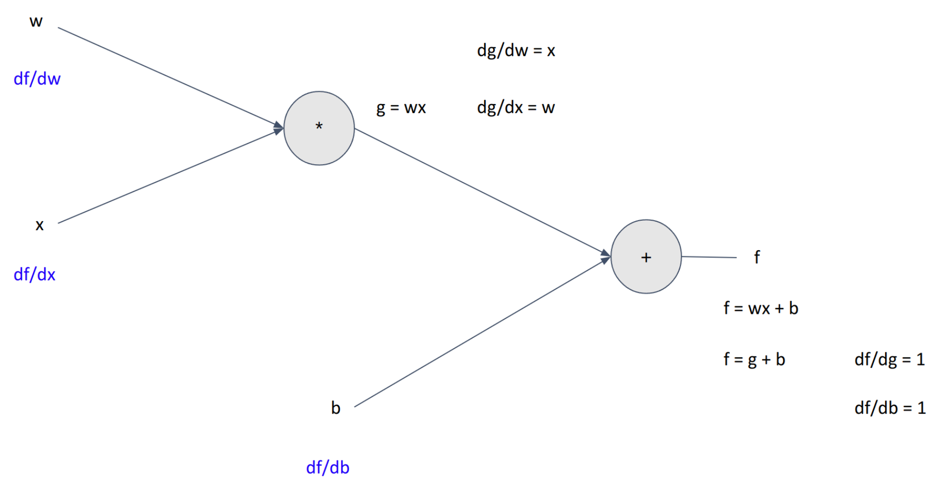
$$f(x) = x \qquad \frac{\partial f}{\partial x} = 1$$

$$f(x) = 2x \qquad \frac{\partial f}{\partial x} = 2$$

Let's also review the chain rule.

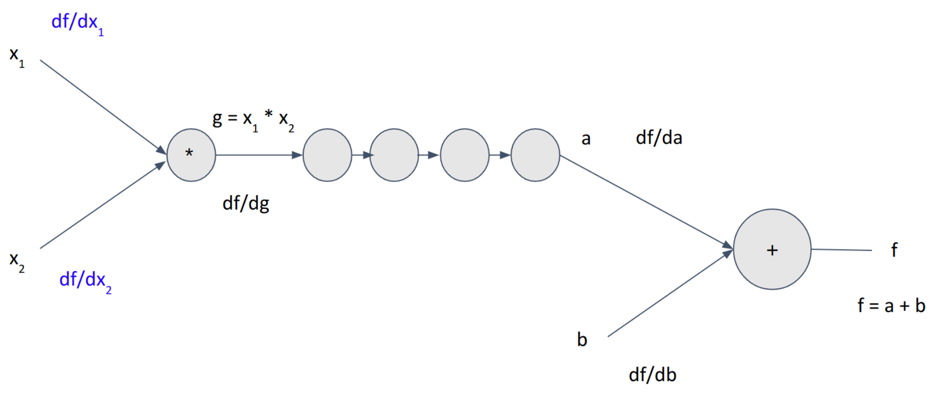
Given $f(g(x))$, $\frac{\partial f}{\partial x} = \frac{\partial g}{\partial x} \cdot \frac{\partial f}{\partial g}$

To explain backpropagation, let's take a look at a simple example.



Here we have a neural network representation of the function f = wx + b. It consists of input nodes w, x, b; the activation function in the hidden layer is *, and the one in the output layer is +. As previously mentioned, what we're interested in is how much influence the input values, i.e., w, x, b, have on the final output value. Knowing the gradient allows us to update these inputs. So how can we calculate the derivative of f with respect to w, df/dw? We apply the chain rule, which means we use **df/dw = dg/dw * df/dg**. Knowing that dg/dw is x and df/dg is 1, we can see that df/dw equals x.

Let's take a look at a slightly more complex case, similar to a deep neural network.



Our goal is to understand the influence of x_1 on the final output layer. In other words, we want to calculate the derivative of x_1. Even if the network is lengthy, as long as we know the derivative with respect to the immediate next value (i.e., g), we can easily calculate df/dx_1 using the chain rule. Of course, the derivative with respect to g can also be calculated from the derivative of the next hidden layer. (By utilizing the chain rule in this way, not only x_1 but also a derivative with respect to any weight w can be calculated.). Thus, the derivative calculation starts from the output layer and goes down to the input layer in order. That's why it's called backpropagation.