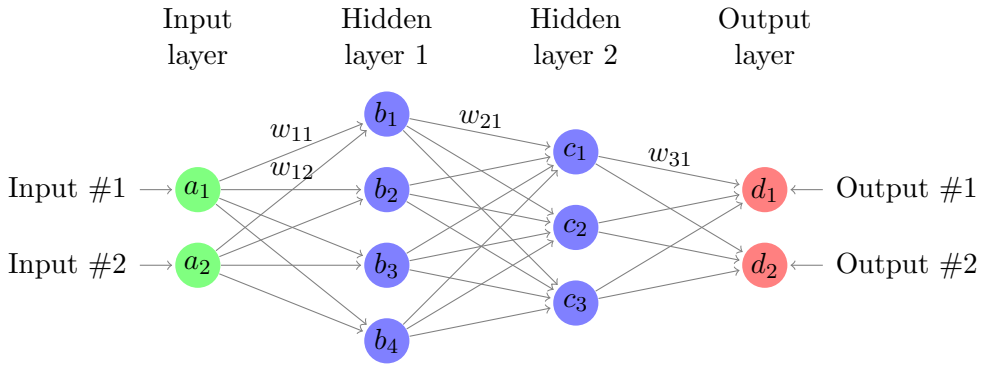
1 Introduction

In my GitHub project general purpose network I tried to figure out, how to apply backpropagation to a network of previously unspecified size. I wanted someone to be able to create a network with a size of his choosing, without the backpropagation algorithm I implemented being broken (unlike my previous project).
Most of the resources about backpropagation I found extremely confusing and the few that were helpful, like this amazing blogpost, derived the gradients per hand for every single weights, not something that I would like to do for large networks.
So I decided to try and figure out an algorithm that works for a network of any size without it having to be specified before and doesn't calculate the gradient for every single weights one after another. This is basically my attempt at explaining backpropagation, with an example, in a more understandable way than most resources I found.
I should note that while I am pretty sure that the algorithm I came up with works, and that I found that it is done pretty much the same way in this book, I don't know about the differences to more advanced projects than my simple exercise project. If you find any error or have some helpful/interesting resource to share, feel free to contact me/create an issue on GitHub.
I will not include biases in this example (or not yet), but once you understood this article, or rather the principle I'm trying to explain, including biases into the backpropagation is very simple.

2 Requirements

To understand what I'm doing in this article, you need to know what the partial derivative is and how to derive it (unless you want to take my word for it).
Also some knowledge about matrix-vector multiplication or a cheatsheet would be helpful. The german wikipedia page has some great images displaying how different multiplications between matrices and vectors work.
The network used in this example is the same as for the other documents in this project and can also be found here (with slight notational differences), but for the sake of completeness, I added it in section 3.

3 The network



The notation of the network should mostly be self explanatory. Each layer is denoted by a letter, starting with "a" at the input layer and then going upwards until "d" for the output layer.
The neurons in each layer are labeled by the letter of its layer and a number as its index, starting with 1 at the top of the layer and counting upwards, so a_1 would be the first neuron in layer a.
A weight is denoted by the letter w and indexed by a letter and a number. The letter is layer the weight leads to and the number the weights position in the layer. For example w_{a1} is the first weight in the first layer, connecting a_1 and b_1. Weight w_{a2} would be the weight connecting a_2 and b_1, w_{a3} the neurons a_1 and b_2 and so on.

4 Forward propagation

Before starting with the backpropagation part, let's look at how we forward propagate inputs to get our corresponding output. Here I will establish the terminology needed when explaining backpropagation and it will help to understand the process of getting the gradient.
We have two inputs a_1 and a_2, lets say we use two binary inputs

$$a_1 = 1, a_2 = 0,$$

that we could use when learning a logic gate.

As weights we will use:

$$W_b = \begin{bmatrix} w_{b1} & w_{b2} \\ w_{b3} & w_{b4} \\ w_{b5} & w_{b6} \\ w_{b7} & w_{b8} \end{bmatrix} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \\ 0.7 & 0.8 \end{bmatrix}$$

$$W_c = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{c5} & w_{c6} & w_{c7} & w_{c8} \\ w_{c9} & w_{c10} & w_{c11} & w_{c12} \end{bmatrix} = \begin{bmatrix} 0.25 & 0.30 & 0.35 & 0.40 \\ 0.45 & 0.50 & 0.55 & 0.60 \\ 0.65 & 0.70 & 0.75 & 0.80 \end{bmatrix}$$

$$W_d = \begin{bmatrix} w_{d1} & w_{d2} & w_{d3} \\ w_{d4} & w_{d5} & w_{d6} \end{bmatrix} = \begin{bmatrix} 0.3 & 0.4 & 0.5 \\ 0.6 & 0.7 & 0.8 \end{bmatrix}$$

Since the net value (before applying an activation function like sigmoid) of a neuron is the sum of all its weights multiplied with its corresponding inputs (e.g. the net value of the neuron b_1 $net_{b_1} = w_{b1} * a_1 + w_{b2} * a_2$, we can use matrix multiplication to calculate all net values for a layer in a single step. When using matrix multiplication to calculate all net inputs for layer b it looks like this

$$net_b = W_b * a = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \\ 0.7 & 0.8 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.1 * 1 + 0.2 * 0 \\ 0.3 * 1 + 0.4 * 0 \\ 0.5 * 1 + 0.6 * 0 \\ 0.7 * 1 + 0.8 * 0 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.3 \\ 0.5 \\ 0.7 \end{bmatrix}$$

Next we want the output out_b of layer b, so we have to apply our activation function, in this case the sigmoid function, to every single element in the vector net_b.
Just as a reminder, the sigmoid function is $sig(x) = \frac{1}{1+e^{-x}}$, so if you are following this example you are probably going to need a calculation from here on out.
Applying the sigmoid function we get

$$out_b = sig(net_b) = sig(\begin{bmatrix} 0.1 \\ 0.3 \\ 0.5 \\ 0.7 \end{bmatrix}) = \begin{bmatrix} 0.5250 \\ 0.5744 \\ 0.6225 \\ 0.6682 \end{bmatrix}$$

I rounded the results to 4 decimals and will continue the calculations with the rounded values. This goes for both this current calculation, as well as the rest of the article.

If we continue propagating these results forward until the output layer we get

$$net_c = W_c * out_b = \begin{bmatrix} 0.25 & 0.30 & 0.35 & 0.40 \\ 0.45 & 0.50 & 0.55 & 0.60 \\ 0.65 & 0.70 & 0.75 & 0.80 \end{bmatrix} * \begin{bmatrix} 0.5250 \\ 0.5744 \\ 0.6225 \\ 0.6682 \end{bmatrix} = \begin{bmatrix} 0.7887 \\ 1.2667 \\ 1.7448 \end{bmatrix}$$

$$out_c = sig(net_c) = sig(\begin{bmatrix} 0.7887 \\ 1.2667 \\ 1.7448 \end{bmatrix}) = \begin{bmatrix} 0.6876 \\ 0.7802 \\ 0.8513 \end{bmatrix}$$

$$net_d = W_d * out_c = \begin{bmatrix} 0.3 & 0.4 & 0.5 \\ 0.6 & 0.7 & 0.8 \end{bmatrix} * \begin{bmatrix} 0.6876 \\ 0.7802 \\ 0.8513 \end{bmatrix} = \begin{bmatrix} 0.9441 \\ 1.6397 \end{bmatrix}$$

$$out_d = sig(net_d) = sig(\begin{bmatrix} 0.9441 \\ 1.6397 \end{bmatrix}) = \begin{bmatrix} 0.7199 \\ 0.8375 \end{bmatrix}$$

Now that we reached the output of layer d we also have the output of our network. Our first output neuron d_1 has the value 0.7199 and our second one d_2 the value 0.8375.
These two values are our networks guess for what the output should be when the input values we chose earlier are plugged in. With that we are basically done with the forward propagation part, since no matter the weights, the process will remain the same. Before moving on to the backpropagation part though, I would like to mention something noteworthy, that will be helpful for understanding backpropagation.
When forward propagating, we started with our inputs and in the end got our outputs. The way we calculated them is by multiplying matrices, applying an activation function, then again multiplying matrices and so on. Basically the entire forward propagation process is nothing but the chaining of functions, the matrix multiplication being one and the activation function being another. Every intermediate result, e.g. net_b is immediately passed on to the next function until we get the output.
This is noteworthy, because later we will need to derive this gigantic chain of functions, so we can train our network. For this, as you might have guessed, the chain rule is perfect.

5 Backpropagation

5.1 The error

Before we can start with the backpropagation itself, we need something we can actually propagate back. Since we want to train the network, we need to know how wrong our guesses were for those inputs and then use the gradient (derivative) to move into the right direction, so it improves.
For this we will use an error function, the squared error

$$E_{total} = \sum \frac{1}{2}(target - out)^2 \tag{1}$$

In this equation we subtract the actual value we got from the target value we wanted. I multiplied by 1/2 here so it cancels out with the square when deriving the function later on.
The total error E_{total} is the sum of all those errors calculated for every single output neuron. For different inputs we could get a totally different output and so our error could be totally different as well.
If we wanted only the error for d_1 (for the inputs we chose earlier) it would be

$$E_{d_1} = \sum \frac{1}{2}(target_{d_1} - out_{d_1})$$

From here on out it will be hard to continue without a target output for out network, so let's say it takes the two inputs a_1 and a_2 and its job is to reverse, or rather switch them. This would mean out inputs and corresponding outputs are.

a_1	a_2		d_1/out_{d_1}	d_2/out_{d_2}
1	1	=	1	1
1	0	=	0	1
0	1	=	1	0
0	0	=	0	0