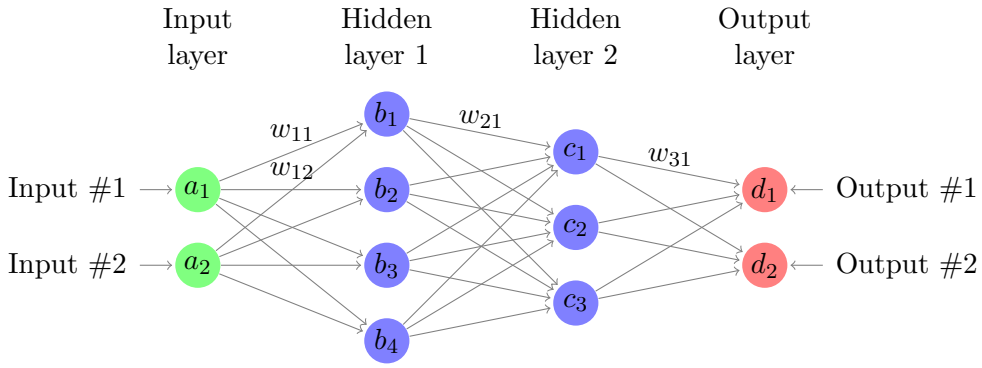
1 Introduction

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 our inputs and corresponding outputs are.

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

Note: You might have noticed that I'm using the output even though we are doing backpropagation. This is because when calculating the gradient you will always need the net and out values of every layer of the network. In other words we have to first propagate forward, save the values and then we can start backpropagating.

Our goal is to make the network learn, which simply means we want its guesses to be as close to the correct output as possible. But you could also put it in other word, we want our error to be as low as possible. Since we have a function 1 that calculates the error, our goal will be to find the minimal possible value for the function.

For this we will use gradient descent (I will not go into detail here, so if you need a refreshed, feel free to look it up) and for that we will calculate the first derivative of the error function. The only value in the function that we can really influence is the output, therefore we will take the partial derivative with respect to it.

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

Now that we have both our error function and its derivative and, in theory, can minimize the error. But right now we only derived with respect to the output, so how do we tell the network how it can get closer to that output? For that let's look at what our network consists of.
We have our inputs and our weights, as well as an activation function. Changing the inputs would make no sense, after all we want to come closer to our target values for exactely those inputs.
Similar for the activation function, while we can choose between several functions and could even choose different ones for different layers, they are fixed once chosen and we can't change them.
So all that is left to change are out weights, which makes sense, since those are the only values that are indepentent for each layer and since they do the actual work in our network.
In other words, the variables we want to derive the error with respect to are the weights. This means we have to derive the error with respect to every single weight to get its gradient and then adjust the network accordingly.
Let's derive for a weight in the last layer, for example w_{d1}. We then, without yet plugging in values or variables, get

$$\frac{\partial E_{total}}{\partial w_{d1}} = \frac{\partial E_{total}}{\partial out_{d_1}} \frac{\partial out_{d_1}}{\partial net_{d_1}} \frac{\partial net_{d_1}}{\partial w_{d1}} \tag{3}$$

If this step isn't clear think about what I wrote at the end of section 4.
The entire network is nothing but a chaining of functions and when deriving the error with respect to w_{d1}, we therefore apply the chain rule. When looking at it from the right (the weight first) we have all weights of the neuron multiplied, equaling the net value of d_1. This net value is chained with the activation function, equaling the out value of d_1. This again is chained, this time with the error function.
As we are deriving with respect to w_{d1}, every connection that does not contain this weight will result in 0 when derived, that is why in 3, only

out_{d_1} and net_{d_1} are relevant. Every other connection doesn't contain w_{d1} and will therefore drop out anyway.

Now let's plug in our formulas. We then get

$$\frac{\partial E_{total}}{\partial w_{d1}} = (out_{d_1} - target_{d_1}) * (out_{d_1} * (1 - out_{d_1})) * out_{c_1}$$

If you are wondering where the second parameter comes from, this is the derivative of the sigmoid function, our activation function.
The sigmoid function is $sig(x) = \frac{1}{1+e^{-x}}$ and it's derivative is
$sig'(x) = sig(x) * (1 - sig(x))$. The value we plug into sig(x) as x is (for this neuron) net_{d_1}. But since $sig(net_{d_1}) = out_{d_1}$, we can just write out_{d_1} instead.
This process is similar for all output neurons, for example

$$\frac{\partial E_{total}}{\partial w_{d4}} = \frac{\partial E_{total}}{\partial out_{d_2}} \frac{\partial out_{d_2}}{\partial net_{d_2}} \frac{\partial net_{d_2}}{\partial w_{d4}}$$

and with the values plugged in

$$\frac{\partial E_{total}}{\partial w_{d1}} = (out_{d_2} - target_{d_2}) * (out_{d_2} * (1 - out_{d_2})) * out_{c_1}$$

For the first layer (or the last, depending on the point of view) this process is relatively easy, most terms just drop out. If you go to the next layer it already gets harder, as more neurons are being influenced by the chosen weight. Continuing this by hand is tedious, but for a few layers with only a few neurons still possible without too much effort. Most networks are far more complex though and would need an overwhelming amount of work to be derived by hand. And even more important, the network would be fixed to it's current size, for every different combination of layers and neurons you would have to do the entire process again.
We would like a more generic solution, one that we can apply no matter the size of the network, just like the simply matrix multiplication when propagating forward.
In other words, we need to find some pattern that we can use and put into a formula.

5.2 Finding a pattern for backpropagation

Earlier in section 3 we split the derivative into three separate factors. The first contains the error with respect to the output of the neuron that is influenced by the weight we want to calculate the gradient for. The second factor is the output with respect the the net value and the third the net value with respect to the weight.
As the error function will only be used once to calculate the error of the network, this part of the equation is specific to the output layer.
But before we move on to different layers, let's try to find a way to calculate the gradient for all our weights in the output layer using matrix multiplication. The matrix W_d is a 2x3 Matrix, so if we want a gradient for every weight in this matrix, a 2x3 gradient matrix should be our goal.
One way to achive this is by multiplying a column vector with a row vector. The resulting matrix would then have the row count of the column vector and the column count of the row vector. To get a 2x3 matrix we would need a 2 row column vector (or 2x1 matrix) and a 3 column row vector (or 1x3 matrix).
To make this part clearer, here a short example

$$\begin{bmatrix}2\\4\end{bmatrix} * \begin{bmatrix}3 & 5 & 7\end{bmatrix} = \begin{bmatrix}6 & 10 & 14\\12 & 20 & 28\end{bmatrix} \tag{4}$$

As our output layer (layer d) has two neurons and it's preceding layer (layer c) has three neurons, lets try to use those. So how could we use our output layer as a two column vector?
For that you might want to check out equation 3 again, because for any specific output neuron d_n, the first two factors will always be the derivative of the error with respect to its output and of the output with respect to its input. Only the last factor contains a value of another layer, namely the output of the neuron that our connection (the weight we derive with respect to) leads to.
Now if the derived values for any output neuron d_n are always the same, let's calculate them. We won't need them just yet, but it hopefully clears up why they stay the same.

$$\frac{\partial E_{total}}{\partial net_{d_1}} = \frac{\partial E_{total}}{\partial out_{d_1}} \frac{\partial out_{d_1}}{\partial net_{d_1}} = (out_{d_1} - target_{d_1}) * out_{d_1} * (1 - out_{d_1})$$

$$\frac{\partial E_{total}}{\partial net_{d_2}} = \frac{\partial E_{total}}{\partial out_{d_2}} \frac{\partial out_{d_2}}{\partial net_{d_2}} = (out_{d_2} - target_{d_2}) * out_{d_2} * (1 - out_{d_2})$$

Before we continue, let's look at how our target matrix would look like, for now without concrete values

$$W_{d_g} = \begin{bmatrix} \frac{\partial E_{total}}{\partial net_{d_1}} * \frac{\partial net_{d_1}}{\partial w_{d1}} & \frac{\partial E_{total}}{\partial net_{d_1}} * \frac{\partial net_{d_1}}{\partial w_{d2}} & \frac{\partial E_{total}}{\partial net_{d_1}} * \frac{\partial net_{d_1}}{\partial w_{d3}} \\ \frac{\partial E_{total}}{\partial net_{d_2}} * \frac{\partial net_{d_1}}{\partial w_{d4}} & \frac{\partial E_{total}}{\partial net_{d_2}} * \frac{\partial net_{d_1}}{\partial w_{d5}} & \frac{\partial E_{total}}{\partial net_{d_2}} * \frac{\partial net_{d_1}}{\partial w_{d6}} \end{bmatrix}$$

I will use g subscript to denote a gradient and in this case the gradient matrix for the weights of layer d.
If we simplify the second factor of every matrix field we get

$$W_{d_g} = \begin{bmatrix} \frac{\partial E_{total}}{\partial net_{d_1}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{d_1}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{d_1}} * out_{c_3} \\ \frac{\partial E_{total}}{\partial net_{d_2}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{d_2}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{d_2}} * out_{c_3} \end{bmatrix}$$

Seeing our goal, you might already realize where this leads to. For our output layer, the gradient of a weight is always the derivative of an output neuron multiplied by the output of the neuron it is connected to (from the preceding layer).
So we have our column vector with its derivatives and we need a row vector with three values. As our preceding layer c has exactely three values and they even are the output values we want, the obvious choice would be to put them in a row vector and try out if we get our desired result.

$$\begin{bmatrix} \frac{\partial E_{total}}{\partial net_{d_1}} \\ \frac{\partial E_{total}}{\partial net_{d_2}} \end{bmatrix} * \begin{bmatrix} out_{c_1} & out_{c_2} & out_{c_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial E_{total}}{\partial net_{d_1}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{d_1}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{d_1}} * out_{c_3} \\ \frac{\partial E_{total}}{\partial net_{d_2}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{d_2}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{d_2}} * out_{c_3} \end{bmatrix}$$

And yes, we actually got exactely what we wanted. We now calculated the gradient of the output layer and, more importantly, did so with a simple matrix multiplication instead of deriving it for every weight by hand.
The only things we have to derive is the neurons, which (in code) is easily implemented by a loop.

Next, let's try and figure out if and how we can use this for further layers. Before we can get to that though, we need to know how the derivative of the next layer would actually look like. If we write it in the same form as in equation 3 we get

$$\frac{\partial E_{total}}{\partial w_{c1}} = \frac{\partial E_{total}}{\partial out_{c_1}} \frac{\partial out_{c_1}}{\partial net_{c_1}} \frac{\partial net_{c_1}}{\partial w_{c1}} \tag{5}$$

and if we split this up a little more we get

$$\frac{\partial E_{total}}{\partial w_{c1}} = (\frac{\partial E_{d_1}}{\partial out_{c_1}} + \frac{\partial E_{d_2}}{\partial out_{c_1}}) \frac{\partial out_{c_1}}{\partial net_{c_1}} \frac{\partial net_{c_1}}{\partial w_{c1}} \tag{6}$$

If you are not sure how I came up with these equations, take your time to go through the math, in the end it is only the chain rule applied over and over again.

You can see that we have three factors, one being a sum of two derivatives. Before we move on to the other derivatives, let's calculate those.

$$\frac{\partial E_{d_1}}{\partial out_{c_1}} = \frac{\partial E_{d_1}}{\partial out_{d_1}} \frac{\partial out_{d_1}}{\partial net_{d_1}} \frac{\partial net_{d_1}}{\partial out_{c_1}} = (out_{d_1} - target_{d_1}) * (out_{d_1} * (1 - out_{d_1})) * w_{d1} \tag{7}$$

$$\frac{\partial E_{d_2}}{\partial out_{c_1}} = \frac{\partial E_{d_2}}{\partial out_{d_2}} \frac{\partial out_{d_2}}{\partial net_{d_2}} \frac{\partial net_{d_2}}{\partial out_{c_1}} = (out_{d_2} - target_{d_2}) * (out_{d_2} * (1 - out_{d_2})) * w_{d4} \tag{8}$$

As long as we always derive with respect to a weight connecting to c_1 the first factor will always stay the same. This obviously goes for every neuron c_n. While the start is different when using a weight connecting to a different neuron, it is always the same when the weight is connected to the same neuron.

When looking at equations 7 and 8, we see that this also means, that the weights will stay the same.
So how can we multiply the weights times the derivative of layer d to get this first factor?
The first choice might be multiplying them just like when forward propagating, which would look like this

$$W_d * derivative(d) = \begin{bmatrix} w_{d1} & w_{d2} & w_{d3} \\ w_{d4} & w_{d5} & w_{d6} \end{bmatrix} * \begin{bmatrix} \frac{\partial E_{d_1}}{\partial net_{d_1}} \\ \frac{\partial E_{d_2}}{\partial net_{d_2}} \end{bmatrix} =?$$

But as the columns of the matrix and the rows of the vector must match up, which they don't, this won't work.
I denoted derivative(d) as the vector of the derivative of all neurons in the layer. Usually this means $derivative(x) = \frac{\partial out_x}{net_x}$. For the output layer this means I will keep using this notation later on, because otherwise equations might get quite a bit longer.
We know that our weight matrix's size is always the number of neurons in the current layer times the number of neurons in the preceding layer. But this was made for forward propagation, now that we are going the other way, we have to reverse it. For this, we will use the transpose of the matrix (nothing complicated, look it up if you are not sure what it is).
With this we get

$$(W_d)^T * derivative(d) = \begin{bmatrix} w_{d1} & w_{d4} \\ w_{d2} & w_{d5} \\ w_{d3} & w_{d6} \end{bmatrix} * \begin{bmatrix} \frac{\partial E_{d_1}}{\partial net_{d_1}} \\ \frac{\partial E_{d_2}}{\partial net_{d_2}} \end{bmatrix}$$

Looks good, let's see what our results are

$$\begin{bmatrix} w_{d1} & w_{d4} \\ w_{d2} & w_{d5} \\ w_{d3} & w_{d6} \end{bmatrix} * \begin{bmatrix} \frac{\partial E_{d_1}}{\partial net_{d_1}} \\ \frac{\partial E_{d_2}}{\partial net_{d_2}} \end{bmatrix} = \begin{bmatrix} w_{d1} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d4} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \\ w_{d2} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d5} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \\ w_{d3} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d6} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \end{bmatrix}$$

Perfect, if you compare this with equation 6, you can see that the first row in the resulting vector equals the first factor of the equation.
Since the first row is the first factor for every weight connecting layer b to c_1, the second row is the factor for every weight connecting layer b to c_2 and the third for c_3.
In other word we now have a vector with all the results we need to continue calculating the gradient for layer c.

Looking back at equation 6 again, we can now try to figure out our next factor. Well $\frac{\partial out_{c_1}}{\partial net_{c_1}}$ is just the derivative of this neuron (derivative(c_1)), which we want to multiply times the first row of the vector we just calculated. This would be if we just want the gradient for the weights connecting layer b to c_1, but we want it for all neurons c_n, so what do we do? Well, the solution is pretty obvious here, we have a vector with 3 rows with each containing the factors we need and we have a vector with 3 rows containing the derivative of layer c. So let's just multiply them (elementwise).

$$\begin{bmatrix} w_{d1} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d4} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \\ w_{d2} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d5} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \\ w_{d3} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d6} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \end{bmatrix} * derivative(c) = \begin{bmatrix} w_{d1} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d4} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \\ w_{d2} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d5} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \\ w_{d3} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d6} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \end{bmatrix} * \begin{bmatrix} \frac{\partial out_{c_1}}{\partial net_{c_1}} \\ \frac{\partial out_{c_2}}{\partial net_{c_2}} \\ \frac{\partial out_{c_3}}{\partial net_{c_3}} \end{bmatrix}$$

$$\begin{bmatrix} w_{d1} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d4} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \\ w_{d2} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d5} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \\ w_{d3} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d6} * \frac{\partial E_{d_2}}{\partial net_{d_2}} \end{bmatrix} * \begin{bmatrix} \frac{\partial out_{c_1}}{\partial net_{c_1}} \\ \frac{\partial out_{c_2}}{\partial net_{c_2}} \\ \frac{\partial out_{c_3}}{\partial net_{c_3}} \end{bmatrix} = \begin{bmatrix} (w_{d1} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d4} * \frac{\partial E_{d_2}}{\partial net_{d_2}}) * \frac{\partial out_{c_1}}{\partial net_{c_1}} \\ (w_{d2} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d5} * \frac{\partial E_{d_2}}{\partial net_{d_2}}) * \frac{\partial out_{c_2}}{\partial net_{c_2}} \\ (w_{d3} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d6} * \frac{\partial E_{d_2}}{\partial net_{d_2}}) * \frac{\partial out_{c_3}}{\partial net_{c_3}} \end{bmatrix}$$

This might get a little confusing, since there is quite a lot going on now, so take your time to check that this equals the formulas we devised earlier. The only difference is that these are the derivatives for all neurons in layer c, not only one.
While there isn't much left until we have the gradient for layer c, continuing with this bloated matrix/vector will only make things more confusing, so I'll write it a little shorter.

$$\begin{bmatrix} (w_{d1} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d4} * \frac{\partial E_{d_2}}{\partial net_{d_2}}) * \frac{\partial out_{c_1}}{\partial net_{c_1}} \\ (w_{d2} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d5} * \frac{\partial E_{d_2}}{\partial net_{d_2}}) * \frac{\partial out_{c_2}}{\partial net_{c_2}} \\ (w_{d3} * \frac{\partial E_{d_1}}{\partial net_{d_1}} + w_{d6} * \frac{\partial E_{d_2}}{\partial net_{d_2}}) * \frac{\partial out_{c_3}}{\partial net_{c_3}} \end{bmatrix} = \begin{bmatrix} \frac{\partial E_{total}}{\partial net_{c_1}} \\ \frac{\partial E_{total}}{\partial net_{c_2}} \\ \frac{\partial E_{total}}{\partial net_{c_3}} \end{bmatrix}$$

Again, if you are not sure why those are equal, compare it to equation 5 or 6.

Alright, we are almost at the end of calculating the gradient for this layer, all that is missing, is the last factor. Checking equation 5 we that this is $\frac{\partial net_{c_1}}{\partial w_{c1}}$.
You might already have noticed, that we are now at exactely the same point as earlier, when we calculated the gradient for the output layer.
We are one step away from our gradient matrix, which once again should be the same size as the weight matrix of layer c (W_c). Since we calculated our column vector just now, all that is left is our row vector. If you need a quick refresher, check out example equation 4.
As the last factor is again the derivative of the net value of the layer with respect to a weight, we can just use the out values of layer b as our row vector(just like the output of layer c earlier).
We then get the following matrix

$$\begin{bmatrix} \frac{\partial E_{total}}{\partial net_{c_1}} \\ \frac{\partial E_{total}}{\partial net_{c_2}} \\ \frac{\partial E_{total}}{\partial net_{c_3}} \end{bmatrix} * \begin{bmatrix} out_{c_1} & out_{c_2} & out_{c_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial E_{total}}{\partial net_{c_1}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{c_1}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{c_1}} * out_{c_3} \\ \frac{\partial E_{total}}{\partial net_{c_2}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{c_2}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{c_2}} * out_{c_3} \\ \frac{\partial E_{total}}{\partial net_{c_3}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{c_3}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{c_3}} * out_{c_3} \end{bmatrix}$$

$$W_{c_g} = \begin{bmatrix} \frac{\partial E_{total}}{\partial net_{c_1}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{c_1}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{c_1}} * out_{c_3} \\ \frac{\partial E_{total}}{\partial net_{c_2}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{c_2}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{c_2}} * out_{c_3} \\ \frac{\partial E_{total}}{\partial net_{c_3}} * out_{c_1} & \frac{\partial E_{total}}{\partial net_{c_3}} * out_{c_2} & \frac{\partial E_{total}}{\partial net_{c_3}} * out_{c_3} \end{bmatrix}$$

And we are done. We just calculated the gradient matrix for layer c. Since this was probably quite a bit to take in, I will provide a short summary in the next section, but before we get to that, what if we had more than two layers? After all, even this example network has more than only those two. If you do backpropagation with matrix multiplication you can, just like when propagating forward, always keep going, no matter how many layers the network has. I didn't want to keep going on to the third layer, since the equations were pretty long for the second layer already and it woud therefore probably be more confusing than helpful.
To understand how you can apply this for any amount of layer I will give a short summary and a code example of how you could implement it.

6 Summary

Alright, when doing backpropagation earlier we started with calculating the gradient for the output layer. For that we needed a column vector of the size of the output layer and we needed a row vector of the size of the preceding layer (here hidden layer 2). If we multiply the column vector times the row vector we get a matrix, that has the exact same size as the weight matrix connecting both layers.
All we needed to figure out was what kind of values the vectors contained. As we wanted to multiply by the outputs of the preceding layer, the row vector contained those. The column vector, in our example, contained the derivative of the error with respect to the net value of layer d.
This is all we need to calculate the output layer gradient. But there is something important to take away here, whenever we have a column vector with the derivative of the error with respect to ANY layer's net value, we can calculate the gradient if we also have a row vector with the preceding layers output.
We did this in the second part, when calculating the gradient for layer c. We first found a way to calculate the gradient of the error with respect to the net value of c. Then we used this as our column vector and multiplied it by the row vector of layer b's outputs and the result was the gradient for layer c.

So if we know that, apart from a layers output, all we need is the derivative of the error with respect to a layers net value, how did we figure that derivative out?
Well, to figure out the derivative of the error with respect to any layer n's net output ($\frac{\partial E_{total}}{\partial net_n}$), we need to know the derivative of the error with respect to the previous layer's net value ($\frac{\partial E_{total}}{net_{n+1}}$).
Then we can do a matrix multiplication of $W_n * \frac{\partial E_{total}}{net_{n+1}}$. The result of this is a vector, which we multiply elementwise with the derivative of layer n ($\frac{\partial out_n}{\partial net_n}$).
The result of this is a vector of the size of layer n's neuron count, that contains the derivative of the error with respect to layer n's net value, so all we need to do is apply the rule from the first paragraph.
Multiply it by the output row vector of the preceding layer (n - 1). That's it you are done and can repeat this until you went through the entire network.
Since I usually find it hard to understand a topic if I first look at it and all I see are formular, I will now go through the entire network as an example.

7 Example

I will use the same weights as in the forward propagation example and again the inputs

$$a_1 = 1, a_2 = 0$$

expecting the network to switch them and therefore produce the output

$$d_1 = 0, d_2 = 1$$

Note: You can try this out yourself if you want, I tested if the chosen network can fulfill this task and it worked for me.