Yet another introduction to backpropagation

Herman Kamper
kamperh@gmail.com

14 November 2017

1. Introduction

This is my attempt at a concise introduction to backpropagation. There are also a number of other, very good introductory texts (references given at the end). One resource I would recommend going through before reading this document, is the CS231n notes on backpropagation [1].

2. The chain rule

Backpropagation is essentially the chain rule applied in a particular order. Here I first review the chain rule in its different forms.

Suppose we have variable $y = f(x)$ (i.e. variable y depends on variable x), and variable $x = g(t)$ (i.e. x in turn depends on t). Then the chain rule for functions of a single variable states [2, p. 967]:

$$\frac{dy}{dt} = \frac{dy}{dx}\frac{dx}{dt}$$

If we have a variable $z = f(x, y)$ that depends on multiple variables (x and y in this case), and variables $x = g(s, t)$ and $y = h(s, t)$ themselves depend on multiple other variables (s and t), then we can calculate partial derivatives using this version of the chain rule [2, p. 969]:

$$\frac{\partial z}{\partial s} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial s} \tag{1}$$

and similarly for $\frac{\partial z}{\partial t}$.

More generally, if we have a scalar variable $y = f(\boldsymbol{u})$ that depends on a vector $\boldsymbol{u} \in \mathbb{R}^M$, and that vector is itself computed as $\boldsymbol{u} = \boldsymbol{g}(\boldsymbol{x})$ from another vector $\boldsymbol{x}$, then the general version of the chain rule states [2, p. 970]; [3, §4]:

$$\frac{\partial y}{\partial x_i} = \sum_{j=1}^{M} \frac{\partial y}{\partial u_j}\frac{\partial u_j}{\partial x_i} \tag{2}$$

All these identities are also given on Wikipedia [4] (you might just need to scroll around a bit).

3. Backpropagation (using an example)

3.1. An example function and computational graph

Suppose we have the function

$$f(x, y) = \frac{x + \sigma(y)}{x^2 + y} \tag{3}$$

where $\sigma(\cdot)$ is the sigmoid function. This function is used only for illustration and is pretty useless otherwise.

A computational graph breaking this function into simple operations (for which we know the derivatives) is shown in Figure 1. This is similar to the graphs described in [5]. Given we perturb the input x by a little, this will cause a change in c, which would cause a change in d, and so forth, until it causes a change in the final output f. We want to know how the change in x affects the output f, and similarly for y. Given such a graph, our aim (for this particular example) is therefore to find $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, and to do so using a generic algorithm with steps that we can follow.

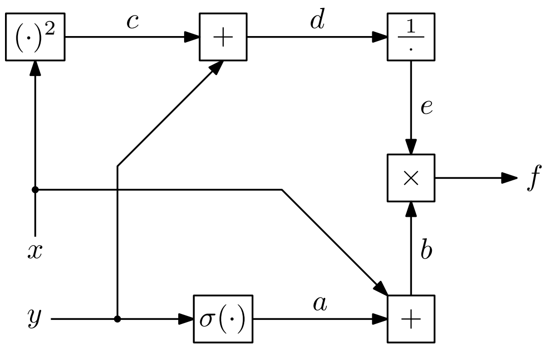

Figure 1: The computational graph for the function $f(x, y) = \frac{x+\sigma(y)}{x^2+y}$.

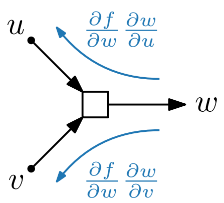
u
$\frac{\partial f}{\partial w} \frac{\partial w}{\partial u}$
w
v
$\frac{\partial f}{\partial w} \frac{\partial w}{\partial v}$

Figure 2: A generic operation within a larger computational graph. The output of the entire computational graph is assumed to be f. Backpropagation of variable w through the operation to input variables u and v are shown in blue.

3.2. The backpropagation algorithm

We will use the *backpropagation algorithm*, which has the following steps:

- **Forward pass.** Start at the inputs and proceed towards the output, calculating the output value of each of the intermediate operations in the graph. In Figure 1, the operations are indicated as blocks. Store these values for use in the backward pass.

- **Backward pass.** Start at the output of the computational graph and move backwards. For each operation you encounter, do the following:
  - Determine and calculate the derivative of the output variable with respect to each of the inputs to that operation. For the operation in the graph fragment shown in Figure 2, we would calculate $\frac{\partial w}{\partial u}$ and $\frac{\partial w}{\partial v}$. This is easy if our blocks are well-known operations for which we know the derivatives.
  - For each input variable u, add $\frac{\partial f}{\partial w}\frac{\partial w}{\partial u}$ to an accumulator for that variable, where w is the output of that operation and f is the final output of the entire computational graph. The term $\frac{\partial f}{\partial w}$ would have been calculated earlier in the backward pass. For the variables u and v in the graph fragment of Figure 2, we would update the accumulators denoted as δ_u and δ_v.
  - After adding to an accumulator all the backpropped values coming from operations that has u as input, that accumulator will contain as its final value $\delta_u = \frac{\partial f}{\partial u}$, which we can then use in subsequent backward steps for other variables. The reason why this works is described in Section 3.4.

3.3. Backpropagation applied to the example

Let us follow these steps for the example of (3), with the corresponding graph in Figure 1. We will write the code in Python.

Assume we start with $x = 3$ and $y = -4$, and our goal is to calculate the gradients at this point. We will have the following code for the forward pass:

```
# Current values
x = 3
y = -4

# Forward pass
a = 1.0 / (1 + np.exp(-y))                # (1)
b = x + a                                 # (2)
c = x**2                                  # (3)
d = y + c                                 # (4)
e = 1.0 / d                               # (5)
f = b * e                                 # (6)
```

Before we write the code for the backward pass, let us find expressions for some of the derivatives. Starting from the output of the final operation, we move backwards calculating derivatives and updating the accumulators for all the variables. We do this in the opposite order that we followed for the forward pass. So let us start with the final output: $f = b \cdot e$. Here $\frac{\partial f}{\partial b} = e$ and $\frac{\partial f}{\partial e} = b$. The values for these expressions are known, since we calculated them in the forward pass. They are now added to the accumulators for b and e. Since a and b are not inputs for any other operations, no other terms are added and we have the final accumulator values $\delta_b = \frac{\partial f}{\partial b} = e$ and $\delta_f = \frac{\partial f}{\partial e} = b$, which we can now use in subsequent backward steps.

Next we "backprop" the operation $e = \frac{1}{d}$, then $d = y + c$, and so forth. Let us see how we backprop $b = x + a$. The derivatives of the output with respect to the inputs for this operation are $\frac{\partial b}{\partial x} = 1$ and $\frac{\partial b}{\partial a} = 1$. We update the accumulators: to δ_x we add $\frac{\partial f}{\partial b}\frac{\partial b}{\partial x} = \delta_b \frac{\partial b}{\partial x} = e \cdot 1$, and to δ_a we add $\frac{\partial f}{\partial b}\frac{\partial b}{\partial a} = \delta_b \frac{\partial b}{\partial a} = e \cdot 1$; from the previous backward steps and the forward pass, we have numeric values for all the required variables.

We continue in this way through all the operations. Using `delta_d` to denote the accumulator δ_d in the Python code, the code for the backward pass are as follows (note this is exactly in the reverse order from the forward pass):

```
# Backward pass

# Backprop f = b * e
delta_b = e                             # (6)
delta_e = b                             # (6)

# Backprop e = 1.0 / d
delta_d = delta_e * (-1.0 / (d**2))     # (5)

# Backprop d = y + c
delta_y = delta_d * 1                   # (4)
delta_c = delta_d * 1                   # (4)

# Backprop c = x**2
delta_x = delta_c * (2 * x)             # (3)

# Backprop b = x + a
delta_x += delta_b * 1                  # (2)
delta_a = delta_b * 1                   # (2)

# Backprop a = 1.0 / (1 + np.exp(-y))
delta_y += delta_a * ((1 - a) * a)      # (1)
```

3.4. Why does this work?

Speaking generally, for any variable u in a graph, we are interested in $\delta_u = \frac{\partial f}{\partial u}$, since this tells us what the effect is on the final output of the computation f when we perturb u. If we proceed from the output and move towards the input, calculating derivatives as we go, the backpropagation algorithm allows us to calculate $\delta_u = \frac{\partial f}{\partial u}$ (for all variables). If u serves as input only for a single computational operation, say with output v, then we just use the simple form of the chain rule: $\delta_u = \frac{\partial f}{\partial u} = \frac{\partial f}{\partial v}\frac{\partial v}{\partial u}$. If u forks out, serving as input for two operations (say with outputs h and g), then we use the version of the chain rule given in (1), i.e.:

$$\delta_u = \frac{\partial f}{\partial u} = \frac{\partial f}{\partial h}\frac{\partial h}{\partial u} + \frac{\partial f}{\partial g}\frac{\partial g}{\partial u}$$

In our example above, both x and y forks in this way, which is why we add to their accumulators when we backprop b and a in the code for the backward pass (the lines marked with `(2)` and `(1)` in the code above). If a variable forks out to more than two operations, the backpropagation algorithm is really just using the general chain rule given in (2). The result of the backpropagation algorithm is that each δ_u contains $\frac{\partial f}{\partial u}$ (the derivative of the final output f with respect to variable u), and we have these derivatives for all the variables in the computational graph.

One other useful property is that if you design a new computational block, it can be inserted easily into the backpropagation algorithm as long as you know two things:

(i) how to calculate the output of the operation using its input variables (you need this for the forward pass, but this is trivial since it is just the definition of the computation); and (ii) how to calculate the derivatives of the output of the block with respect to all the inputs (for the backward pass). Before packages like Theano and TensorFlow came along with automatic differentiation, some neural network packages like MatConvNet was structured in this way [6].

4. Feedforward neural network (using vectors)

4.1. Vector and matrix calculus

In Section 3 we only used scalar variables. It is relatively easy to generalise the backpropagation algorithm to vectors (or even matrices). But ultimately, vector and matrix operations can still be reduced to simple scalar operations. However, there is a benefit in vectorisation: we can take advantage of efficient matrix algebra packages (like NumPy).

I start by giving the definitions and identities we will use. I use the denominator layout to do vector and matrix calculus [7,8]. The Wikipedia article [7] also gives all the identities used here.

The derivative of a scalar function $f : \mathbb{R}^N \to \mathbb{R}$ with respect to vector $\boldsymbol{x} \in \mathbb{R}^N$ is defined as

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \triangleq \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x_1} \\ \frac{\partial f(\boldsymbol{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\boldsymbol{x})}{\partial x_N} \end{bmatrix} \tag{4}$$

The derivative of a vector function $\boldsymbol{f} : \mathbb{R}^N \to \mathbb{R}^M$, where $\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_1(\boldsymbol{x}) & f_2(\boldsymbol{x}) & \cdots & f_M(\boldsymbol{x}) \end{bmatrix}^\mathrm{T}$, with respect to vector $\boldsymbol{x} \in \mathbb{R}^N$, is

$$\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}} \triangleq \begin{bmatrix} \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial x_1} \\ \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial x_2} \\ \vdots \\ \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial x_N} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_2(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial f_M(\boldsymbol{x})}{\partial x_1} \\ \frac{\partial f_1(\boldsymbol{x})}{\partial x_2} & \frac{\partial f_2(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_M(\boldsymbol{x})}{\partial x_2} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_1(\boldsymbol{x})}{\partial x_N} & \frac{\partial f_2(\boldsymbol{x})}{\partial x_N} & \cdots & \frac{\partial f_M(\boldsymbol{x})}{\partial x_N} \end{bmatrix} \tag{5}$$

The derivative of a scalar function $f : \mathbb{R}^{M \times N} \to \mathbb{R}$ with respect to matrix $\mathbf{X} \in \mathbb{R}^{M \times N}$ is

$$\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}} \triangleq \begin{bmatrix} \frac{\partial f(\mathbf{X})}{\partial X_{1,1}} & \frac{\partial f(\mathbf{X})}{\partial X_{1,2}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial X_{1,N}} \\ \frac{\partial f(\mathbf{X})}{\partial X_{2,1}} & \frac{\partial f(\mathbf{X})}{\partial X_{2,2}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial X_{2,N}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f(\mathbf{X})}{\partial X_{M,1}} & \frac{\partial f(\mathbf{X})}{\partial X_{M,2}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial X_{M,N}} \end{bmatrix} \tag{6}$$

Given these definitions, we can generalise the chain rule. Given $\boldsymbol{u} = \boldsymbol{h}(\boldsymbol{x})$ (i.e. $\boldsymbol{u}$ is a function of $\boldsymbol{x}$), the vector-by-vector chain rule states:

$$\frac{\partial \boldsymbol{g}(\boldsymbol{u})}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{g}(\boldsymbol{u})}{\partial \boldsymbol{u}} \tag{7}$$

where $\boldsymbol{g}$ is a vector function. For the vectorised version of backpropagation, we will use this version of the chain rule.

4.2. Backpropagation for a feedforward neural network

Using the vectorised version of the approach of Section 3, let us derive the backpropagation equations for a traditional feedforward neural network, and see if we obtain similar equations to the more traditional way of explaining neural networks, as e.g. in [9] and [10, §16.5.4].

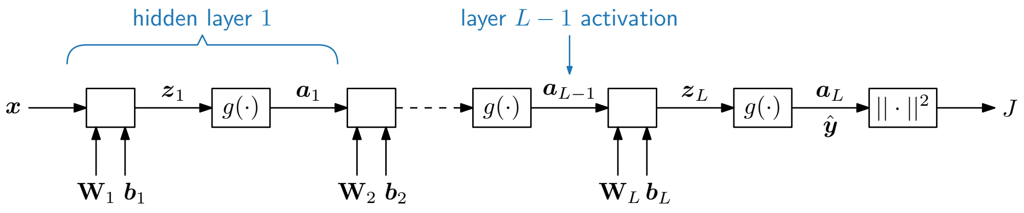

Figure 3: The computational graph for a feedforward neural network. Here all the intermediate variables are vectors.

Figure 3 shows the computational graph for an L-layer feedforward neural network. We consider a single training example $(\boldsymbol{x}, \boldsymbol{y})$ with input $\boldsymbol{x} \in \mathbb{R}^K$ and true output $\boldsymbol{y} \in \mathbb{R}^D$. The prediction from the network is $\hat{\boldsymbol{y}} \in \mathbb{R}^D$. Our aim is to find the derivatives of all the parameters with respect to the cost $J = ||\boldsymbol{y} - \hat{\boldsymbol{y}}||^2$. If we have all these gradients, we will be able to optimise the network parameters using gradient descent.

The feedforward equations are as follows:

$$\boldsymbol{z}_1 = \mathbf{W}_1\,\boldsymbol{x} + \boldsymbol{b}_1 \tag{8}$$
$$\boldsymbol{a}_1 = g(\boldsymbol{z}_1) \tag{9}$$
$$\cdots$$
$$\boldsymbol{z}_l = \mathbf{W}_l\,\boldsymbol{a}_{l-1} + \boldsymbol{b}_l \tag{10}$$
$$\boldsymbol{a}_l = g(\boldsymbol{z}_l) \tag{11}$$
$$\cdots$$
$$\boldsymbol{z}_L = \mathbf{W}_L\,\boldsymbol{a}_{L-1} + \boldsymbol{b}_L \tag{12}$$
$$\boldsymbol{a}_L = g(\boldsymbol{z}_L) = \hat{\boldsymbol{y}} \tag{13}$$
$$J = ||\boldsymbol{y} - \hat{\boldsymbol{y}}||^2 = ||\boldsymbol{y} - \boldsymbol{a}_L||^2 \tag{14}$$

with $g(\cdot)$ the nonlinearity (e.g. sigmoid, tanh or ReLU) applied element-wise. As illustrated in Figure 3, the term *hidden layer* refers to the entire computation from the input (the output of the previous layer) to the output of that layer; the output $\boldsymbol{a}_l$ itself is normally called the *activation* of the layer.

For the backward pass, we just follow the steps from Section 3. Because there are no forks, each backpropogation step will give us the final accumulator values, so we do not really need to think of these in terms of accumulators (they immediately take on the values of the gradients). We start at the final operation, $||\cdot||^2$, and backprop to its input:

$$\boldsymbol{\delta}_{\boldsymbol{a}_L} = \frac{\partial J}{\partial \boldsymbol{a}_L} = \frac{\partial}{\partial \boldsymbol{a}_L}\,||\boldsymbol{y} - \boldsymbol{a}_L||^2 = \frac{\partial}{\partial \boldsymbol{a}_L}(\boldsymbol{y} - \boldsymbol{a}_L)^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{a}_L) = -2(\boldsymbol{y} - \boldsymbol{a}_L) \tag{15}$$

where we use (7) together with the identity:

$$\frac{\partial \boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}}{\partial \boldsymbol{x}} = 2\boldsymbol{x}$$

No we backprop $\boldsymbol{a}_L = g(\boldsymbol{z}_L)$ to the variable $\boldsymbol{z}_L$:

$$\boldsymbol{\delta}_{\boldsymbol{z}_L} = \frac{\partial J}{\partial \boldsymbol{z}_L} = \frac{\partial \boldsymbol{a}_L}{\partial \boldsymbol{z}_L}\frac{\partial J}{\partial \boldsymbol{a}_L} = \frac{\partial \boldsymbol{a}_L}{\partial \boldsymbol{z}_L}\boldsymbol{\delta}_{\boldsymbol{a}_L} \tag{16}$$

The last term on the right hand side we know, since we already calculated it in (15). The first term is:

$$\frac{\partial \boldsymbol{a}_L}{\partial \boldsymbol{z}_L} = \frac{\partial}{\partial \boldsymbol{z}_L} g(\boldsymbol{z}_L) = \operatorname{diag}\left(g'(\boldsymbol{z}_L)\right) \tag{17}$$

where $g'(\boldsymbol{z}_L)$ is the element-wise derivative of the nonlinearity with respect to the input vector $\boldsymbol{z}_L$. The diagonalisation is necessary from the definition in (6). We can just check that all the dimensions work out for the backpropogation step in (16). We know that $\boldsymbol{\delta}_{\boldsymbol{z}_L}$ is a vector in $\mathbb{R}^D$ (since $\boldsymbol{z}_L$ has the same dimensionality as the predicted output $\hat{\boldsymbol{y}}$). The term in (17) will have dimensionality $\frac{\partial \boldsymbol{a}_L}{\partial \boldsymbol{z}_L} \in \mathbb{R}^{D\times D}$, and since $\boldsymbol{\delta}_{\boldsymbol{a}_L} = \frac{\partial J}{\partial \boldsymbol{a}_L} \in \mathbb{R}^D$, all the dimensions in (16) works out. Taking everything together, we have the following backpropogation step:

$$\boldsymbol{\delta}_{\boldsymbol{z}_L} = \frac{\partial J}{\partial \boldsymbol{z}_L} = \operatorname{diag}(g'(\boldsymbol{z}_L))\,\boldsymbol{\delta}_{\boldsymbol{a}_L} = \boldsymbol{\delta}_{\boldsymbol{a}_L} \odot g'(\boldsymbol{z}_L)$$

where $\odot$ indicates element-wise multiplication.

Next we backprop $\boldsymbol{z}_L = \mathbf{W}_L\,\boldsymbol{a}_{L-1} + \boldsymbol{b}_L$ to the input variables $\mathbf{W}_L \in \mathbb{R}^{D\times D_{L-1}}$, $\boldsymbol{a}_{L-1} \in \mathbb{R}^{D_{L-1}}$ and $\boldsymbol{b}_L \in \mathbb{R}^D$, where D_{L-1} is the dimensionality of the penultimate hidden layer $L-1$. The gradients for the last two variables are relatively easy:

$$\boldsymbol{\delta}_{\boldsymbol{b}_L} = \frac{\partial J}{\partial \boldsymbol{b}_L} = \frac{\partial \boldsymbol{z}_L}{\partial \boldsymbol{b}_L}\frac{\partial J}{\partial \boldsymbol{z}_L} = \mathbf{I}\,\boldsymbol{\delta}_{\boldsymbol{z}_L} = \boldsymbol{\delta}_{\boldsymbol{z}_L}$$

and

$$\boldsymbol{\delta}_{\boldsymbol{a}_{L-1}} = \frac{\partial J}{\partial \boldsymbol{a}_{L-1}} = \frac{\partial \boldsymbol{z}_L}{\partial \boldsymbol{a}_{L-1}}\frac{\partial J}{\partial \boldsymbol{z}_L} = \mathbf{W}_L^{\mathrm{T}}\,\boldsymbol{\delta}_{\boldsymbol{z}_L}$$

which uses the identity [7]:

$$\frac{\partial \mathbf{A}\,\boldsymbol{x}}{\partial \boldsymbol{x}} = \mathbf{A}^{\mathrm{T}}$$

Backpropping to the variable $\mathbf{W}_L$ is a bit more involved. We want to find the term

$$\boldsymbol{\delta}_{\mathbf{W}_L} = \frac{\partial J}{\partial \mathbf{W}_L} \in \mathbb{R}^{D\times D_{L-1}} \tag{18}$$

with the dimensions according to (6). But there is no easy chain rule for the derivative of a scalar (or vector) with respect to a matrix [7]. A number of solutions are possible; I mention four. First, you can try and get an expression for the chain rule that includes matrices, but things gets difficult because the derivative of a vector with respect to a matrix is a tensor; this approach is mentioned in [11]. Another approach is to flatten the matrix $\mathbf{W}_L$ into a single vector, and then just use the vector chain rule in (7). This is the approach followed in [6], and is a pretty good option. The third approach is to wing it and just use the matrix dimensionalities to figure out (more-or-less) what to do. In this case, we know that $\frac{\partial J}{\partial \mathbf{W}_L}$ will consist of a term times $\boldsymbol{\delta}_{\boldsymbol{z}_L} = \frac{\partial J}{\partial \boldsymbol{z}_L}$. The missing term would probably be something looking like $\boldsymbol{a}_{L-1}$, and from the dimensionalities we can figure out the correct orientation (i.e. whether we need to transpose anything). This is the approach used at the end of [1].

I follow the fourth option, which is to just write out the individual matrix elements, and then subsequently vectorise the expressions again. Just for now, I drop the L and $L-1$ subscripts and use subscripts instead to denote element indices. Without the vector and matrix subscripts, we are currently considering $\boldsymbol{z} = \mathbf{W}\,\boldsymbol{a} + \boldsymbol{b}$.

Each element of $\boldsymbol{z}$ is given by:

$$z_i = b_i + \sum_{j=1}^{D_{L-1}} W_{i,j}\, a_j \tag{19}$$

which means we can write

$$\frac{\partial J}{\partial W_{i,j}} = \frac{\partial J}{\partial z_i}\frac{\partial z_i}{\partial W_{i,j}} = [\boldsymbol{\delta}_{\boldsymbol{z}}]_i\, a_j$$

According to the definition in (6), this can be written in vectorised form as

$$\frac{\partial J}{\partial \mathbf{W}} = \boldsymbol{\delta}_{\boldsymbol{z}}\, \boldsymbol{a}^{\mathrm{T}} \tag{20}$$

Adding back in the appropriate vector and matrix subscripts, we have

$$\boldsymbol{\delta}_{\mathbf{W}_L} = \frac{\partial J}{\partial \mathbf{W}_L} = \boldsymbol{\delta}_{\boldsymbol{z}_L}\, \boldsymbol{a}_{L-1}^{\mathrm{T}}$$

and since $\boldsymbol{\delta}_{\boldsymbol{z_L}} \in \mathbb{R}^D$ and $\boldsymbol{a} \in \mathbb{R}^{D-1}$, we get the right dimensionality $\boldsymbol{\delta}_{\mathbf{W}_L} \in \mathbb{R}^{D\times D_{L-1}}$.

We therefore have the following equations for an arbitrary layer l:

$$\boldsymbol{\delta}_{\boldsymbol{z}_l} = \boldsymbol{\delta}_{\boldsymbol{a}_l} \odot g'(\boldsymbol{z}_l) \tag{21}$$
$$\boldsymbol{\delta}_{\boldsymbol{b}_l} = \boldsymbol{\delta}_{\boldsymbol{z}_l} \tag{22}$$
$$\boldsymbol{\delta}_{\mathbf{W}_l} = \boldsymbol{\delta}_{\boldsymbol{z}_l}\, \boldsymbol{a}_{l-1}^{\mathrm{T}} \tag{23}$$
$$\boldsymbol{\delta}_{\boldsymbol{a}_{l-1}} = \mathbf{W}_l^{\mathrm{T}}\, \boldsymbol{\delta}_{\boldsymbol{z}_l} \tag{24}$$

These match up exactly with the equations given in [12]. In more traditional explanations such as [9] and [10, §16.5.4], equations (21) and (24) are typically combined into one:

$$\boldsymbol{\delta}_{\boldsymbol{z}_l} = \left(\mathbf{W}_{l+1}^{\mathrm{T}}\, \boldsymbol{\delta}_{\boldsymbol{z}_{l+1}}\right) \odot g'(\boldsymbol{z}_l)$$

This has the benefit of making the recursive nature of the backpropagation algorithm clear since $\boldsymbol{\delta}_{\boldsymbol{z}_l}$ is calculated using $\boldsymbol{\delta}_{\boldsymbol{z}_{l+1}}$. In these explanations, the $\boldsymbol{\delta}$'s are sometimes referred to as "error terms" or "error signals", since you could see them as indication of how much a particular layer (or unit) is "responsible for any errors in the output" [9].

The issue is that many of these traditional explanations can become quite rigid. It is then difficult to see how flexible backpropagation is in that it can be applied to much more complicated architectures than the simple feedforward neural network of Figure 3. For example, this network does not contain any forks (in contrast to the example of Section 3), and seeing how weight sharing would be implemented is a bit hard.

References

[1] CS231n: Optimization 2. [Online]. Available: http://cs231n.github.io/optimization-2/

[2] J. Stewart, *Calculus*, 5th ed. Thomson Learning, 2003.

[3] R. Lipshitz. Linear maps, the total derivative and the chain rule. [Online]. Available: https://www.math.columbia.edu/~lipshitz/teaching/Linearization.pdf

[4] Chain rule. [Online]. Available: https://en.wikipedia.org/wiki/Chain_rule

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016.

[6] A. Vedaldi and A. Zisserman. (2016) VGG convolutional neural networks practical. [Online]. Available: http://www.robots.ox.ac.uk/~vgg/practicals/cnn/

[7] Matrix calculus. [Online]. Available: http://en.wikipedia.org/wiki/Matrix_calculus

[8] H. Kamper. (2013) Vector and matrix calculus. [Online]. Available: http://www.kamperh.com/notes/kamper_matrixcalculus13.pdf

[9] UFLDL: Backpropagation algorithm. [Online]. Available: http://deeplearning.stanford.edu/wiki/index.php/Backpropagation_Algorithm

[10] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA: MIT Press, 2012.

[11] M. Deisenroth. (2017) Deep Learning Indaba: Mathematics for deep learning. [Online]. Available: http://www.deeplearningindaba.com/uploads/1/0/2/6/102657286/2017-09-10-deep-learning-indaba.pdf

[12] I. Murray. (2016) MLPR: Backpropagation of derivatives. [Online]. Available: http://www.inf.ed.ac.uk/teaching/courses/mlpr/2016/notes/w5a_backprop.html