EMAT31530, Part 5: Backprop

Laurence Aitchison

The key thing about PyTorch that *seems* magical is `backward`, which automatically computes gradients of parameters (or anything else) wrt a loss function. Ultimately, `backward` applies the backprop algorithm, and the backprop algorithm is just an automatic way of applying the chain rule. In this part, we'll:

1. Start with a reminder of the multivariable chain rule and total / partial derivatives.
2. Remind ourselves of the neural network setting. We're going to slightly alter exactly how we write down the neural network, so it'll be easier to talk about backprop later on.
3. Discuss the high-level strategy for backprop.
4. Discuss how to backprop is efficiently implemented in PyTorch.

1 The chain rule, partial and total derivatives

Unfortunately, the field of deep learning has gotten a bit confused about partial, ∂, and total, d, derivatives. I'm going to try to adopt clear and consistent notation. But I won't always succeede. And the field itself is a bit of a mess, so if you look at other resources, you will see other choices of notation.

It is actually simplest to define the partial derivative. Consider a function of two arguments, $f(x, y)$. There are two partial derivatives for f, one for each argument,

$$\frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x} \tag{1a}$$

$$\frac{\partial f}{\partial y} = \lim_{\Delta y \to 0} \frac{f(x, y + \Delta y) - f(x, y)}{\Delta y}. \tag{1b}$$

Ultimately, the partial derivative takes a *function* (here, f) and tells us what happens to the output if you perturb one (and only one) of that function's arguments.

Then we come on to the total derivative. While you may be more familiar with the total derivative, the total deriative is actually harder to define. That's because the total derivative describes how changes propagate through a "compute graph" (a term from deep learning that turns out to be very useful in our context), not just a single function. For instance, consider the setting where we have $f(x, y(x))$, i.e. y is itself a function of x. The resulting compute graph looks like:

$$\begin{tikzcd} x \arrow[rr] \arrow[dr] & & f \\ & y \arrow[ur] & \end{tikzcd}$$

The partial and total derivatives for y are the same, as y only appears as a direct argument to f,

$$\frac{df}{dy} = \frac{\partial f}{\partial y}. \tag{2}$$

Importantly though, the partial and total derivatives for x are different. In the partial derivative, we just treat x as the first input argument to f, so we just compute the change in f implied by a change in the first argument (Eq. 1a). In contrast, for the total derivative, we treat x as a component of the compute graph. In that case, we can see how x is not just the first argument to f; the second argument $y(x)$ also depends on x. The total derivative gives the full change in f induced by a change in x under the full compute graph. We can compute this change using the multivariable chain rule,

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{dy}{dx}. \tag{3}$$

And this is the form for the multivariable chain rule you'll see in most textbook/references.

To see other differences between the partial and total derivatives, consider a slightly different compute graph, with $f(x(t), y(t))$,

$$\begin{tikzcd} & x \arrow[dr] & \\ t \arrow[ur] \arrow[dr] & & f \\ & y \arrow[ur] & \end{tikzcd}$$

How does the partial derivative work in this context? The informal definition of the partial derivative usually states "the partial deriative is the derivative, holding everything else fixed". Now, what happens if we try to calculate the partial derivative of f wrt t? This informal definition of the partial derivative would seem to suggest the answer is 0, as f can't change if we hold x and y fixed,

$$\frac{\partial f}{\partial t} \overset{?}{=} 0. \tag{4}$$

However, it isn't clear that this makes sense if we look back at our more formal definition of the partial derivative, as "taking a *function* and telling us what happens to the output if you perturb one (and only one) of that function's arguments". Under this more formal definition, it simply does not make sense to ask for the partial derivative of f wrt t, as t is not an argument to f. Instead, f only has two arguments: x and y. That would suggest,

$$\frac{\partial f}{\partial t} \overset{?}{=} \text{undefined}. \tag{5}$$

Of course, this is ultimately a matter of definitions, and we are free to choose the definition that is most useful in our specific context. In deep learning, it's far better to use the more formal definition, as it'll help us to make sense of some of the notational confusion.

In contrast, the total derivative does make sense in this context, as it describes how changes (e.g. in t) propagate through the full compute graph. To compute the total derivative, we use the multivariate chain rule,

$$\frac{df}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}. \tag{6}$$

This is again the form for the multivariate chain rule you'll find in most textbooks / references. Note that the chain rule uses total derivatives for $\frac{dx}{dt}$ and $\frac{dy}{dt}$. In this simple setting where x and y are univariate functions of t, we could equally use partial derivatives. But this form for the chain rule continues to work in a more general computational graph where x and y aren't directly functions of t, but there's other variables in between.

$$\begin{tikzcd}
& a \arrow[r] \arrow[rdd] & x \arrow[rd] & \\
t \arrow[ru] \arrow[rd] & & & f \\
& b \arrow[r] \arrow[ruu] & y \arrow[ru] &
\end{tikzcd}$$

For instance, in this graph, we introduce new functions, a, b between t and x, y.

[Non-examinable]. Hopefully, the previous discussion made some sense, and we'll be using the previously described conventions in the rest of the notes + exam. However, deep learning folks have made a complete mess of things. They often write the chain rule entirely in partial deriative form,

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}. \tag{7}$$

If we directly apply the definitions above, this does not make sense. Specifically, $\frac{\partial f}{\partial t}$ is undefined, as f is a function of x and y; it is not a function of t. Thus, you could argue that specifying the chain rule in the above form is just wrong (and I'd have alot of sympathy for that argument). There is an alternative though. We could choose to write f_t as a function with one argument, t, which gives the same values as $f(x(t), y(t))$,

$$f_t(t) = f(x(t), y(t)). \tag{8}$$

Then the argument goes that as $\frac{\partial f}{\partial t}$ is undefined, I must really have meant $\frac{\partial f_t}{\partial t}$. Indeed it is correct to write,

$$\frac{\partial f_t}{\partial t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}. \tag{9}$$

The problem is that using partial derivatives in this way requires us to constantly implicitly redefine the arguments to our functions. It's therefore a really bad idea if we're trying to be precise. But the fact that deep learning people have got themselves in such a mess probably indicates that it doesn't actually matter too much.

2 Setting up the problem of backprop in a neural network

To understand what backprop is doing, we'll consider a neural network written in terms of a linear and sum-squared error functions,

$$\operatorname{linear}(\mathbf{h}, \mathbf{W}, \mathbf{b}) = \mathbf{h}\mathbf{W} + \mathbf{b}, \tag{10a}$$

$$\operatorname{sqerr}(\mathbf{f}, \mathbf{y}) = \tfrac{1}{2}\sum_i (f_i - y_i)^2. \tag{10b}$$

In this entire document, we're going to consider just a single input datapoint. So e.g. the vector $\mathbf{h}$ represents multiple intermediate layer features for one datapoint. This extends to $\operatorname{sqerr}(\mathbf{f}, \mathbf{y})$, which represents a squared-error loss for regression with multiple output features, f_i, and targets, y_i, for each datapoint. In $\operatorname{sqerr}(\mathbf{f}, \mathbf{y})$, we use $\mathbf{f}$ to represent the output from our network, while $\mathbf{y}$ is the targets from the training data. We'll consider a 3-layer network (note the three linear layers),

$$\mathbf{a}_1 = \operatorname{linear}(\mathbf{x}, \mathbf{W}_1, \mathbf{b}_1) \tag{11a}$$

$$\mathbf{h}_1 = \operatorname{relu}(\mathbf{a}_1) \tag{11b}$$

$$\mathbf{a}_2 = \operatorname{linear}(\mathbf{h}_1, \mathbf{W}_2, \mathbf{b}_2) \tag{11c}$$

$$\mathbf{h}_2 = \operatorname{relu}(\mathbf{a}_2) \tag{11d}$$

$$\mathbf{f} = \operatorname{linear}(\mathbf{h}_2, \mathbf{W}_3, \mathbf{b}_3) \tag{11e}$$

$$\mathcal{L} = \operatorname{sqerr}(\mathbf{f}, \mathbf{y}). \tag{11f}$$

Here, the subscripts all indicate layers, not features/datapoints. The implied compute graph is,

$$\begin{tikzcd}
\mathbf{x} \arrow[rr] & & \mathbf{a}_1 \arrow[rr] & & \mathbf{h}_1 \arrow[rr] & & \mathbf{a}_2 \arrow[rr] & & \mathbf{h}_2 \arrow[rr] & & \mathbf{f} \arrow[rr] & & \mathcal{L} \\
& \mathbf{W}_1 \arrow[ur] & & \mathbf{b}_1 \arrow[ul] & & \mathbf{W}_2 \arrow[ur] & & \mathbf{b}_2 \arrow[ul] & & \mathbf{W}_3 \arrow[ur] & & \mathbf{b}_3 \arrow[ul] & \mathbf{y} \arrow[u]
\end{tikzcd}$$

Now, our goal is to compute the gradient of the loss wrt all the parameters,

$$\frac{d\mathcal{L}}{d\mathbf{W}_1} \qquad \frac{d\mathcal{L}}{d\mathbf{W}_2} \qquad \frac{d\mathcal{L}}{d\mathbf{W}_3} \qquad \frac{d\mathcal{L}}{d\mathbf{b}_1} \qquad \frac{d\mathcal{L}}{d\mathbf{b}_2} \qquad \frac{d\mathcal{L}}{d\mathbf{b}_3}. \tag{12}$$

3 Backprop

The problem with the multivariate chain rule is that it applies only to one function, whereas we have a huge graph of functions. We therefore need to find a way of repeatedly applying the chain rule that gives us what we want. In particular, we're going to work backwards through the compute graph, computing the required total derivatives.

$$\begin{tikzcd}
\frac{d\mathcal{L}}{d\mathbf{x}} & \frac{d\mathcal{L}}{d\mathbf{a}_1} \arrow[l] \arrow[d] \arrow[dr] & & \frac{d\mathcal{L}}{d\mathbf{h}_1} \arrow[ll] & \frac{d\mathcal{L}}{d\mathbf{a}_2} \arrow[l] \arrow[d] \arrow[dr] & & \frac{d\mathcal{L}}{d\mathbf{h}_2} \arrow[ll] & \frac{d\mathcal{L}}{d\mathbf{f}} \arrow[l] \arrow[d] \arrow[dr] & & \mathcal{L} \arrow[ll] \arrow[d] \\
& \frac{d\mathcal{L}}{d\mathbf{W}_1} & \frac{d\mathcal{L}}{d\mathbf{b}_1} & & \frac{d\mathcal{L}}{d\mathbf{W}_2} & \frac{d\mathcal{L}}{d\mathbf{b}_2} & & \frac{d\mathcal{L}}{d\mathbf{W}_3} & \frac{d\mathcal{L}}{d\mathbf{b}_3} & \frac{d\mathcal{L}}{d\mathbf{y}}
\end{tikzcd}$$

We have greyed-out $\frac{d\mathcal{L}}{d\mathbf{x}}$ and $\frac{d\mathcal{L}}{d\mathbf{y}}$ because these are gradients wrt data ($\mathbf{x}$ and $\mathbf{y}$), not parameters. While research papers do sometimes use these papers, that's rare. In standard settings, data are fixed, not tunable, so we're going to throw away these gradients.

That's great, but how do we actually compute these gradients? By applying the chain rule *alot*. We start with $\frac{d\mathcal{L}}{df_m}$, which can be obtained directly by differentiating Eq. (11f) using the definition of the squared error function in Eq. (10b). Then we use the chain rule to compute total derivatives for variables and parameters that appear progressively earlier in the compute graph, (I'm not including all the equations here, because it gets quite repetitive),

$$\frac{d\mathcal{L}}{df_m} = \frac{\partial \mathcal{L}}{\partial f_m} \tag{13a}$$

$$\frac{d\mathcal{L}}{dh_{2;l}} = \sum_m \frac{\partial f_m}{\partial h_{2;l}} \frac{d\mathcal{L}}{df_m} \tag{13b}$$

$$\frac{d\mathcal{L}}{da_{2;k}} = \sum_l \frac{\partial h_{2;l}}{\partial a_{2;k}} \frac{d\mathcal{L}}{dh_{2;l}} \tag{13c}$$

$$\frac{d\mathcal{L}}{db_{2;i}} = \sum_k \frac{\partial a_{2;k}}{\partial b_{2;i}} \frac{d\mathcal{L}}{da_{2;k}} \tag{13d}$$

$$\frac{d\mathcal{L}}{dW_{2;ij}} = \sum_k \frac{\partial a_{2;k}}{\partial W_{2;ij}} \frac{d\mathcal{L}}{da_{2;k}} \tag{13e}$$

$$\cdots \quad \cdots \tag{13f}$$

4 Implementing backprop

At a mathematical level that's all there is to backprop. However, it is worth thinking a bit harder to understand how backprop is usually implemented efficiently. This is going to help you:

- Reason about memory usage, and get bigger models to run.
- Extend deep learning frameworks to do new, exciting and weird things.

4.1 Implementing backprop using explicit Jacobians

Each individual application of the chain rule involves the partial-derivative of the output wrt the input of some operation (e.g. $\frac{\partial f_m}{\partial h_{2;l}}$ in Eq. 13b). When there are multiple inputs and outputs, these derivatives form big matrices/tensors which are sometimes called Jacobians. The most obvious approach to implementing backprop is to directly use Eq. (13). And to do that, we could implement func.jacobian, which computes the Jacobians, alongside the usual "forward" computation,

$$\text{outputs} = \text{func}\left(\text{inputs}\right) \tag{14a}$$

$$\frac{\partial\,\text{outputs}}{\partial\,\text{inputs}} = \text{func.jacobian}\left(\text{inputs}\right). \tag{14b}$$

Concretely, for the relu function,

$$\mathbf{h} = \text{relu}\left(\mathbf{a}\right) \tag{15a}$$

$$\frac{\partial \mathbf{h}}{\partial \mathbf{a}} = \text{relu.jacobian}\left(\mathbf{a}\right) \tag{15b}$$

Note that $\frac{\partial \mathbf{h}}{\partial \mathbf{a}}$ is just $\frac{\partial h_i}{\partial a_j}$ interpreted as a matrix (like $\mathbf{A}$ vs A_{ij}). We aren't usually going to e.g. do matrix multiplication with Jacobians written as $\frac{\partial \mathbf{h}}{\partial \mathbf{a}}$, as things quickly get confusing. But we can work with total derivatives wrt the loss, e.g. $\frac{d\mathcal{L}}{d\mathbf{h}}$. We'll follow PyTorch and use the convention is always that $\frac{d\mathcal{L}}{d\mathbf{h}}$ has the same shape as $\mathbf{h}$. As we treat $\mathbf{h}$ as a row-vector, that means $\frac{d\mathcal{L}}{d\mathbf{h}}$ is also a row-vector.

We can do exactly the same thing for the linear function,

$$\mathbf{a} = \text{linear}\left(\mathbf{h}, \mathbf{W}, \mathbf{b}\right) \tag{16a}$$

$$\left(\frac{\partial \mathbf{a}}{\partial \mathbf{h}}, \frac{\partial \mathbf{a}}{\partial \mathbf{W}}, \frac{\partial \mathbf{a}}{\partial \mathbf{b}}\right) = \text{linear.jacobian}\left(\mathbf{h}, \mathbf{W}, \mathbf{b}\right) \tag{16b}$$

Then, we could compute the required gradients of the objective wrt the parameters by computing all these Jacobians, then doing the required sums (as in Eq. 13).

However, this approach turns out to be highly inefficient. For instance, consider the relu. Remember the univariate relu is defined as,

$$h = \text{relu}(a) = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

So the univariate partial derivative is,

$$\frac{\partial h}{\partial a} = \Theta(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

where $\Theta(a)$ is known as the Heaviside step function. When we use the relu in deep learning, we apply it to the whole vector, and it applies pointwise to each element.

$$h_j = \text{relu}_j(\mathbf{a}) = \text{relu}(a_j) = \begin{cases} a_j & \text{if } a_j > 0 \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

Thus, the partial derivatives are,

$$\frac{\partial h_j}{\partial a_\alpha} = \delta_{j\alpha}\Theta(a_j). \tag{20}$$

Critically, the partial derivatives are zero when $j \neq \alpha$, as the jth output, h_j only depends on the αth input, a_α. Now, we can apply the chain rule as in Eq. (13c),

$$\frac{d\mathcal{L}}{da_\alpha} = \sum_j \frac{\partial h_j}{\partial a_\alpha}\frac{d\mathcal{L}}{dh_j}. \tag{21}$$

Substituting the Jacobian,

$$\frac{d\mathcal{L}}{da_\alpha} = \sum_j \delta_{j\alpha}\Theta(a_\alpha)\frac{d\mathcal{L}}{dh_j}. \tag{22}$$

The Kronecker-delta picks out one element of the sum, for which $j = \alpha$,

$$\frac{d\mathcal{L}}{da_\alpha} = \Theta(a_\alpha)\frac{d\mathcal{L}}{dh_\alpha}. \tag{23}$$

Intuitively, this passes the gradient through unchanged if we're on the linear part of the relu (i.e. if $a_\alpha > 0$). In contrast, this zeros-out the gradient if we're on the flat part of the relu (i.e. if $a_\alpha < 0$).

Critically, it is far more efficient to implement this final expression (Eq. 23) than to explicitly compute the Jacobian then use Eq. (21). That's because the final expression (Eq. 21) only involves $\mathcal{O}(D)$ operations, where D is the number of features in the hidden vectors. In contrast, the Jacobian (Eq. 20) has D^2 elements, so explicitly computing the chain rule (Eq. 21) requires $\mathcal{O}(D^2)$ operations. It turns out that in almost every operations there are huge efficiencies like this that can be exploited, and PyTorch is implemented to make sure that these efficiencies are exploited!

4.2 Implementing backprop without explicit Jacobians

In the previous subsection, we saw that there were two ways to compute the chain rule: Eq. (21) and Eq. (23). The first (Eq. 21) explicitly summed over the Jacobian. The key insight in the second equation (Eq. 23) was that we didn't need the Jacobian explicitly. Instead, we just need an expression to compute $\frac{d\mathcal{L}}{d\mathbf{a}}$ from $\frac{d\mathcal{L}}{d\mathbf{h}}$. And such an expression can often be much simpler and more efficient than working with Jacobians explicitly. This is the key idea behind how backprop is implemented in practice. Specifically, PyTorch implements "forward" for each operation (e.g. linear or relu),

$$\text{outputs} = \text{func}\left(\text{inputs}\right). \tag{24}$$

PyTorch also gives us a way to apply the chain rule. Remember that the chain rule is usually defined in terms of the Jacobian,

$$\frac{d\mathcal{L}}{d\,\text{inputs}} = \sum_j \frac{\partial\,\text{outputs}_j}{\partial\,\text{inputs}} \frac{d\mathcal{L}}{d\,\text{outputs}_j}. \tag{25}$$

But PyTorch doesn't implement this function directly. Instead, it defines a backward function for each operation,

$$\frac{d\mathcal{L}}{d\,\text{inputs}} = \text{func.backward}\left(\frac{d\mathcal{L}}{d\,\text{outputs}}; \text{inputs}\right) \tag{26}$$

This function takes the original inputs, along with the gradient of the objective wrt the output, $\frac{d\mathcal{L}}{d\,\text{outputs}}$, and returns the gradient of the objective wrt the inputs, $\frac{d\mathcal{L}}{d\,\text{inputs}}$. The result is equal to multiplying by the Jacobian (Eq. (25)), but can be much more efficient, as we don't have to explicitly form the Jacobian.

Concretely, we for the ReLU:

$$\mathbf{h} = \text{relu}\left(\mathbf{a}\right) \tag{27a}$$

$$\frac{d\mathcal{L}}{d\mathbf{a}} = \text{relu.backward}\left(\frac{d\mathcal{L}}{d\mathbf{h}}; \mathbf{a}\right) \tag{27b}$$

And for the linear function,

$$\mathbf{a} = \text{linear}\left(\mathbf{h}, \mathbf{W}, \mathbf{b}\right) \tag{28a}$$

$$\left(\frac{d\mathcal{L}}{d\mathbf{h}}, \frac{d\mathcal{L}}{d\mathbf{W}}, \frac{d\mathcal{L}}{d\mathbf{b}}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{d\mathbf{a}}; \mathbf{h}, \mathbf{W}, \mathbf{b}\right) \tag{28b}$$

For the actual value of these derivatives, see the exercises. Finally, for the sum-square error.

$$\mathcal{L} = \text{sqerr}\left(\mathbf{f}, \mathbf{y}\right) \tag{29a}$$

$$\left(\frac{d\mathcal{L}}{d\mathbf{f}}, \frac{d\mathcal{L}}{d\mathbf{y}}\right) = \text{sqerr.backward}\left(\frac{d\mathcal{L}}{d\mathcal{L}} = 1; \mathbf{f}, \mathbf{y}\right) \tag{29b}$$

Note that something a little weird happens for the sum-squared error. Specifically, its output is the loss! Therefore, $\frac{d\mathcal{L}}{d\,\text{outputs}}$ becomes $\frac{d\mathcal{L}}{d\mathcal{L}}$. And of course, $\frac{d\mathcal{L}}{d\mathcal{L}} = 1$.

Now, func.backward gives us the gradients of the objective wrt the inputs of the function. The inputs to one operation are either:

- Parameters, in which case, $\frac{d\mathcal{L}}{d\,\text{inputs}}$ are the gradients we want (e.g.
- The output from operation. In that case, $\frac{d\mathcal{L}}{d\,\text{inputs}}$ from this operation are $\frac{d\mathcal{L}}{d\,\text{outputs}}$ from the previous operation, so we can pass the gradients backwards.
- Data (i.e. **x** or **y**), in which case we don't use the gradients.

Overall, the full backward pass looks like,

$$\left(\frac{d\mathcal{L}}{d\mathbf{f}}, \frac{d\mathcal{L}}{d\mathrm{y}}\right) = \text{sqerr.backward}\left(\frac{d\mathcal{L}}{d\mathcal{L}} = 1; \mathbf{f}, \mathbf{y}\right) \tag{30a}$$

$$\left(\frac{d\mathcal{L}}{d\mathbf{h}_2}, \frac{d\mathcal{L}}{d\mathbf{W}_3}, \frac{d\mathcal{L}}{d\mathbf{b}_3}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{d\mathbf{f}}; \mathbf{h}_2, \mathbf{W}_3, \mathbf{b}_3\right) \tag{30b}$$

$$\frac{d\mathcal{L}}{d\mathbf{a}_2} = \text{relu.backward}\left(\frac{d\mathcal{L}}{d\mathbf{h}_2}; \mathbf{a}_2\right) \tag{30c}$$

$$\left(\frac{d\mathcal{L}}{d\mathbf{h}_1}, \frac{d\mathcal{L}}{d\mathbf{W}_2}, \frac{d\mathcal{L}}{d\mathbf{b}_2}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{d\mathbf{a}_2}; \mathbf{h}_1, \mathbf{W}_2, \mathbf{b}_2\right) \tag{30d}$$

$$\frac{d\mathcal{L}}{d\mathbf{a}_1} = \text{relu.backward}\left(\frac{d\mathcal{L}}{d\mathbf{h}_2}; \mathbf{a}_2\right) \tag{30e}$$

$$\left(\frac{d\mathcal{L}}{d\mathrm{x}}, \frac{d\mathcal{L}}{d\mathbf{W}_2}, \frac{d\mathcal{L}}{d\mathbf{b}_2}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{d\mathbf{a}_1}; \mathbf{x}, \mathbf{W}_1, \mathbf{b}_1\right) \tag{30f}$$

Note that the gradients required as input at each step are computed in the previous step!

Overall, the backprop algorithm is:

1. Run the forward pass (Eq. 11), recording the order of and outputs from all operations.
2. Run the backward pass (Eq. 30) to compute all the gradients, by working backwards through the operations recorded in the forward pass.

4.3 Memory consumption of backprop

Note that to perform the backward pass, we need to retain all the values of the variables in the forward pass (e.g. Eq. 26 and 30). This requirement to keep copies of all the variables from the forward pass implies a big memory consumption. This memory consumption is largely unavoidable, and causes lots of issues (in fact, huge amounts of deep learning research is about reducing/mitigating memory issues). In contrast, if we *just* run the forward pass at test time (also known as "inference"), then the memory consumption is much less. That's because you don't need to retain all variables in the whole forward pass: you can almost always throw away variables that are earlier in the compute graph that you aren't going to need any more. For instance, in Eq. 11, once you have computed $\mathbf{a}_2$, you don't need $\mathbf{h}_1$ anymore, so you can throw away $\mathbf{h}_1$, and reuse its memory.

5 Exercises

All these exercises relate to the linear function,

$$\mathbf{a} = \text{linear}(\mathbf{h}, \mathbf{W}, \mathbf{b}) = \mathbf{h}\mathbf{W} + \mathbf{b}. \tag{31}$$

The resulting $\mathbf{a}$ can be written in index-notation as,

$$a_j = \sum_i h_i W_{ij} + b_j. \tag{32}$$

Exercise 1. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial h_\alpha} \tag{33}$$

Exercise 2. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} \tag{34}$$

Exercise 3. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial b_\alpha} \tag{35}$$

Exercise 4. *Do the backward computation, by writing $\frac{d\mathcal{L}}{dh_\alpha}$ in terms of $\frac{d\mathcal{L}}{da_j}$. Write the implied expression for $\frac{d\mathcal{L}}{d\mathbf{h}}$.*

Exercise 5. *Do the backward computation, by writing $\frac{d\mathcal{L}}{dW_{\alpha\beta}}$ in terms of $\frac{d\mathcal{L}}{da_k}$. Write the implied expression for $\frac{d\mathcal{L}}{d\mathbf{W}}$.*

Exercise 6. *Do the backward computation, by writing $\frac{d\mathcal{L}}{db_\alpha}$ in terms of $\frac{d\mathcal{L}}{da_j}$. Reinterpret this expression in terms of e.g. matrix-vector products rather than indices. Write the implied expression for $\frac{d\mathcal{L}}{d\mathbf{b}}$.*

6 Answers

Answer 1. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial h_\alpha} = \frac{\partial}{\partial h_\alpha}\left(\sum_i h_i W_{ij} + b_j\right) \tag{36}$$

As $0 = \frac{\partial W_{ij}}{\partial h_\alpha}$ *and* $0 = \frac{\partial b_j}{\partial h_\alpha}$*,*

$$\frac{\partial a_j}{\partial h_\alpha} = \sum_l \frac{\partial h_i}{\partial h_\alpha} W_{ij} \tag{37}$$

As $\frac{\partial h_i}{\partial h_\alpha}$ *is* 1 *when* $i = \alpha$ *and is zero otherwise,*

$$\frac{\partial a_j}{\partial h_\alpha} = \sum_i \delta_{i\alpha} W_{ij} \tag{38}$$

The Kronecker delta picks out the $i = \alpha$*th element of the sum,*

$$\frac{\partial a_j}{\partial h_\alpha} = W_{\alpha j}. \tag{39}$$

Answer 2. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} = \frac{\partial}{\partial W_{\alpha\beta}}\left(\sum_i h_i W_{ij} + b_j\right) \tag{40}$$

As $0 = \frac{\partial h_j}{\partial W_{\alpha\beta}}$ *and* $0 = \frac{\partial b_i}{\partial W_{\alpha\beta}}$*,*

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} = \sum_i h_i \frac{\partial W_{ij}}{\partial W_{\alpha\beta}} \tag{41}$$

As $\frac{\partial W_{ij}}{\partial W_{\alpha\beta}}$ *is* 1 *when* $i = \alpha$ *and* $j = \beta$ *and is zero otherwise,*

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} = \sum_i h_i \delta_{i\alpha} \delta_{j\beta} \tag{42}$$

The Kronecker delta picks out the $i = \alpha$*th element of the sum,*

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} = h_\alpha \delta_{j\beta} \tag{43}$$

Answer 3. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial b_\alpha} = \frac{\partial}{\partial h_\alpha}\left(\sum_i h_i W_{ij} + b_j\right) \tag{44}$$

As $0 = \frac{\partial W_{ij}}{\partial b_\alpha}$ *and* $0 = \frac{\partial h_i}{\partial b_\alpha}$,

$$\frac{\partial a_j}{\partial b_\alpha} = \frac{\partial b_j}{\partial b_\alpha} \tag{45}$$

As $\frac{\partial b_j}{\partial b_\alpha}$ *is* 1 *when* $j = \alpha$ *and is zero otherwise,*

$$\frac{\partial a_j}{\partial b_\alpha} = \delta_{j\alpha}. \tag{46}$$

Answer 4. *Apply the chain rule,*

$$\frac{d\mathcal{L}}{dh_\alpha} = \sum_j \frac{\partial a_j}{\partial h_\alpha} \frac{d\mathcal{L}}{da_j} \tag{47}$$

Substitute the Jacobian from the previous question,

$$\frac{d\mathcal{L}}{dh_\alpha} = \sum_j W_{\alpha j} \frac{d\mathcal{L}}{da_j} \tag{48}$$

If you wanted to write this as a matrix-vector multiplication, where $\frac{d\mathcal{L}}{d\mathbf{h}}$ *has the same shape as* $\mathbf{h}$, *you'd use,*

$$\frac{d\mathcal{L}}{d\mathbf{h}} = \frac{d\mathcal{L}}{d\mathbf{a}} \mathbf{W}^T \tag{49}$$

As both $\mathbf{h}$ *and* $\mathbf{a}$, *and hence* $\frac{d\mathcal{L}}{d\mathbf{h}}$ *and* $\frac{d\mathcal{L}}{d\mathbf{a}}$ *are row-vectors.*

Answer 5. *Apply the chain rule,*

$$\frac{d\mathcal{L}}{dW_{\alpha\beta}} = \sum_j \frac{\partial a_j}{\partial W_{\alpha\beta}} \frac{d\mathcal{L}}{da_j} \tag{50}$$

Substitute the Jacobian from the previous question,

$$\frac{d\mathcal{L}}{dW_{\alpha\beta}} = \sum_j h_\alpha \delta_{j\beta} \frac{d\mathcal{L}}{da_j} \tag{51}$$

The Kronecker delta picks out the $j = \beta$*th element of the sum,*

$$\frac{d\mathcal{L}}{dW_{\alpha\beta}} = h_\alpha \frac{d\mathcal{L}}{da_\beta} \tag{52}$$

If you wanted to write this as a vector outer product,

$$\frac{d\mathcal{L}}{d\mathbf{W}} = \mathbf{h}^T \frac{d\mathcal{L}}{d\mathbf{a}} \tag{53}$$

As both $\mathbf{h}$ *and* $\frac{d\mathcal{L}}{d\mathbf{a}}$ *are row-vectors.*

Answer 6. *Apply the chain rule,*

$$\frac{d\mathcal{L}}{db_\alpha} = \sum_j \frac{\partial a_j}{\partial b_\alpha} \frac{d\mathcal{L}}{da_j} \tag{54}$$

Substitute the Jacobian from the previous question,

$$\frac{d\mathcal{L}}{db_\alpha} = \sum_j \delta_{j\alpha} \frac{d\mathcal{L}}{da_j}. \tag{55}$$

The Kronecker delta picks out the $j = \alpha$*th element of the sum,*

$$\frac{d\mathcal{L}}{db_\alpha} = \frac{d\mathcal{L}}{da_\alpha} \tag{56}$$

If you wanted to write this with vectors,

$$\frac{d\mathcal{L}}{d\mathbf{b}} = \frac{d\mathcal{L}}{d\mathbf{a}}. \tag{57}$$