SEMT20003, 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,

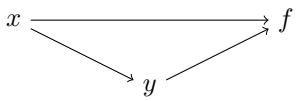
$$\frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x} \tag{1a}$$

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

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:



The total derivative is what happens in the compute graph, if I add a small perturbation to one of the variables. For instance, if I add a small perturbation, dy to y, then the resulting perturbation to f is,

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

Alternatively, if I add a small perturbation, dx to x, then the resulting perturbation to y is,

$$dy = \frac{\partial y}{\partial x} dx \tag{3}$$

Of course, that same perturbation to x also changes f. The perturbation to f from dx now arises from two sources: directly from x and indirectly through y,

$$df = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} dy \tag{4}$$

substituting the perturbation for y (Eq. 3),

$$df = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} \frac{\partial y}{\partial x} dx. \tag{5}$$

We get the usual total derivatives by "dividing through" by dy in Eq. (2)

$$\frac{df}{dy} = \frac{\partial f}{\partial y}. \tag{6}$$

i.e. the partial and total derivatives for y are the same, because y can only affect f directly, and not through any other "path". And we "divide through" by dx in Eq. (5),

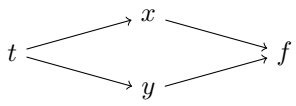
$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial x}. \tag{7}$$

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

Thus, the partial and total derivatives for x are different, because they are answering fundamentally different questions. The partial derivative, $\frac{\partial f}{\partial x}$ is *really*

asking how f changes as a function of its first argument, so x is acting as just a "label" for the first argument (Eq. 1a). In contrast, for the total derivative, we treat x as a component of the compute graph, and ask how a perturbation to x affects f as that perturbation flows through the full graph.

Now, its worth relating my notion of the partial derivative back to something you've heard before, "the partial deriative is the derivative, holding everything else fixed". To see how this alternatie definition differs from my approach, consider a slightly different compute graph, with $f(x(t), y(t))$,



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,

$$\frac{\partial f}{\partial t} \stackrel{?}{=} 0. \tag{8}$$

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,

$$\frac{\partial f}{\partial t} \stackrel{?}{=} \text{undefined}. \tag{9}$$

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,

$$\frac{df}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}. \tag{10}$$

This is again the form for the multivariate chain rule you'll find in most textbooks / references.

[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,

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}. \tag{11}$$

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 $f(x(t), y(t))$,

$$f_t(t) = f(x(t), y(t)). \tag{12}$$

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,

$$\frac{\partial f_t}{\partial t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}. \tag{13}$$

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 this mess, but still seem to do useful things probably indicates that it doesn't actually matter too much.

2 A Scalar Example to Motivate Backprop

Before diving into the full backprop algorithm, let's look at a simplified example that helps motivate why backprop is useful. Consider a very simple neural network where all variables are scalars rather than vectors. This means:

- Instead of weight matrices $\mathbf{W}_1$, we have scalar weights w_1
- Instead of bias vectors $\mathbf{b}_1$, we have scalar biases b_1
- Instead of hidden vectors $\mathbf{h}_1$, we have scalar hidden units h_1
- Instead of activation vectors $\mathbf{a}_1$, we have scalar activations a_1

Note that the subscripts all indicate layers, not features/datapoints. The network equations become:

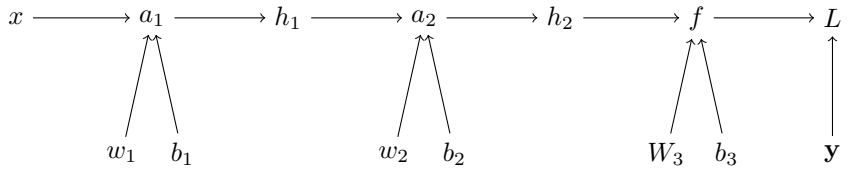
$$a_1 = \text{linear}(x, w_1, b_1) = w_1 x + b_1 \tag{14a}$$
$$h_1 = \text{relu}(a_1) \tag{14b}$$
$$a_2 = \text{linear}(h_1, w_2, b_2) = w_2 h_1 + b_2 \tag{14c}$$
$$h_2 = \text{relu}(a_2) \tag{14d}$$
$$f = \text{linear}(h_2, w_3, b_3) = w_3 h_2 + b_3 \tag{14e}$$
$$\mathcal{L} = \text{sqerr}(f, y) \tag{14f}$$

where,

$$\text{sqerr}(f, y) = \tfrac{1}{2}(f - y)^2 \tag{15}$$

$$\text{linear}(h, w, b) = wh + b. \tag{16}$$

The implied compute graph is,



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

$$\frac{d\mathcal{L}}{dw_1} \qquad \frac{d\mathcal{L}}{dw_2} \qquad \frac{d\mathcal{L}}{dw_3} \qquad \frac{d\mathcal{L}}{db_1} \qquad \frac{d\mathcal{L}}{db_2} \qquad \frac{d\mathcal{L}}{db_3}. \tag{17}$$

Lets do that then! Applying the chain rule, we get,

$$\frac{d\mathcal{L}}{dw_3} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial w_3} \tag{18a}$$

$$\frac{d\mathcal{L}}{db_3} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial b_3} \tag{18b}$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial w_2} \tag{18c}$$

$$\frac{d\mathcal{L}}{db_2} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial b_2} \tag{18d}$$

Note that, all the partial derivatives are pretty easy to compute. For instance,

$$\frac{\partial f}{\partial w_3} = \frac{\partial}{\partial w_3}\,\text{linear}(h_2, w_3, b_3) = \frac{\partial}{\partial w_3}\left[w_3 h_2 + b_3\right] = h_2. \tag{19}$$

or,

$$\frac{\partial h_2}{\partial a_2} = \Theta(a_2) \tag{20}$$

where,

$$\Theta(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

is the gradient of the relu. etc. While this is all possible in-principle, it is quickly starting to look like alot of painful book-keeping...

Backprop is an algorithm that lets the computer keep track of all the book-keeping for you! To develop backprop, there are three key insights.

The first key insight is that when we apply the chain rule to compute these derivatives, many of the terms are *shared*,

$$\frac{d\mathcal{L}}{dw_3} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial w_3} \tag{22a}$$

$$\frac{d\mathcal{L}}{db_3} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial b_3} \tag{22b}$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial w_2} \tag{22c}$$

$$\frac{d\mathcal{L}}{db_2} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial b_2} \tag{22d}$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial h_1}\frac{\partial h_1}{\partial a_1}\frac{\partial a_1}{\partial w_2} \tag{22e}$$

$$\frac{d\mathcal{L}}{db_1} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial h_1}\frac{\partial h_1}{\partial a_1}\frac{\partial a_1}{\partial b_2} \tag{22f}$$

The second key insight is that these shared terms actually mean something: they're the gradients of the loss wrt the activations:

$$\frac{d\mathcal{L}}{df} = \frac{\partial \mathcal{L}}{\partial f} \tag{23a}$$

$$\frac{d\mathcal{L}}{da_2} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2} \tag{23b}$$

$$\frac{d\mathcal{L}}{da_1} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial h_1}\frac{\partial h_1}{\partial a_1}. \tag{23c}$$

And the third insight is that the gradient of the activations at one layer can be written in terms of the gradients of the activations later in the network,

$$\frac{d\mathcal{L}}{df} = \frac{\partial \mathcal{L}}{\partial f} \tag{24a}$$

$$\frac{d\mathcal{L}}{da_2} = \frac{d\mathcal{L}}{df}\frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial a_2} \tag{24b}$$

$$\frac{d\mathcal{L}}{da_1} = \frac{d\mathcal{L}}{da_2}\frac{\partial a_2}{\partial h_1}\frac{\partial h_1}{\partial a_1}. \tag{24c}$$

Going even further, and dropping the colours,

$$\frac{d\mathcal{L}}{df} = \frac{\partial \mathcal{L}}{\partial f} \tag{25a}$$

$$\frac{d\mathcal{L}}{dh_2} = \frac{d\mathcal{L}}{df}\frac{\partial f}{\partial h_2} \tag{25b}$$

$$\frac{d\mathcal{L}}{da_2} = \frac{d\mathcal{L}}{dh_2}\frac{\partial h_2}{\partial a_2} \tag{25c}$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{da_2}\frac{\partial a_2}{\partial h_1} \tag{25d}$$

$$\frac{d\mathcal{L}}{da_1} = \frac{d\mathcal{L}}{dh_1}\frac{\partial h_1}{\partial a_1}. \tag{25e}$$

And this is backprop! We're just going backwards through the network, computing the gradient of the activations at one layer from the gradient at next layer. We can include parameters too,

$$\frac{d\mathcal{L}}{df} = \frac{\partial \mathcal{L}}{\partial f} \tag{26a}$$

$$\frac{d\mathcal{L}}{dh_2} = \frac{d\mathcal{L}}{df}\frac{\partial f}{\partial h_2} \qquad \frac{d\mathcal{L}}{dw_3} = \frac{d\mathcal{L}}{df}\frac{\partial f}{\partial w_3} \qquad \frac{d\mathcal{L}}{db_3} = \frac{d\mathcal{L}}{df}\frac{\partial f}{\partial b_3} \tag{26b}$$

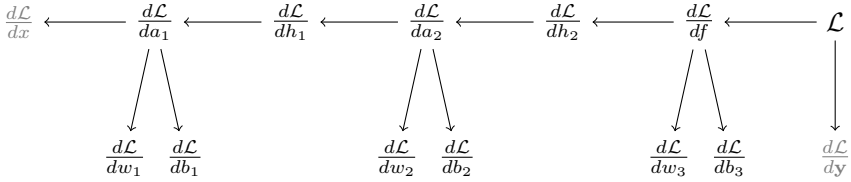
$$\frac{d\mathcal{L}}{da_2} = \frac{d\mathcal{L}}{dh_2}\frac{\partial h_2}{\partial a_2} \tag{26c}$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{da_2}\frac{\partial a_2}{\partial h_1} \qquad \frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{da_2}\frac{\partial a_2}{\partial w_2} \qquad \frac{d\mathcal{L}}{db_2} = \frac{d\mathcal{L}}{da_2}\frac{\partial a_2}{\partial b_2} \tag{26d}$$

$$\frac{d\mathcal{L}}{da_1} = \frac{d\mathcal{L}}{dh_1}\frac{\partial h_1}{\partial a_1} \tag{26e}$$

$$\frac{d\mathcal{L}}{dx} = \frac{d\mathcal{L}}{da_1}\frac{\partial a_1}{\partial x} \qquad \frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{da_1}\frac{\partial a_1}{\partial w_1} \qquad \frac{d\mathcal{L}}{db_1} = \frac{d\mathcal{L}}{da_1}\frac{\partial a_1}{\partial b_1} \tag{26f}$$

Note that everything in each row can be computed using stuff from the previous row. The full computation here looks like,



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.

3 Implementing backprop

At a mathematical level, that's all there is to backprop. However, notice that there are still *alot* of equations, and it isn't clear how you'd automated it. The answer is to notice that I've carefully arranged Eq. (26) such that every row computes the gradients of the *inputs* to a particular operation. For instance, Eq. (26d) computes the gradients of all the inputs to $a_2 = \text{linear}(h_1, w_2, b_2)$. PyTorch mirrors this arrangement in rows in Eq. (26) Specifically, PyTorch implements "forward" for each operation (e.g. linear or relu),

$$\text{outputs} = \text{func}\left(\text{inputs}\right). \tag{27}$$

PyTorch also gives us a way to apply the chain rule, to compute the gradient of all the inputs, as a function of the gradient of all of the outputs.

$$\frac{d\mathcal{L}}{d\,\text{inputs}} = \text{func.backward}\left(\frac{d\mathcal{L}}{d\,\text{outputs}}; \text{inputs}\right) \tag{28}$$

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, $\frac{d\mathcal{L}}{d\,\text{inputs}}$.

Concretely, for the ReLU:

$$h = \text{relu}\,(a) \tag{29a}$$

$$\frac{d\mathcal{L}}{da} = \text{relu.backward}\left(\frac{d\mathcal{L}}{dh}; a\right) \tag{29b}$$

And for the linear function,

$$a = \text{linear}\,(h, w, b) \tag{30a}$$

$$\left(\frac{d\mathcal{L}}{dh}, \frac{d\mathcal{L}}{dw}, \frac{d\mathcal{L}}{db}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{da}; h, w, b\right) \tag{30b}$$

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

$$\mathcal{L} = \text{sqerr}\,(f, y) \tag{31a}$$

$$\left(\frac{d\mathcal{L}}{df}, \frac{d\mathcal{L}}{dy}\right) = \text{sqerr.backward}\left(\frac{d\mathcal{L}}{d\mathcal{L}} = 1; f, y\right) \tag{31b}$$

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. $\frac{d\mathcal{L}}{dw_3}$).
- 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. $\mathbf{x}$ or $\mathbf{y}$), in which case we don't use the gradients.

Overall, the full backward pass looks like,

$$\left(\frac{d\mathcal{L}}{df}, \frac{d\mathcal{L}}{dy}\right) = \text{sqerr.backward}\left(\frac{d\mathcal{L}}{d\mathcal{L}} = 1; f, y\right) \tag{32a}$$

$$\left(\frac{d\mathcal{L}}{dh_2}, \frac{d\mathcal{L}}{dw_3}, \frac{d\mathcal{L}}{db_3}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{df}; h_2, w_3, b_3\right) \tag{32b}$$

$$\frac{d\mathcal{L}}{da_2} = \text{relu.backward}\left(\frac{d\mathcal{L}}{dh_2}; a_2\right) \tag{32c}$$

$$\left(\frac{d\mathcal{L}}{dh_1}, \frac{d\mathcal{L}}{dw_2}, \frac{d\mathcal{L}}{db_2}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{da_2}; h_1, w_2, b_2\right) \tag{32d}$$

$$\frac{d\mathcal{L}}{da_1} = \text{relu.backward}\left(\frac{d\mathcal{L}}{dh_2}; a_2\right) \tag{32e}$$

$$\left(\frac{d\mathcal{L}}{dx}, \frac{d\mathcal{L}}{dw_2}, \frac{d\mathcal{L}}{db_2}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{da_1}; x, w_1, b_1\right) \tag{32f}$$

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. 33), recording the order of and outputs from all operations.
2. Run the backward pass (Eq. 32) to compute all the gradients, by working backwards through the operations recorded in the forward pass.

4 From a scalar to a full neural network

The full network is,

$$\mathbf{a}_1 = \text{linear}\left(\mathbf{x}, \mathbf{W}_1, \mathbf{b}_1\right) \tag{33a}$$

$$\mathbf{h}_1 = \text{relu}\left(\mathbf{a}_1\right) \tag{33b}$$

$$\mathbf{a}_2 = \text{linear}\left(\mathbf{h}_1, \mathbf{W}_2, \mathbf{b}_2\right) \tag{33c}$$

$$\mathbf{h}_2 = \text{relu}\left(\mathbf{a}_2\right) \tag{33d}$$

$$\mathbf{f} = \text{linear}\left(\mathbf{h}_2, \mathbf{W}_3, \mathbf{b}_3\right) \tag{33e}$$

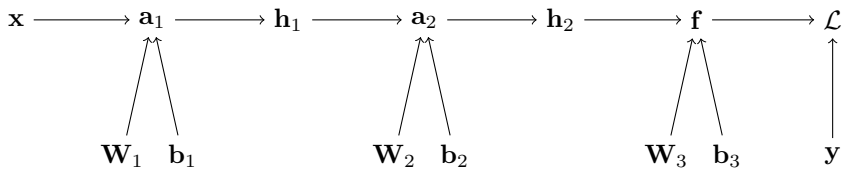
$$\mathcal{L} = \text{sqerr}\left(\mathbf{f}, \mathbf{y}\right). \tag{33f}$$

where,

$$\text{linear}\left(\mathbf{h}, \mathbf{W}, \mathbf{b}\right) = \mathbf{h}\mathbf{W} + \mathbf{b}, \tag{34a}$$

$$\text{sqerr}\left(\mathbf{f}, \mathbf{y}\right) = \tfrac{1}{2}\sum_i (f_i - y_i)^2. \tag{34b}$$

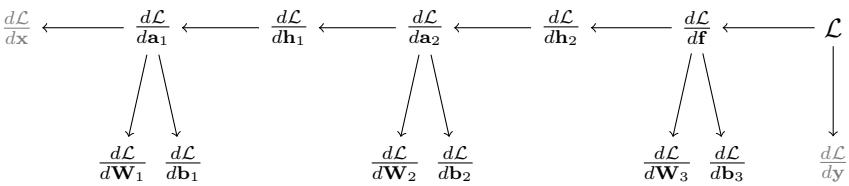
The implied compute graph is,



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

$$\frac{d\mathcal{L}}{d\mathbf{W}_1} \qquad \frac{d\mathcal{L}}{d\mathbf{W}_2} \qquad \frac{d\mathcal{L}}{d\mathbf{W}_3} \qquad \frac{d\mathcal{L}}{d\mathbf{b}_1} \qquad \frac{d\mathcal{L}}{d\mathbf{b}_2} \qquad \frac{d\mathcal{L}}{d\mathbf{b}_3}. \tag{35}$$

The backward pass again looks like,



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? Using the multivariate chain rule. Specifically, taking Eq. (26), and swapping out all the scalars for vectors, (I'm not including all the equations here, because it gets quite repetitive),

$$\frac{d\mathcal{L}}{df_m} = \frac{\partial \mathcal{L}}{\partial f_m} \tag{36a}$$

$$\frac{d\mathcal{L}}{dh_{2;i}} = \sum_k \frac{\partial f_k}{\partial h_{3;i}} \frac{d\mathcal{L}}{df_k} \qquad \frac{d\mathcal{L}}{dW_{3;ij}} = \sum_k \frac{\partial f_k}{\partial W_{3;ij}} \frac{d\mathcal{L}}{df_k} \qquad \frac{d\mathcal{L}}{db_{3;i}} = \sum_k \frac{\partial f_k}{\partial b_{3;i}} \frac{d\mathcal{L}}{df_k}$$

$$\frac{d\mathcal{L}}{da_{2;i}} = \sum_k \frac{\partial h_{2;k}}{\partial a_{2;i}} \frac{d\mathcal{L}}{dh_{2;k}} \tag{36b}$$

$$\ldots \quad \ldots \tag{36c}$$

Then, we can implement these full Jacobians in code, using,

$$\left(\frac{d\mathcal{L}}{d\mathbf{f}}, \frac{d\mathcal{L}}{d\mathbf{y}}\right) = \text{sqerr.backward}\left(\frac{d\mathcal{L}}{d\mathcal{L}} = 1; \mathbf{f}, \mathbf{y}\right) \tag{37a}$$

$$\left(\frac{d\mathcal{L}}{d\mathbf{h}_2}, \frac{d\mathcal{L}}{d\mathbf{W}_3}, \frac{d\mathcal{L}}{d\mathbf{b}_3}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{d\mathbf{f}}; \mathbf{h}_2, \mathbf{W}_3, \mathbf{b}_3\right) \tag{37b}$$

$$\frac{d\mathcal{L}}{d\mathbf{a}_2} = \text{relu.backward}\left(\frac{d\mathcal{L}}{d\mathbf{h}_2}; \mathbf{a}_2\right) \tag{37c}$$

$$\left(\frac{d\mathcal{L}}{d\mathbf{h}_1}, \frac{d\mathcal{L}}{d\mathbf{W}_2}, \frac{d\mathcal{L}}{d\mathbf{b}_2}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{d\mathbf{a}_2}; \mathbf{h}_1, \mathbf{W}_2, \mathbf{b}_2\right) \tag{37d}$$

$$\frac{d\mathcal{L}}{d\mathbf{a}_1} = \text{relu.backward}\left(\frac{d\mathcal{L}}{d\mathbf{h}_2}; \mathbf{a}_2\right) \tag{37e}$$

$$\left(\frac{d\mathcal{L}}{d\mathbf{x}}, \frac{d\mathcal{L}}{d\mathbf{W}_2}, \frac{d\mathcal{L}}{d\mathbf{b}_2}\right) = \text{linear.backward}\left(\frac{d\mathcal{L}}{d\mathbf{a}_1}; \mathbf{x}, \mathbf{W}_1, \mathbf{b}_1\right) \tag{37f}$$

Note that you'll derive these functions in the exercises!

4.1 Alternative (bad) approach using explicit Jacobians

To understand just how clever backprop is, its worth going back and trying some alternative approaches. Specifically, the most obvious approach to implementing backprop is to directly use Eq. (36). And to do that, we could implement func.jacobian, which computes the Jacobians, alongside the usual "forward" computation,

$$\text{outputs} = \text{func}\left(\text{inputs}\right) \tag{38a}$$

$$\frac{\partial\, \text{outputs}}{\partial\, \text{inputs}} = \text{func.jacobian}\left(\text{inputs}\right). \tag{38b}$$

Concretely, for the relu function,

$$\mathbf{h} = \text{relu}\left(\mathbf{a}\right) \tag{39a}$$

$$\frac{\partial \mathbf{h}}{\partial \mathbf{a}} = \text{relu.jacobian}\left(\mathbf{a}\right) \tag{39b}$$

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 can do exactly the same thing for the linear function,

$$\mathbf{a} = \text{linear}\left(\mathbf{h}, \mathbf{W}, \mathbf{b}\right) \tag{40a}$$

$$\left(\frac{\partial \mathbf{a}}{\partial \mathbf{h}}, \frac{\partial \mathbf{a}}{\partial \mathbf{W}}, \frac{\partial \mathbf{a}}{\partial \mathbf{b}}\right) = \text{linear.jacobian}\left(\mathbf{h}, \mathbf{W}, \mathbf{b}\right) \tag{40b}$$

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. 36).

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

$$h = \text{relu}(a) = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{41}$$

So the univariate partial derivative is,

$$\frac{\partial h}{\partial a} = \Theta(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{42}$$

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.

$$h_j = \text{relu}_j(\mathbf{a}) = \text{relu}(a_j) = \begin{cases} a_j & \text{if } a_j > 0 \\ 0 & \text{otherwise} \end{cases} \tag{43}$$

Thus, the partial derivatives are,

$$\frac{\partial h_j}{\partial a_\alpha} = \delta_{j\alpha}\Theta(a_j). \tag{44}$$

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,

$$\frac{d\mathcal{L}}{da_\alpha} = \sum_j \frac{\partial h_j}{\partial a_\alpha}\frac{d\mathcal{L}}{dh_j}. \tag{45}$$

Substituting the Jacobian,

$$\frac{d\mathcal{L}}{da_\alpha} = \sum_j \delta_{j\alpha}\Theta(a_\alpha)\frac{d\mathcal{L}}{dh_j}. \tag{46}$$

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

$$\frac{d\mathcal{L}}{da_\alpha} = \Theta(a_\alpha)\frac{d\mathcal{L}}{dh_\alpha}. \tag{47}$$

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. 47) than to explicitly compute the Jacobian then use Eq. (45). That's because the final expression (Eq. 45) only involves $\mathcal{O}(D)$ operations, where D is the number of features in the hidden vectors. In contrast, the Jacobian (Eq. 44) has D^2 elements, so explicitly computing the chain rule (Eq. 45) requires $\mathcal{O}(D^2)$ operations. It turns out that in almost every operations there are huge efficiencies like this that can be exploited. PyTorch is implemented to make sure that these efficiencies are exploited, by using the approach in Eq. (28) of providing functions that compute $\frac{d\mathcal{L}}{d\text{inputs}}$ from $\frac{d\mathcal{L}}{d\text{outputs}}$, rather than providing functions that compute Jacobians.

4.2 Memory consumption of backprop

Note that to perform the backward pass (e.g. 32), we need to retain all the values of the variables in the forward pass. 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. 33, 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,

$$\mathbf{a} = \text{linear}(\mathbf{h}, \mathbf{W}, \mathbf{b}) = \mathbf{h}\mathbf{W} + \mathbf{b}. \tag{48}$$

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

$$a_j = \sum_i h_i W_{ij} + b_j. \tag{49}$$

Exercise 1. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial h_\alpha} \tag{50}$$

Exercise 2. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} \tag{51}$$

Exercise 3. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial b_\alpha} \tag{52}$$

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,*

$$\frac{\partial a_j}{\partial h_\alpha} = \frac{\partial}{\partial h_\alpha}\left(\sum_i h_i W_{ij} + b_j\right) \tag{53}$$

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

$$\frac{\partial a_j}{\partial h_\alpha} = \sum_l \frac{\partial h_i}{\partial h_\alpha} W_{ij} \tag{54}$$

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

$$\frac{\partial a_j}{\partial h_\alpha} = \sum_i \delta_{i\alpha} W_{ij} \tag{55}$$

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

$$\frac{\partial a_j}{\partial h_\alpha} = W_{\alpha j}. \tag{56}$$

Answer 2. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} = \frac{\partial}{\partial W_{\alpha\beta}}\left(\sum_i h_i W_{ij} + b_j\right) \tag{57}$$

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

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} = \sum_i h_i \frac{\partial W_{ij}}{\partial W_{\alpha\beta}} \tag{58}$$

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

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} = \sum_i h_i \delta_{i\alpha} \delta_{j\beta} \tag{59}$$

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

$$\frac{\partial a_j}{\partial W_{\alpha\beta}} = h_\alpha \delta_{j\beta} \tag{60}$$

Answer 3. *Calculate the Jacobian,*

$$\frac{\partial a_j}{\partial b_\alpha} = \frac{\partial}{\partial h_\alpha}\left(\sum_i h_i W_{ij} + b_j\right) \tag{61}$$

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

$$\frac{\partial a_j}{\partial b_\alpha} = \frac{\partial b_j}{\partial b_\alpha} \tag{62}$$

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

$$\frac{\partial a_j}{\partial b_\alpha} = \delta_{j\alpha}. \tag{63}$$

Answer 4. *Apply the chain rule,*

$$\frac{d\mathcal{L}}{dh_\alpha} = \sum_j \frac{\partial a_j}{\partial h_\alpha} \frac{d\mathcal{L}}{da_j} \tag{64}$$

Substitute the Jacobian from the previous question,

$$\frac{d\mathcal{L}}{dh_\alpha} = \sum_j W_{\alpha j} \frac{d\mathcal{L}}{da_j} \tag{65}$$

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,*

$$\frac{d\mathcal{L}}{d\mathbf{h}} = \frac{d\mathcal{L}}{d\mathbf{a}} \mathbf{W}^T \tag{66}$$

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,*

$$\frac{d\mathcal{L}}{dW_{\alpha\beta}} = \sum_j \frac{\partial a_j}{\partial W_{\alpha\beta}} \frac{d\mathcal{L}}{da_j} \tag{67}$$

Substitute the Jacobian from the previous question,

$$\frac{d\mathcal{L}}{dW_{\alpha\beta}} = \sum_j h_\alpha \delta_{j\beta} \frac{d\mathcal{L}}{da_j} \tag{68}$$

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

$$\frac{d\mathcal{L}}{dW_{\alpha\beta}} = h_\alpha \frac{d\mathcal{L}}{da_\beta} \tag{69}$$

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

$$\frac{d\mathcal{L}}{d\mathbf{W}} = \mathbf{h}^T \frac{d\mathcal{L}}{d\mathbf{a}} \tag{70}$$

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

Answer 6. *Apply the chain rule,*

$$\frac{d\mathcal{L}}{db_\alpha} = \sum_j \frac{\partial a_j}{\partial b_\alpha} \frac{d\mathcal{L}}{da_j} \tag{71}$$

Substitute the Jacobian from the previous question,

$$\frac{d\mathcal{L}}{db_\alpha} = \sum_j \delta_{j\alpha} \frac{d\mathcal{L}}{da_j}. \tag{72}$$

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

$$\frac{d\mathcal{L}}{db_\alpha} = \frac{d\mathcal{L}}{da_\alpha} \tag{73}$$

If you wanted to write this with vectors,

$$\frac{d\mathcal{L}}{d\mathbf{b}} = \frac{d\mathcal{L}}{d\mathbf{a}}. \tag{74}$$