Lecture 6: Backpropagation

Roger Grosse

1 Introduction

So far, we've seen how to train "shallow" models, where the predictions are computed as a linear function of the inputs. We've also observed that deeper models are much more powerful than linear ones, in that they can compute a broader set of functions. Let's put these two together, and see how to train a multilayer neural network. We will do this using backpropagation, the central algorithm of this course. Backpropagation ("backprop" for short) is a way of computing the partial derivatives of a loss function with respect to the parameters of a network; we use these derivatives in gradient descent, exactly the way we did with linear regression and logistic regression.

If you've taken a multivariate calculus class, you've probably encountered the Chain Rule for partial derivatives, a generalization of the Chain Rule from univariate calculus. In a sense, backprop is "just" the Chain Rule — but with some interesting twists and potential gotchas. This lecture and Lecture 8 focus on backprop. (In between, we'll see a cool example of how to use it.) This lecture covers the mathematical justification and shows how to implement a backprop routine by hand. Implementing backprop can get tedious if you do it too often. In Lecture 8, we'll see how to implement an *automatic differentiation* engine, so that derivatives even of rather complicated cost functions can be computed automatically. (And just as efficiently as if you'd done it carefully by hand!)

This will be your least favorite lecture, since it requires the most tedious derivations of the whole course.

1.1 Learning Goals

- Be able to compute the derivatives of a cost function using backprop.

1.2 Background

I would highly recommend reviewing and practicing the Chain Rule for partial derivatives. I'd suggest Khan Academy[1], but you can also find lots of resources on Metacademy[2].

[1] `https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/multivariable-chain-rule/v/multivariable-chain-rule`

[2] `https://metacademy.org/graphs/concepts/chain_rule`

2 The Chain Rule revisited

Before we get to neural networks, let's start by looking more closely at an example we've already covered: a linear classification model. For simplicity, let's assume we have univariate inputs and a single training example (x, t). The predictions are a linear function followed by a sigmoidal nonlinearity. Finally, we use the squared error loss function. The model and loss function are as follows:

$$z = wx + b \tag{1}$$
$$y = \sigma(z) \tag{2}$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2 \tag{3}$$

Now, to change things up a bit, let's add a *regularizer* to the cost function. We'll cover regularizers properly in a later lecture, but intuitively, they try to encourage "simpler" explanations. In this example, we'll use the regularizer $\frac{\lambda}{2}w^2$, which encourages w to be close to zero. (λ is a hyperparameter; the larger it is, the more strongly the weights prefer to be close to zero.) The cost function, then, is:

$$\mathcal{R} = \frac{1}{2}w^2 \tag{4}$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}. \tag{5}$$

In order to perform gradient descent, we wish to compute the partial derivatives $\partial\mathcal{E}/\partial w$ and $\partial\mathcal{E}/\partial b$. the parameters or features

This example will cover all the important ideas behind backprop; the only thing harder about the case of multilayer neural nets will be the cruftier notation.

2.1 How you would have done it in calculus class

Recall that you can calculate partial derivatives the same way you would calculate univariate derivatives. In particular, we can expand out the cost function in terms of w and b, and then compute the derivatives using re-

peated applications of the univariate Chain Rule.

$$
\begin{aligned}
\mathcal{L}_{\text{reg}} &= \frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2 \\
\frac{\partial \mathcal{L}_{\text{reg}}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial w}w^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) + \lambda w \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) + \lambda w \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)x + \lambda w \\
\frac{\partial \mathcal{L}_{\text{reg}}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial b}w^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) + 0 \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)
\end{aligned}
$$

This gives us the correct answer, but hopefully it's apparent from this example that this method has several drawbacks:

1. The calculations are very cumbersome. In this derivation, we had to copy lots of terms from one line to the next, and it's easy to accidentally drop something. (In fact, I made such a mistake while writing these notes!) While the calculations are doable in this simple example, they become impossibly cumbersome for a realistic neural net.

2. The calculations involve lots of redundant work. For instance, the first three steps in the two derivations above are nearly identical.

3. Similarly, the final expressions have lots of repeated terms, which means lots of redundant work if we implement these expressions directly. For instance, $wx+b$ is computed a total of four times between $\partial\mathcal{E}/\partial w$ and $\partial\mathcal{E}/\partial b$. The larger expression $(\sigma(wx+b)-t)\sigma'(wx+b)$ is computed twice. If you happen to notice these things, then perhaps you can be clever in your implementation and factor out the repeated expressions. But, as you can imagine, such efficiency improvements might not always jump out at you when you're implementing an algorithm.

Actually, even in this derivation, I used the "efficiency trick" of not expanding out σ'. If I had expanded it out, the expressions would be even more hideous, and would involve *six* copies of $wx+b$.

The idea behind backpropagation is to share the repeated computations wherever possible. We'll see that the backprop calculations, if done properly, are very clean and modular.

2.2 Multivariable chain rule: the easy case

We've already used the univariate Chain Rule a bunch of times, but it's worth remembering the formal definition:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(g(t)) = f'(g(t))g'(t). \tag{6}$$

Roughly speaking, increasing t by some infinitesimal quantity h_1 "causes" g to change by the infinitesimal $h_2 = g'(t)h_1$. This in turn causes f to change by $f'(g(t))h_2 = f'(g(t))g'(t)h_1$.

The multivariable Chain Rule is a generalization of the univariate one. Let's say we have a function f in two variables, and we want to compute $\frac{d}{dt} f(x(t), y(t))$. Changing t slightly has two effects: it changes x slightly, and it changes y slightly. Each of these effects causes a slight change to f. For infinitesimal changes, these effects combine additively. The Chain Rule, therefore, is given by:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}. \tag{7}$$

g: R -> R^2
t |—> (x(t), y(t)) = z(t)
f : R^2 -> R
z(t)

Df(g(t)) = gradient(f(x(t), y(t))) = (f/x, f/y)
Dg(t) = (x/t, y/t)^T

2.3 An alternative notation

It will be convenient for us to introduce an alternative notation for the derivatives we compute. In particular, notice that the left-hand side in all of our derivative calculations is $\mathrm{d}\mathcal{L}/\mathrm{d}v$, where v is some quantity we compute in order to compute $\mathcal{L}$. (Or substitute for $\mathcal{L}$ whichever variable we're trying to compute derivatives of.) We'll use the notation

$$\overline{v} \triangleq \frac{\partial \mathcal{L}}{\partial v}. \tag{8}$$

This notation is less crufty, and also emphasizes that $\overline{v}$ is a value we compute, rather than a mathematical expression to be evaluated. This notation is nonstandard; see the appendix if you want more justification for it.

We can rewrite the multivariable Chain rule (Eqn. 7) using this notation:

$$\bar{t} = \overline{x}\frac{\mathrm{d}x}{\mathrm{d}t} + \overline{y}\frac{\mathrm{d}y}{\mathrm{d}t}. \tag{9}$$

Here, we use $\mathrm{d}x/\mathrm{d}t$ to mean we should actually evaluate the derivative algebraically in order to determine the formula for $\bar{t}$, whereas $\overline{x}$ and $\overline{y}$ are values previously computed by the algorithm.

2.4 Using the computation graph

In this section, we finally introduce the main algorithm for this course, which is known as **backpropagation**, or **reverse mode automatic differentiation (autodiff)**.[3]

[3] Automatic differentiation was invented in 1970, and backprop in the late 80s. Originally, backprop referred to the special case of reverse mode autodiff applied to neural nets, although the derivatives were typically written out by hand (rather than using an autodiff package). But in the last few years, neural nets have gotten so diverse that we basically think of them as compositions of functions. Also, very often, backprop is now implemented using an autodiff software package. For these reasons, the distinction between autodiff and backprop has gotten blurred, and we will use the terms interchangeably in this course. Note that there is also a forward mode autodiff, but it's rarely used in neural nets, and we won't cover it in this course.

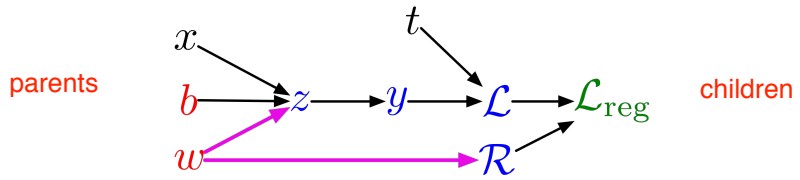

Figure 1: Computation graph for the regularized linear regression example in Section 2.4. The magenta arrows indicate the case which requires the multivariate chain rule because w is used to compute both z and $\mathcal{R}$.

Now let's return to our running example, written again for convenience:

$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2 \\
\mathcal{R} &= \frac{1}{2}w^2 \\
\mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda\mathcal{R}.
\end{aligned}
$$

Let's introduce the **computation graph**. The nodes in the graph correspond to all the values that are computed, with edges to indicate which values are computed from which other values. The computation graph for our running example is shown in Figure 1.

Note that the computation graph is *not* the network architecture. The nodes correspond to values that are computed, rather than to units in the network.

The goal of backprop is to compute the derivatives $\overline{w}$ and $\overline{b}$. We do this by repeatedly applying the Chain Rule (Eqn. 9). Observe that to compute a derivative using Eqn. 9, you first need the derivatives for its *children* in the computation graph. This means we must start from the result of the computation (in this case, $\mathcal{E}$) and work our way backwards through the graph. It is because we work backward through the graph that backprop and reverse mode autodiff get their names.

Let's start with the formal definition of the algorithm. Let $v_1, \ldots, v_N$ denote all of the nodes in the computation graph, in a topological ordering. (A topological ordering is any ordering where parents come before children.) We wish to compute all of the derivatives $\overline{v_i}$, although we may only be interested in a subset of these values. We first compute all of the values in a **forward pass**, and then compute the derivatives in a **backward pass**. As a special case, v_N denotes the result of the computation (in our running example, $v_N = \mathcal{E}$), and is the thing we're trying to compute the derivatives of. Therefore, by convention, we set $\overline{v_N} = 1$. The algorithm is as follows:

computation of derivatives depend on values

$\overline{\mathcal{E}} = 1$ because increasing the cost by h increases the cost by h.

For $i = 1, \ldots, N$

  Compute v_i as a function of $\text{Pa}(v_i)$

parent

$v_N = 1$

For $i = N - 1, \ldots, 1$

  $\overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$

child

Here $\mathrm{Pa}(v_i)$ and $\mathrm{Ch}(v_i)$ denote the parents and children of v_i.

This procedure may become clearer when we work through the example in full:

$$\begin{aligned}
\overline{\mathcal{L}_{\text{reg}}} &= 1 \\
\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{R}} \\
&= \overline{\mathcal{L}_{\text{reg}}}\, \lambda \\
\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{L}} \\
&= \overline{\mathcal{L}_{\text{reg}}} \qquad \text{keep in this form even if derivative of L_reg is 1} \\
\overline{y} &= \overline{\mathcal{L}} \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} \\
&= \overline{\mathcal{L}}\,(y - t) \\
\overline{z} &= \overline{y} \frac{\mathrm{d}y}{\mathrm{d}z} \\
&= \overline{y}\, \sigma'(z) \\
\overline{w} &= \overline{z} \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} \\
&= \overline{z}\, x + \overline{\mathcal{R}}\, w \\
\overline{b} &= \overline{z} \frac{\partial z}{\partial b} \\
&= \overline{z}
\end{aligned}$$

Since we've derived a procedure for computing $\overline{w}$ and $\overline{b}$, we're done. Let's write out this procedure without the mess of the derivation, so that we can compare it with the naïve method of Section 2.1:

$$\begin{aligned}
\overline{\mathcal{L}_{\text{reg}}} &= 1 \\
\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}}\, \lambda \\
\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \\
\overline{y} &= \overline{\mathcal{L}}\,(y - t) \\
\overline{z} &= \overline{y}\, \sigma'(z) \\
\overline{w} &= \overline{z}\, x + \overline{\mathcal{R}}\, w \\
\overline{b} &= \overline{z}
\end{aligned}$$

The derivation, and the final result, are much cleaner than with the naïve method. There are no redundant computations here. Furthermore, the procedure is *modular*: it is broken down into small chunks that can be reused for other computations. For instance, if we want to change the loss function, we'd only have to modify the formula for $\overline{y}$. With the naïve method, we'd have to start over from scratch.

Actually, there's one redundant computation, since $\sigma(z)$ can be reused when computing $\sigma'(z)$. But we're not going to focus on this point.

3 Backprop on a multilayer net

Now we come to the prototypical use of backprop: computing the loss derivatives for a multilayer neural net. This introduces no new ideas beyond

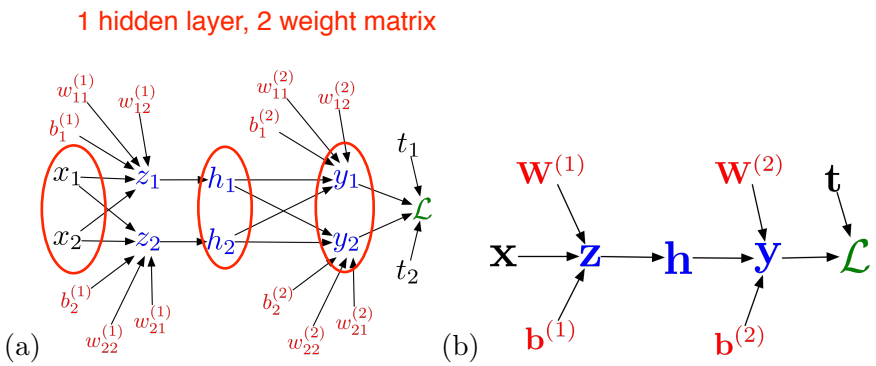

Figure 2: **(a)** Full computation graph for the loss computation in a multilayer neural net. **(b)** Vectorized form of the computation graph.

what we've already discussed, so think of it as simply another example to practice the technique. We'll use a multilayer net like the one from the previous lecture, and squared error loss with multiple output units:

$$
\begin{aligned}
z_i &= \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \\
h_i &= \sigma(z_i) \\
y_k &= \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \\
\mathcal{L} &= \frac{1}{2} \sum_k (y_k - t_k)^2
\end{aligned}
$$

As before, we start by drawing out the computation graph for the network. The case of two input dimensions and two hidden units is shown in Figure 2(a). Because the graph clearly gets pretty cluttered if we include all the units individually, we can instead draw the computation graph for the vectorized form (Figure 2(b)), as long as we can mentally convert it to Figure 2(a) as needed.

Based on this computation graph, we can work through the derivations of the backwards pass just as before.

One you get used to it, feel free to skip the step where we write down $\overline{\mathcal{L}}$.

$$
\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{y_k} &= \overline{\mathcal{L}}\,(y_k - t_k) \\
\overline{w_{ki}^{(2)}} &= \overline{y_k}\, h_i \\
\overline{b_k^{(2)}} &= \overline{y_k} \\
\overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)} \\
\overline{z_i} &= \overline{h_i}\, \sigma'(z_i) \\
\overline{w_{ij}^{(1)}} &= \overline{z_i}\, x_j \\
\overline{b_i^{(1)}} &= \overline{z_i}
\end{aligned}
$$

Focus especially on the derivation of $\overline{h_i}$, since this is the only step which actually uses the multivariable Chain Rule.

Once we've derived the update rules in terms of indices, we can find the vectorized versions the same way we've been doing for all our other calculations. For the forward pass:

$$\begin{aligned}
\mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{h} &= \sigma(\mathbf{z}) \\
\mathbf{y} &= \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \\
\mathcal{L} &= \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2
\end{aligned}$$

And the backward pass:

$$\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{\mathbf{y}} &= \overline{\mathcal{L}}\,(\mathbf{y} - \mathbf{t}) \\
\overline{\mathbf{W}^{(2)}} &= \overline{\mathbf{y}}\mathbf{h}^\top \qquad \text{outer product: } (uv^T)_{ij} = u_i v_j \\
\overline{\mathbf{b}^{(2)}} &= \overline{\mathbf{y}} \\
\overline{\mathbf{h}} &= \mathbf{W}^{(2)\top}\overline{\mathbf{y}} \\
\overline{\mathbf{z}} &= \overline{\mathbf{h}} \cdot \sigma'(\mathbf{z}) \\
\overline{\mathbf{W}^{(1)}} &= \overline{\mathbf{z}}\mathbf{x}^\top \\
\overline{\mathbf{b}^{(1)}} &= \overline{\mathbf{z}}
\end{aligned}$$

4 Appendix: why the weird notation?

Recall that the partial derivative $\partial\mathcal{E}/\partial w$ means, how much does $\mathcal{E}$ change when you make an infinitesimal change to w, holding everything else fixed? But this isn't a well-defined notion, because it depends what we mean by "holding everything else fixed." In particular, Eqn. 5 defines the cost as a function of two arguments; writing this explicitly,

$$\mathcal{E}(\mathcal{L}, w) = \mathcal{L} + \frac{\lambda}{2}w^2. \tag{10}$$

change w holding L constant

Computing the partial derivative of this function with respect to w,

$$\frac{\partial\mathcal{E}}{\partial w} = \lambda w. \tag{11}$$

But in the previous section, we (correctly) computed

$$\frac{\partial\mathcal{E}}{\partial w} = (\sigma(wx + b) - t)\sigma'(wx + b)x + \lambda w. \tag{12}$$

change w holding x,b constant

What gives? Why do we get two different answers?

The problem is that mathematically, the notation $\partial\mathcal{E}/\partial w$ denotes the partial derivative of a *function* with respect to one of its *arguments*. We make an infinitesimal change to one of the arguments, while holding the rest of the arguments fixed. When we talk about partial derivatives, we need to be careful about what are the arguments to the function. When we compute the derivatives for gradient descent, we treat $\mathcal{E}$ as a function of the parameters of the model — in this case, w and b. In this context,

$\partial \mathcal{E}/\partial w$ means, how much does $\mathcal{E}$ change if we change w while holding b fixed? By contrast, Eqn. 10 treats $\mathcal{E}$ as a function of $\mathcal{L}$ and w; in Eqn. 10, we're making a change to the second argument to $\mathcal{E}$ (which happens to be denoted w), while holding the first argument fixed.

Unfortunately, we need to refer to *both* of these interpretations when describing backprop, and the partial derivative notation just leaves this difference implicit. Doubly unfortunately, our field hasn't consistently adopted any notational conventions which will help us here. There are dozens of explanations of backprop out there, most of which simply ignore this issue, letting the meaning of the partial derivatives be determined from context. This works well for experts, who have enough intuition about the problem to resolve the ambiguities. But for someone just starting out, it might be hard to deduce the meaning from context.

That's why I picked the bar notation. It's the least bad solution I've been able to come up with.