Backpropagation of Derivatives

Derivatives for neural networks and other functions with multiple stages and parameters can be expressed by mechanical application of the chain rule. Computing these derivatives efficiently requires ordering the computation with a little care.

A quick review of the chain rule

Pre-requisite: given a simple multivariate function, you should be able to write down its partial derivatives. For example:

$$f(x,y) = x^2\,y, \quad \text{means that } \frac{\partial f}{\partial x} = 2xy, \;\; \frac{\partial f}{\partial y} = x^2.$$

You should also know how to use these derivatives to predict how much a function will change if its inputs are perturbed. Then you can run a check:

```
fn = @(x, y) (x.^2) * y; % Python: fn = lambda x,y: (x**2)*y
xx = randn(); yy = randn(); hh = 1e-5;
2*xx*yy                                  % analytic df/dx
(fn(xx+hh, yy) - fn(xx-hh, yy)) / (2*hh) % approximate df/dx
```

A function might be defined in terms of a series of computations. For example, the variables x and y might be defined in terms of other quantities: $x = r\cos\theta$, and $y = r\sin\theta$. The chain rule of differentiation gives the derivatives with respect to the earlier quantities:

$$\frac{\partial f}{\partial r} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial r}, \quad \text{and} \quad \frac{\partial f}{\partial \theta} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial \theta}.$$

A small change δ_r in r causes the function to change by about $\frac{\partial f}{\partial r}\delta_r$. That change is caused by small changes in x and y of $\delta_x \approx \frac{\partial x}{\partial r}\delta_r$ and $\delta_y \approx \frac{\partial y}{\partial r}\delta_r$.

You could write code for $f(\theta, r)$ and find its derivatives by evaluating the expressions above. You don't need answers from me: you can check your derivatives by finite differences.

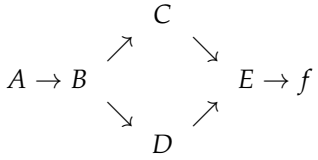
In the example above, you could also substitute the expressions for $x(\theta, r)$ and $y(\theta, r)$ into the equation for $f(x,y)$, and then differentiate with respect to θ and r. You should get the same answers. Eliminating intermediate quantities may seem easier here, but is not the best approach for larger computations, such as neural network functions.

In general, given a function $f(\mathbf{x})$, where the vector of inputs is computed from another vector $\mathbf{u}$, the chain rule states:

$$\frac{\partial f}{\partial u_i} = \sum_{d=1}^{D} \frac{\partial f}{\partial x_d}\frac{\partial x_d}{\partial u_i}.$$

Application to computation graphs

A function expressed as a sequence of operations in code can be represented as a Directed Acyclic Graph (DAG)[1], for example:



1. Caveats: 1) Computing functions does not normally require keeping the whole DAG in memory. Computing derivatives as advocated in this note can sometimes have prohibitive memory costs. 2) If you take the PMR course, you will see DAG's defining probability distributions rather than deterministic functions as here. The diagram here is not a probabilistic graphical model.

In the diagram above, f is a function of the initial inputs A. A function value is computed through intermediate quantities B, C, D, and E. That is,

$$f(A) = f(E(C(B(A)), D(B(A)))).$$

Each quantity could be a scalar, vector, or matrix (or even a larger array).

To continue the example, we will assume that the inputs and intermediate quantities are all matrices, and the function is a scalar. The derivatives of this function could be obtained by repeated application of the chain rule, which results in:

$$\frac{\partial f}{\partial A_{ij}} = \sum_{k,l,q,r} \frac{\partial B_{kl}}{\partial A_{ij}} \left(\sum_{m,n} \frac{\partial D_{mn}}{\partial B_{kl}} \frac{\partial E_{qr}}{\partial D_{mn}} + \sum_{o,p} \frac{\partial C_{op}}{\partial B_{kl}} \frac{\partial E_{qr}}{\partial C_{op}} \right) \frac{\partial f}{\partial E_{qr}}.$$

[If you don't immediately see where this expression comes from, that's ok, you can just keep reading.] If we perturb an element of A, it will affect the elements of B. These changes in turn affect C and D, and the perturbations to those matrices each affect E, which results in a perturbation to the final output. The equation above states how these effects add up.

We don't want to evaluate the expression above in the form it's written. If all the quantities are $M \times M$ matrices, explicitly looping through every element of terms such as $\frac{\partial B_{kl}}{\partial A_{ij}}$ would cost $O(M^4)$, which is worse than the cost of most functions that appear in machine learning. The two most obvious (not necessarily best) approaches to computing the expression above are to work from left-to-right or right-to-left, contracting each sum as efficiently as possible.

Working from left-to-right is called "forwards-mode differentiation". A perturbation to an element of the input A_{ij} is tracked forwards through the computation graph, noting the effect on each intermediate quantity. Implemented efficiently (I won't detail how here), the cost turns out to be similar to one function evaluation for each A_{ij}. Computing the derivatives of K inputs has similar cost to K function evaluations. That is, the cost is similar to approximating each derivative with finite differences. If we need to compute a million derivatives per optimization update, that's probably too expensive!

Working from right-to-left is called "reverse-mode differentiation", which is what we'll use for neural networks. The rest of this note outlines how the reverse-mode computation proceeds.

We start by evaluating each element $\frac{\partial f}{\partial E_{qr}}$, the derivatives of the final scalar outcome with respect to the previous stage of the computation. We put these derivatives in an array $\bar{E}$, which is the same size as E, such that $\bar{E}_{qr} = \frac{\partial f}{\partial E_{qr}}$.

For each intermediate quantity, X, we aim to find an array of derivative signals $\bar{X}$, such that $\bar{X}_{st} = \frac{\partial f}{\partial X_{st}}$. These derivative signals are computed by starting at the end of the computation and working backwards. Given a reverse-mode derivative signal, we aim to *backpropagate* it to the previous stage(s) of the computation.

In the example above we start by finding $\bar{C}$ and $\bar{D}$ from $\bar{E}$. Mathematically, these quantities are equal to:

$$\bar{C}_{op} = \frac{\partial f}{\partial C_{op}} = \sum_{qr} \frac{\partial E_{qr}}{\partial C_{op}} \frac{\partial f}{\partial E_{qr}} = \sum_{qr} \frac{\partial E_{qr}}{\partial C_{op}} \bar{E}_{qr},$$

and similarly

$$\bar{D}_{mn} = \sum_{qr} \frac{\partial E_{qr}}{\partial D_{mn}} \bar{E}_{qr}.$$

Then backpropagating to the previous stage:

$$\bar{B}_{kl} = \sum_{op} \frac{\partial C_{op}}{\partial B_{kl}} \bar{C}_{op} + \sum_{mn} \frac{\partial D_{mn}}{\partial B_{kl}} \bar{D}_{mn}.$$

Then finally to the previous and first stage:

$$\bar{A}_{ij} = \sum_{kl} \frac{\partial B_{kl}}{\partial A_{ij}} \bar{B}_{kl},$$

which contains all of the derivatives of the final function value with respect to the initial inputs, $\frac{\partial f}{\partial A_{ij}}$.

Given a function representing an intermediate step, $Z(Y)$ or $Z(X, Y)$, it is always possible to compute $\bar{Y}$ (and $\bar{X}$) from $\bar{Z}$ in a similar cost[2] to one function evaluation. Performing this backpropagation step isn't usually done as in the maths above: we don't evaluate a *huge* array of derivatives $\frac{\partial Z_{ij}}{\partial Y_{kl}}$. We work out how to perform the step efficiently. Either by doing some linear algebra (see Giles reference below), or by breaking down the operation into a sequence of scalar operations and applying backpropagation to those.

In practice, it's known how to backpropagate through common matrix operations. For example, for the matrix product $Z = XY$ it's not too hard to show that $\bar{X} = \bar{Z}Y^\top$ and $\bar{Y} = X^\top \bar{Z}$. These propagation rules are ordinary matrix multiplications like the original function (no $O(N^4)$ operations!).

Tools like Google's TensorFlow and Theano build a computation graph from your code and know how to efficiently backpropagate derivatives through many standard operations. Thus they can automatically differentiate many functions for you, including most neural networks.

Application to straightforward feedforward networks

The computation of a neural network's error on a training example is easily expressed as a graph, starting at the input features and moving through the layers of the neural network. Each layer introduces more parameters, which are additional inputs to the computation. In one reverse mode sweep through the network we can work out the derivative signals for all of the parameters in all of the layers.

In each layer of a standard feedforward neural network we form an *activation* using the weights and biases for the layer and the values of the units in the layer below:

$$\mathbf{a}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$$

These activations then have an element-wise function applied:

$$\mathbf{h}^{(l)} = g^{(l)}(\mathbf{a}^{(l)}).$$

The output of the final layer is a function value f (sometimes a scalar, sometimes vector-valued), from which a scalar cost or error function is evaluated given a target y:

$$E = L(f, y).$$

Traditional explanations of backpropagation (e.g., Murphy 16.5.4) refer to "error signals" δ (not to be confused with small perturbations δ as used in the opening section of this note). These error signals are the derivative signals for the activations of each layer $\delta_i^{(l)} = \frac{\partial E}{\partial a_i^{(l)}} = \bar{a}_i^{(l)}$. (Murphy also adds an additional index n stating which training example we are evaluating.)

We can backpropagate these $\delta = \bar{\mathbf{a}}$ error signals through the layer equations above by deriving the updates from scratch, or by combining standard rules for matrix operations. For now I

2. typically 1 to 4 times the cost, maybe as bad as six times the cost

will just state the results, so you can see that they are easy to implement:

$$\begin{aligned}
\bar{\mathbf{b}}^{(l)} &= \bar{\mathbf{a}}^{(l)} \\
\bar{W}^{(l)} &= \bar{\mathbf{a}}^{(l)} \mathbf{h}^{(l-1)\top} \\
\bar{\mathbf{h}}^{(l-1)} &= W^{(l)\top} \bar{\mathbf{a}}^{(l)} \\
\bar{\mathbf{a}}^{(l-1)} &= g^{(l-1)'}(\mathbf{a}^{(l-1)}) \odot \bar{\mathbf{h}}^{(l-1)},
\end{aligned}$$

where $\odot$ means elementwise multiply, and $g^{(l)'}$ is the derivative of the non-linearity in the lth layer. We obtain the gradients for all of the parameters in the layer, and a new error signal to backpropagate to the previous layer.

Check your understanding

The updates are given for each layer of a neural network once backpropagation has begun. Can you see how to use the derivatives of the training loss function to start the backpropagation procedure?

Why learn this stuff if derivatives can be done by software?

The code for the derivatives in many methods can be short and simple. You won't always want to bring in Theano or TensorFlow as a dependency.

You need to understand how backpropagation works to structure your software correctly, even if you lean on automatic differentiation in parts. Currently most extensible "Gaussian process" frameworks are slower than they should be, because they have structured part of their derivative computations incorrectly. Update: GPML updated its implementation of gradients in September 2016 to have the correct scaling. Widespread understanding of how to do differentation efficiently is surprisingly recent!

Moving outside neural networks, software support for automatic differentiation still isn't perfect. There are highly regarded machine learning papers, even with authors who are early adopters of backpropagation, that contain inefficient expressions for derivatives[3]. You might avoid these mistakes if you study reverse mode differentiation more generally than the common neural network case.

Further Reading

For keen students:

Reverse mode differentiation and its automatic application to code is an *old* idea:

Who invented the reverse mode of differentiation?, Andreas Griewank, *Optimization Stories*, *Documenta Matematica*, Extra Volume ISMP (2012), pp389–400.

Automatic differentiation in machine learning: a survey. Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind (2015). An overview of automatic differentiation and some common misconceptions.

In my opinion, the Baydin et al. survey misses the real reasons that adoption in machine learning has been so slow. Software tools that work easily with the sort of code written by machine learning researchers haven't existed. Theano and TensorFlow aren't as general as traditional AD compilers, but they do most of what people want, and are easy to use, which is why they've been adopted. Now the gap of what's possible and what's easy needs closing further.

3. e.g., `https://papers.nips.cc/paper/2566-neighbourhood-components-analysis`

What if we want machine learning tools to be able to pass derivatives through some function of a matrix, like the Cholesky decomposition? I recently wrote a fairly tutorial note discussing the options. TensorFlow has now adopted the matrix-based backpropagation approach championed by my note. Although there are still linear algebra operations it does not know how to differentiate.

As the Cholesky note explains, I think many of the existing guides on differentiating matrix computations are misleading. The note I found most useful was An extended collection of matrix derivative results for forward and reverse mode automatic differentiation, Mike B Giles, 2008. The introduction is succinct, so requires some careful reading. However, the note focusses on the right primitive task: given a matrix function $C(A, B)$, backpropagate the derivative signal $\bar{C}$ to obtain $\bar{A}$ and $\bar{B}$, without creating large intermediate objects. Giles also provides example Matlab/Octave code, and demonstrates how to check everything. This is the note that finally made me believe I could differentiate anything without it being a huge chore.