Backpropagation of Derivatives

Derivatives for neural networks, and other functions with multiple parameters and stages of computation, can be expressed by mechanical application of the chain rule. Computing these derivatives efficiently requires ordering the computation carefully, and expressing each step using matrix computations.

[The first parts of this note cover general material on differentiation. The symbols used ($f, x, y, \ldots$) do not imply machine learning meanings — such as model outputs, inputs, or outputs — as they do in the rest of the course notes.]

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:

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

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 directly with respect to θ and r. You should get the same answers. You might have found it easier to eliminate the intermediate quantities x and y in this example: there was no need to reason about the chain rule. However, the chain rule approach is better for larger computations, such as neural network functions, because we can apply it mechanically.

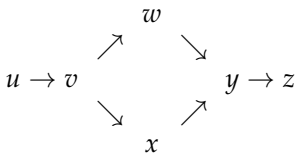
In general, given a function $f(\mathbf{x})$, where the vector of inputs is computed from another vector $\mathbf{u}$, the chain rule states:

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

When a quantity u_i is used multiple times in a function computation, we sum over the effect it has through each of the quantities that we compute directly from it.

Application to graphs of scalar computations

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



Each child variable is computed as a function of its parents. As a running example, the function $z = \exp(\sin(u^2)\log(u^2))$ can be written as a series of elementary steps following the graph above. We list the local functions below, along with the corresponding local derivatives of each child variable with respect to its parents:

$$v = u^2, \qquad w = \sin(v), \qquad x = \log(v), \qquad y = wx, \qquad z = \exp(y).$$

$$\frac{\partial v}{\partial u} = 2u, \quad \frac{\partial w}{\partial v} = \cos(v), \quad \frac{\partial x}{\partial v} = 1/v, \quad \frac{\partial y}{\partial w} = x, \quad \frac{\partial y}{\partial x} = w, \quad \frac{\partial z}{\partial y} = \exp(y) = z.$$

Forward-mode differentiation computes the derivative of every variable in the graph with respect to a scalar input. As the name suggests, we accumulate these derivatives in a forward pass through the graph. Here we differentiate with respect to the input u, and notate these derivatives with a dot: $\dot{\theta} = \frac{\partial \theta}{\partial u}$, where θ is any intermediate quantity. The chain rule of differentiation gives us each derivative in terms of a local derivative, and the derivatives that we have already computed for the parent quantities:

$$\begin{aligned}
\dot{v} &= \frac{\partial v}{\partial u} & \dot{w} &= \frac{\partial w}{\partial u} & \dot{x} &= \frac{\partial x}{\partial u} & \dot{y} &= \frac{\partial y}{\partial u} & \dot{z} &= \frac{\partial z}{\partial u} \\
&= 2u & &= \frac{\partial w}{\partial v}\frac{\partial v}{\partial u} & &= \frac{\partial x}{\partial v}\frac{\partial v}{\partial u} & &= \frac{\partial y}{\partial w}\frac{\partial w}{\partial u} + \frac{\partial y}{\partial x}\frac{\partial x}{\partial u} & &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial u} \\
& & &= \cos(v)\dot{v} & &= (1/v)\dot{v} & &= x\dot{w} + w\dot{x} & &= z\dot{y}
\end{aligned}$$

To compute the numerical value of each $\dot{\theta}$, we only need the derivatives of the elementary function used at that stage, and the numerical values of the parents' derivative signals.

A stage of the derivative computation can be computationally cheaper than computing the function in the corresponding stage. For example $\dot{z} = z\dot{y}$ requires one floating-point multiply operation, whereas $z = \exp(y)$ usually has the cost of many floating point operations. Propagating derivatives can also be more expensive: y requires one multiply, whereas $\dot{y}$ requires two multiplies and an addition. However, for the elementary mathematical functions we use in code, the derivatives are never much more expensive.

The computation of the derivatives can be done alongside the original function computation. Only a constant factor of extra memory is required: we need to track a $\dot{\theta}$ derivative for every θ intermediate quantity currently in memory. Because only derivatives of elementary functions are needed, the process can be completely automated, and is then called forwards-mode *automatic differentiation* (AD).[2]

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.

2. The (non-examinable) complex step trick mentioned in the previous note is a hacky proof of concept: for some cases (analytic functions of real-valued inputs) it tracks a small multiple of the $\dot{\theta}$ derivative quantities in the complex part of each number. More general AD tools exist.

Reverse-mode differentiation computes the derivative of a scalar output variable with respect to every other variable in the graph. As the name suggests, we accumulate these derivatives in a reverse pass through the graph. Here we differentiate the output z, and notate its derivatives with a bar: $\bar{\theta} = \frac{\partial z}{\partial \theta}$, where θ is any intermediate quantity. The chain rule of differentiation gives us each derivative in terms of a local derivative, and the derivatives that we have already computed for the child quantities. The quantities are computed starting in the right column below, and then moving to the left:

$$\begin{aligned}
\bar{u} &= \frac{\partial z}{\partial u} & \bar{v} &= \frac{\partial z}{\partial v} & \bar{w} &= \frac{\partial z}{\partial w} & \bar{x} &= \frac{\partial z}{\partial x} & \bar{y} &= \frac{\partial z}{\partial y} \\
&= \frac{\partial z}{\partial v}\frac{\partial v}{\partial u} & &= \frac{\partial z}{\partial w}\frac{\partial w}{\partial v} + \frac{\partial z}{\partial x}\frac{\partial x}{\partial v} & &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial w} & &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} & &= \exp(y) \\
&= \bar{v}(2u) & &= \bar{w}\cos(v) + \bar{x}(1/v) & &= \bar{y}x & &= \bar{y}w & &= z
\end{aligned}$$

To compute the numerical value of each $\bar{\theta}$, we only need the derivatives of the elementary function used at that stage, and the numerical values of the childrens' derivative signals. Reverse-mode automatic differentiation tools also exist, which can construct a graph and compute reverse-mode derivatives automatically from your code for a function.

Again the number of floating point operations is at most a small constant factor times the number required for one function evaluation. However, the computation can no longer be done in parallel with the original function computation. We need to traverse the graph in reverse after computing the function, using stored values of the intermediate quantities. The memory cost of reverse mode differentiation can be much higher than for the function, if we wouldn't normally keep the whole computation graph in memory.[3]

Comparison: At the end of the forwards-mode computation we had accumulated $\dot{z} = \frac{\partial z}{\partial u}$. At the end of the reverse-mode computation we accumulated the same quantity: $\bar{u} = \frac{\partial z}{\partial u}$. For this example:

$$\dot{z} = \bar{u} = \frac{\partial z}{\partial u} = \frac{\partial v}{\partial u}\left(\frac{\partial w}{\partial v}\frac{\partial y}{\partial w} + \frac{\partial x}{\partial v}\frac{\partial y}{\partial x}\right)\frac{\partial z}{\partial y},$$

with forwards-mode running left-to-right, and reverse-mode running right-to-left.

Reverse-mode tools are harder to implement, and use more memory, but have a major advantage for machine learning applications. One reverse-mode sweep simultaneously accumulates the derivatives of the output with respect to every quantity in the computation graph. If a function has many inputs, we can obtain *all* of its partial derivatives in one reverse sweep, which only costs a constant times the number of operations of one function evaluation. However, the forwards-mode procedure, like finite-differences, would require multiple sweeps through the graph, one sweep for each partial derivative. For a neural network with millions of parameters, reverse-mode differentiation (or backpropagation[4]) is millions of times faster than forwards-mode differentiation and finite differences!

It's easy to dismiss differentiation as "just the chain rule". But reverse-mode differentiation is *amazing*. We can get a million partial-derivatives, which describe the whole tangent-hyperplane of a function, usually for the cost of only a couple of evaluations of the scalar function!

---

3. A literature on reducing the memory consumption of reverse-mode differentiation exists. There are tricks to avoid storing everything, at the cost of more computation, including "check-pointing" and recomputing the inputs of reversible operations from their outputs.
4. *Backpropagation* is the term from the neural network literature for reverse-mode differentiation. Backpropagation is also sometimes used to mean the whole procedure of training a network with a gradient descent method, where the gradients come from reverse-mode differentiation.

Application to graphs of array-based operations

Cost functions in machine learning are usually based on matrix computations. *In theory*, matrix computations can be broken down into primitive scalar operations. Then reverse-mode automatic differentiation could be applied. This theory tells us that there is always an algorithm for computing derivatives of a cost function that uses at most a few times as many operations as the cost function itself.

However, *in practice*, machine learning code is not usually written at the level of primitive scalar operations. Where possible, code is written using high-level matrix operations, which are provided by a library like BLAS or LAPACK, often optimized for the hardware we're using. Our derivatives need to use these optimized libraries too, or they will be slow.[5]

As before, we consider a computation graph that ends with the computation of a scalar z. An intermediate computation creates a matrix C from two parent matrices A and B.

$$\begin{array}{ccc} \cdots \to A & \searrow & \\ & & C \to \cdots \to z \\ \cdots \to B & \nearrow & \end{array}$$

In reverse-mode differentiation we will compute $\bar{C}$, a matrix the same size as C where $\bar{C}_{ij} = \frac{\partial z}{\partial C_{ij}}$. We then need to *backpropagate* this derivative signal to compute $\bar{A}$ and $\bar{B}$. Where $\bar{A}_{ij} = \frac{\partial z}{\partial A_{ij}}$ and $\bar{B}_{ij} = \frac{\partial z}{\partial B_{ij}}$.

The chain rule gives a general equation[6] for backpropagating matrix derivatives:

$$\bar{A}_{ij} = \frac{\partial z}{\partial A_{ij}} = \sum_{k,l} \frac{\partial z}{\partial C_{kl}} \frac{\partial C_{kl}}{\partial A_{ij}} = \sum_{k,l} \bar{C}_{kl} \frac{\partial C_{kl}}{\partial A_{ij}}.$$

However, we shouldn't evaluate all of the terms in this equation. If A and C are $N \times N$ matrices, there are N^4 elements $\left\{ \frac{\partial C_{kl}}{\partial A_{ij}} \right\}$, when considering all combinations of i, j, k, and l. We've argued that derivatives can have the same computational scaling as the cost function, but most matrix functions scale better than $O(N^4)$. Therefore, it is usually possible to evaluate the sum above without explicitly computing all of its terms.

Given a standard matrix function, we don't want to differentiate the function! That is, we usually won't compute the partial derivatives of all the outputs with respect to all the inputs. Instead we derive a propagation rule that takes the derivatives of a scalar cost function with respect to the output, $\bar{C}$, and returns the derivatives with respect to parent quantities: in the above example $\bar{A}$, and $\bar{B}$. These reverse-mode propagation rules are the building blocks for differentiating larger matrix functions. By chaining them together we can differentiate any function built up of primitives for which we know the propagation rules.

Standard results: The general backpropagation rule above simplifies for some standard matrix operations as follows:

- matrix product: $C = AB \;\; \Rightarrow \;\; \bar{A} = \bar{C}B^\top$ and $\bar{B} = A^\top \bar{C}$,
- matrix addition: $C = A + B \;\; \Rightarrow \;\; \bar{A} = \bar{C}$ and $\bar{B} = \bar{C}$,
- element-wise function: $C = g(A) \;\; \Rightarrow \;\; \bar{A} = g'(A) \odot \bar{C}$, where g' is the gradient of g and $\odot$ is the element-wise or Hadamard product (`.*` in Matlab, `*` for NumPy arrays).
- masking: $C = r(A) \;\; \Rightarrow \;\; \bar{A} = r(\bar{C})$, where r is any operation that masks out elements of the matrix A. Example: extract lower-triangle, $r(A) = \text{tril}(A)$.

5. How slow? It's hardware and compiler dependent. The last time I compared my naive C code for a matrix operation to a hardware-optimized BLAS implementation, the difference in speed was about $50\times$.
6. If A is a parent to more than one child in the graph, then $\bar{A}$ is the sum over this equation for all its children C.

- reshaping: $C = r(A) \quad \Rightarrow \quad \bar{A} = r^{-1}(\bar{C})$, where r is any operation that rearranges the elements of the matrix A, and r^{-1} is the reverse operation. Examples: transpose, $r(A) = A^\top$; or rearrange into a column vector, $r(A) = \text{vec}(A)$. If a subset $\mathcal{S}$ of elements of A are ignored, such that C contains fewer elements than A, then $\bar{A}_i = 0$ for $i \in \mathcal{S}$.

Matrix-matrix multiplication is an expensive operation — $O(LMN)$ for $L \times M$ and $M \times N$ matrices[7] — it's often a computational bottleneck, and so this operation is also heavily optimized in library routines. The derivative propagation rule for matrix multiplication above can use the same optimized routines, and so has similar cost to the corresponding stage of the original function.

You could break down any matrix operation into a sequence of scalar operations, apply backpropagation to those, and then try to implement the result using matrix operations. However, there is also an elegant framework for deriving matrix-based backpropagation rules, described by Giles (2008) — full reference below.

Tools like Theano (developed 2008–2017) made automated application of reverse-mode differentiation popular in machine learning. Theano knows the backpropagation rules for many standard matrix operations, and can thus automatically differentiate matrix-based functions built from these primitives, including most neural networks. Recently a large number of alternative tools have emerged. If you want to add a new matrix function primitive to one of these tools, you often need to add the corresponding gradient propagation rule by hand.

Application to straightforward feedforward networks

The computation of a feedforward 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:

$$\mathbf{a}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \qquad \text{or} \qquad A^{(l)} = W^{(l)}H^{(l-1)} + \mathbf{b}^{(l)}\mathbf{1}_B^\top.$$

The equation on the left is for a single $K^{(l)} \times 1$ vector of activations, using a $K^{(l)} \times K^{(l-1)}$ weight matrix, and a $K^{(l)} \times 1$ vector of biases. On the right we compute a $K^{(l)} \times B$ matrix of activations $A^{(l)}$ from a matrix of hidden unit values in the previous layer, for a minibatch of B examples.[8] The biases are added to the activation for each example with the help of a $B \times 1$ vector of ones, $\mathbf{1}_B$.[9] Computing the layer for a whole minibatch at once lets us use a matrix-matrix product, which in most frameworks is heavily optimized.

These activations then have an element-wise function applied:

$$\mathbf{h}^{(l)} = g^{(l)}(\mathbf{a}^{(l)}) \qquad \text{or} \qquad H^{(l)} = g^{(l)}(A^{(l)}),$$

to give hidden unit values. These values are inputs into the next layer, or in the final layer give the network's function value $f = h^{(L)}$ or values $\mathbf{f} = \mathbf{h}^{(L)}$. We could also create a matrix of function values for a minibatch $F = H^{(L)\top}$. I've added a transpose to make the output

7. In practice. There are algorithms with better scaling, but they have large constant factors.

8. Apologies: I've put the activations and hiddens for each example in columns rather than rows in this section. The single example and mini-batch cases look more similar this way, but it's probably less standard, and doesn't match the orientation of the design matrix, so I'd have to say $H^{(0)} = X^\top$.

9. The vector of ones isn't required in NumPy code, or recent Matlab, because these languages *broadcast* the addition operator automatically. However, using the vector of ones makes the "+" a standard matrix addition, and helps us get the derivatives correct.

$B \times K$, because in this example the neural net internally has examples in columns, but data is often stored with examples in rows.[10]

Finally, we compare the network's function values to the training targets, to compute a scalar training cost

$$\text{for a single example: } c = L(f, y), \quad \text{or} \quad c = L(\mathbf{f}, \mathbf{y}), \qquad \text{or a minibatch: } c = L(F, Y).$$

We find derivatives of this scalar cost with respect to everything in the network by reverse-mode differentiation, or backpropagation. For example, $\bar{W}^{(l)}_{ij} = \frac{\partial c}{\partial W^{(l)}_{ij}}$.

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 reverse-mode derivative signals for the activations of each layer: $\delta^{(l)}_{nk} = \frac{\partial c}{\partial A^{(l)}_{kn}}$.

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

$$\begin{aligned}
\bar{\mathbf{b}}^{(l)} &= \bar{\mathbf{a}}^{(l)} & \bar{\mathbf{b}}^{(l)} &= \bar{A}^{(l)}\mathbf{1}_B = \textstyle\sum_{b=1}^{B} \bar{A}^{(l)}_{:,b} \\
\bar{W}^{(l)} &= \bar{\mathbf{a}}^{(l)}\mathbf{h}^{(l-1)\top} & \bar{W}^{(l)} &= \bar{A}^{(l)}H^{(l-1)\top} \\
\bar{\mathbf{h}}^{(l-1)} &= W^{(l)\top}\bar{\mathbf{a}}^{(l)} & \bar{H}^{(l-1)} &= W^{(l)\top}\bar{A}^{(l)} \\
\bar{\mathbf{a}}^{(l-1)} &= g^{(l-1)'}(\mathbf{a}^{(l-1)}) \odot \bar{\mathbf{h}}^{(l-1)} & \bar{A}^{(l-1)} &= g^{(l-1)'}(A^{(l-1)}) \odot \bar{H}^{(l-1)}.
\end{aligned}$$

We need to derive or look up $g^{(l)'}$, 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

Do you understand the scalar example of forward and reverse propagation well enough to numerically check that the maths in that section is correct for some random settings of u?

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?

You want to backpropagate through the linear algebra operation $C = A^\top B$. You recall that $\bar{A}$ involves B and $\bar{C}$ somehow. It's something like $\bar{A} = B\bar{C}$, $\bar{A} = B^\top\bar{C}$, $\bar{A} = B\bar{C}^\top$, $\bar{A} = B^\top\bar{C}^\top$, $\bar{A} = \bar{C}B$, $\bar{A} = \bar{C}B^\top$, $\bar{A} = \bar{C}^\top B$, or $\bar{A} = \bar{C}^\top B^\top$. How can you immediately know which of these is the only one that could be correct in general? Could you check it numerically?

Given a rule to compute $\bar{B}$ from $\bar{C}$, how could you compute a derivative of the local function $\frac{\partial C_{ij}}{\partial B_{kl}}$ if you wanted to know it? You should know why we don't usually compute all of these derivatives.

By combining the propagation rules for $C = AB$ and matrix transposition, can you backpropagate through the quadratic form $c = \mathbf{x}^\top A\mathbf{y}$? You could check your answer by deriving the expression a different way, and/or numerically. Can you then write down the $\bar{\mathbf{x}}$ and $\bar{A}$ rules for $c = \mathbf{x}^\top A\mathbf{x}$?

---

10. Earlier in the notes (and later), when considering N scalar targets, I used $\mathbf{y}$ and $\mathbf{f}$ as $N \times 1$ vectors. Whereas here I just used $\mathbf{f}$ for a single K-dimensional network output. Another option is to use $B \times K$ matrices F and Y for predictions and targets for B examples, even if $K = 1$. I'm afraid keeping track of the orientation and meaning of matrices is generally a pain.

Why study this material 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, or provide new derivative propagation operators, even if you lean on automatic differentiation in parts. Currently several extensible "Gaussian process" frameworks are slower than they should be, because they have structured part of their derivative computations incorrectly. GPML updated its implementation of gradients in September 2016 to have the correct scaling. Widespread understanding of how to do differentiation efficiently is surprisingly recent!

Some learning procedures involving modifying (hacking) parts of the gradient computation. Implementing such procedures probably requires understanding how the derivatives are computed.

Some neural network code still resorts to computing derivatives by hand. The automatic differentiation engines in machine learning frameworks aren't yet clever enough to save memory in all of the creative ways people can implement by hand.

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[11]. You might avoid these mistakes if you study reverse mode differentiation more generally than backpropagation for standard feedforward neural networks.

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). A good tutorial overview of automatic differentiation and some common misconceptions.

Although the Baydin et al. survey has "machine learning" in the title, early versions didn't cover the real reasons that adoption in machine learning has been so slow at all. Software tools that work easily with the sort of code written by machine learning researchers didn't exist. Even now (version 4), the survey has little on using efficient linear algebra operations. Theano wasn't as general as traditional AD compilers, but it did most of what people wanted, was easy to use, so it revolutionized how machine learning code was written. Now the gap of what's possible and what's easy needs closing further.

What if we want machine learning tools to automatically pass derivatives through a useful function of a matrix, like the Cholesky decomposition, that isn't commonly used in neural networks? I recently wrote a fairly tutorial note `https://arxiv.org/abs/1602.07527` discussing the options. Matrix-based approaches from this note were rapidly adopted by several tools, including TensorFlow, Autograd, MXNet, PyTorch, and Theano. However, it's still the case that there are mainstream machine learning packages that still can't differentiate some reasonably common linear algebra operations. I'm sure the situation will improve.

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"* (Giles, 2008). The introduction is succinct, so requires some careful reading. However, the note focusses on the correct primitive task: given a matrix function $C(A, B)$, backpropagate the derivative

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

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 any reasonable matrix-based code without it being a huge chore.