Automatic Differentiation

David Barber

What is AutoDiff?

- AutoDiff takes a function $f(\mathbf{x})$ and returns an exact value (up to machine accuracy) for the gradient

$$g_i(\mathbf{x}) \equiv \left.\frac{\partial}{\partial x_i} f\right|_{\mathbf{x}}$$

- Note that this is not the same as a numerical approximation (such as central differences) for the gradient.
- One can show that, if done efficiently, one can always calculate the gradient in less than 5 times the time it takes to compute $f(\mathbf{x})$.
- This is also *not* the same as symbolic differentiation.

Symbolic Differentiation

- Given a function $f(x) = \sin(x)$, symbolic differentiation returns an algebraic expression for the derivative. This is not necessarily efficient since it may contain a great number of terms.
- As an (overly!) simple example, consider

$$f(x_1, x_2) = \left(x_1^2 + x_2^2\right)^2$$

$$\frac{\partial f}{\partial x_1} = 2\left(x_1^2 + x_2^2\right) 2x_1, \qquad \frac{\partial f}{\partial x_2} = 2\left(x_1^2 + x_2^2\right) 2x_2$$

The algebraic expression is not computationally efficient. However, by defining $y = 4(x_1^2 + x_2^2)$,

$$\frac{\partial f}{\partial x_1} = yx_1, \qquad \frac{\partial f}{\partial x_2} = yx_2$$

Which is a more efficient *computational* expression.

- Also, more generally, we want to consider computational subroutines that contain loops and conditional `if` statements; these do not correspond to simple closed algebraic expressions. We want to find a corresponding subroutine that can return the exact derivative efficiently for such subroutines.

Forward and Reverse Differentiation

Forward

- This is (usually) easy to implement
- However, it is not (generally) computationally efficient.
- It cannot easily handle conditional statements or loops.

Reverse

- This is exact and computationally efficient.
- It is, however, harder to code and requires a parse tree of the subroutine.
- If possible, one should always attempt to do reverse differentiation.
- As we will discuss, the famous backprop algorithm is just a special case of reverse differentiation.
- Reverse differentiation is also important since, with it, one can understand (for example) how to deal easily with calculating the derivative of a function subject to parameter tying.

Forward Differentiation

Consider $f(x) = x^2$.

Complex arithmetic

$$f(x + i\epsilon) = (x + i\epsilon)^2 = x^2 - \epsilon^2 + 2i\epsilon x$$

$$f'(x) = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \mathrm{Im}\left(f(x + i\epsilon)\right)$$

- This also holds for any smooth function (one that can be expressed as a Taylor series).
- For finite ϵ this gives an *approximation* only.
- More accurate approximation than standard finite differences since we do not subtract two small quantities and divide by a small quantity – the complex arithmetic approach is more numerically stable.
- To implement, we need to overload all functions so that they can deal with complex arithmetic.

Forward Differentiation

Consider $f(x) = x^2$.

Dual arithmetic

Define an idempotent variable, ϵ such that $\epsilon^2 = 0$.

$$f(x + \epsilon) = (x + \epsilon)^2 = x^2 + 2x\epsilon$$

Hence

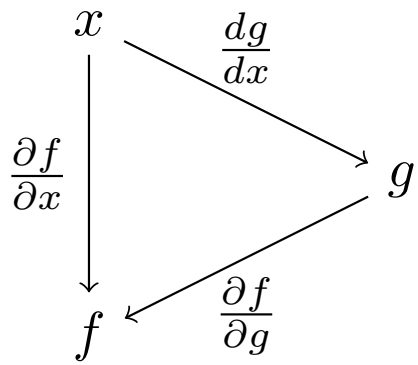
$$f'(x) = \text{DualPart} f(x + \epsilon)$$

- This holds for any smooth function $f(x)$ and non-zero value of ϵ.
- Need to overload every function in the subroutine to work in dual arithmetic.
- Numerically *exact*.
- Whilst exact, this is not necessarily efficient.

Reverse Differentiation

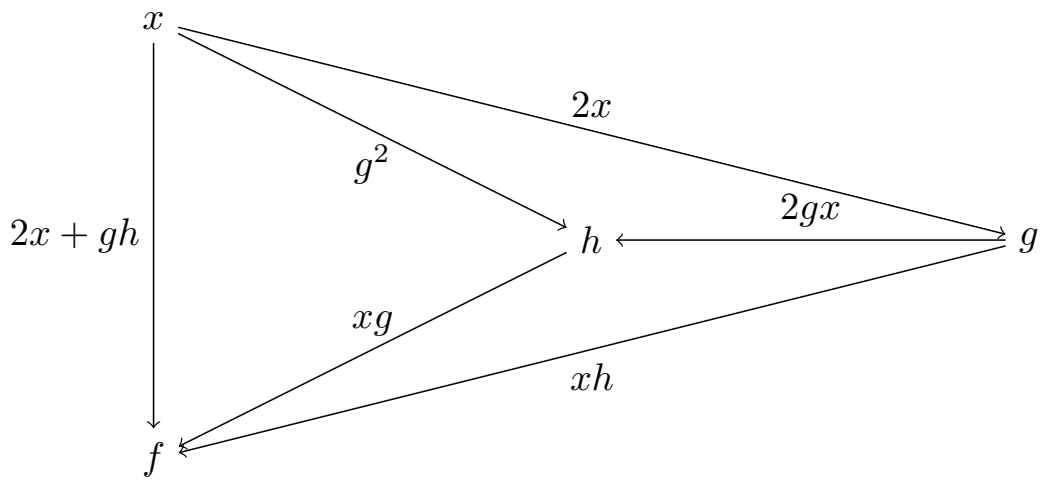
A useful graphical representation is that the total derivative of f with respect to x is given by the sum over all path values from x to f, where each path value is the product of the partial derivatives of the functions on the edges:

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial g}\frac{dg}{dx}$$



Example

For $f(x) = x^2 + xgh$, where $g = x^2$ and $h = xg^2$



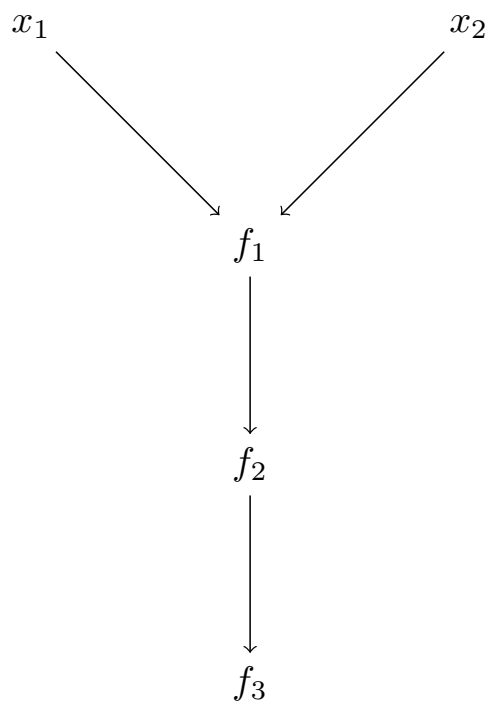
$$f'(x) = (2x + gh) + (g^2 xg) + (2x2gxxg) + (2xxh) = 2x + 8x^7$$

Reverse Differentiation

Consider

$$f(x_1, x_2) = \cos\left(\sin(x_1 x_2)\right)$$

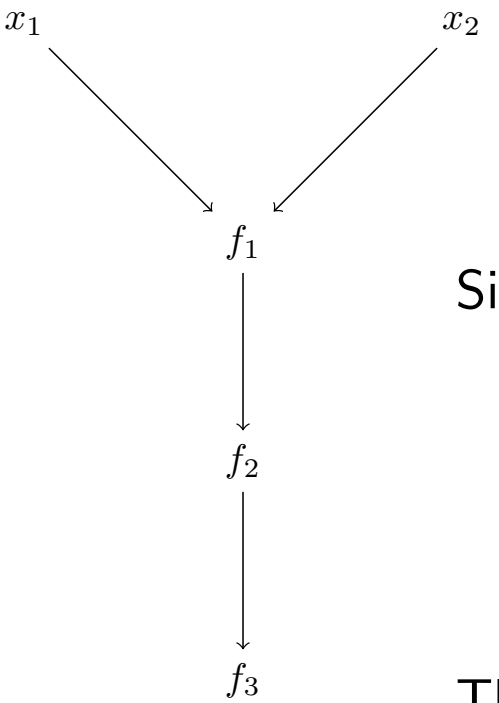
We can represent this computationally using an Abstract Syntax Tree (AST):



$$f_1(x_1, x_2) = x_1 x_2$$
$$f_2(x) = \sin(x)$$
$$f_3(x) = \cos(x)$$

Given values for x_1, x_2, we first run forwards through the tree so that we can associate each node with an actual function value.

Reverse Differentiation



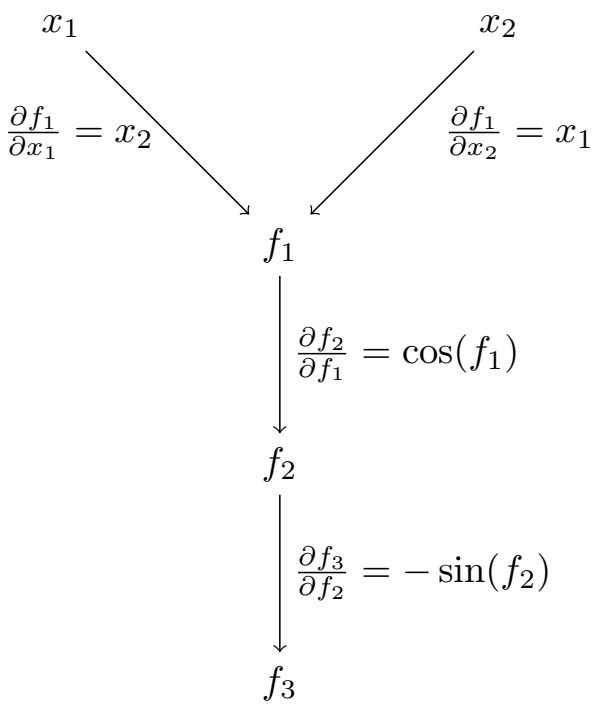
$$\frac{df_3}{dx_1} = \frac{\partial f_3}{\partial f_2}\frac{df_2}{dx_1} = \underbrace{\frac{\partial f_3}{\partial f_2}\frac{df_2}{df_1}}_{\frac{df_3}{df_1}}\frac{df_1}{dx_1}$$

Similarly,

$$\frac{df_3}{dx_2} = \underbrace{\frac{\partial f_3}{\partial f_2}\frac{df_2}{df_1}}_{\frac{df_3}{df_1}}\frac{df_1}{dx_2}$$

The two derivatives share the same computation branch and we want to exploit this.

Reverse Differentiation



1. Find the reverse ancestral (backwards) schedule of nodes $(f_3, f_2, f_1, x_1, x_2)$.
2. Start with the first node n_1 in the reverse schedule and define $t_{n_1} = 1$.
3. For the next node n in the reverse schedule, find the child nodes $\mathrm{ch}(n)$. Then define

$$t_n = \sum_{c \in \mathrm{ch}(n)} \frac{\partial f_c}{\partial f_n} t_c$$

4. The total derivatives of f with respect to the root nodes of the tree (here x_1 and x_2) are given by the values of t at those nodes.

This is a general procedure that can be used to automatically define a subroutine to efficiently compute the gradient. It is efficient because information is collected at nodes in the tree and split between parents only when required.

Hessian-Vector product

- Consider a function $E(\theta)$ and its Hessian

$$H_{ij} \equiv \frac{\partial^2 E}{\partial \theta_i \theta_j}$$

- In the Newton optimisation method, we update a vector θ by the inverse Hessian times the gradient of the objective:

$$\mathbf{x} = \mathbf{H}^{-1}\mathbf{g}$$

- In practice, we find the update $\mathbf{x}$ by solving the linear system

$$\mathbf{H}\mathbf{x} = \mathbf{g}$$

  typically by conjugate gradients.
- For a neural net with W parameters, just storing the Hessian takes $O\left(W^2\right)$ space and computing the Hessian-vector product $O\left(W^2\right)$ time. This is too expensive.
- Magically, there is a way to compute a Hessian-vector product in $O\left(W\right)$ time and space!

Hessian-Vector product

- Define the Directional Derivative in the direction $\mathbf{v}$ as

$$D_{\mathbf{v}}(f) \equiv \lim_{\delta \to 0} \frac{f(\mathbf{x} + \delta \mathbf{v}) - f(\mathbf{x})}{\delta} = \sum_i v_i \frac{\partial f}{\partial x_i} = \mathbf{v}^{\mathsf{T}} \nabla f$$

  This is a scalar value and represents the rate of change of the function along the direction $\mathbf{v}$.

- This is a derivative and so all usual rules (chain rule etc) apply as well.

- $$\begin{aligned} D_{\mathbf{v}}\left(\frac{\partial E}{\partial \theta_i}\right) &= \lim_{\delta \to 0} \frac{\frac{\partial E}{\partial \theta_i}(\theta + \delta \mathbf{v}) - \frac{\partial E}{\partial \theta_i}}{\delta} \\ &= \lim_{\delta \to 0} \frac{\frac{\partial E}{\partial \theta_i} + \delta \sum_j v_j \frac{\partial^2 E}{\partial \theta_i \partial \theta_j} + O\left(\delta^2\right) - \frac{\partial E}{\partial \theta_i}}{\delta} \\ &= [\mathbf{H}\mathbf{v}]_i \end{aligned}$$

  where $\mathbf{H}$ is the Hessian.

- Hence, we can find Hessian-vector products by taking the directional derivative of the gradient in the direction of the vector.

Hessian-Vector product example

- For the function

$$E(\boldsymbol{\theta}) = \left(\mathbf{x}^{\mathsf{T}}\boldsymbol{\theta}\right)^2$$

the Hessian is

$$\mathbf{H} = 2\mathbf{x}\mathbf{x}^{\mathsf{T}}$$

Hence,

$$\mathbf{H}\mathbf{v} = 2\left(\mathbf{x}^{\mathsf{T}}\mathbf{v}\right)\mathbf{x}$$

- We want to see how to calculate this efficiently within the autodiff framework
- According to our approach, we first need to get the gradient which we will do using the usual forward and backward reverse autodiff equations.

Hessian-Vector product example

For the function

$$f(\boldsymbol{\theta}) = \left(\mathbf{x}^\mathsf{T}\boldsymbol{\theta}\right)^2$$

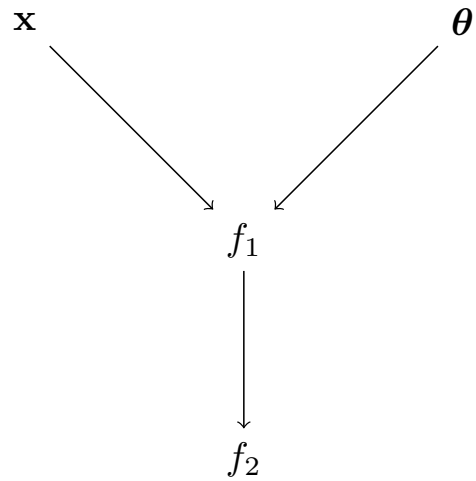
The nodes of the computation graph are

$\boldsymbol{\theta}$

$\mathbf{x}$

$f_1 = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$

$f_2 = (f_1)^2$



Hessian-Vector product example

- If we run reverse Autodiff, this defines the equations

$$
\begin{aligned}
t_2 &= 1 \\
t_1 &= \frac{\partial f_2}{\partial f_1} t_2 \\
t_\theta &= \frac{\partial f_1}{\partial \theta} t_1
\end{aligned}
$$

According to our theory, the Hessian-vector product is given by the directional derivative of the gradient, namely $D_{\mathbf{v}}(t_\theta)$. We can calculate this as follows:

$$
\begin{aligned}
D_{\mathbf{v}}(t_2) &= 0 \\
D_{\mathbf{v}}(t_1) &= \frac{\partial f_2}{\partial f_1} D_{\mathbf{v}}(t_2) + t_2 D_{\mathbf{v}}\left(\frac{\partial f_2}{\partial f_1}\right) \\
D_{\mathbf{v}}(t_\theta) &= \frac{\partial f_1}{\partial \theta} D_{\mathbf{v}}(t_1) + t_1 D_{\mathbf{v}}\left(\frac{\partial f_1}{\partial \theta}\right)
\end{aligned}
$$

Hessian-Vector product example

- Let's look at some of these terms:

$$D_{\mathbf{v}}\left(\frac{\partial f_2}{\partial f_1}\right) = \sum_i v_i \frac{\partial^2 f_2}{\partial f_1 \partial \theta_i} = \sum_i v_i \frac{\partial^2 f_2}{\partial f_1^2}\frac{\partial f_1}{\partial \theta_i} = \frac{\partial^2 f_2}{\partial f_1^2}\sum_i v_i \frac{\partial f_1}{\partial \theta_i}$$
$$= \frac{\partial^2 f_2}{\partial f_1^2} D_{\mathbf{v}}\left(f_1\right)$$

  Similarly

$$D_{\mathbf{v}}\left(\frac{\partial f_1}{\partial \theta}\right) = \frac{\partial^2 f_1}{\partial \theta^2} D_{\mathbf{v}}\left(\theta\right)$$

- Hence we can relate directional derivatives of gradients to directional derivatives of the nodes.
- We then just need to calculate the node directional derivatives, which we can do by going back to the equations.

Hessian-Vector product example

- We can find the directional derivative of each node by a forward pass through each of the nodes:
-

$$
\begin{aligned}
D_{\mathbf{v}}(\boldsymbol{\theta}) &= \mathbf{v} \\
D_{\mathbf{v}}(\mathbf{x}) &= \mathbf{0} \\
D_{\mathbf{v}}(f_1) &= \mathbf{v}^{\mathsf{T}}\mathbf{x} \\
D_{\mathbf{v}}(f_2) &= 2 f_1 D_{\mathbf{v}}(f_1)
\end{aligned}
$$

- We now have everything we need to get the Hessian-vector product. Using the reverse pass, with results computed from the two forward passes and the gradient reverse pass. Using

$$
D_{\mathbf{v}}(t_1) = 2\mathbf{v}^{\mathsf{T}}\mathbf{x}, \qquad D_{\mathbf{v}}\left(\frac{\partial f_1}{\partial \theta}\right) = 0
$$

we get

$$
D_{\mathbf{v}}(t_\theta) = \mathbf{x} D_{\mathbf{v}}(t_1) = 2\left(\mathbf{v}^{\mathsf{T}}\mathbf{x}\right)\mathbf{x}
$$

which is the correct result.

Hessian-vector product

If each node f_i is a function of its parent nodes $\pi_1^i, \ldots,$, there are then 4 passes required:

Standard forward pass: With i indexing a forward ordering calculate the values of the nodes:

$$f_i = f_i(\pi_1^i, \ldots, \pi_k^i)$$

Standard Reverse Pass: With n indexing a reverse ordering:

$$t_n = \sum_{c \in \mathrm{ch}(n)} \frac{\partial f_c}{\partial f_n} t_c$$

Directional Derivative Forward Pass:

$$D_{\mathbf{v}}(f_i) = \sum_k \frac{\partial f_i}{\partial \pi_k^i} D_{\mathbf{v}}\left(\pi_k^i\right)$$

Directional Derivative Reverse Pass:

$$D_{\mathbf{v}}(t_n) = \sum_{c \in \mathrm{ch}(n)} \left[\frac{\partial f_c}{\partial f_n} D_{\mathbf{v}}(t_c) + t_c \frac{\partial^2 f_c}{\partial f_n^2} D_{\mathbf{v}}(f_n) \right]$$

Gauss-Newton-vector product

We can write the loss as a function of the output $y = N(\theta)$ of the network

$$E(\theta) = L(N(\theta))$$

Then

$$\frac{\partial E}{\partial \theta_i} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \theta_i}$$

and

$$\frac{\partial^2 E}{\partial \theta_i \theta_j} = \frac{\partial L}{\partial y}\frac{\partial^2 y}{\partial \theta_i \partial \theta_j} + \frac{\partial y}{\partial \theta_i}\frac{\partial^2 L}{\partial y^2}\frac{\partial y}{\partial \theta_j}$$

Provided the loss L is a convex function of the output of the network (which is true for squared loss with a linear output and also for log loss with softmax output) then, by ignoring the first term we can definite a positive semidefinite Gauss-Newton matrix

$$G_{ij} \equiv \frac{\partial y}{\partial \theta_i} \frac{\partial^2 L}{\partial y^2} \frac{\partial y}{\partial \theta_j}$$

and

$$\sum_j G_{ij} v_j = \frac{\partial y}{\partial \theta_i} \sum_j \frac{\partial^2 L}{\partial y^2} \frac{\partial y}{\partial \theta_j} v_j = \frac{\partial y}{\partial \theta_i} \sum_j \frac{\partial}{\partial \theta_j} \left(\frac{\partial L}{\partial y} \right) v_j = \frac{\partial y}{\partial \theta_i} D_{\mathbf{v}} (t_y)$$

This means that we can efficiently compute a Gauss-Newton-vector product by only doing the reverse directional derivative pass up to the network output layer.

Net with multiple outputs

- If we have multiple outputs y_k, then the above becomes

$$G_{ij} \equiv \sum_k \frac{\partial y_k}{\partial \theta_i} \frac{\partial^2 L}{\partial y_k^2} \frac{\partial y_k}{\partial \theta_j}$$

and

$$\sum_j G_{ij} v_j = \sum_k \frac{\partial y_k}{\partial \theta_i} D_{\mathbf{v}} (t_{ky})$$

- This seems like a problem since we need the derivative of each y_k wrt each input θ_i.
- However, we note that $D_{\mathbf{v}}(t_{ky}) \equiv \mu_k$ can be calculated by doing the reverse direction derivative up to the $\mathbf{y}$ layer and then stored as known values μ_k. To compute

$$\sum_j G_{ij} v_j = \sum_k \frac{\partial y_k}{\partial \theta_i} \mu_k$$

we note that we can consider this as the gradient wrt θ of an 'auxiliary' loss function

$$\sum_k y_k \mu_k$$

We can calculate the gradient of this auxiliary loss function by simply using the standard reverse autodiff backpass using this loss. No additional forward pass is needed since the auxiliary loss and the original loss share the same layers up to layer $\mathbf{y}$.

Software

- AutoDiff has been around a long time (since the 1960's).
- There are tons of tools out there with varying degrees of sophistication.
- The most efficient tools use special purpose optimisers to first obtain the most compact AST.
- Stan is a popular recent C++ tools from Stanford.
- Theano is a popular tool in python, developed by Montreal Machine Learners.
- TensorFlow is becoming a standard package (though still slower than Theano).