Automatic Differentiation

Aleksnadar Botev, Dmitry Adamskiy, David Barber

What is AutoDiff?

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

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

- 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.

A better numerical difference

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

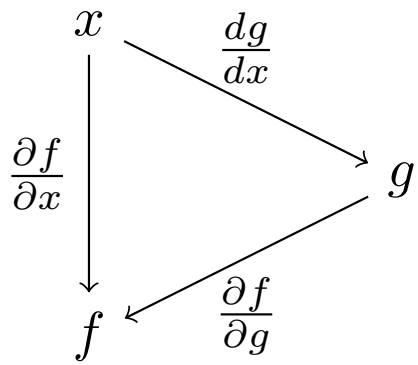
$$f'(x) = \text{DualPart} f(x + \epsilon)$$

- 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.

Computation Graph

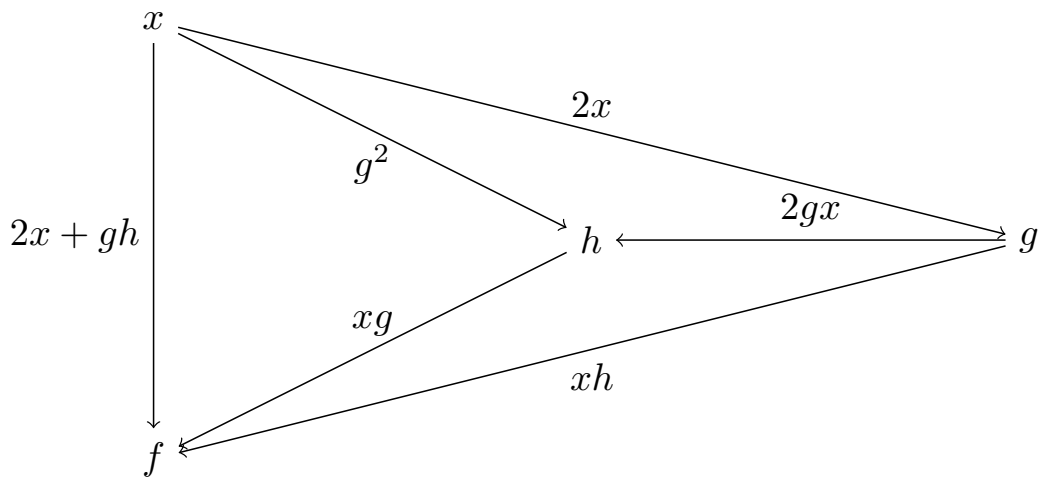
A useful graphical representation is that the total derivative of f with respect to x is given by the sum over all path values from x to f, where each path value is the product of the partial derivatives of the functions on the edges:

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



Example

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



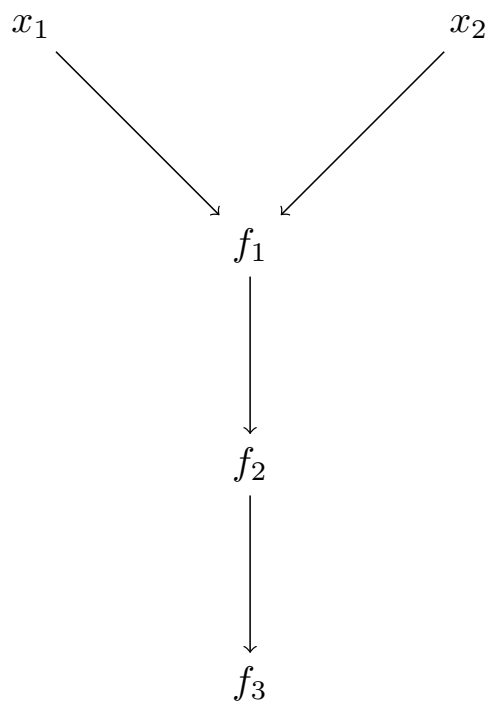
$$f'(x) = (2x + gh) + (g^2 xg) + (2x2gxxg) + (2xxh) = 2x + 8x^7$$

Computation Graph

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.

Forward Mode using a Computation Graph

- We can calculate the derivative with respect to x_1 in our example by starting at node x_1 with the value v_1, and then going to it's child f_1.
- At f_1 we calculate $d = \left.\frac{\partial f_1}{\partial x_1}\right|_{v_1}$ and $v = f_1(v_1)$
- We then go to the child f_2 and update $d \to d \times \left.\frac{\partial f_2}{\partial f_1}\right|_{v}$ and $v \to f_2(v)$
- We then go to the child f_3 and update $d \to d \times \left.\frac{\partial f_3}{\partial f_2}\right|_{v}$ and $v \to f_3(v)$
- d will then contain the derivative and the value

Forward Mode using a Computation Graph

More generally, to calculate the derivative of f with respect to a variable x evaluated at v

1. Define a valid forward schedule $n_1, \ldots$ so that node n_i can be calculated provided that previous nodes n_j $(j < i)$ have already been calculated.
2. Set $t_n = 0$ for all nodes that have no parents (except node x).
3. Set $v_x = v$ and $t_x = 1$.
4. Starting from the first (in the ancestral sense) child of node x, for each node n in the forward schedule compute

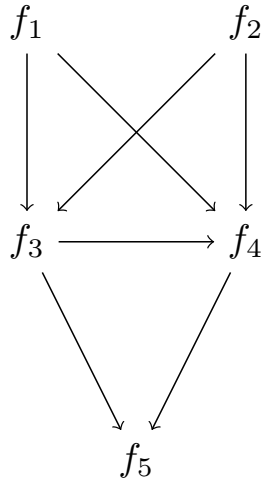
$$v_n = f_n(\cup_{p \in \mathrm{pa}(n)} f_p) \qquad t_n = \sum_{p \in \mathrm{pa}(n)} \left. \frac{\partial f_n}{\partial f_p} \right|_{v_p} t_p$$

5. Repeat the above until the tree is descended to node f.
6. The value of f evluated at v is then given by v_f
7. The total derivative of f with respect x evaluated at v is then given by t_f.

This general procedure defines a subroutine to compute the gradient with respect to x using a single pass through the graph. As we go through the graph, we can free the memory of nodes whose children have been computed. The memory requirement is roughly twice the memory required to compute the function itself.

Example Forward Pass using Computation Graph

Let's say we have a function with the Tree on the right and that we wish to calculate $\frac{\partial f_5}{\partial f_2}$.

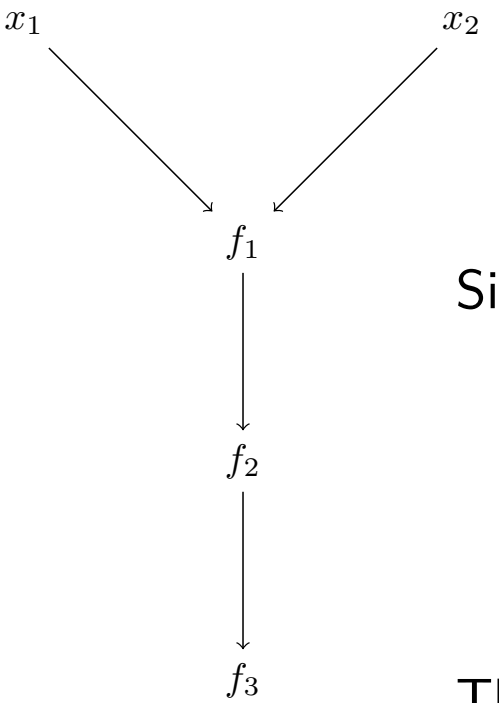


We assume that nodes f have been placed in ancestral order, meaning that we can calculate the value of f_i provided we have calculated all f_j for $j < i$.

- Set v_1, v_2 and $t_{f_1} = 0$, $t_{f_2} = 1$.
- Children of f_2 are (in ancestral order) f_3 and f_4.
- Set $t_{f_3} = \left.\frac{\partial f_3}{\partial f_2}\right|_{v_2} t_{f_2}$ and $v_3 = f_3(v_1, v_2)$
- Set $t_{f_4} = \left.\frac{\partial f_4}{\partial f_3}\right|_{v_3} t_{f_3} + \left.\frac{\partial f_4}{\partial f_2}\right|_{v_2} t_{f_2}$ and $v_4 = f_4(v_1, v_2, v_3)$
- Set $t_{f_5} = \frac{\partial f_5}{\partial f_3} t_{f_3} + \frac{\partial f_5}{\partial f_4} t_{f_4}$ and $v_5 = f_5(v_3, v_4)$

Note that as soon as t_3, v_3, t_4, v_4 have been calculated v_1, t_1, v_2, t_2 can be freed from memory.

Reverse Differentiation



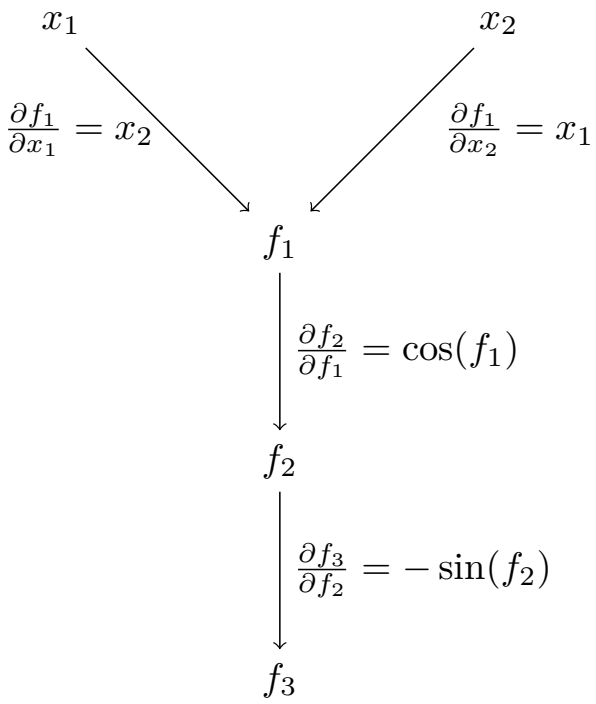
$$\frac{df_3}{dx_1} = \frac{\partial f_3}{\partial f_2}\frac{df_2}{dx_1} = \underbrace{\frac{\partial f_3}{\partial f_2}\frac{df_2}{df_1}}_{\frac{df_3}{df_1}}\frac{df_1}{dx_1}$$

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

$$t_n = \sum_{c \in \mathrm{ch}(n)} \left.\frac{\partial f_c}{\partial f_n}\right|_{v_n} t_c$$

5. 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.

The Jacobian Matrix

For a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ the Jacobian matrix $\mathbf{J}$ is defined as the $m \times n$ matrix of all partial derivatives

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_2} & \cdots & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_n} \\ \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_2} & \cdots & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{f}_m}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_m}{\partial \mathbf{x}_2} & \cdots & \frac{\partial \mathbf{f}_m}{\partial \mathbf{x}_n} \end{bmatrix}$$

The mutlivariate Chain Rule is

$$\frac{d\mathbf{f}}{d\mathbf{x}} = \sum_i J_{\mathbf{y}_i} \frac{d\mathbf{y}_i}{d\mathbf{x}}$$

where all variables are vectors.

The difference between Forward and Reverse AutoDiff

Forward mode starts by initializing all of the total derivatives of the variable x with respect to we are differentiating f. For a scalar x we used $t_x = 1$, but what about when x is a vector?

* If we initialize $t_{\mathbf{x}_i} = \mathbf{v}_i$ than Forward mode AutoDiff computes $\mathbf{J}\mathbf{v}$

Backward mode starts by initializing all of the total derivatives of the variable function nodes f wich we are differentiating with respect to x. For a scalar f we used $t_f = 1$, but what about when f is a vector?

* If we initialize $t_{\mathbf{f}_i} = \mathbf{v}_i$ than Backward mode AutoDiff computes $\mathbf{J}^T\mathbf{v}$

There is a fundamental difference between the two AutoDiff modes which is not aparent in the scalar cases. Although, both can be used to compute the Jacobian matrix their efficiency is mainly dependable on the structure of the problem - whether $\dim(\mathbf{x})$ is greater than $\dim(\mathbf{f})$.

Forward and Reverse Differentiation

Denote with $O(\mathbf{f})$ time complexity for calculating the function value of $\mathbf{f}(\mathbf{x})$.

Forward

- Calculates a directional derivative using a single forward pass trough the computation graph - $\mathbf{Jv}$
- Computational complexity is $O(\dim(\mathbf{x})) * O(\mathbf{f})$
- Generally as memory efficient as calcualting the function value of $\mathbf{f}$.
- Not efficient for standard Machine Learning where $\dim(\mathbf{x})$ is large
- Practically used for Hessian or Gauss-Newton vector products and hyper-parameter optimization

---

Reverse

- Calculates the gradient using a signle forward and backward pass - $\mathbf{J}^T\mathbf{v}$
- Computational complexity is $O(\dim(\mathbf{f})) * O(\mathbf{f})$
- This is memory intensive as it requires storing the values of all the nodes' values of the graph before running the backward pass.
- Efficient for standard Machine Learning where we have a scalar objective.
- Practically used for any model trained by gradient based optimizers. Backpropagation, Backpropagation Trough Time and many other algorithms are just a special case of Reverse mode AutoDiff.

Formalizing the two AutoDiff modes

For clearer notations of when we use Forward mode and when Backward mode we define the corresponding mathematical operators

- The $\mathcal{L}$ operator (L-op) takes as inputs a function f, a variable x and vector v with the same dimensionality as f and performs Reverse mode Autodiff

$$\mathcal{L}(f, x, v) = J_x^f{}^T v = J^T v$$

- The $\mathcal{R}$ operator (R-op) takes as inputs a function f, a variable x and a vector v with the same dimensionality as x and performs Forward mode Autodiff

$$\mathcal{R}(f, x, v) = J_x^f v = Jv$$

- Using these two operators we will be able to perform products of higher order derivatives without explicitely calculating them.

$$\mathcal{L}(f, x, v) = \mathcal{L}(f^T v, x, 1)$$

- For scalar functions f we can calculate directional derivatives in two ways:

$$\nabla^T v = \mathcal{R}(f, x, v) = \mathcal{L}(f, x, 1)v$$

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

- Notice that the Hessian is the Jacobian of the gradient

$$J_x^{\nabla} = H$$

- Since the Hessian is symmetric we can can calculate the Hessian-vector product in two ways:

$$H = \mathcal{L}(\nabla, x, v) = \mathcal{R}(\nabla, x, v)$$

- We can calculate the gradient as $\mathcal{L}(f, x, v)$. This gives backward-after-backward and forward-after-backward diff modes.
- Alternatively we notice that the Hessian-vector product is the gradient of the directional derivative

$$Hv = \frac{d(\nabla^T v)}{dx} = \mathcal{L}(\nabla^T v, x, 1)$$

- We can calculate the directional derivative using either the L-op or the R-op. This gives backward-after-backward and backward-after-forward.

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 define a positive semidefinite Gauss-Newton matrix

Software

- AutoDiff has been around a long time (since the 1970's). Seppo Linnainmaa introduced Reverse Mode AutoDiff as part of his 1970's Masters Thesis at the University of Helsinki.
- There are tons of tools out there with varying degrees of sophistication.
- The most efficient tools use additional optimization steps to make the graph smaller and remove any redundant computation.
- Currently TensorFlow is the most widely used package, developed by Google
- Packages which can work with dynamic graphs are popular in academia (Pytorch, Dynet)