1 Introduction

Automatic differentiation (AD or autodiff throughout) is a set of techniques for numerically evaluating a gradient. It achieves this by breaking the function-to-be-differentiated into a set of elementary operations and using the chain rule. We can contrast this with two other flavours of differentiation: symbolic and finite-differences. Symbolic differentiation aims to calculate a symbolic mathematical expression for the derivative of a particular function, while finite-differences numerically approximates a derivative at a particular point. Compared to the latter of these, AD achieves machine precision accuracy without sacrificing computational complexity and does not slow down as much when input dimensionality is high [1]. Compared to symbolic differentation, AD is not limited to closed-form functions, and can handle loops, recursion and control flow.

Not only is autodiff powerful compared to other differentiation techniques, but it is already used widely. Gradient-based optimization, using the negative gradient of a function to step towards its local minimum, is pivotal to deep learning implementations. The importance of such optimization led us to develop an optimization feature as part of autodiff30, which provides a simple user interface and uses AD in the backend. These have come to dominate many machine learning domains including computer vision and natural language processing. Implementations of computational methods for probabilistic inference, such as Hamiltonian Monte Carlo and Variational Inference, have also come to rely on automatic differentiation for the computation of necessary gradients.

2 Background

2.1 Automatic Differentiation

We can consider any function that we may want the derivative of as being made up of elementary operations applied sequentially, such as arithmetic operations, switching sign, exponential, logarithmic, and trigonometric operators. We can easily compute the derivatives of such operations. Combining these derivatives through the chain rule gives the foundations of automatic differentiation.

2.2 The Chain Rule

In its simplest form, for real-valued functions of one variable, with $y = f(u)$ and $u = g(x)$ then the chain rule states

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Generalizing to an input vector x of dimensionality m and a scalar function f action on a vector of n other functions $y_i(x)$, we also have a more general form of the chain rule:

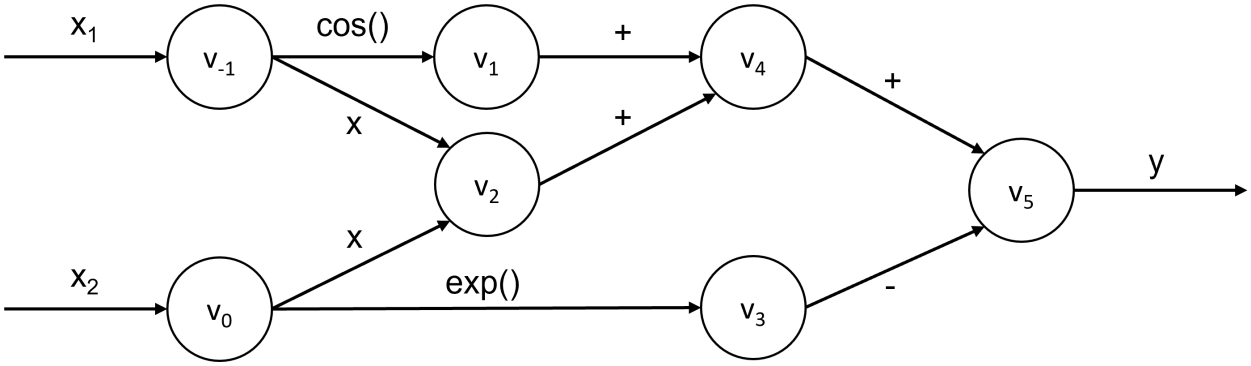
$$\nabla_x f = \sum_{i=1}^{n} \frac{\partial f}{\partial y_i} \nabla y_i(x)$$

2.3 Example

This example is motivated by [1]. Consider the function $y = f(x_1, x_2) = \cos(x_1) + x_1 x_2 - \exp(x_2)$ where we want to find $\frac{\partial y}{\partial x_1}$ at $(x_1, x_2) = (2, 4)$. We can decompose this problem into the steps in the table below.

Elementary operations	Derivative w.r.t x_1
$v_{-1} = x_1$	$\dot{v_{-1}} = 1$
$v_0 = x_2$	$\dot{v_0} = 0$
$v_1 = \cos(v_{-1})$	$\dot{v_1} = -\sin(v_{-1})\dot{v_{-1}}$
$v_2 = v_{-1}v_0$	$\dot{v_2} = \dot{v_{-1}}v_0 + v_{-1}\dot{v_0}$
$v_3 = \exp(v_0)$	$\dot{v_3} = \dot{v_0}\exp(v_0)$
$v_4 = v_1 + v_2$	$\dot{v_4} = \dot{v_1} + \dot{v_2}$
$v_5 = v_4 - v_3$	$\dot{v_5} = \dot{v_4} - \dot{v_3}$
$y = v_5$	$\dot{y} = \dot{v_5}$

The function can equivalently be displayed as a computational graph, where the nodes are intermediate variables and edges represent elementary operators:



Automatic differentiation can operate in two modes:

- Forward mode
  - The principal mode this package will focus on
  - Follow the chain rule from "inside" to "outside", i.e. in this example starting with $\frac{\partial v_{-1}}{\partial x_1} = 1$ and ending with $\frac{\partial v_5}{\partial x_1} = \frac{\partial y}{\partial x_1}$
  - Goes in the same direction as the evaluation of the function itself
  - Starts from the right-hand end of the chain rule expression above
  - Preferred when output space is much bigger than input space
- Reverse mode
  - Follow the chain rule from "outside" to "inside"
  - Goes in the reverse order as the evaluation of the function

- Starts from the left-hand side of the chain rule expression above
- Preferred when input space is much bigger than output space

The first column in the table is called the **Primal trace** and the second column is the **Forward tangent trace**. We can see how evaluating the intermediate steps (primals) as well as their derivatives (tangents) at the required point $(x_1, x_2) = (2, 4)$ starting from v_{-1} and $\dot{v_{-1}}$ eventually gives the required derivative evaluated at the correct point.

We can equivalently define a directional derivative operator $D_p y_i = \sum_{j=1}^{m} \frac{\partial y_i}{\partial x_j} p_j$, were we are projecting the gradient vector in the direction of a vector $\mathbf{p}$ with elements p_i. By selecting a p vector appropriate for the derivative we want, and letting the y_i be the intermediate variables in the trace, we achieve the same thing, except strictly now we are computing $\mathbf{J} \cdot \mathbf{p}$ were $\mathbf{J}$ is the Jacobian. It takes m passes to compute the full Jacobian.

2.4 Dual numbers

Forward mode autodiff can be achieved through as representing a function using dual numbers $v + \dot{v}\epsilon$ were $\epsilon \neq 0$ and $\epsilon^2 = 0$. The primal trace is carried in hte real part, and the tangent trace in the dual part.

This is a useful format because addition and multiplication of these dual numbers results in dual parts that respect differentiation rules w.r.t their real parts, e.g. $(v + \dot{v}\epsilon)(u + \dot{u}\epsilon) = (vu) + (v\dot{u} + \dot{v}u)\epsilon$. Most significantly, this formulation respects the chain rule, in that if we have $z_j = v_j + D_p v_j \epsilon$ then applying a function f gives $f(z_j) = f(v_j) + f'(v_j) D_p v_j$. So defining our dual number in this way we can progress through the intermediate steps of AD and the real and dual parts will give us the function and derivative evaluation naturally.

More detail can be found here.

2.5 Optimization

Optimization means selecting the best option from a set of possible options, according to some criterion. Finding the minimum or maximum of some function is a classic example of this, e.g. finding the (x, y) that maximize $f(x, y) = (x^2 + y^2) + 6$, giving $(x^*, y^*) = (0, 0)$. This is a trivially simple example, with a simple closed form expression for f. But in general we will not have such a form. For example in neural networks, we aim to minimize the loss function with respect to the network weights and biases, of which there can be billions forming a highly non-linear function. To tackle this we proceed in an iterative manner, taking small steps from some starting point in the function space and heading in the directions we believe will lead us to a maximum or minimum. We will explain two algorithms to achieve this next.

2.6 Gradient Descent

Assume we have a function $f(\mathbf{x})$ defined on an n-dimensional space, such that $\mathbf{x}$ has dimension n. Assuming f is differentiable, then the n-dimensional gradient of f always points in the direction of greatest increase of f (and similarly the negative of the gradient points in the direction of greatest decrease). So, for some starting point $\mathbf{x_0}$ we can move iteratively towards a minimum using the equation $\mathbf{x_{i+1}} = \mathbf{x_i} - \gamma f(\mathbf{x_i})$, where γ is a tunable parameter called the learning rate that determines the size of step we take at each iteration. Note that this only guarantees that we will find a local minima, not a global minima, unless additional constraints are imposed on f.

2.7 Adam

Adam optimization looks to improve upon some of the drawbacks of gradient descent, and is one of the most common algorithms used in training neural networks. First, rather than actually computing the gradient from the entire data set, it estimates it through random sampling. This is useful as taking the gradient of high-dimensional, complex functions can be intractable or very slow. Second, Adam uses exponential moving

averages of previous gradients to include a sense of "momentum" into the changes. Gradient descent suffers from the drawback that if it gets into areas of parameter space where the gradient is very small (e.g. at a minimum, or at a saddle/inflexion point) it tends to get stuck there as the changes become small. The "momentum" in Adam counteracts this. More details can be found here https://arxiv.org/abs/1412.6980.