MKNeuralNet

Morten Kals

February 4, 2017

Introduction

This is an overview of the mechanisms of the neural network framework MKNeuralNet, and the mathematics backing the algorithm. It was developed as a personal exercise to promote understanding of neural nets and the learning process.
I worked on this project the spring of 2016 using Swift 2 and an object-oriented approach. The functionality is based on the Apple Accelerate framework, developed for fast large scale mathematical computations. Matrix, vector and scalar operations are however wrapped in Swift code for readability, and speed-tests thus show this algorithm to suffer severe performance penalties.

Back Propagation

In general we can express the partial derivative of the cost function J with respect to each weight matrix W as

$$\frac{\partial J}{\partial W^{(n)}} = \left(a^{(n-1)}\right)^T * \delta^{(n)}$$

where

$$\delta^{(n)} = \left(\delta^{(n+1)} * \left(W^{(n+1)}\right)^T\right) f'(z^{(n)})$$

We examine the edge cases:

- Lower edge case, $n = 0$,

$$\frac{\partial J}{\partial W^{(0)}} = (X)^T * \delta^{(n)}$$

- Upper edge case, $n = i$,

$$\delta^{(i)} = (\overline{y} - y) f'(z^{(i)})$$

The algorithm is implemented recursively so as to perform both forward propagation and back propagation in one pass. This elegantly enables values of activities and activations to be directly captured rather than passed around between function calls.

The wrapper function serves to deal with the first edge case $n = 0$, passing in the input data as the zeroth layer activity. The wrapper also serves to filter the $deltaTerm$, vital to getting the recursive function working, but not meant to be exposed to the user of the function.

The nextLayer assignment is conditional and will end the recursive call-stack when the layer reached is the final layer of the network. It will assign the edge case values for $n = i$ to the nextLayer variable, and despite this not being strictly a good syntax as there really is no "next layer" to talk about, it suffices to convey the concept.

Automatic Differentiation, AD

Automatic differentiation (AD) is defined as a set of techniques used to numerically evaluate the derivative of a function specified by a computer program. The approach has great potential in that numeric derivatives of arbitrary order functions can be computed automatically, accurately and using a small constant factor more arithmetic operations than required for the evaluation of the original method. When working in arbitrary dimensions in multivariable and vector calculus, computations such as directional derivatives and gradients quickly become too complex to reasonably evaluate by hand. AD may thus be considered a natural and important extension of multivariable calculus to enable working with large and complex datasets. It provides an important backing for algorithms such as gradient descent, commonly used in applications of optimisation and machine learning as well as atmospheric chemistry, breast cancer biostatistical analysis and semiconductor device simulation to name but a few. In this epic, I will explore implementations of AD, focusing on the underlying mathematical notions on which its operation is based. I will start by comparing AD to other more

conventional methods of computer aided differentiation, discuss some practical details of implementing forward mode AD and conclude with a short discussion on the power of the main mathematical concept backing these algorithms - the chain rule.

Methods of computer aided differentiation

The classical methods for digital computation of gradients are numeric differentiation and symbolic differentiation. Both of these approaches have their disadvantages, and will in this discussion serve to highlight some of the main advantages to AD.
Numeric differentiation is based on the limit definition of the derivative, and is easy to code, but is subject to large rounding and truncation errors. Techniques developed to mitigate this, but these result in rapidly increasing programming complexity and thus slow performance.
Symbolic differentiation on the other hand, is the algorithmic encoding of the rules of differentiation used by human beings. It will, given a simple enough function, produce a legible symbolic description of the derivative, which can be very valuable in itself. This method does however suffer from rapidly increasing complexity. Consider, as an example, the product rule:

$$h'(x) = \frac{d}{dx} f(x)g(x) = f'(x)g(x) + f(x)g'(x)$$

When the function h consists of a product, then h and h' have some computations in common (namely f and g). Notice also that f and f' appear separately. When proceeding to symbolically differentiate f, any computation that appears in common between f and f' will have been duplicated. Deeply nested duplications of this sort will quickly arise as the function complexities increases, producing exponentially large symbolic expressions which will take exponentially long to evaluate.
This leads to the great advantage of AD. Simply put, instead of letting the symbolic expressions bulge, the symbolic differentiation is divided into layers and numerically evaluated after the computation of each layer. This concept is in fact so powerful, that far from encountering an exponential program complexity, the increase will follow a linear relationship. The cost of this potentially huge increase in speed is the loss of a symbolic result.
When performing AD, the chain rule is repeatedly applied to the expression at hand until arriving at partials simple enough to be easily evaluated. The order in which these operations are performed will have a large impact on performance, as it determines the number times the graph will have to be

traversed (hereafter referred to as number of sweeps) to obtain the full Jacobean matrix. The two extremes are known as forward mode (evaluates the partial derivative of a node with respect to an input variable) and reverse mode(evaluates the partial derivative of each node with respect an output variable). Determining the minimum number of arithmetic operations required to traverse the graph is known as the optimal Jacobean accumulation problem. Solving it would give the best path for traversing the graph. The optimal Jacobean accumulation problem is however NP-complete (in short, meaning it requires far more computational power than performing the AD itself).

Forward Mode

Consider the function

$$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$$

In forward mode, the partial derivative with respect to one input variable is calculated per sweep, producing a m dimensional vector function $\frac{\partial \mathbf{f}}{\partial x_i}$. Computing the Jacobean

$$J = \frac{d\mathbf{f}}{dx} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \cdots \frac{\partial \mathbf{f}}{\partial x_n}\right]$$

of this function would thus require n sweeps, resulting in a n by m matrix. Forward mode is thus the preferred choice when $n \ll m$.
Forward mode is very natural to implement, as the flow of the derivative information coincides with the order of evaluation. In a algorithmic encoding, each variable

$$x$$

would simply be augmented with its derivative stored as a numerical value (not a symbolic expression, this to maintain linear evaluation time), see section on dual numbers. The derivatives are then computed in synchronisation with the evaluation steps and combined with other derivatives via the chain rule.

Reverse Mode

As the name suggests, reverse mode AD will traverse the graph in the opposite direction of evaluation. Calculating the Jacobean of the function $\mathbf{f}$ would thus require m weeps, each producing an n dimensional vector output. This would thus be the preferred method when $n \gg m$.

Table 1: Example Computation

Variable	Value	Forward mode derivative $w_i{}'$	Reverse mode derivative $\overline{w}_i$
w_1	x_1	1 and 0 (seed)	$\overline{w}_3 + \overline{w}_5 w_2$
w_2	x_2	0 and 1 (seed)	$\overline{w}_5 w_1$
w_3	$w_1 - 2$	$w_1{}'$	$3\overline{w}_4 {w_3}^2$
w_4	${w_3}^3$	$3{w_3}^2 w_3{}'$	$\overline{w}_7 w_5$
w_5	$w_1 w_2$	$w_1{}' w_2 + w_1 w_2{}'$	$-\overline{w}_6 \sin(w_5)$
w_6	$\cos(w_5)$	$-\sin(w_5) w_5{}'$	$\overline{w}_7 w_4$
w_7	$w_5 w_6$	$w_5{}' w_6 + w_5 w_6{}'$	1 (seed)

Reverse mode is harder to encode as the derivative information flows opposite to the direction of evaluation.
For many applications, the gradient of functions
$f : \mathbb{R}^n \to \mathbb{R}$
is much more valuable than the directional derivatives produced by forward mode. The gradients enable for example multidimensional optimisation methods. and multi-parameter sensitivity analyses. This has resulted in backward propagation being the most popular method of performing AD.

Example evaluation procedure

In order to better underline the difference between forward and reverse mode AD, I have included the respective evaluations of an example function $f : \mathbb{R}^2 \to \mathbb{R}$. Consider

$$y = f(x_1, x_2) = (x_1 - 2)^3 \cos(x_1 x_2)$$

We beak this up into a computational graph to underline the components of the computation. Each node w_i consists of a mathematical operation.
In the table below, the operations performed when traversing this graph is outlined. In column three, the steps of forward mode AD are outlined. Column three shows the equivalent procedure for reverse mode. Forward mode evaluates the partial derivative of a node with respect to a input variable x_i , ${w_j}' = \frac{\partial w_i}{\partial x_i}$. Reverse mode evaluates the derivative of a node with respect to the output variable y , $\overline{w}_j = \frac{\partial y}{\partial w_j}$.
In order to calculate the partial derivative with respect to x_1 and x_2 in forward mode, we need to perform two passes seeding ${x_1}'$ and ${x_2}'$ with one alternately. Reverse mode would only require one pass to compute both partial derivatives.

Form the table above, by choosing the final expression for the derivatives (w_7' for forward mode and $\overline{w}_1$ for reverse mode) and substituting in the other expressions accordingly, the symbolic antiderivatives is obtained (omitted due to space constraint). When AD is implemented in an algorithm, the numerical evaluations will be performed at every layer of the computational graph in order to keep the evaluation-time linear (as opposed to the exponential plow-up seen with symbolic differentiation). Forward mode AD can thus be thought of as a way to organise symbolic differentiation to retain the sharing of intermediate results between the main computation and the derivative, which coincidentally also extends nicely to non-numeric programming constructs.

Dual numbers in forward mode

One common way of implementing forward mode AD is by the use of the mathematical notation of dual numbers. We define dual numbers as formal truncated Taylor series of the form $x+\epsilon x'$, where ϵ is called the infinitesimal. Arithmetic with dual numbers works much like arithmetic with complex numbers, with the difference that the infinitesimal is defined as $\epsilon^2 = 0$. Any non-dual number y is interpreted as $y + \epsilon \cdot 0$. Thus,

$$(x + \epsilon x') + (y + \epsilon y') = (x + y) + \epsilon(x' + y')$$

$$(x + \epsilon x')(y + \epsilon y') = xy + \epsilon(xy' + x'y)$$

Differential calculus with dual numbers works as follows:

$$f(x + \epsilon x') = f(x) + \epsilon f'(x)x'$$

This can be proved to work using arithmetic when f is a polynomial of arbitrary length, but I have had to remove this calculation due to lack of space. This result may be further generalised by considering the Taylor series of any continuos function f:

$$f(x + \epsilon x') = \sum_{n=0}^{\infty} \frac{f^{(n)} x^n \epsilon^n}{n!} = f(x) + \epsilon f'(x)x'$$

From a computer science perspective, these dual numbers may be used as data-structures for carrying the derivative information around together with the undifferentiated answer. To perform forward mode AD, the chain rule is repeatedly applied.

$$f(g(x+\epsilon x')) = f(g(x)+\epsilon g'(x)x') = f(g(x)) + \epsilon f'(g(x))g'(x)x'$$

where the coefficient of ϵ on the right hand side is exactly the derivative of the composition of f and g. We implement the primitive operations to respect invariance, meaning that all compositions of them will too. The derivative of functions by thus be extracted by evaluating the function this nonstandard way on an initial input with a 1 coefficient of ϵ. This property also generalises naturally to taking directional derivatives of $f : \mathbb{R}^n \to \mathbb{R}^m$. This is the outline of a completely mechanical way of evaluating directional derivatives at a point. It works for any function composed of the primitive operations that we have extended to dual numbers. It turns every arithmetic operation into a fixed number of new arithmetic operations, so the computational cost goes up by a small constant factor. This, combined with the natural extension from evaluating functions to evaluating arbitrary program constructs makes it an ideal approach for computer generated numeric derivatives at a point. In general, we may as well expect the accuracy to be comparable to the accuracy of the original function as the loss in accuracy due to truncation-errors during computation will be similar for both as they are evaluated in the same way.

For computation of numerical derivatives with multivariable functions, automatic differentiation is a good approach to get both performance and accuracy. Typically, forward mode would be used to compute directional derivatives whereas reverse mode would produce directional gradients. The algorithm finds its solid foundations in the chain rule, whose power I am just starting to realise. A computer program will, no matter how complicated, execute as a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division etc.) and elementary functions (exponentials, logarithms, trigonometric functions etc.) The repeated application of the chain rule to these operations will thus result in the ability to compute derivatives of arbitrary order. Using a concept called edge pushing, automatic differentiation is as an example used to calculate the Hessian matrix in applications of image processing and computer vision.

Structure

Below is an overview of variables used.

Symbol	Term
X	input data
Y	target data
$\overline{Y}$	output data
W	weights
z	activations
a	activities
f	activation function
J	cost function

Sources

- http://alexey.radul.name/ideas/2013/introduction-to-automatic-differentiation/
- http://www.pvv.ntnu.no/~berland/resources/autodiff-triallecture.pdf
- https://en.wikipedia.org/wiki/Automatic_differentiation
- https://en.wikipedia.org/wiki/Hessian_matrix