A Project Report

On

Algorithmic Differentiation

BY

C. S. Adityakrishna

2013A7PS387H

Under the supervision of

Dr. N. Anil

MATH F367 : DESIGN ORIENTED PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(APRIL 2016)

ACKNOWLEDGMENTS

I would like to thank Dr N. Anil for constant guidance and support. I would also like to thank the Department of Mathematics for providing the opportunity to pursue this project under Dr Anil's supervision.

Birla Institute of Technology and Science-Pilani,

Hyderabad Campus

Certificate

This is to certify that the project report entitled "**Algorithmic Differentiation"** submitted by Mr/Ms. Chivukula Satya Adityakrishna (ID No. 2013A7PS387H) in fulfillment of the requirements of the course MATH F376, Design Oriented Project Course, embodies the work done by him under my supervision and guidance.

Date: 28/4/16

(Dr N. Anil)

BITS- Pilani, Hyderabad Campus

ABSTRACT

Algorithmic Differentiation (herein referred to as AD) is a novel approach to computing derivatives of functions. The concept of AD is by no means recent[1] . Firstly, we disucss some of the traditional approaches, including Symbolic Differentiation, Method of Finite Differences (Numerical Differentiation) and CTSE, along with their respective advantages and disadvantages. Further, we provide rationale as to why Algorithmic Differentiation remains the best of all these approaches. We then delve into the black box that is AD and highlight two of the fundamental methods of applying AD, i.e, the Forward Mode and the Reverse Mode. We discuss properties of both these methods and evaluate the crietria under which one mode operates better than the other. Further we analyse the applicability of AD to Neural Networks (further referred to as NN). Training of NNs is fundamentally a gradient based optimization of a cost function. We reflect on the various classes of Network architecture and provide improvements to the Computational Graph that would be generated by a general purpose AD tool. Armed with this information we can contrast with a non-AD approach to the problem domain.

CONTENTS

TRADITIONAL APPROACHES

Symbolic Differentiation

In this method, analytical derivates are obtained by deriving the function expression as a person would. The program reads and understands the function expression and produces corresponding derivative code. This approach breaks down when the function expression requies the usage of loops and branches.

$$\frac{\partial l}{\partial \alpha_i} = \frac{l(\alpha_i + \Delta\alpha_i) - l(\alpha_i - \Delta\alpha_i)}{2\Delta\alpha_i} \cdot O(\Delta\alpha_i)^2$$

Method of Finite Differences

Simplistically, this method involves calculating the difference coefficients of functions around given point h, by calculating the limit of h tending to zero. While there are several numerical analysis improvements to this basic idea, this approach is prone to several types of calculation and computational(precision) errors. This method also involves high computational cost as it requires a function evaluation for every dependant variable in the system.

Complex Taylor Series Expansion

The Complex Taylor Series Expansion method uses an imaginary perturbation of iΔa. The Taylor Series expansion of the perturbed function is used to determine the derivative. This method removes the issue of cancellation errors as seen while calculating Numerical Derivatives, but is still computationaly very expensive.

ALGORITHMIC DIFFERENTIATION

AD works on the principle of Chain Rule at the operator level. This is similar to symbolic differentiation, however, the derivative function code is not explicitly generated by the AD tool. The AD tool will interpret the input program as a sequence of elementary operations along with some intrinsic functions. The tool will then output a derivative program that is obtained by applying chain rule sequentially to each of the elementary operations. Methods of implementing (in code) and complexity of these transformations are discussed in later sections.

Consider the function,

$$f = f_1(f_2(f_3 \cdots f_n(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial f_1}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_n} \cdot \frac{\partial f_n}{\partial x}$$

In Forward Mode, the chain rule propagates from right to left. i.e. the inner most function to outermost. In Reverse Mode, the direction of propagation is from left to right. However it must be noted that Forward and Reverse Modes are just two (extreme) ways of traversing the chain rule. The problem of computing a full Jacobian of f: R n → R m with a minimum number of arithmetic operations is known as the *optimal Jacobian accumulation (OJA)* problem, which is NP-complete under mild assumptions. This is central to our reasoning that AD Tools can be tailored for the purposes of training arbitrarily designed Neural Networks by virtue of the basic architecture that is common to all networks.

The propagation of the chain rule is best illustrated via a computational graph. There exists multiple possible computational graphs for a single given expression (function) by rearranging the associativity of the operations or by treating repeated sub-expressions as unique. We demonstrate both Modes with a toy example:

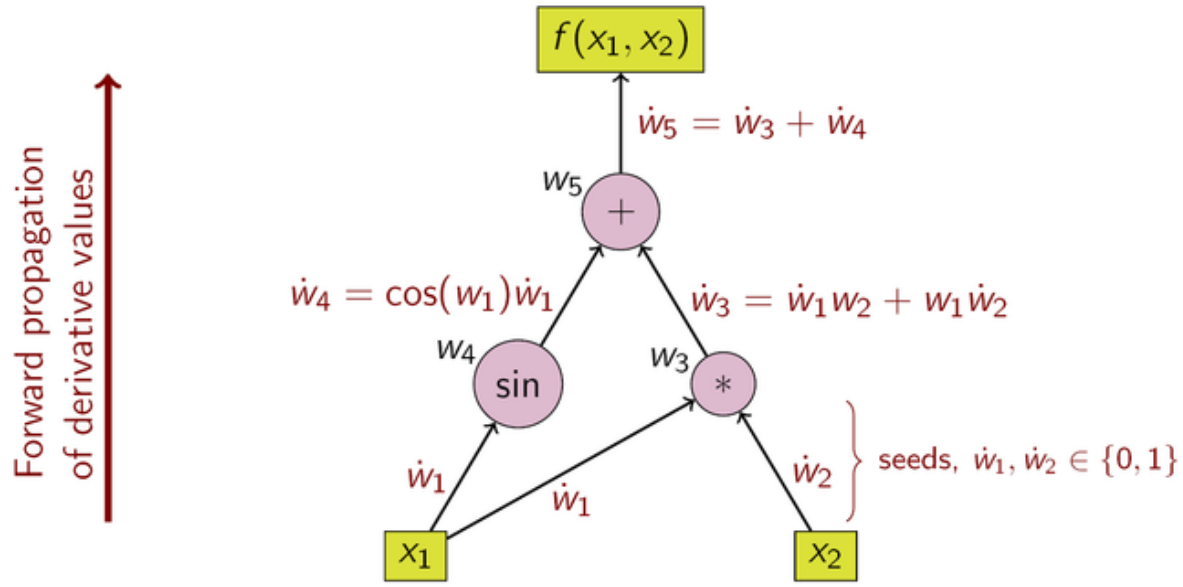
Consider the function,

We need to compute,

$$\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}$$

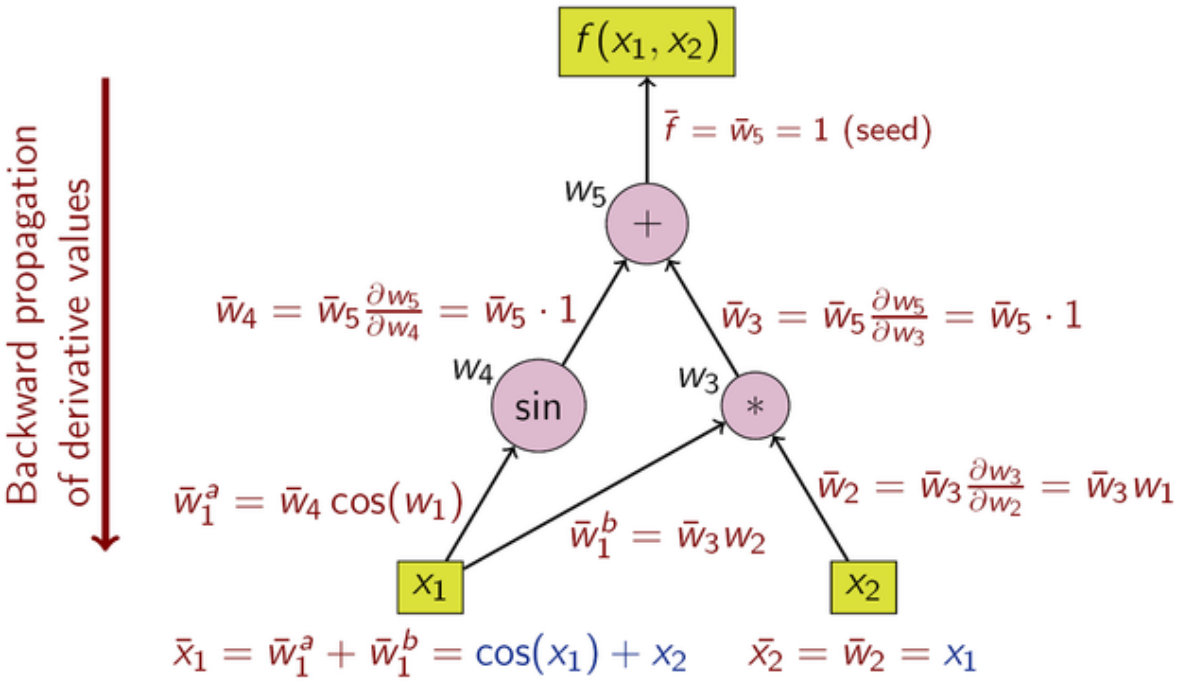
Forward Mode

In Forward Mode, a forward (evaluative pass) is required for each independant variable. For each such variable, it is initialised to 1 and the remaining variables to 0.



Reverse Mode

In Reverse Mode, a single forward evaluation is performed after which a backward pass provides the derivative of the output function with respect to all the input variables. While this seems magical, it is at the cost of high memory requirement. Every intermediate variable must be stored during the forward pass for purpose of evaluating the derivative.



Comparison of Modes

For any function $f:\mathbb{R}^n \to \mathbb{R}^m$

Choosing the optimal technique for computing the derivative of requires consideration of computational complexity and memory requirements. With these criteria in mind a simple thumb rule comes to surface.

Forward Mode requires **n** passes to evaluate the derivative whereas Reverse Mode requires **m** passes. Therefore, for m >> n, Reverse Mode is preferred and Forward Mode for n >> m. It sohuld also noted that Reverse Mode *may* require high memory storage as all intermediate variable in the forward pass need to be stored on a tape. Some methods like checkpointing help reduce the memory requirements for the process but Reverse Mode is still memory intensive.

In terms of complexity, we can place some bounds on the running time of the appended/modified function evaluation code. Reverse method provides a complete set of derivatives for a cost of between one and four function evaluations. As mentioned earlier, determining the OJA is an NP-complete problem.

Computational Graphs

Consider now the (directed) *Computation Graph* that represents the structure of the program to evaluate $f(X)$, $X \in \mathbb{R}^n$

where V consists of three disjoint sets . The vertices in V_x represent the input variables. The vertices in V_y represent the internal variable with an in degree of either one or two. The vertices each correspond to an intermediate variable and an basic/elementary function that can differentiated at an operator level as these functions are unary/binary in nature. Further, V_z is the set of vertices that represents the output variables, i.e. they have an out degree of zero.

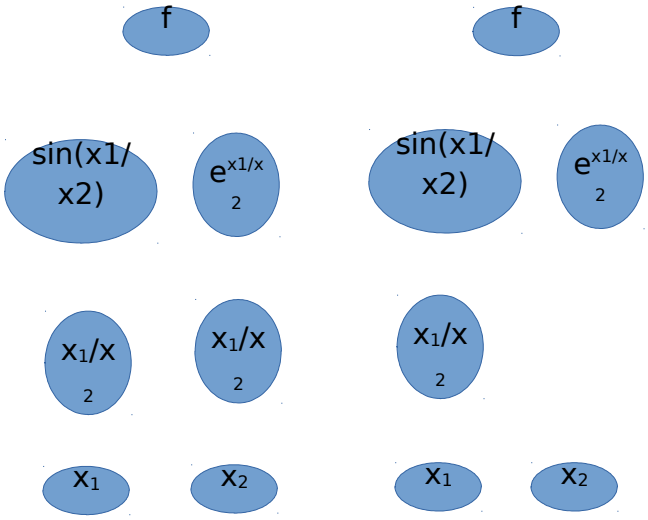
This rigorous definition allows application of several graph based optimisations to the function evaluations and reconciles the forward/reverse mode definitions illiustrated in the previous sections. Theseoptimisations are briefly discussed in the next section.

Graph Optimisation

It is possible to illiustrate the concept of Graph-based Optimisation of function evaluation using a simple example.

Let $f(x_1, x_2) = \sin\left(\frac{x_1}{x_2}\right) + e^{x_1/x_2}$

We can draw to unique graphs for the same function. Notice from the definition of computational graph in the previous section, the number of elementary operations required to evaluate the specified function is proportional to the $|V|, |E|$.



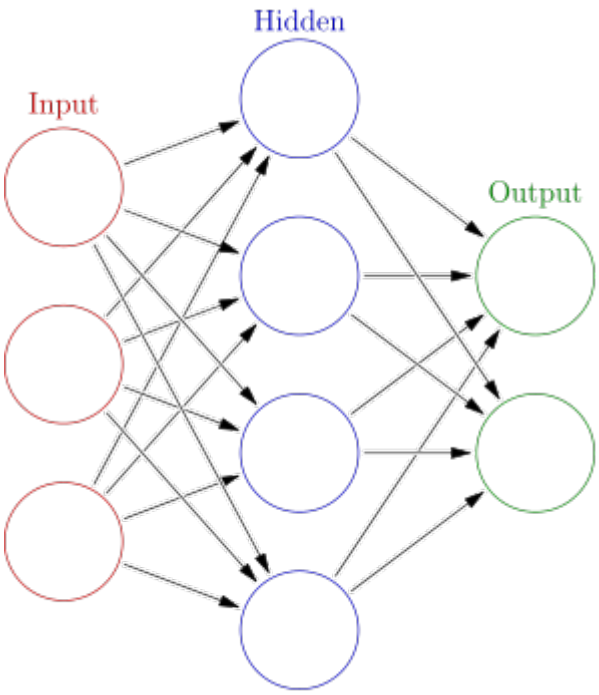
Optimisations performed on the Computational Graph, are performed by **Matrix Chaining** and **Vertex Elimination**. These concepts are discussed in later section with regard to Neural Networks.

NEURAL NETWORKS

Neural Networks (further referred to as NN) simulate a network of neurons passing “messages” in the brain. While the concept has existed for over 3 decades. This computing paradigm has now come to the fore due to advances in computing techniques and novel learning techniques. It has varied application in the fiel of Machine Learning and Artificial Intelligence. AD is another tool that augments the expressivity of this technique.

Structure and Representation

The word *network* in a NN refers to the inter–connections between the neurons in the different layers of each system. An example system has three layers. The first layer has input neurons which send data via synapses to the second layer of neurons, and then via more synapses to the third layer of output neurons. More complex systems will have more layers of neurons, some having increased layers of input neurons and output neurons. The synapses store parameters called "weights" that manipulate the data in the calculations.



Operations

The NN, is modelled as a function. $f : X \to Y$, s.t. $X \in \mathbb{R}^n$ and $Y \in \mathbb{R}^m$. Training this network for pattern recognition involves using an arbitrary error function as required by the problem domain. Stochastic Gradient Descent can then be used to minimze this error function. For the simple 3-layer network shown above, using L_2 error, the function can be written as,

$f_l(X) = \sum_D \sum_j \sigma\left(\sum_i w_{ij} x_i\right)$ for each layer *l* in the network.

Here, sigma is an activation function, w is the weight of each synapse and x is an input data feature.

$$E = \sum_D \left(f(X) - y_d\right)^2$$

$$E = \left(\sum_D \left(\sum_j \sigma\left(\sum_i w_{ij} x_i\right)\right) - y_d\right)^2$$

Therefore, to compute accurate synapse weights we need to find gradient of E w.r.t. to W.

$\frac{\partial E}{\partial w_{ij}}$ For all i, j in each layer.

This gradient information is now fed to a Stochastic Gradient Descent program to optimize the cost function.

Backpropagation algorithm was a crucial breakthrough for the implementation of NN. It is a function level application of the concept of AD. It requires one feedforward pass followed by a backward accumulation of gradient information for each forward pass.

We shall know look at the multiple classes of NN and analyse the applicability of AD to the netoworks.

Classes of Neural Networks

Feedforward NN

These networks have similar architecture as the diagram above, every layer has fixed number of nodes and each layer is fully connected to its adjacent layers. Backpropagation algorithm provides simple derivatives for these type of networks.

Recurrent NN

Recurrent NN include several subclasses of networks. These type of networks exploit the power of AD significantly as backpropagation cannot provide gradients for such networks, This is due to the Acyclic nature of the network architecture. Data Flow in an RNN can be linked back to a node in the previous layer, however this link will correspond to a weight different from the weight possesed by the forward pass link. Using AD for RNN has become the norm for training RNN as hand computing gradients for each network is a tedious task.

Convolutional NN

CNNs work on an intuitive understanding of NN. Each layer in the CNN corresponds to a higher level of abstraction among features in the data. The basic concept of CNN involves weight sharing of the synaptic links between layers to provide spatial independence of the feature detector. This weight sharing allows for a convenient trace evaluation and taping mechanism discussed in further section.

IMPLEMENTATION

Source Code Transformation

AD Tool would pre-process the function written in specified language. The tool would then append/modify the code accordingly and the forward to the compiler. This allows the compiler to perform compile time optimizations as the compiler is unaware of the intermediate pre-processing.

Operator Overloading

Overloaded behaviour for every elemetary operation and intrinsic function is fed to the compiler along with the original code. The derivative code is interleaved with the function evaluation code. This approach requires redefinition/restructuring of some basic data types to allow for this overloaded behaviour.

Rationale

For purposes of NN argument, Operator Overloading suits the problem best for the below reasons:

1. The existence of activation functions to be inculcated a an elementary function, for example, the sigmoid function. The derivative of which can be represented in terms of the function itself.
2. Source Code transformation would require additional compiler constructs making design of the AD tool complicated.
3. Operator Overloading allows optimization of the Computational Graph as the Graph can be constructed on the fly with each operator adding a sub tree to the Graph.

Code Intricacies

Variables

Input and output variables need to redefined so as for the AD tool to recognise the dependant and indepedant variables to seed before computing gradient. Adjoints of the corresponding variables must be initialised.

Memory

The evaluation of gradients is accumulative in nature, therefore allowing for easy taping mechanism and efficient memory usage. The trace of every forward pass need not be taped. The evaluation sweep can be cleared after every forward pass as this information is not required for subsequent accumulations of the gradient.

Three Address Code

Using a standard bottom up parser, a three address code version of function evaluation is generated. Operator overloading can allow conversion of this syntax tree into a Directed Acyclic Graph. Matrix Chaining and Vertex Elimination can now be applied to this generated DAG.

SCOPE AND LIMITATIONS

Second Order Derivatives

Neural Netoworks are also trained using Second derivative information via the Hessian Free Optimisation algorithm. This method has been shown to provide greater accuracy in weights generated that while using only Stochastic Gradient Descent.

Activation Functions

Experimentation with multiple activation functions for every layer is possible as we are now not concerned about the derivative of these functions, the AD tool simply includes a overhead in computation time. This allows for much faster protoytyping of a variety of networks as the programmer does not need to hand calculate or even hand code the gradients.

Limitations

The memory usage of the AD tool can be of an issue if dimensionality of the input data is significantly large, in such cases hand coding gradients is unavoidable.

CONCLUSION

An AD tool tailored for the purpose of training NNs is beneficial as it can reduce training time by a significant factor. An AD tool has the added benefit of reducing prototyping time of the programmer allowing for smaller, legible code. The techniques discussed above provide gurantees on the reduction of computational complexity and memory efficiency over a general purpose AD tool.

REFERENCES

[]Recent Advances in Algorithmic Differentiation - Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, Andrea Walther

Evaluating Derivatives – Andreas Griewank

Automatic differentiation in machine learning: a survey – Baydin et al.

https://en.wikipedia.org/wiki/Automatic_differentiation

https://en.wikipedia.org/wiki/Artificial_neural_network