

Introduction to Neural Networks and Deep Learning

Backpropagation and Automatic Differentiation

Andres Mendez-Vazquez

June 22, 2020

Outline

1 Backpropagation

- Introduction
- Derivatives of Network Functions
- Function Composition, Weights and Addition
- The Backpropagation Algorithm Works
- Moving everything to Tensors

2 Automatic Differentiation

- Introduction
- Advantages of Automatic Differentiation
 - Avoiding Truncation Errors
 - Differences with Symbolic Differentiation
 - Difference Quotients May be Useful
 - RNN Example
- A Simple Example
- The Forward and Reverse Mode
 - Forward propagation of Tangents
 - Forward Mode of a ML Perceptron
 - Complexity of the Forward Procedure
 - The Reverse Mode
 - Dual Process in Reverse Process
 - Incremental Adjoint Recursion
 - Example
- What Method to Use Forward or Reverse Mode?

3 Basic Implementation of Automatic Differentiation

- Source Transformation and Overloading
- Building the Computational Graph
- Memory Structures
- Way More...

4 Conclusions

- The Problem of Backpropagation

A Remarkable Revenant

This algorithm has been used by many communities

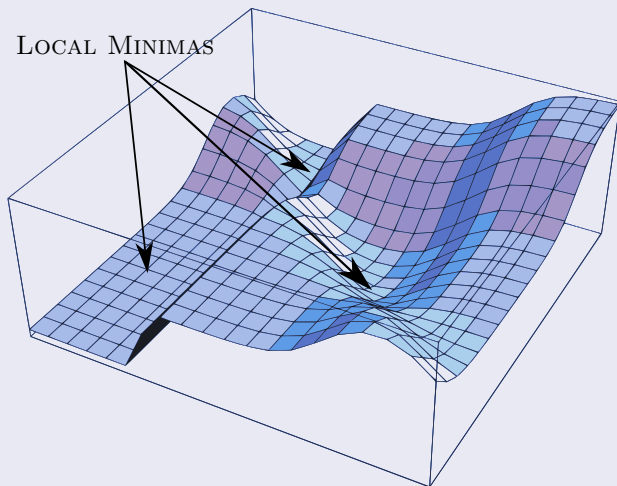
- Discovered and rediscovered, until 1985 it reached the AI community [1]

Basically

- The Basis of the modern neural networks

One Big Problem, a lot of Local Minimums

A Lot of Them!!!



This is due to the fact that

Yes, we have a convex function

$$\frac{1}{2} (z_i - t_i)^2$$

With an intermediate non-linear activation function

$$z_i = f \left(\sum_{j=1}^d w_{ij} y_j \right)$$

Making the surface to be searched for the optimum

- A non linear function map from \mathbb{R}^d to \mathbb{R}^m

Recall The Learning Problem

Neural Networks

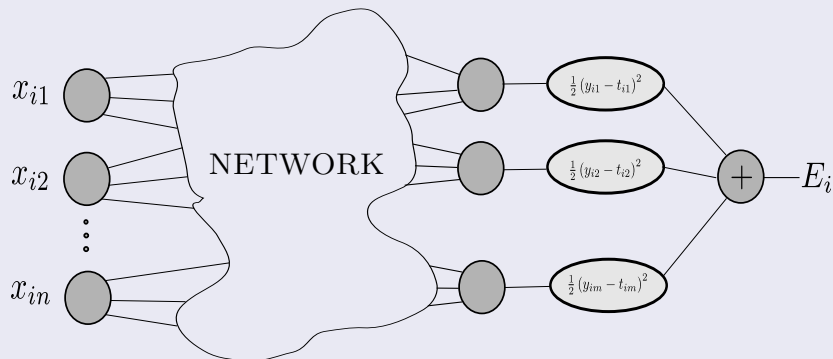
- You can see the network as a computational graph...
 - ▶ Transmitting information from node to node...

Therefore, the network

- It is a particular implementation of a composite function from input space to output space.

Extended Network

The computation of the error by the network [2]



Thus

The network can calculate the total error

$$E = \sum_{i=1}^N E_i$$

Therefore, the network can be updated using

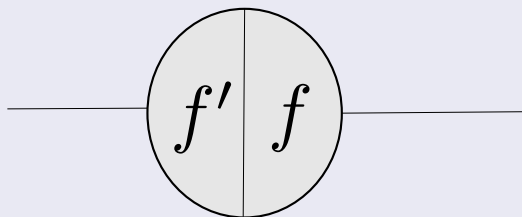
$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_l} \right)$$
$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i} \text{ for } i = 1, \dots, l$$

Now, if we forget everything about learning

Given that the network is a complex composition of functions

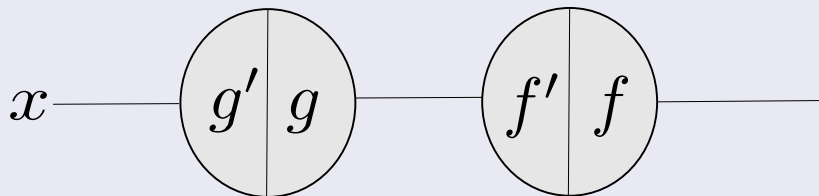
$$E = f_1 \circ f_2 \circ \cdots \circ f_K$$

Now, each node has a left and right side



Furthermore

Separation of integration and activation function

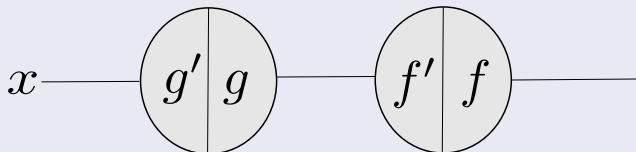


Then, we can use this notation to build the forward/backward steps

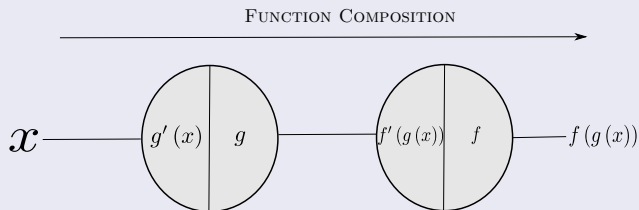
- Actually the basis for automatic differentiation

First, we have

The sequence of derivatives

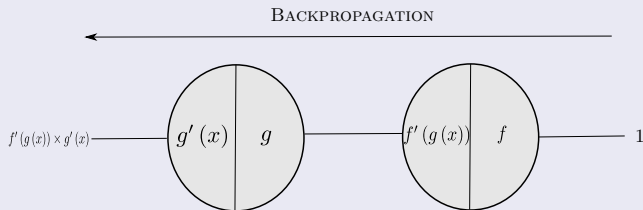


Then, we can do the forward step getting the function compositions



Now, Backpropagation

Here the interesting part, you can collect such information

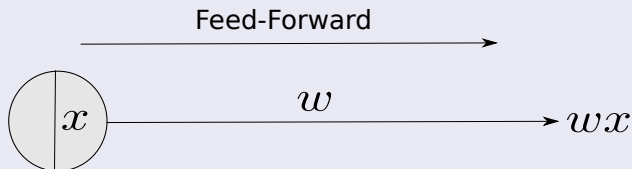


Now, what else?

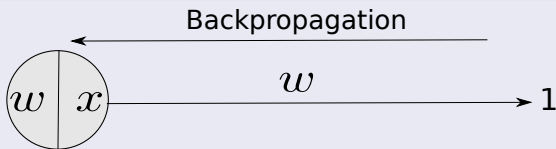
- The aggregation of functions toward the activation functions!!!

We add an extra caveat to the graph representation

A weight into the graph

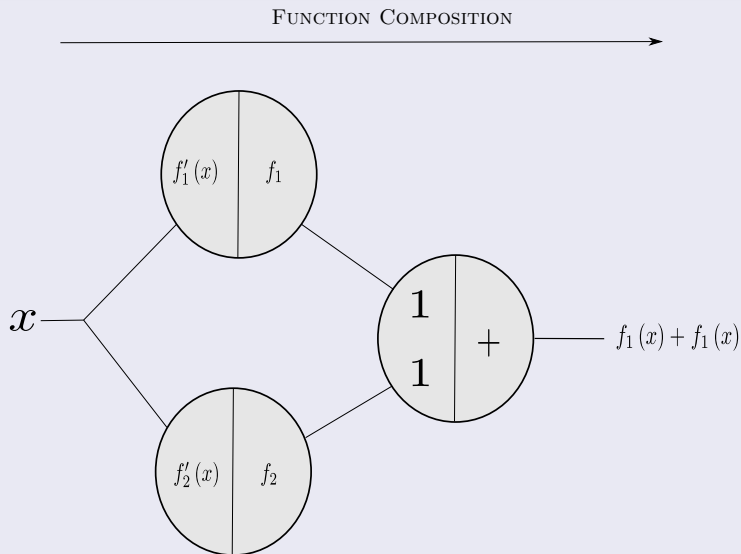


We have the backward process



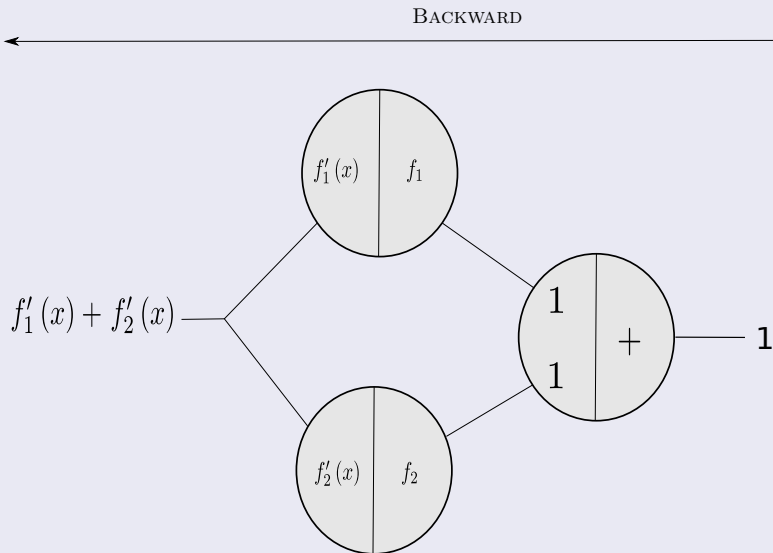
Function Addition

We have the forward step



Then

At the Backward Step, we have



Backpropagation Algorithm

Consider a network with a single input and a network function F

- The Derivative $F'(x)$ is computed in two phases.
 - ① Feed-forward:
 - ★ The input x is fed into the network.
 - ★ The primitive functions at the nodes and their derivatives are evaluated at each node.
 - ★ The derivatives are stored at the left side of the node.
 - ② Backpropagation:
 - ★ The constant 1 is fed into the output unit and the network is run backwards.
 - ★ Incoming information to a node is added and the result is multiplied by the value stored in the left part of the unit.
 - ★ The result is transmitted to the left of the unit.
 - ★ The result collected at the input unit is the derivative of the network function with respect to x .

Proof of Correctness about the derivatives

Proposition

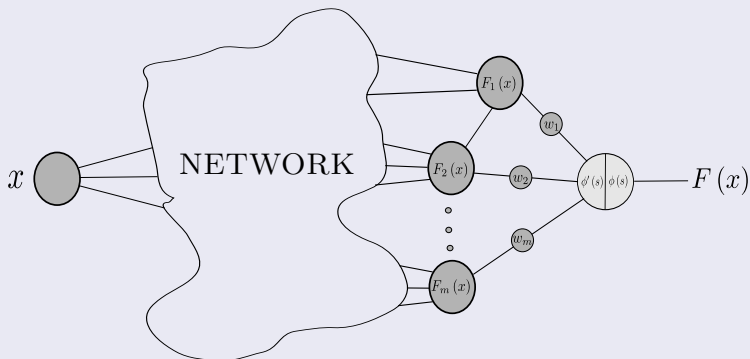
- The Backpropagation algorithm computes the derivative of the network function F with respect to the input x correctly.

Proof

- By induction assume that the algorithm works with n or fewer nodes

Consider

The following network with $n + 1$ nodes



Thus

We have that

$$F(x) = \phi(w_1 F_1(x) + w_2 F_2(x) + \cdots + w_m F_m(x))$$

We have that the derivative

$$F'(x) = \phi'(s) [w_1 F_1'(x) + w_2 F_2'(x) + \cdots + w_m F_m'(x)]$$

- With $s = w_1 F_1(x) + w_2 F_2(x) + \cdots + w_m F_m(x)$

Now, we use induction

The subgraph of the main graph which contains all the nodes to $F_1(x)$

- Thus, by induction, we can calculate the derivative of $F_1(x)$ by introducing a 1 into the last unit and doing backpropagation

The same happens to all the other units

- Now if instead of multiplying by 1 we introduce $\phi'(s)$ and multiply by w_j , we get

$$w_j F_j'(x) \phi'(s)$$

This can be accomplished by

- Introducing a 1 into the output unit, multiplying by the stored value $\phi'(s)$ and distributing the result to the m units through edge weight nodes.

Basically, we get the derivative

We get then

$$\phi'(s) [w_1 F'_1(x) + w_2 F'_2(x) + \dots + w_m F'_m(x)]$$

Basically the networks is run backward

$$F'(x) = \phi'(s) [w_1 F'_1(x) + w_2 F'_2(x) + \dots + w_m F'_m(x)]$$

The algorithms works for $n + 1$

- QED

Why not using matrices to process all the individual parts?

Imagine the following, a simple idea

$$X = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{pmatrix}$$

We know the fields are created in input to hidden as

$$g(X) = XW = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{pmatrix} \begin{pmatrix} \mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_d \end{pmatrix}$$

Where

We have these construct $g_{ij}(\mathbf{x}_i^T) = \mathbf{x}_i^T \mathbf{w}_j$

$$g(X) = \begin{pmatrix} g_{11}(\mathbf{x}_1^T) & g_{12}(\mathbf{x}_1^T) & \cdots & g_{1d}(\mathbf{x}_1^T) \\ g_{21}(\mathbf{x}_2^T) & g_{22}(\mathbf{x}_2^T) & \cdots & g_{2d}(\mathbf{x}_2^T) \\ \vdots & \vdots & \ddots & \vdots \\ g_{N1}(\mathbf{x}_N^T) & g_{N2}(\mathbf{x}_N^T) & \cdots & g_{Nd}(\mathbf{x}_N^T) \end{pmatrix}$$

Then

We have that the $f_{ij}(x) = \frac{1}{1+\exp\{-x\}}$

$$f(g(X)) = \begin{pmatrix} f_{11}(g_{11}(\mathbf{x}_1^T)) & f(g_{12}(\mathbf{x}_1^T)) & \cdots & f(g_{1d}(\mathbf{x}_1^T)) \\ f(g_{21}(\mathbf{x}_2^T)) & f(g_{22}(\mathbf{x}_2^T)) & \cdots & f(g_{2d}(\mathbf{x}_2^T)) \\ \vdots & \vdots & \ddots & \vdots \\ f(g_{N1}(\mathbf{x}_N^T)) & f(g_{N2}(\mathbf{x}_N^T)) & \cdots & f(g_{Nd}(\mathbf{x}_N^T)) \end{pmatrix}$$

Finally, we can do the following modification when forward

Then the matrix can be extended

$$g'(X) | g(X) = \begin{pmatrix} \frac{dg_{11}(\mathbf{x}_1^T)}{d\mathbf{w}_1} | \mathbf{x}_1^T \mathbf{w}_1 & \frac{dg_{12}(\mathbf{x}_1^T)}{d\mathbf{w}_2} | \mathbf{x}_1^T \mathbf{w}_2 & \dots & \frac{dg_{1d}(\mathbf{x}_1^T)}{d\mathbf{w}_d} | \mathbf{x}_1^T \mathbf{w}_d \\ \frac{dg_{21}(\mathbf{x}_2^T)}{d\mathbf{w}_1} | \mathbf{x}_2^T \mathbf{w}_1 & \frac{dg_{22}(\mathbf{x}_2^T)}{d\mathbf{w}_2} | \mathbf{x}_2^T \mathbf{w}_2 & \dots & \frac{dg_{2d}(\mathbf{x}_2^T)}{d\mathbf{w}_d} | \mathbf{x}_2^T \mathbf{w}_d \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dg_{N1}(\mathbf{x}_N^T)}{d\mathbf{w}_1} | \mathbf{x}_N^T \mathbf{w}_1 & \frac{dg_{N2}(\mathbf{x}_N^T)}{d\mathbf{w}_2} | \mathbf{x}_N^T \mathbf{w}_2 & \dots & \frac{dg_{Nd}(\mathbf{x}_N^T)}{d\mathbf{w}_d} | \mathbf{x}_N^T \mathbf{w}_d \end{pmatrix}$$

Finally, we have

The next function $f'(g(X)) | f(g(X)) =$

$$\left(\begin{array}{ccc} \frac{df_{11}(x)}{dx} (g_{11}(\mathbf{x}_1^T)) | f_{11}(g_{11}(\mathbf{x}_1^T)) & \cdots & \frac{df_{1d}(x)}{dx} (g_{1d}(\mathbf{x}_1^T)) | f(g_{1h}(\mathbf{x}_1^T)) \\ \frac{df_{21}(x)}{dx} (g_{21}(\mathbf{x}_1^T)) | f(g_{21}(\mathbf{x}_2^T)) & \cdots & \frac{df_{2d}(x)}{dx} (g_{2d}(\mathbf{x}_1^T)) | f(g_{2h}(\mathbf{x}_2^T)) \\ & \vdots & \vdots \\ \frac{df_{N1}(x)}{dx} (g_{N1}(\mathbf{x}_1^T)) | f(g_{N1}(\mathbf{x}_N^T)) & \cdots & \frac{df_{Nd}(x)}{dx} (g_{Nd}(\mathbf{x}_1^T)) | f(g_{Nh}(\mathbf{x}_N^T)) \end{array} \right)$$

Using the Hadamard Product

We have for the backpropagation

$$f'(g(X)) \circ g'(X)$$

In particular for a position ij

$$\frac{dg_{ij}(\mathbf{x}_i^T)}{d\mathbf{w}_j} \times \frac{df_{ij}(x)}{dx} (g_{ij}(\mathbf{x}_i^T)) = \frac{df_{ij}(x)}{dx} (g_{ij}(\mathbf{x}_i^T)) \times \begin{pmatrix} x_{1i} \\ x_{2i} \\ \vdots \\ x_{di} \end{pmatrix}$$

Then using a vertical sum

We get the change that is imposed into the possible vector w_j

$$\text{sum} (f' (g (X)) \circ g' (X), \text{axis} = 0) = \left\{ \sum_{i=1}^N \frac{dg_{ij} (x_i^T)}{dw_j} \times \frac{df_{ij} (x)}{dx} (g_{ij} (x_i^T)) \right\}_{j=1}^h$$

Now a Historical Perspective

The idea of a Graph Structure was proposed by Raul Rojas

- “Neural Networks - A Systematic Introduction” by Raul Rojas in **1996...**

TensorFlow was initially released in **November 9, 2015**

- Originally an inception of the project “Google Brain” (Circa 2011)
- So TensorFlow started around 2012-2013 with internal development and DNNResearch’s code (Hinton’s Company)

However, the graph idea was introduced in 2002 in torch, the basis of Pytorch (Circa 2016)

- One of the creators, Samy Bengio, is the brother of Joshua Bengio [3]

Backpropagation a little brother of Automatic Differentiation (AD)

We have a crude way to obtain derivatives [4, 5, 6][7]

$$D_{+h}f(x) \approx \frac{f(x+h) - f(x)}{2h} \text{ or } D_{\mp h}f(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Huge Problems

- If h is small, then cancellation error reduces the number of significant figures in $D_{+h}f(x)$.
- if h is not small, then truncation errors (terms such as $h^2 f'''(x)$) become significant.
- Even if h is optimally chosen, the values of $D_{+h}f(x)$ and $D_{\mp h}f(x)$ will be accurate to only about $\frac{1}{2}$ or $\frac{2}{3}$ of the significant digits of f .

Avoiding Truncation Errors

We have that

- Algorithmic differentiation does not incur truncation errors.

For example

$$f(x) = \sum_{i=1}^n x_i^2 \text{ at } x_i = i \text{ for } i = 1 \dots n$$

Then for $e_1 \in \mathbb{R}^n$

$$\frac{f(x + he_1) - f(x)}{h} = \frac{\partial f(x)}{\partial x_1} + h = 2x_1 + h = 2 + h$$

Floating Points

Given that the quantity needs floating point number representation in machine accuracy of 64 bits

$$\text{Roundoff error} = f(x + he_1) \epsilon \approx n^3 \frac{\epsilon}{3} \text{ with } \epsilon = 2^{-54} \approx 10^{-16}$$

For $h = \sqrt{\epsilon}$, as often is recommended

- The difference quotient has a rounding error of size

$$\frac{1}{3} n^3 \sqrt{\epsilon} \approx \frac{1}{3} n^3 10^{-8}$$

Now, Imagine $n = 1000$

Then Rounding Error

$$\frac{1}{3}1000^3\sqrt{\epsilon} \approx \frac{1}{3}1000000000 \times 10^{-8} = \frac{1}{3}100 \approx 33.333\dots$$

Ouch

- We cannot even get the sign correctly!!!

$$\frac{f(x + he_1) - f(x)}{h}$$

In contrast Automatic Differentiation

It yields

- $2x_i$ in both its forward and reverse modes

You could assume that the derivatives are generated symbolically

- Actually is true in some sense, but $2x_i$ will never be generated by Symbolic Differentiation

In Symbolic Differentiation

- The numerical value of x_i is multiplied by 2 then returned as the gradient value.

Example using Forward Differentiation

We will see the forward procedure later on

$$f(\mathbf{x}) = \sum_{i=1}^n x_i^2 \text{ with } x_i = i \text{ for } i = 1, \dots, n$$

AD Initializes (Do not worry we will see this in more detail)

$$v_{i-n} = i \text{ for } i = 1, \dots, n$$

$$\dot{v}_{i-n} = 0, \text{ but } \dot{v}_{1-n} = 1$$

Then, we have that

Apply the compositions

ϕ Functions	Derivatives
$v_1 = 1^2$	$\dot{v}_1 = \frac{\partial v_1}{\partial v_{1-n}} \dot{v}_{1-n} = 2 \times (1) \times 1 = 2$
\vdots	\vdots
$v_n = n^2$	0

Therefore, we have at the end

$$\frac{\partial f}{\partial \mathbf{x}}(x) = (2, 0, \dots, 0)$$

Quite different from

Using a numerical difference, we have

$$\frac{f(\mathbf{x} + \mathbf{e}_1 h) - f(\mathbf{x})}{h} - 2 < 0$$

Then for $n = 10^j$ and $h = 10^{-k}$

$$10^k [(h + 1)^2 - 1] < 2$$

Finally, we have

$$k > -\log_{10} 3$$

Therefore

It is possible to get into underflow

- by getting a $k > -\log_{10} 3$

Therefore, we have that

- Automatic Differentiation allows to obtain the correct answer!!!

For example

You have the following equation

$$f(x) = \prod_{i=1}^n x_i$$

Then, the gradient

$$\begin{aligned} \nabla f(x) &= \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right) = \left(\prod_{j \neq i} x_j \right)_{i=1 \dots n} \\ &= (x_2 \times x_3 \times \dots \times x_i \times x_{i+1} \times \dots \times x_{n-1} \times x_n, \\ &\dots \\ &x_1 \times x_2 \times \dots \times x_{i-1} \times x_{i+1} \times \dots \times x_{n-1} \times x_n, \\ &\dots \\ &x_1 \times x_2 \times \dots \times x_{i-1} \times x_i \times \dots \times x_{n-2} \times x_{n-1},) \end{aligned}$$

Actually

Symbolic Differentiation will consume a lot of memory

- Instead AD will reuse the common expressions to improve performance and memory.

However, Symbolic and Automatic Differentiation

- They make use of the chain rule to achieve their results

However, the chain rules in AD

- It is used not into the symbolic expressions but the actual numerical values.

The User Insight

Difference quotients may sometimes be useful too

$$\frac{f(x + he_1) - f(x)}{h}$$

Computer Algebra packages

- They have really neat ways to simplify expressions.

In contrast, current AD packages assume that

- That the given program calculates the underlying function efficiently

There

AD can automatize the gradient generation

- The best results will be obtained when AD takes advantage
 - ▶ the user's insight into the structure underlying the program

RNN Example

When you look at the recurrent neural network Elman [8]

$$\mathbf{h}_t = \sigma_h (W_{sd}\mathbf{x}_t + U_{sh}\mathbf{h}_{t-1} + b_h)$$

$$\mathbf{y}_t = \sigma_y (V_{os}\mathbf{h}_t)$$

$$L = \frac{1}{2} (\mathbf{y}_t - \mathbf{z}_t)^2$$

Here if you do blind AD sooner or later you have

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \times \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}} \times \frac{\partial \mathbf{h}_{t-2}}{\partial \mathbf{h}_{t-3}} \times \dots \times \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k}$$

- This is known as Back Propagation Through Time (BPTT)

This is a problem given

- The Vanishing Gradient or Exploding Gradient

Here, you can modify the architecture

Using an intermediate layer using the Hadamard product \circ we have

$$L = \frac{1}{2} (\mathbf{y}_t - \mathbf{z}_t)^2$$

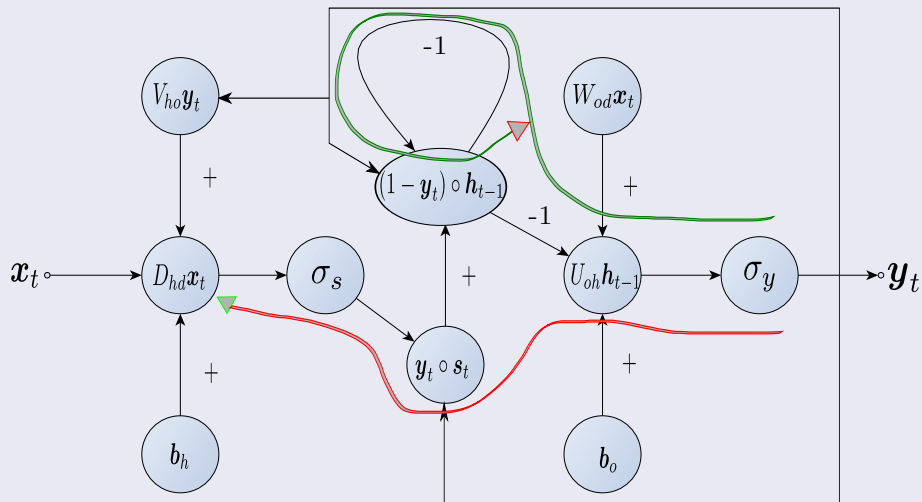
$$\mathbf{y}_t = \sigma_y (W_{od}\mathbf{x}_t + U_{oh}\mathbf{h}_{t-1} + \mathbf{b}_o)$$

$$\mathbf{s}_t = \sigma_s (V_{ho}\mathbf{y}_t + D_{hd}\mathbf{x}_t + \mathbf{b}_h)$$

$$\mathbf{h}_t = (1 - \mathbf{y}_t) \circ \mathbf{h}_{t-1} + \mathbf{y}_t \circ \mathbf{s}_t$$

Therefore

You have multiple paths of derivatives



One of them

It can be seen

- That one of the paths can take you to BPTT

The Other One

The other gets you into a more Markovian Property

- This allows to to get a Backpropagation that does not require the BPTT

How? For example, the derivative of L with respect to D_{hd}

$$\frac{\partial L}{\partial D_{hd}} = \frac{\partial L}{\partial \mathbf{y}_t} \times \frac{\partial \mathbf{y}_t}{\partial net_y} \times \frac{\partial net_y}{\partial \mathbf{h}_{t-1}} \times \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{s}_{t-2}} \times \frac{\partial \mathbf{s}_{t-2}}{net_s} \times \frac{net_s}{\partial D_{hd}}$$

Therefore

You do not have

- The Backpropagation through time... you can avoid it all together!!!

Because Backpropagation Through Time

- Makes the process of obtaining the gradients unstable...

Thus

A great simplifying step

- Here resound trues the phrase
 - ▶ “AD taking advantage of the user’s insight”

A Simple Example

Here, we have the following ideas

- Some of the floating point values, generated by the AD, will be stored in variables of the program,
- Other operations will be held until overwritten or discarded.

Thus, we will introduce the concept

- **Evaluation Trace** which is basically a record of a particular run of a given program.

This Evaluation Trace stores

- Input variables,
- Sequence of floating point generated by the CPU
- Operations that are used for it

Example

A simple example

$$y = f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - \exp(x_2) \right] \times \left[\frac{x_1}{x_2} - \exp(x_2) \right]$$

We wish to calculate $y = f(x_1, x_2)$

- With $x_1 = 1.5$, $x_2 = 0.5$

Evaluation Trace/Forward Procedure

We have the table for the evaluation of the function

$$v_{-1} = x_1 = 1.5$$

$$v_0 = x_2 = 0.5$$

$$v_1 = \frac{v_{-1}}{v_0} = \frac{1.5}{0.5} = 3.0$$

$$v_2 = \sin(v_1) = \sin(3.0) = 0.1411$$

$$v_3 = \exp(v_0) = \exp(0.5) = 1.6487$$

$$v_4 = v_1 - v_3 = 3.0 - 1.6487 = 1.3513$$

$$v_5 = v_2 + v_4 = 0.1411 + 1.3413 = 1.4924$$

$$v_6 = v_5 \times v_4 = 1.4924 \times 1.3513 = 2.0167$$

$$y = v_6 = 2.0167$$

A Cautionary Note

Normally

- Programmers will try to rearrange this execution trace to improve performance through parallelism.

Thus

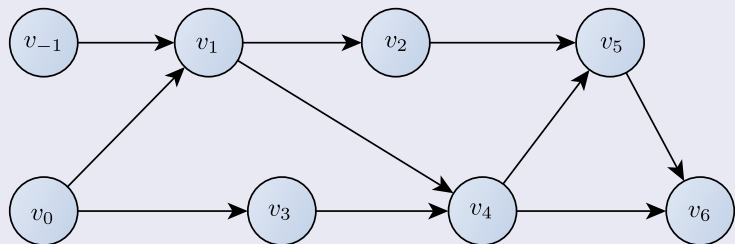
- Subexpressions will be algorithmically exploited by the AD to improve performance.

It is usually more convenient to use

- The so called “computational graph”

Computational Graph

A Simpler Version



Please take a look at section in **Chapter 2 A Framework for Evaluating Functions**

- At the book [7]
 - ▶ Andreas Griewank and Andrea Walther, **Evaluating derivatives: principles and techniques of algorithmic differentiation** vol. 105, (Siam, 2008).

A Little Bit of Notation

In general, we assume quantities v_i such

$$\underbrace{v_{1-n}, \dots, v_0}_x v_1, \dots, v_{l-m-1} \underbrace{v_{l-m+1}, \dots, v_l}_y$$

Then, we have

- 1 v_{1-n}, \dots, v_0 are the initial input variables
- 2 v_{l-m+1}, \dots, v_l the output variables
- 3 v_1, \dots, v_{l-m-1} the intermediate functions

Additionally

Where each value v_i with $i > 0$ is obtained by applying an elemental function ϕ

$$v_i = \phi_i(v_j)_{j \prec i}$$

- $j \prec i$ v_i depends directly on v_j

Then, for the application of the chain rule

It is useful to associate with each elemental function ϕ_i the state transformation

$$v_i = \Phi_i(v_{i-1}) \text{ with } \Phi_i : \mathbb{R}^{n+l} \rightarrow \mathbb{R}^{n+l}$$

where

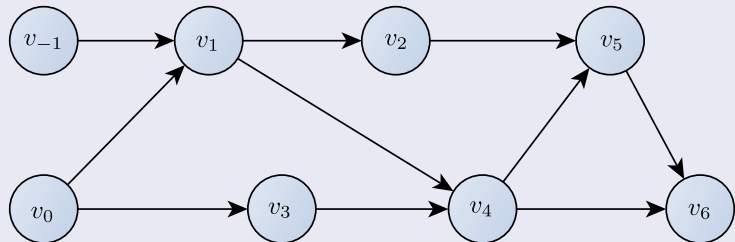
$$v_i = (v_{1-n}, \dots, v_i, 0, \dots, 0)^T$$

In other words

- Φ_i sets of v_i to $\phi_i(v_j)_{j < i}$ and keeps all other components v_j for $j \neq i$ unchanged.

Basically the Computational Graph

A Simpler Version



Example of the Forward Mode

Suppose we want to differentiate $y = f(x_1, x_2)$ with respect to x_1

- We consider x_1 as an independent variable and y as a dependent variable.

We can work the numerical value of the $y = f(x_1, x_2)$

- By getting the numerical derivative of each of its components

Something like

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1}$$

Therefore, we get

We have the Procedure

$v_{-1} = x_1 = 1.5$ $v_0 = x_2 = 0.5$	$\dot{v}_{-1} = 1.0$ $\dot{v}_1 = 0.0$
$v_1 = \frac{v_{-1}}{v_0} = \frac{1.5}{0.5} = 3.0$ $v_2 = \sin(v_1) = \sin(3.0) = 0.1411$ $v_3 = \exp(v_0) = \exp(0.5) = 1.6487$ $v_4 = v_1 - v_3 = 3.0 - 1.6487 = 1.3513$ $v_5 = v_2 + v_4 = 0.1411 + 1.3413 = 1.4924$ $v_6 = v_5 \times v_4 = 1.4924 \times 1.3513 = 2.0167$	$\dot{v}_1 = \frac{\partial v_1}{\partial v_{-1}} \dot{v}_{-1} + \frac{\partial v_1}{\partial v_0} \dot{v}_0 = 2.0$ $\dot{v}_2 = \cos(v_1) \dot{v}_1 = -1.98$ $\dot{v}_3 = v_3 \dot{v}_1 = 0.0$ $\dot{v}_4 = \dot{v}_1 - \dot{v}_3 = 2.0$ $\dot{v}_5 = \dot{v}_2 + \dot{v}_4 = 0.02$ $\dot{v}_6 = \dot{v}_5 \times v_4 + v_5 \times \dot{v}_4 = 3.0118$
$y = v_6 = 2.0167$	$\dot{y} = 3.0118$

The first Column of this process

It can be seen as an automatic procedure

v_{i-n}	$i = 1 \dots n$
$v_i = \varphi_i(v_j)_{j < i}$	$i = 1 \dots l$
$y_{m-i} = v_{l-i}$	$i = m - 1 \dots 0$

In a similar way

We can obtain $\frac{\partial f(x_1, x_2)}{\partial x_2}$

- However, it can be more efficient to redefine the \dot{v}_i as vectors for efficiency!!!

Forward propagation of Tangents

Remarks

- As you can see the second column of the evaluation procedure is done in a mechanical way

This increase the size

- Basically, twice the size of the original simple evaluation.

We have the following

We have the chain rule

$$\dot{y}(t) = \frac{\partial F(x(t))}{\partial t} = F'(x(t)) \dot{x}(t)$$

Where

- $F'(x) \in \mathbb{R}^{m \times n}$ is the Jacobian Matrix

Here, we will be tempted to calculate $\dot{y}(t)$

- By evaluating the full Jacobian $F'(x)$ then multiplying by $\dot{x}(t)$

However

Such approach is quite uneconomically

- Unless many tangents need to be calculated as in the Newton Step.

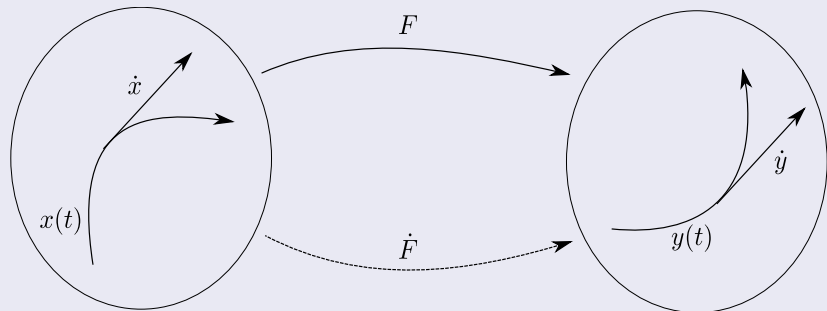
A simpler version, differentiate the first column of the table

$v_{i-n} = x_i$	$i = 1, \dots, n$
$v_i = \phi_i(v_j)_{j \prec i}$	$i = 1, \dots, l$
$y_{m-i} = v_{l-i}$	$i = m - 1, \dots, 0$

- $j \prec i$ v_i depends directly v_j (The graph propagation of the dependencies)

Which can be seen as Forward Propagation of Tangents

Basically, we can think of the forward mode as a propagation of tangents



The Automatic Procedure

Therefore, we have the following automatic procedure

- $j \prec i$ v_i **depends directly on** v_j and $u_i = (v_j)_{j \prec i} \in \mathbb{R}^{n_i}$

$v_{i-n} \equiv x_i$	$i = 1 \dots n$
$\dot{v}_{i-n} \equiv \dot{x}_i$	
$v_i \equiv \phi_i(v_j)_{j \prec i}$	$i = 1 \dots l$
$\dot{v}_i \equiv \sum_{j \prec i} \frac{\partial \phi_i(u_j)}{\partial v_j} \dot{v}_j$	$i = 1 \dots l$
$y_{m-i} \equiv v_{l-i}$	$i = m - 1 \dots 0$
$\dot{y}_{m-i} \equiv \dot{v}_{l-i}$	

Therefore

Each element assignment $v_i = \phi_i(u_i)$

- You have the corresponding

$$\dot{v}_i = \sum_{j \prec i} \frac{\partial \phi_i(u_j)}{\partial v_j} \times \dot{v}_j = \sum_{j \prec i} c_{ij} \times \dot{v}_j$$

Abbreviating $\dot{u}_i = (\dot{v}_j)_{j \prec i}$

$$\dot{v}_i = \dot{\phi}_i(u_i, \dot{u}_i) = \phi'_i(u_i) \dot{u}_i$$

Where $\dot{\phi}_i = \mathbb{R}^{2n_i} \rightarrow \mathbb{R}$

- It is called the tangent function associated with the elemental ϕ_i .

Now

Question

- What is the correct order of evaluation?

Why the question?

Until now, we have always placed the tangent statement yielding \dot{v}_i after the underlying value v_i

- This order of calculation seems natural and certainly yields correct results as long as there is no overwriting.

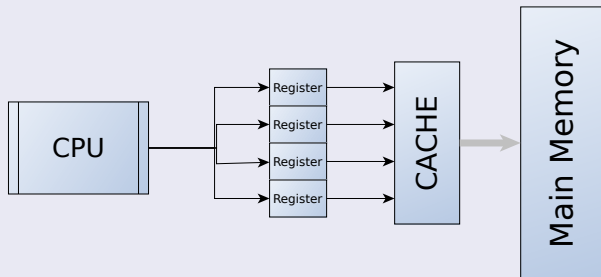
Then the order of $2l$ statements in the middle part of Table does not matter

$v_{i-n} \equiv x_i$ $\dot{v}_{i-n} \equiv \dot{x}_i$	$i = 1 \dots n$
$v_i \equiv \phi_i(v_j)_{j < i} \quad i = 1 \dots l$ $\dot{v}_i \equiv \sum_{j < i} \frac{\partial \phi_i(u_j)}{\partial v_j} \dot{v}_j$	$i = 1 \dots l$
$y_{m-i} \equiv v_{l-i}$ $\dot{y}_{m-i} \equiv \dot{v}_{l-i}$	$i = m - 1 \dots 0$

Here, we have a big problem in Cache

Imagine that we have a single block of memory to hold

- For v_i and its arguments v_j live in the same memory cell on the cache memory



This is known as Cache Aliasing

Definition

- Cache aliasing occurs when multiple mappings to a physical page of memory have conflicting caching states, such as cached and uncached.
 - ▶ the same physical address can be mapped to multiple virtual addresses.

On ARMv4 and ARMv5 processors, cache is organized as a virtual-indexed, virtual-tagged (VIVT)

- Cache lookups are faster because the translation look-aside buffer (TLB) is not involved in matching cache lines for a virtual address.

However

- This caching method does require more frequent cache flushing because of cache aliasing.

Then

The value of $\dot{v}_i = \dot{\phi}_i(u_i, \dot{u}_i)$ it will incorrect

- Once we update $v_i = \phi_i(u_i)$

ADIFOR and Tapenade [9, 5]

- They put the derivative statement ahead of the original assignment and update before the erasing the original statement.

On the other hand

- For most univariate functions $v = \phi(u)$ is better to obtain the undifferentiated value first
 - ▶ Then to use it into the tangent function $\dot{\phi}$

In this presentation

We will list φ and $\dot{\varphi}$

Side by side in a common bracket to indicate that they should be evaluated simultaneously

Then

- sharing results is immediate.

Classic Tangent Operations

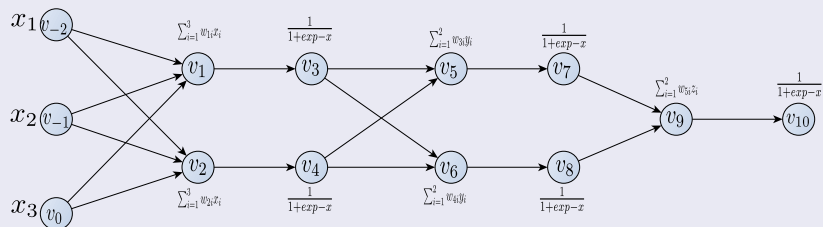
We have a series of improvements on the tangent equations

ϕ	$[\phi, \dot{\phi}]$
$v = c$	$v = c, \dot{v} = 0$
$v = v \pm w$	$v = v \pm w$ $\dot{v} = \dot{v} \pm \dot{w}$
$v = u \times w$	$\dot{v} = \dot{u} \times w + u \times \dot{w}$ $v = u \times w$
$v = 1/u$	$v = 1/u$ $\dot{v} = -v \times (v \times \dot{u})$

ϕ	$[\phi, \dot{\phi}]$
$v = u^c$	$v = \frac{\dot{u}}{u}; v = u^c$ $\dot{v} = v \times (v \times \dot{u})$
$v = \sqrt{u}$	$v = \sqrt{u}$ $v = 0.5 \times \frac{\dot{u}}{v}$
$v = \exp(u)$	$v = \exp(u)$ $\dot{v} = v * \dot{u}$
$v = \log(u)$	$\dot{v} = \dot{u}/u$ $v = \log(u)$
$v = \sin(u)$	$\dot{v} = \cos(u) \times \dot{u}$ $v = \sin(u)$

Now Imagine the following network

Something simple for our sake



Forward mode to get gradient of x_1

$v_{-11} = w_{11}, \dots, v_{-6} = w_{16}, v_{-5} = w_{21}, \dots, v_{-2} = w_{24}, v_{-1} = v_{31}, v_{-1} = w_{41}$
$\dot{v}_{-11} = 1, \dot{v}_{-10} = 0, \dots, \dot{v}_0 = 0$
$v_1 = \sum_{i=1}^3 w_{1i} x_i, \dot{v}_1 = x_1$
$v_2 = \sum_{i=1}^3 w_{2i} x_i, \dot{v}_2 = 0$
$v_3 = \frac{1}{1+\exp(-v_1)}, \dot{v}_3 = v_3 [1 - v_3] x_{11}$
$v_4 = \frac{1}{1+\exp(-v_2)}, \dot{v}_4 = 0$
$v_5 = \sum_{i=1}^3 w_{3i} v_i, \dot{v}_5 = w_{31} \times \dot{v}_3$
$v_6 = \sum_{i=1}^3 w_{4i} v_i, \dot{v}_6 = w_{41} \times \dot{v}_3$
$v_7 = \frac{1}{1+\exp(-v_5)}, \dot{v}_7 = v_7 [1 - v_7] \times \dot{v}_5$
$v_8 = \frac{1}{1+\exp(-v_6)}, \dot{v}_8 = v_8 [1 - v_8] \times \dot{v}_6$
$v_9 = \sum_{i=1}^2 w_{5i} v_i, \dot{v}_9 = w_{51} \times \dot{v}_7 + w_{52} \times \dot{v}_8$
$v_{10} = \frac{1}{1+\exp(-v_9)}, \dot{v}_{10} = v_{10} [1 - v_{10}] \times \dot{v}_9$

Complexity of the Procedure

Time Complexity

$$TIME \{F(x), F'(x) \dot{x}\} \leq w_{tan} TIME \{F(x)\}$$

- Where $w_{tan} \in \left[2, \frac{5}{2}\right]$

Space Complexity

$$SPACE \{F(x), F'(x) \dot{x}\} \leq 2SPACE \{F(x)\}$$

Here, an essential observation

The cost of evaluating derivatives by propagating them forward

- it increases linearly with number of directions \hat{x} along which we want to differentiate.

It looks inevitable

- But it is possible to avoid these complexity by
 - ▶ Observing that the gradient of a single dependent variable could be obtained for a fixed multiple of the cost of evaluating the underlying scalar-valued function.

We choose instead an output variable

We use the term “reverse mode” for this technique

- Because the label “backward differentiation” is well established [10, 11].

Therefore, for an output $f(x_1, x_2)$

- We have for each variable v_1

$$\bar{v}_i = \frac{\partial y}{\partial v_i} \text{ (Adjoint Variable)}$$

Actually

This is an abuse of notation

- We mean a new independent variable δ_i

$$\bar{v}_i = \frac{\partial y}{\partial \delta_i} \text{ (Adjoint Variable)}$$

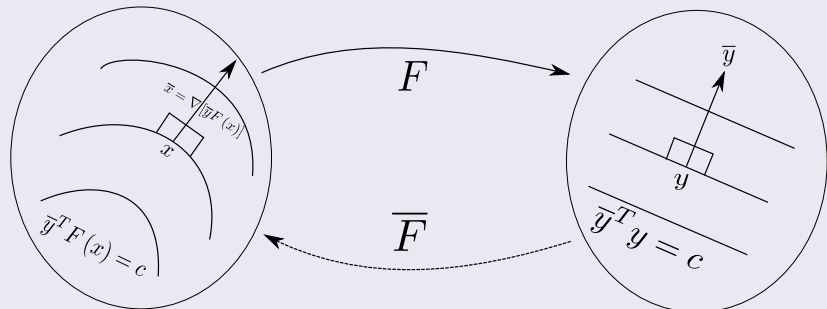
Which can be thought as adding a small numerical value δ_i to v_i

$$v_i + \delta_i \rightarrow f(x_1, x_2) + \bar{v}_i \delta_i$$

- As a perturbation in variational calculus

Actually, you propagate the Normal vectors

Actually, \bar{y} and \bar{v}_i are normals or cotangents



Then, we have

The following sought mapping

$$\bar{x} = \nabla \left[\bar{y}^T F(x) \right] = \bar{y}^T F'(x)$$

Observation

- Here, \bar{y} is a fixed vector that plays a dual role to the domain direction \dot{x} .

In the Forward Procedure, you compute

$$\dot{y} = F'(x) \dot{x} = \dot{F}(x, \dot{x})$$

Instead

In the Reverse Procedure, you compute

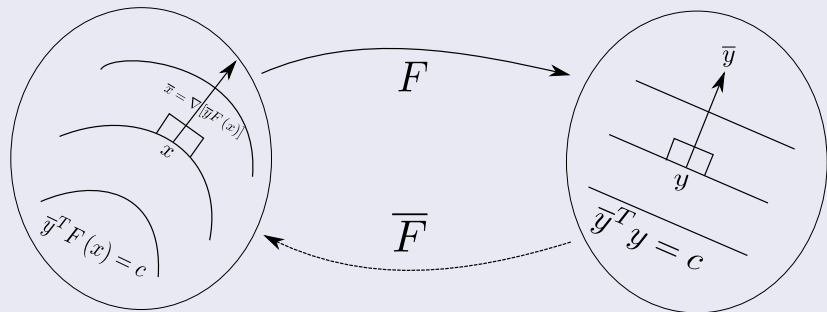
$$\bar{x}^T = \bar{y}^T F'(x) \equiv \bar{F}(x, \bar{y})$$

Where we solve F and \bar{F} are evaluated together

- Thus, we have a dual process

Dual Process

Here, we have that the hyperplane $\bar{y}^T \bar{y} = c$ in the range of F has inverse image $\{x | \bar{y}^T F(x) = c\}$



The implicit function theorem

Theorem

- Let $F : \mathbb{R}^{n+m} \rightarrow \mathbb{R}^m$ be a continuously differentiable function, and a point $(x_1^0, x_2^0, \dots, x_{m+n}^0)$ so $F(x_1^0, x_2^0, \dots, x_{m+n}^0) = c$. If $\frac{\partial F(x_1^0, x_2^0, \dots, x_{m+n}^0)}{\partial x_{m+n}} \neq 0$, then there exist a neighborhood of $(x_1^0, x_2^0, \dots, x_{m+n}^0)$ so whatever (x_1, \dots, x_{n+m-1}) is close enough to $(x_1^0, \dots, x_{n+m-1}^0)$, there is a unique z so that $F(x_1, \dots, x_{n+m-1}, z) = c$. Furthermore, $z = g(x_1, \dots, x_{n+m-1})$ a continuous function of (x_1, \dots, x_{n+m-1}) .

Therefore

The set $\{x | \bar{y}^T F(x) = c\}$

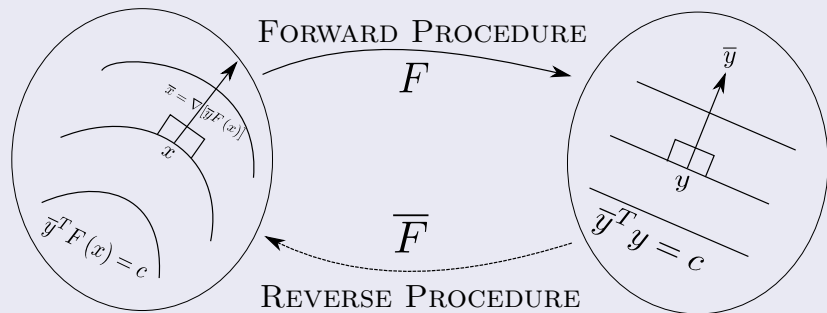
- It is a smooth hyper-surface with the normal

$$\bar{x}^T = \bar{y}^T F'(x)$$

at x provided that \bar{x} does not vanishes.

The Process

Here, we have that the hyperplane $\bar{y}^T \bar{y} = c$ in the range of F has inverse image $\{x | \bar{y}^T F(x) = c\}$



Therefore

When $m = 1$, then $F = f$ is scalar-valued

- We obtain $\bar{y} = 1 \in \mathbb{R}$ the familiar gradient $\nabla f(x) = \bar{y}^T F'(x)$.

Something Notable

- We will look only at the main procedure of Incremental Adjoint Recursion

Please take a look at section in **Derivation by Matrix-Product Reversal**

- At the book [7]
 - ▶ Andreas Griewank and Andrea Walther, **Evaluating derivatives: principles and techniques of algorithmic differentiation** vol. 105, (Siam, 2008).

The derivation of the reversal mode

For this, we will use

$v_{i-n} \equiv x_i$	$i = 1 \dots n$
$\dot{v}_{i-n} \equiv \dot{x}_i$	
$v_i \equiv \phi_i(v_j)_{j < i} \quad i = 1 \dots l$	$i = 1 \dots l$
$\dot{v}_i \equiv \sum_{j < i} \frac{\partial \phi_i(v_j)}{\partial v_j} \dot{v}_j$	
$y_{m-i} \equiv v_{l-i}$	$i = m - 1 \dots 0$
$\dot{y}_{m-i} \equiv \dot{v}_{l-i}$	

And the identity

$$\bar{y}^T \dot{y} = \bar{x}^T \dot{x}$$

Now, using the state transformation Φ

We map from x to $y = F(x)$ as the composition

$$y = Q_m \Phi_l \circ \Phi_{l-1} \circ \cdots \circ \Phi_2 \circ \Phi_1 \left(P_n^T x \right)$$

- Where $P_n \equiv [I, 0, \dots, 0] \in \mathbb{R}^{n \times (n+l)}$ and $Q_m \equiv [0, 0, \dots, I] \in \mathbb{R}^{m \times (n+l)}$

They are matrices that project an arbitrary $(n + l)$ -vector

- Onto its first n and last m components.

Where

The c_{ij} 's represent partial differential

$$c_{ij} \equiv c_{ij}(u_i) \equiv \frac{\partial \phi_i}{\partial v_j} \text{ for } 1 - n \leq i, j \leq l$$

Labelin the elemental partials as c_{ij}

We get the state Jacobian

$$A_i \equiv \Phi'_i \equiv \begin{bmatrix} 1 & 0 & \dots & 0 & \dots & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \dots & \dots & \\ 0 & 0 & \dots & 1 & \dots & \dots & 0 \\ c_{i1-n} & c_{i2-n} & \dots & c_{ii-n} & \dots & \dots & 0 \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \dots & \dots & \dots & 1 \end{bmatrix} \in \mathbb{R}^{(n+l) \times (n+l)}$$

- where the c_{ij} occur in the $(n+i)$ th row of A_i .

Remarks

The square matrices A_i are lower triangular

- It may also be written as rank-one perturbations of the identity,

$$A_i = I + e_{n+i} [\nabla \phi_i(u_i) - e_{n+i}]^T$$

- ▶ Where e_j denotes the j th Cartesian basis vector in \mathbb{R}^{n+l}

The differentiating the composition of functions

$$\dot{y} = Q_m A_l A_{l-1} \cdots A_2 A_1 P_n^T \dot{x}$$

Embeddings

The multiplication by $P_n^T \in \mathbb{R}^{(n+l) \times n}$

- It embeds \dot{x} into \mathbb{R}^{n+l}

Meaning

- corresponding to the first part of the tangent recursion

The subsequent multiplications by the A_i

- It generates the component \dot{v}_i at a time, according to the middle part

Finally

Q_m extracts the last m components as \dot{y} corresponding to the third part of the table

$v_{i-n} \equiv x_i$ $\dot{v}_{i-n} \equiv \dot{x}_i$	$i = 1 \dots n$
$v_i \equiv \phi_i(v_j)_{j < i} \quad i = 1 \dots l$ $\dot{v}_i \equiv \sum_{j < i} \frac{\partial \phi_i(v_j)}{\partial v_j} \dot{v}_j$	$i = 1 \dots l$
$y_{m-i} \equiv v_{l-i}$ $\dot{y}_{m-i} \equiv \dot{v}_{l-i}$	$i = m - 1 \dots 0$

Now

By comparison with

$$\dot{y}(t) = \frac{\partial F(x(t))}{\partial t} = F'(x(t)) \dot{x}(t)$$

We have in fact a product representation of the full Jacobian

$$F'(x) = Q_m A_l A_{l-1} \cdots A_2 A_1 P_n^T \in \mathbb{R}^{m \times n}$$

Then

By transposing the product we obtain the adjoint relation

$$\bar{x} = P_n A_1^T A_2^T \cdots A_{l-1}^T A_l^T \bar{y}$$

Given that

$$A_i^T = I + [\nabla \phi_i(u_i) - e_{n+i}] e_{n+i}^T$$

Therefore

The transformation of any vector $(\bar{v}_j)_{1-n \leq j \leq l}$

- By multiplication with A_i^T representing an incremental operation.

In detail, one obtains for $i = l, \dots, 1$ the operations

For all j with $i \neq j \not\prec i$

- \bar{v}_j is left unchanged

For all j with $i \neq j \prec i$

- \bar{v}_i is augmented by $\bar{v}_i c_{ij}$

$$c_{ij} \equiv c_{ij}(u_i) \equiv \frac{\partial \phi_i}{\partial v_j} \text{ for } 1 - n \leq i, j \leq l$$

Subsequently

- \bar{v}_i is set to zero.

Some Remarks

Using the C-style abbreviation

- $a+ \equiv b$ for $a \equiv a + b$
 - ▶ We may rewrite the matrix- vector product as the adjoint evaluation procedure in the following table

Incremental Adjoint Recursion

We have the following procedure ($u_i = (v_j)_{j \prec i} \in \mathbb{R}^{n_i}$)

$\bar{v}_i \equiv 0$	$i = 1 - n \dots l$
$\bar{v}_{i-n} \equiv x_i$	$i = 1 \dots n$
$v_i \equiv \phi_i(v_j)_{j \prec i}$	$i = m - 1 \dots l$
$y_{m-i} \equiv v_{l-i}$	$i = 0 \dots m - 1$
$\bar{v}_{l-i} \equiv \bar{y}_{m-i}$	$i = 0 \dots m - 1$
$\bar{v}_{j+} \equiv \bar{v}_i \frac{\partial \phi_i(u_i)}{\partial v_j}$ for $j \prec i$	$i = l \dots 1$
$\bar{x}_i \equiv \bar{v}_{i-n}$	$i = n \dots 1$

Explanation

It is assumed as a precondition that the adjoint quantities

- \bar{v}_i for $1 \leq i \leq l$ have been initialized to zero

As indicated by the range specification $i = l, \dots, 1$

- we think of the incremental assignments as being executed in reverse order, i.e., for $i = l, l - 1, l - 2, \dots, 1$.

Only then is it guaranteed

- Each \bar{v}_i will reach its full value before it occurs on the right-hand side.

Furthermore

We can combine the incremental operations

- Affected by the adjoint of ϕ_i to

$$\bar{u}_{i+} = \bar{v}_i \cdot \nabla \phi_i(u_i) \text{ where } \bar{u}_i \equiv (\bar{u}_j)_{j \prec i} \in \mathbb{R}^{n_i}$$

Something Remarkable

- We can do something different
 - ▶ one can directly compute the value of the adjoint quantity \bar{v}_j by collecting all contributions to it as a sum ranging over all successors $i \succ j$.

This no-incremental

- Requires global information that is not easy to come by.

Complexity

Something Notable

$$TIME \{ F(x), \bar{y}^T F'(x) \} \leq w_{grad} TIME \{ F(x) \}$$

- Where $w_{grad} \in [3, 4]$ (The cheap gradient principle)

Remember

Time Complexity

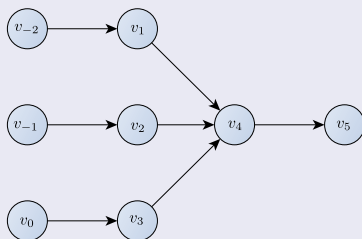
$$TIME \{F(x), F'(x) \dot{x}\} \leq w_{tan} TIME \{F(x)\}$$

- Where $w_{tan} \in \left[2, \frac{5}{2}\right]$

Example a single layer perceptron

We have

$$y = \sigma \left(\sum_{i=1}^3 w_i x_i \right)$$



First Phase

Forward Step

Forward Step

$v_{-2} = w_1$

$v_{-1} = w_2$

$v_0 = w_3$

$v_1 = x_1 v_{-2}$

$v_2 = x_2 v_{-1}$

$v_3 = x_3 v_0$

$v_4 = v_1 + v_2 + v_3$

$v_5 = \sigma(v_4)$

$y_1 = v_5$

Second Phase

Incremental Return

Forward Step
$v_{-2} = w_1$
$v_{-1} = w_2$
$v_0 = w_3$
$v_1 = x_1 v_{-2}$
$v_2 = x_2 v_{-1}$
$v_3 = x_3 v_0$
$v_4 = v_1 + v_2 + v_3$
$v_5 = \sigma(v_4)$
$y_1 = v_5$

Incremental Return
$\bar{v}_5 = \bar{y}_1 = 1$
$\bar{v}_4 = \frac{\partial v_5}{\partial v_4} \bar{y}_1 = \sigma'(v_4)$
$\bar{v}_3 + = \frac{\partial v_4}{\partial v_3} \bar{v}_4 = 1 \times \sigma'(v_4)$
$\bar{v}_0 = \frac{\partial v_3}{\partial v_0} \bar{v}_3 = x_3 \times \sigma'(v_4)$
$\bar{v}_2 + = \frac{\partial v_4}{\partial v_2} \bar{v}_4 = 1 \times \sigma'(v_4)$
$\bar{v}_{-1} = \frac{\partial v_2}{\partial v_{-1}} \bar{v}_2 = x_2 \times \sigma'(v_4)$
$\bar{v}_1 + = \frac{\partial v_4}{\partial v_1} \bar{v}_4 = 1 \times \sigma'(v_4)$
$\bar{v}_{-2} = \frac{\partial v_1}{\partial v_{-2}} \bar{v}_1 = x_1 \times \sigma'(v_4)$
$\bar{w}_3 = x_3 \times \sigma'(v_4)$
$\bar{w}_2 = x_2 \times \sigma'(v_4)$
$\bar{w}_1 = x_1 \times \sigma'(v_4)$

How does it compares with the Forward Mode?

We noticed that you do the following for each gradient variable

Forward Step; Gradient of Forward Step
$v_{-2} = w_1; \dot{v}_{-2} = \dot{w}_1 = 0$
$v_{-1} = w_2; \dot{v}_{-1} = \dot{w}_2 = 0$
$v_0 = w_3; \dot{v}_0 = \dot{w}_3 = 1$
$v_1 = x_1 v_{-2}$
$\dot{v}_1 = x_1 \dot{v}_{-2} = 0$
$v_2 = x_2 v_{-1}$
$\dot{v}_2 = x_2 \dot{v}_{-1} = 0$
$v_3 = w_3 v_0$
$\dot{v}_3 = x_3 \dot{v}_0 = x_3$
$v_4 = v_1 + v_2 + v_3$
$\dot{v}_4 = \dot{v}_1 + \dot{v}_2 + \dot{v}_3 = x_3$
$v_5 = \sigma(v_4)$
$\dot{v}_5 = \dot{v}_4 = x_3 \times \sigma'(v_4)$
$y_1 = v_5; \dot{y}_1 = \dot{v}_5$

Let us to look at the following example

We have the following system of equations

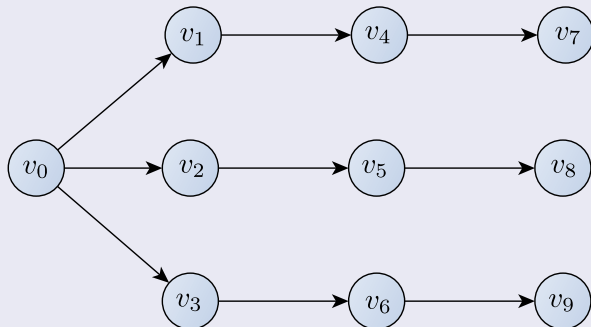
$$y_1 = \sigma(w_1 x)$$

$$y_2 = \sigma(w_2 x)$$

$$y_3 = \sigma(w_2 x)$$

With the following graph

Notice the difference with a neural network



The Forward mode looks like

We have that

$$v_0 = x; \dot{v}_0 = \dot{x} = 1$$

$$v_1 = w_1 v_0$$

$$\dot{v}_1 = w_1 \dot{v}_0 = w_1$$

$$v_2 = w_2 v_1$$

$$\dot{v}_2 = w_2 \dot{v}_1 = w_2 w_1$$

$$v_3 = w_3 v_2$$

$$\dot{v}_3 = w_3 \dot{v}_2 = w_3 w_2 w_1$$

$$v_4 = \sigma(v_1)$$

$$\dot{v}_4 = \sigma'(v_1) \times \dot{v}_1 = w_1 \times \sigma'(v_1)$$

$$v_5 = \sigma(v_2)$$

$$\dot{v}_5 = \sigma'(v_2) \times \dot{v}_2 = w_2 \times \sigma'(v_2)$$

$$v_6 = \sigma(v_3)$$

$$\dot{v}_6 = \sigma'(v_3) \times \dot{v}_3 = w_3 \times \sigma'(v_3)$$

$$y_1 = v_4; \dot{y}_1 = \dot{v}_4$$

$$y_2 = v_5; \dot{y}_2 = \dot{v}_5$$

$$y_3 = v_6; \dot{y}_3 = \dot{v}_6$$

Now you can see it

Forward and Reverse Mode

- They depend on the input and output size!!!

A More Formal Definition

- For a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, suppose we wish to compute all the elements of the $m \times n$ Jacobian matrix

Ignoring the overhead of building the expression graph

- Under this situation Reverse Mode requires m sweeps performs better when $n > m$.

Consequences for Deep Learning

With a relatively small overhead

- The performance of reverse-mode AD is superior when $n \gg m$, that is when we have many inputs and few outputs.

As we saw it in the previous examples

- If $n \leq m$ forward mode performs better

Special Cases

Nevertheless when we have a comparable number of outputs and inputs

- Forward mode can be more efficient,
 - ▶ less overhead associated with storing the expression graph in memory in forward mode.

For Example

- If you have $f : \mathbb{R}^n \rightarrow \mathbb{R}$, when $n = 1$ forward mode is more efficient, but the result flips as n increases.

Be Aware

Be Careful

- A computationally naive implementation of AD can result in slow code and excess use of memory.

Additionally

- There exists no standard set of problems spanning the diversity of AD applications.

Source Transformation in Fortran and C

First

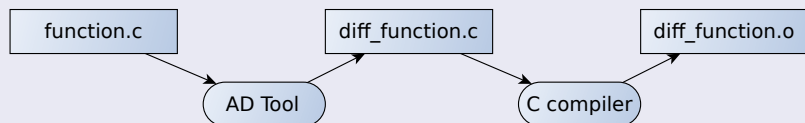
- We start with the source code of a computer program that implements our target function.

Second

- A preprocessor then applies differentiation rules to the code generating new source code which calculates derivatives.
 - ▶ Remember our basic tables

Example on how source code transformation could work

We could have the following pipeline



Limitations of Source Transformation

Severe limitations with source transformation

- it can only use information available at compile time
 - ▶ It cannot handle more sophisticated programming statements, such as while loops, C++ templates, and other object-oriented features

For this, it is better to use operator overloading

- Operator overloading is the appropriate technology.

Operator overloading

The key idea is to introduce a new class of objects

- Containing the value of a variable on the expression graph and a differential component.

Not all variables on the expression graph will belong to this class

- But the root variables, which require sensitivities, and all the intermediate variables.

In a forward mode framework

- The differential component is the derivative with respect to one input.

Furthermore

In a reverse mode framework

- it is the adjoint with respect to one output.

Something Notable

- Operators and math functions are overloaded to handle these new types.

Basically

The operators are overloaded

- So it is possible to handle the dual numbers under their new arithmetic (Forward Mode)

Building a Computational Graph

Parse the equations

$$y_1 = \sigma(w_1x)$$

$$y_2 = \sigma(w_2x)$$

$$y_3 = \sigma(w_2x)$$

Generate variables for intermediate values

- $v_0 = x$ Then $V = V \cup \{v_0\}$

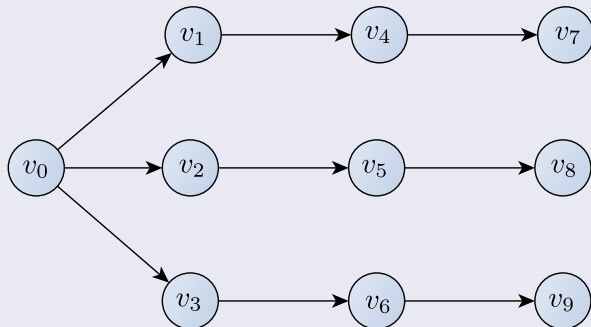
Generate edges between the intermediate values

- If $v_1 = w_1x$ Then $E = E \cup \{\langle v_0, v_1 \rangle\}$

Topological sort for Evaluation

Run the topological sort, then you get the order of evaluation

- Using the graph built in the previous step



For the Reversal Mode

We could use a stack

- When assignments occur at the Forward Mode of the process

$\bar{v}_i \equiv 0$	$i = 1 - n \dots l$
$\bar{v}_{i-n} \equiv x_i$	$i = 1 \dots n$
$v_i \equiv \phi_i(v_j)_{j < i}$	$i = m - 1 \dots l$
$y_{m-i} \equiv v_{l-i}$	$i = 0 \dots m - 1$

Then we pop the necessary elements

- At the reversal process

Nevertheless

There are many techniques to improve the efficiency and avoid aliasing problems of these modes [12]

- 1 Taping for adjoint recursion
- 2 Caching
- 3 Checkpoints
- 4 Expression Templates
- 5 etc

You are invited to read more about them

- Given that these techniques are already being implemented in languages as swift...
 - ▶ “First-Class Automatic Differentiation in Swift: A Manifesto”
<https://gist.github.com/rxwei/30ba75ce092ab3b0dce4bde1fc2c9f1d>

Between Two Extremes

Something Notable

- Forward and reverse accumulation are just two (extreme) ways of traversing the chain rule.

The problem of computing a full Jacobian of $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with a minimum number of arithmetic operations

- It is known as the Optimal Jacobian Accumulation (OJA) problem, which is NP-complete [13].






Finally






Using all the previous ideas




- The Graph Structure Proposed in [2]
- The Computational Graph of AD
- The Forward and Reversal Methods

It has been possible to develop the Deep Learning Frameworks

- TensorFlow
- Torch
- Pytorch
- Keras
- etc...

-  D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.
-  R. Rojas, *Neural networks: a systematic introduction*. Springer Science & Business Media, 1996.
-  R. Collobert, S. Bengio, and J. Mariéthoz, “Torch: a modular machine learning software library,” Idiap-RR Idiap-RR-46-2002, IDIAP, 2002.
-  A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: a survey,” *Journal of machine learning research*, vol. 18, no. 153, 2018.
-  C. H. Bischof, A. Carle, P. Khademi, and A. Mauer, “ADIFOR 2.0: Automatic differentiation of Fortran 77 programs,” *IEEE Computational Science & Engineering*, vol. 3, no. 3, pp. 18–32, 1996.

-  C. Elliott, “The simple essence of automatic differentiation,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, p. 70, 2018.
-  A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, vol. 105. Siam, 2008.
-  J. L. Elman, “Finding structure in time,” *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.
-  L. Hascoët and V. Pascual, “The Tapenade automatic differentiation tool: Principles, model, and specification,” *ACM Transactions on Mathematical Software*, vol. 39, no. 3, pp. 20:1–20:43, 2013.
-  Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural networks: Tricks of the trade*, pp. 9–48, Springer, 2012.

-  R. Alexander, “Solving ordinary differential equations i: Nonstiff problems (e. hairer, sp norsett, and g. wanner),” *Siam Review*, vol. 32, no. 3, p. 485, 1990.
-  C. C. Margossian, “A review of automatic differentiation and its efficient implementation,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 4, p. e1305, 2019.
-  U. Naumann, “Optimal jacobian accumulation is np-complete,” *Mathematical Programming*, vol. 112, no. 2, pp. 427–441, 2008.