# Automatic Differentiation

In this talk:

- How to calculate derivatives inside computer?
- Forward and reverse mode automatic differentiation
- Computational graphs and evaluation traces
- Engineering trade-offs

Annada Behera

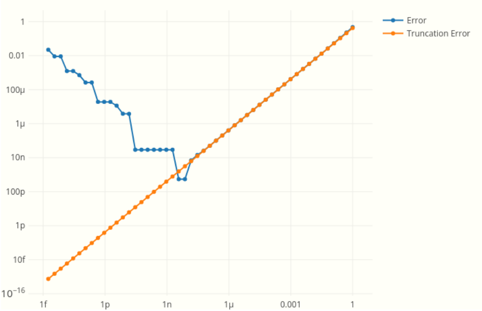
June 10, 2022

# Computing derivatives: Numerical approximation

For a function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient can be computed by Netwon's method,

$$\nabla f(x) = \lim_{h \to 0} \frac{f(x + e_i h) - f(x)}{h} \tag{1}$$

where $e_i$ is the unit vector in $i$-th dimension.

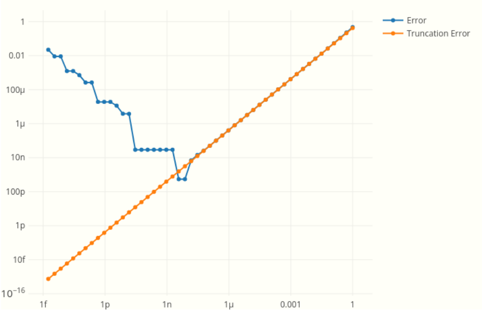


# Computing derivatives: Numerical approximation

For a function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient can be computed by Netwon's method,

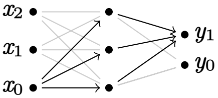
$$\nabla f(x) = \lim_{h \to 0} \frac{f(x + e_i h) - f(x)}{h} \tag{1}$$

where $e_i$ is the unit vector in $i$-th dimension.




**Idea:** Use arbitrary-precision arithmatic. Takes $O(nk^2)$ time for $n$ variables with $k$ digits of precision. And this is still just an approximation!

# Symbolic differentiation




$$y_1 = \sigma(w_{00}x_0) + \sigma(w_{10}x_1) + \sigma(w_{20}x_2) + \sum_j \sigma\left(\sum_i w_{ij}x_i\right) \tag{2}$$

where $\sigma$ is any non-linear function. Or, if $x = x_i$, $y = y_i$, $W^0 = w_{ij}$, and $W^1 = w_{ij}$, then,

$$y = W^1 \sigma(W^0 x) \tag{3}$$

Consider the product rule,

$$\frac{\partial fg}{\partial x} = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x} \tag{4}$$

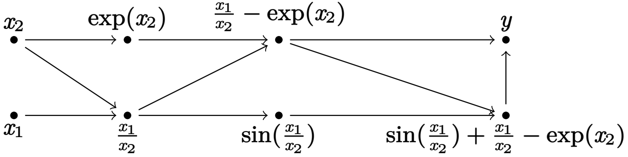
This method of computing derivative of $n$ variables takes $O(2^n)$ time!

# Computational graph and evaluation traces

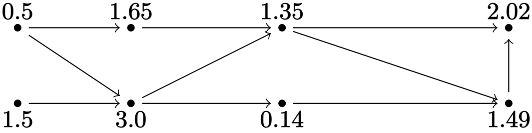
Consider the equation,

$$y = [\sin(x_1/x_2) + x_1/x_2 - \exp(x_2)][x_1/x_2 - \exp(x_2)] \tag{5}$$

The computational graph (DAG):

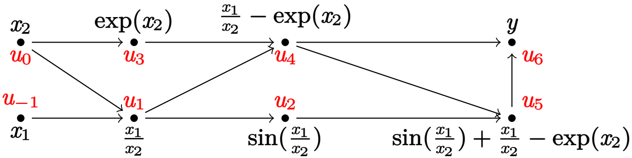



The decompsition of calculations into elementary steps forms the *evaluation trace.* Say, $x = 1.5$ and $x_2 = 0.5$, then the evaluation trace will look like,




These intermidate values are called the *primal trace.*

# Forward mode: Accumulating the *tangent trace*

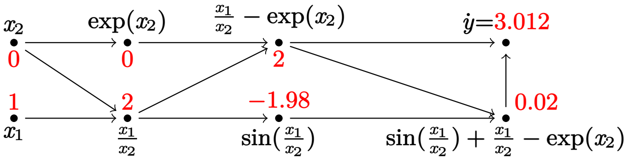



For $x_1 = 1.5$ and $x_2 = 0.5$, the primal and tangent trace with respect to $x_1$ will be,

$$
\begin{aligned}
u_{-1} &= x_1 \\
u_0 &= x_2 \\
u_1 &= u_{-1}/u_0 \\
u_2 &= \sin(u_1) \\
u_3 &= \exp(u_0) \\
u_4 &= u_1 - u_3 \\
u_5 &= u_2 + u_4 \\
u_6 &= u_5 u_4
\end{aligned}
$$

$$
\begin{aligned}
\dot{u}_{-1} &= \frac{\partial u_{-1}}{x_1} = \partial_{x_1} x_1 = 1 \\
\dot{u}_0 &= \partial_{x_1} x_2 = 0 \\
\dot{u}_1 &= \partial(u_{-1}/u_0) = u_{-1}\partial_{x_1}(1/u_0) + \partial_{x_1} u_{-1} \cdot \frac{1}{u_0} \\
&= 0 + 1/0.5 = 2 \\
\dot{u}_2 &= \partial_{x_1} \sin(u_1) = \cos(u_1)\dot{u}_1 = \cos(3.0) \times 2 = -1.98 \\
\dot{u}_3 &= \partial_{x_1} \exp(u_0) = \exp(u_0)\dot{u}_0 = 0
\end{aligned}
$$

# Accumulating tangent trace (contd.)




Mathematica:

```
In[1]:= D[(Sin[x/0.5]+x/0.5-Exp[0.5])*(x/0.5-Exp[0.5]),x]/.x->1.5
Out[0]= 3.01184
```

We can do the same and get the derivative with respect to $x_2$. Therefore for any function, $f : \mathbb{R}^n \to \mathbb{R}^m$, it takes $O(n)$ time.

Forward mode is a great choice for $f : \mathbb{R} \to \mathbb{R}^m$ since, takes **constant** time but it is a poor choice for $f : \mathbb{R}^n \to \mathbb{R}$.

## How to code?

A *dual number* takes the form,

$$x = a + b\epsilon \quad \text{where, } \epsilon^2 = 0 \text{ and } a, b \in \mathbb{R} \tag{6}$$

The component-wise addition and multiplication is given by,

$$(a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\epsilon \tag{7}$$

It is not hard to see that dual numbers form a *commutative algebra* over two dimension.

The Taylor series expansion of any arbitrary function $f$ at the dual number $a + \epsilon$ is,

$$f(a + b\epsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a) b^n \epsilon^n}{n!} = f(a) + b\dot{f}(a)\epsilon \tag{8}$$

Evaluate any function at $a + \epsilon$, the co-efficient of $\epsilon$ gives the derivative of the function.

# Forward mode autodiff: an example

- Product rule:

$$
\begin{aligned}
f(a+\epsilon)g(a+\epsilon) &= [f(a)+\dot{f}(a)\epsilon][g(a)+\dot{g}(a)\epsilon] && (9)\\
&= f(a)g(a)+[\dot{f}(a)g(a)+f(a)\dot{g}(a)]\epsilon && (10)
\end{aligned}
$$

- Chain rule:

$$
\begin{aligned}
f(g(a+\epsilon)) &= f(g(a)+\dot{g}(a)\epsilon) && (11)\\
&= f(g(a))+\dot{f}(g(a))\dot{g}(a)\epsilon && (12)
\end{aligned}
$$

- An example: What is the derivative of $f(x) = kx + \sin(x)$?

$$
\begin{aligned}
f(a+\epsilon) &= k\cdot(a+\epsilon)+\sin(a+\epsilon) && (13)\\
&= ka+k\epsilon+\sin(a)\cos(\epsilon)+\cos(a)\sin(\epsilon) && (14)\\
&= \underbrace{ka+\sin(a)}_{\text{primal trace}}+\epsilon(\underbrace{k+\cos(a)}_{\text{tangent trace}}) && (15)
\end{aligned}
$$

# Reverse mode autodiff: Propagate from the end

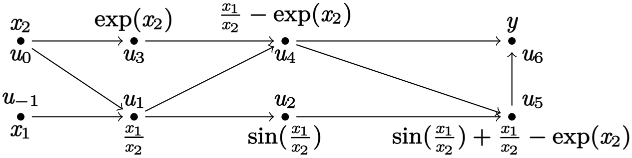
Chain rule: For any arbitrary function, $y = f(x(t))$,

$$\partial_t y = \partial_x y \cdot \partial_t x \tag{16}$$

Consider the same function again,

$$y = [\sin(x_1/x_2) + x_1/x_2 - \exp(x_2)][x_1/x_2 - \exp(x_2)] \tag{17}$$

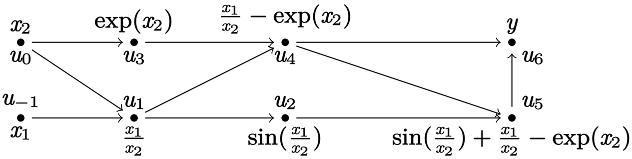
along with it's variables and DA computational graph,




Define the *adjoint* $\bar{u}_i$ as,

$$\bar{u}_i = \partial_{u_i} y \tag{18}$$

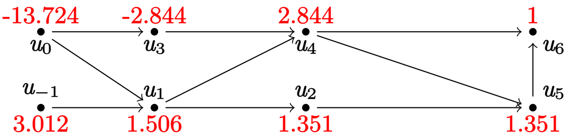
# Reverse mode autodiff (contd.)




$$\begin{aligned}
u_6 &= u_5 u_4 = 2.016 \\
u_5 &= u_2 + u_4 = 1.492 \\
u_4 &= u_1 - u_3 = 1.351 \\
u_3 &= \exp(u_0) = 1.648 \\
u_2 &= \sin(u_1) = 0.141 \\
u_1 &= u_{-1}/u_0 = 3 \\
u_0 &= x_2 = 0.5 \\
u_{-1} &= x_1 = 1.5
\end{aligned}$$

$$\begin{aligned}
\partial_{u_6} y &= 1 \\
\partial_{u_5} y &= \partial_{u_6} y \cdot \partial_{u_5} u_6 = 1.351 \\
\partial_{u_4} y &= \partial_{u_6} y \cdot \partial_{u_4} u_6 + \partial_{u_5} y \cdot \partial_{u_4} u_5 \\
&= 1.351 + 1.492 = 2.844
\end{aligned}$$

# Reverse mode autodiff (contd.)




$$\bar{x}_1 = \partial_{x_1} y = 3.012 \quad \text{and,} \quad \bar{x}_2 = \partial_{x_2} y = -13.724 \tag{19}$$

- For a function, $f : \mathbb{R}^n \to \mathbb{R}^m$, the reverse mode autodiff takes $O(m)$ time!
- Great choice for $f : \mathbb{R}^n \to \mathbb{R}$ but poor choice for $g : \mathbb{R} \to \mathbb{R}^m$.
- In neural networks, the loss function is, $L : \mathbb{R}^n \to \mathbb{R}$. Therefore the gradients can be computed in **constant** time.
- We are essenitally trading space for time.

# Engineering trade-offs: Static graphs

Import the Tensorflow 1.0 package:

```python
import tensorflow as tf
```

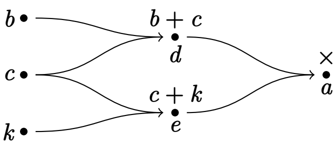
Creating variables:

```python
k = tf.Variable(2.0, name='k')
b = tf.Variable(2.0, name='b')
c = tf.Variable(1.0, name='c')
```

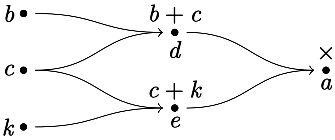
Doing calculations:

```python
d = tf.add(b, c,       name='d')
e = tf.add(c, k,       name='e')
a = tf.multiply(d, e, name='a')
```

Computational graph:



# Dynamic graphs




```
from torch import nn
import torch
class calc(nn.Module):
    def __init__(self):
        super().__init__()
        self.k = torch.tensor([2],
        requires_grad=True)
    def forward(self, b, c):
        d = b+c
        e = c+self.k
        return e*d
b = 2
c = 1
a = calc(b, c)
```

- All computations must be a part of nn.Module.
- All parameters must be top level variables in the constructor.
- All computation is stored on a ‘tape’ during the forward pass.
- Hence, dynamic graphs are slow.

## Summary

- Numerical differentiation is slow and inaccurte.
- Symbolic differentiation is extremely slow and hard to code.
- Algorithmic/automatic differentiation is fast and exact.
- Forward-mode autodiff is better for $f : \mathbb{R} \to \mathbb{R}^m$.
- Reverse-mode autodiff is better for $f : \mathbb{R}^m \to \mathbb{R}$.
- Forward mode takes less space but needs more time.
- Reverse mode takes more space and needs less time.
- Neural network libraries use reverse mode autodiff.
- Dynamic computation graph is slow but easy to write, e.g, PyTorch.
- Static computation graph is fast but hard to write, e.g, Tensorflow.
- JAX and Julia make trade-offs in between static and dynamic graphs using *jit*.

Thank you.