AUTOMATIC DIFFERENTIATION

KAIM
Anand Krish

Agenda

- Differentiation Strategies
- Nuts & Bolts of Autodiff
  - *Modes of Operation*
- Autodiff for Deep Learning
- Tutorial

Differentiation

- ML algorithms require Gradients & Hessians for optimization
- Computers can perform differentiation in 3 ways
  - *Numerical Differentiation*
    - Easy to implement ✓
    - Finite Approximations ✗
    - Prone to numerical error ✗
    - Slow and inefficient (scales poorly $\approx \mathcal{O}(d)$ in d dimensions) ✗
  - *Symbolic Differentiation*
    - Efficient and accurate ✓
    - Requires *closed form* expressions ✗
    - Difficult to implement ✗
    - Mathematica, Maple etc
  - **Automatic Differentiation**
    - Best of both worlds!! ✓

Automatic Differentiation

- Breakdown complex function → list of Elementary functions (Wengert List)
  - *Use Chain Rule!*
- Can be applied for *any* computational structure –
  - *Sequential, recursive, branched or iterative*
    - These does not alter the numeric values
  - *Represented as a* **computation graph**
- Which elementary functions?
  - *Transcendental functions (exp, log, trigonometric)*
  - *Arithmetic*
- Requires pre-computed derivatives of elementary functions
- Nothing “Automatic” in autodiff – **Algorithmic Differentiation** is more proper.

Computation Graph

Computation graph of $f(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$

Intermediate Variables

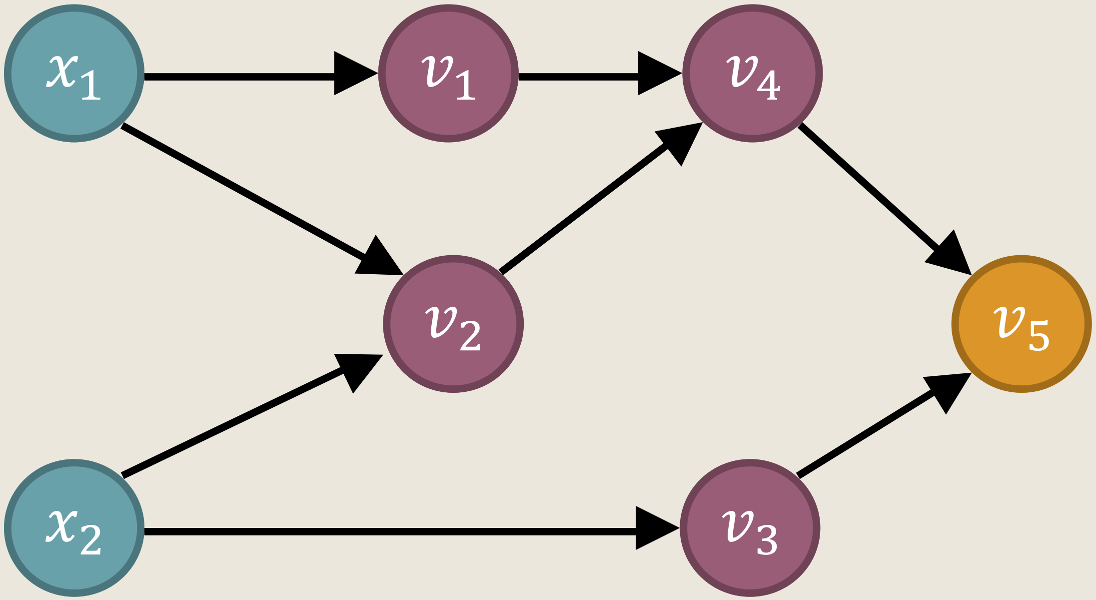
$v_1 = \log(x_1)$

$v_2 = x_1 x_2$

$v_3 = sin(x_2)$

$v_4 = v_1 + v_2$

$f = v_5 = v_4 - v_3$



Backprop rule -

$$\boxed{\dot{v}_i = \dot{v}_i + \dot{v}_{\{i+1\}} \frac{\partial v_{\{i+1\}}}{\partial v_i}}$$

Forward Accumulation Mode

- Straight-forward chain rule
- Idea – Jitter the input to see how to the output changes
- Suitable for functions with **No. of inputs << No. of outputs**
  - Single pass can compute derivatives of all outputs for one input
  - N passes for N inputs, irrespective of no. of outputs.
  - Best suited for computing Jacobians (one pass computes one column);

$$\mathbf{J} = \begin{bmatrix} \nabla f_1 \\ \nabla f_2 \\ \nabla f_3 \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_1} \end{bmatrix}$$

Reverse Accumulation Mode

- Chain rule in reverse (Essentially the backpropagation algorithm)
- Idea – jitter the output(s) and check how the inputs vary
- Suitable for functions with **No. of inputs ≥ No. of outputs**
  - This is the case for almost all neural networks
  - M passes for M outputs, irrespective of the no. of inputs
- Two phases:
  - **Forward phase** *: Compute all the intermediate values*
  - **Reverse phase** *: Compute the gradients with respect to the previous values (fancy term - adjoint)*
- Seems complex but number of operations (flops) are in fact less
  - *Higher space complexity*
- Speed Vs Space complexity trade-off

Autograd Package

- Developed by HIPS Group, Harvard University
- The following operations are done *dynamically* –
  - *Decompose a complex function into a compound list of elementary functions*
  - *Construct Computation graphs*
  - *Derivatives of complex functions*
    - Fourier transforms, logsumexp, tensor operations etc.
- Pytorch and Chainer extends the functionality through -
  - *In-place computations (no additional memory)*
  - *Require only the subset of computation graph*
    - Enables multi-threading

QUESTION TIME