THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

RELAXED SYSTEM LAB

Automatic Differentiation

COMP4901Y

Binhang Yuan

RELAXED
SYSTEM LAB

Numerical Differentiation

Numerical Differentiation

- Numerical differentiation is the finite difference approximation of derivatives using values of the original function evaluated at some sample points.
- It is based on the limit definition of a derivative of function $f\colon \mathbb{R}^n \to \mathbb{R}$:

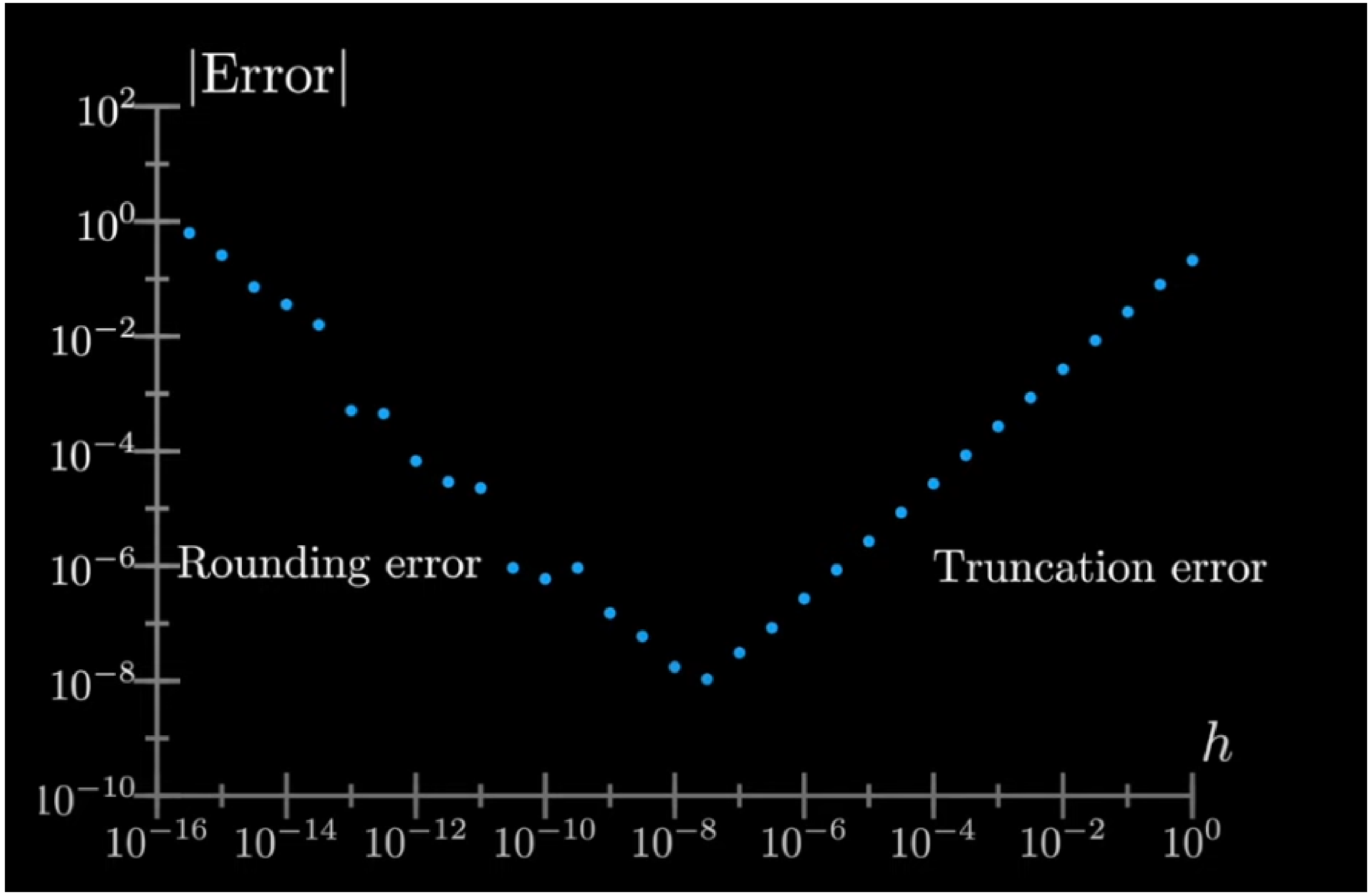
$$\frac{\partial f}{\partial x_i} = \lim_{\epsilon \to 0} \frac{f(\boldsymbol{x} + \epsilon \boldsymbol{e}_i) - f(\boldsymbol{x})}{\epsilon} \approx \frac{f(\boldsymbol{x} + h\boldsymbol{e}_i) - f(\boldsymbol{x})}{h}$$

- $\boldsymbol{e}_i$ is the i-th unit vector, $h > 0$ is a small step size.

Pros and Cons

- Advantage:
  - Easy to implement.
- Disadvantage:
  - Perform $\mathcal{O}(n)$ evaluatoins of f for a gradient in n dimensions.
  - Requires careful consideration in selecting the step size h.

Choose Step Size h



- Truncation Error:
  - The error of approximation that one gets from h not actually being zero.
  - Proportional to a power of h.
- Rounding Error:
  - The inaccuracy that is inflicted by the limited precision of computations.
  - Inversely proportional to a power of h.

RELAXED SYSTEM LAB

Symbolic Differentiation

Derivative Computation Rules

- Assume $f(x): \mathbb{R} \rightarrow \mathbb{R}, \quad g(x): \mathbb{R} \rightarrow \mathbb{R}$:
- Derivative of sum or difference: $u = f(x), v = g(x)$:

$$\frac{d}{dx}(u \pm v) = \frac{du}{dx} \pm \frac{dv}{dx}$$

- Product Rule: $u = f(x), v = g(x)$:

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$$

- Chain Rule: $\mathrm{y} = f(u), u = g(x)$:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Derivative of Common Functions

- $f(x) = c, \quad \frac{df(x)}{dx} = 0$
- $f(x) = x, \quad \frac{df(x)}{dx} = 1$
- $f(x) = cx, \quad \frac{df(x)}{dx} = c$
- $f(x) = x^n, \quad \frac{df(x)}{dx} = nx^{n-1}$
- $f(x) = e^x, \quad \frac{df(x)}{dx} = e^x$

- $f(x) = \ln(x), \quad \frac{df(x)}{dx} = \frac{1}{x}$
- $f(x) = \sin(x), \quad \frac{df(x)}{dx} = \cos(x)$
- $f(x) = \cos(x), \quad \frac{df(x)}{dx} = -\sin(x)$
- $f(x) = \tan(x), \quad \frac{df(x)}{dx} = \sec^2(x)$

Main Idea

- Symbolic differentiation is the automatic manipulation of expressions for obtaining derivative expressions carried out by applying derivative computation rules.

- When formulae are represented as data structures, symbolically differentiating an expression tree is a perfectly mechanistic process.

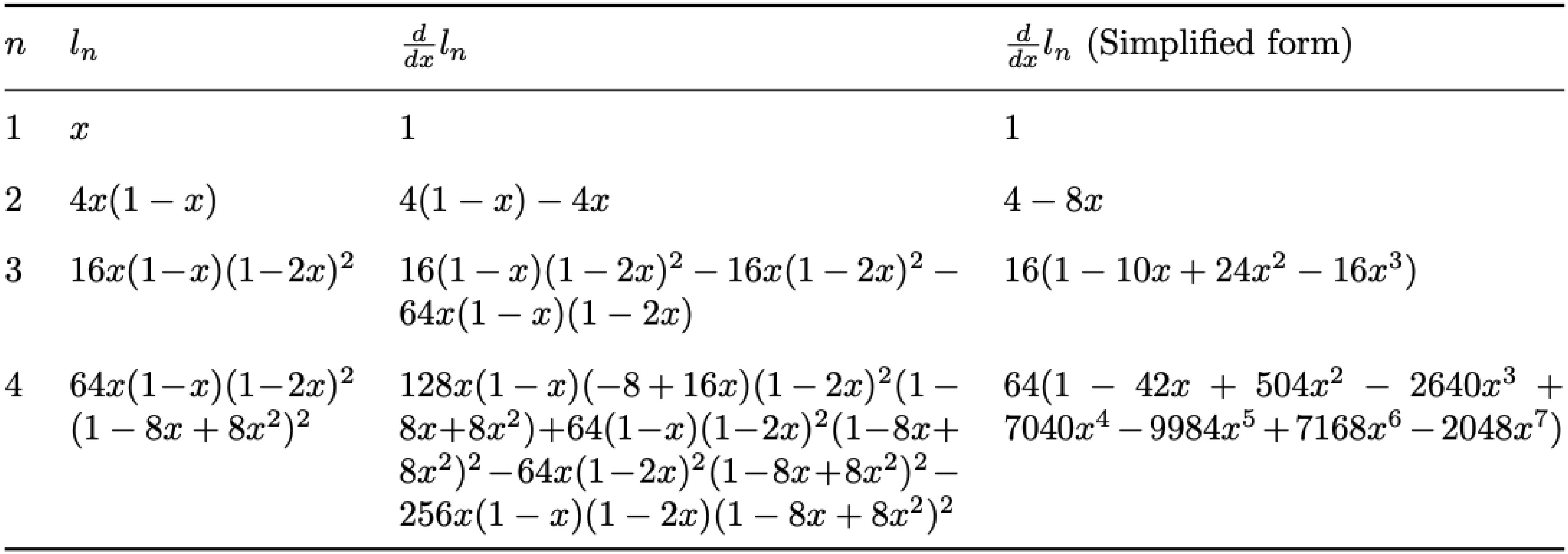
- This is realized in modern computer algebra systems such as Mathematica.

Problem

- Symbolic derivatives do not lend themselves to efficient runtime calculation of derivative values, as they can get exponentially larger than the expression whose derivative they represent.
- Expression swell: careless symbolic differentiation can easily produce exponentially large symbolic expressions that take correspondingly long to evaluate.

Expression Swell

Iterations of the logistic map $l_{n+1} = 4l_n(1 - l_n), l_1 = x$ and the corresponding derivatives of l_n with respect to x, illustrating expression swell.



n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

Automatic Differentiation

Main Idea

- An automatic differentiation (AD) system will convert the program into a sequence of elementary operations with specified routines for computing derivatives:
  - Apply symbolic differentiation at the elementary operation level;
  - Keep intermediate numerical results;
  - Combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition.

Notations

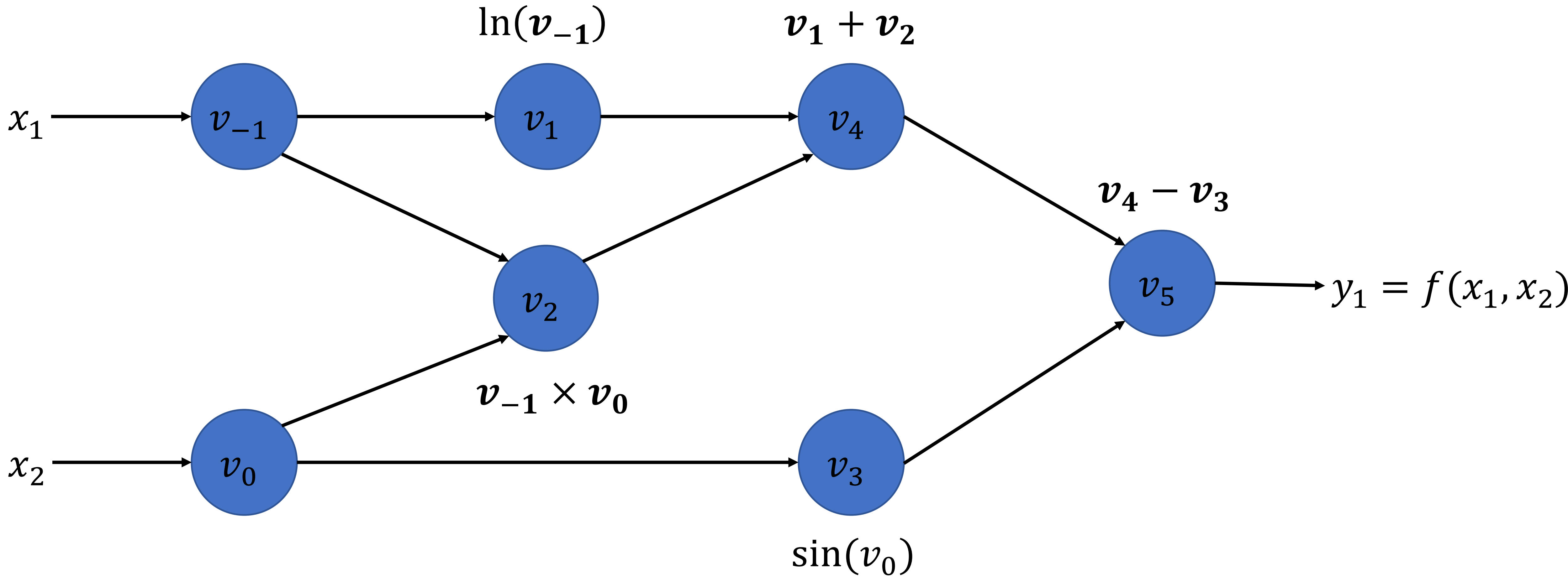
- The *Jacobian matrix* of a function $f\colon\ \mathbb{R}^n \rightarrow\ \mathbb{R}^m$ is defined by a $m \times n$ matrix noded by $\mathbf{J}$ where $\mathrm{J}_{ij} = \frac{\partial y_i}{\partial x_j}$, or explicitly:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Notations

- A function $f\colon\ \mathbb{R}^n \rightarrow\ \mathbb{R}^m$ is constucted using intermidate variable v_i such that:
    - Variable $v_{j-n} = x_j,\ j = 1, \dots, n$ are the input variables;
    - Variable $v_i,\ i = 1, \dots, l$ are the intermidate variables;
    - Variable $y_{m-k} = v_{l-k},\ k = 1, \dots, m$ are the output variables;

Example: $\boldsymbol{f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)}$



Forward Mode AD

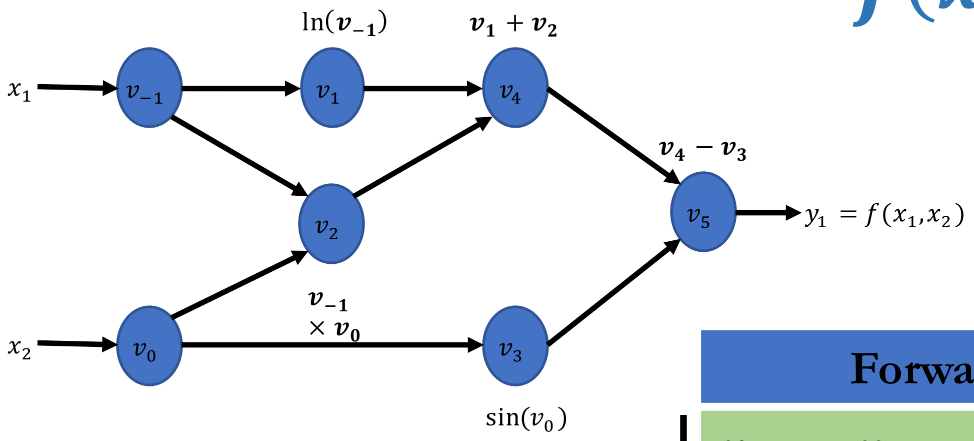
- For computing the derivative of f with respect to x_1, we start by associating with each intermediate variable v_i a derivative (tangent):

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

- Apply the chain rule to each elementary operation in the forward primal trace;
- Generate the corresponding tangent (derivative) trace;
- Evaluating the primals v_i in lockstep with their corresponding tangents $\dot{v}_i$ gives us the required derivative in the final variable $\dot{v}_5 = \frac{\partial y_1}{\partial x_1}$.

Forward Mode AD:

$$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$



Forward Primal Trace	
$v_{-1} = x_1$	$= 2$
$v_0 = x_2$	$= 5$
$v_1 = \ln(v_{-1})$	$= \ln(2) = 0.693$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5 = 10$
$v_3 = \sin v_0$	$= \sin 5 = 0.959$
$v_4 = v_1 + v_2$	$= 0.693 + 10$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$
$y_1 = v_5$	$= 11.652$

Forward Tangent (Derivative) Trace	
$\dot v_{-1} = \dot x_1$	$= 1$
$\dot v_0 = \dot x_2$	$= 0$
$\dot v_1 = \dot v_{-1} / v_{-1}$	$= 1/2$
$\dot v_2$ $= \dot v_{-1} \times v_0 + \dot v_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot v_3 = \dot v_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot v_4 = \dot v_1 + \dot v_2$	$= 0.5 + 5$
$\dot v_5 = \dot v_4 - \dot v_3$	$= 5.5 - 0$
$\dot y_1 = \dot v_5$	$= 5.5$

$\dot v_{-1} = \frac{\partial x_1}{\partial x_1}$=1

Forward Mode AD

- Compute the Jacobian of a function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ with n independent/input variable x_i and m dependent/output variable y_j:
  - Each forward pass of AD is initialized by setting only one of the input variable x_i and setting the rest to 0 (i.e., $\dot{\boldsymbol{x}} = \boldsymbol{e}_i$, where $\boldsymbol{e}_i$ is the i-th unit vector).
  - One exeucution of forward mode AD computes: $\dot{y}_j = \frac{\partial y_j}{\partial x_i}|_{\boldsymbol{x}=\boldsymbol{a}}, j = 1, \ldots, m$
  - Give us one columne of the Jacobian matrix at point $\boldsymbol{a}$ (the full jacobian can be computed by n evaluations):

$$\boldsymbol{J}_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}|_{\boldsymbol{x}=\boldsymbol{a}}$$

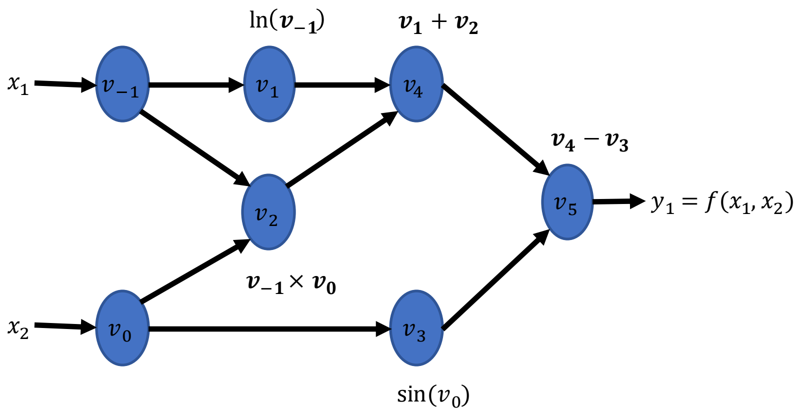
Reverse Mode AD

- Reverse mode AD propagates derivatives backward from a given output.
- We start by complementing each intermediate variable v_i with an adjoint (cotangent) representing the sensitivity of a considered output y_j with respect to changes in v_i:
$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$$
- In the first phase, the original function code is run forward, populating intermediate variables v_i and recording the dependencies in the computational graph.
- In the second phase, derivatives are calculated by propagating adjoints $\bar{v}_i$ in reverse, from the outputs to the inputs.

Chain rule in the multivariable case:

- $y = f(g_1(x), g_2(x), \dots, g_n(x))$;
- $\frac{\partial y}{\partial x} = \sum_{i=1}^{n} \frac{\partial y}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x}$.

Reverse Mode AD: $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



Forward Primal Trace	
$v_{-1} = x_1$	$= 2$
$v_0 = x_2$	$= 5$
$v_1 = \ln(v_{-1})$	$= \ln(2) = 0.693$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5 = 10$
$v_3 = \sin v_0$	$= \sin 5 = 0.959$
$v_4 = v_1 + v_2$	$= 0.693 + 10$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$
$y_1 = v_5$	$= 11.652$

The way v_{-1} Influences y is through v_1 and v_2:

$$\bar{v}_{-1} = \bar{v}_1 \frac{\partial v_1}{\partial v_0} + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$$

The way v_0 Influences y is through v_2 and v_3:

$$\bar{v}_0 = \bar{v}_2 \frac{\partial v_2}{\partial v_0} + \bar{v}_3 \frac{\partial v_3}{\partial v_0}$$

$$\bar{v}_4 = \frac{\partial y_1}{\partial v_4} = \frac{\partial y_1}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_4} = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$$

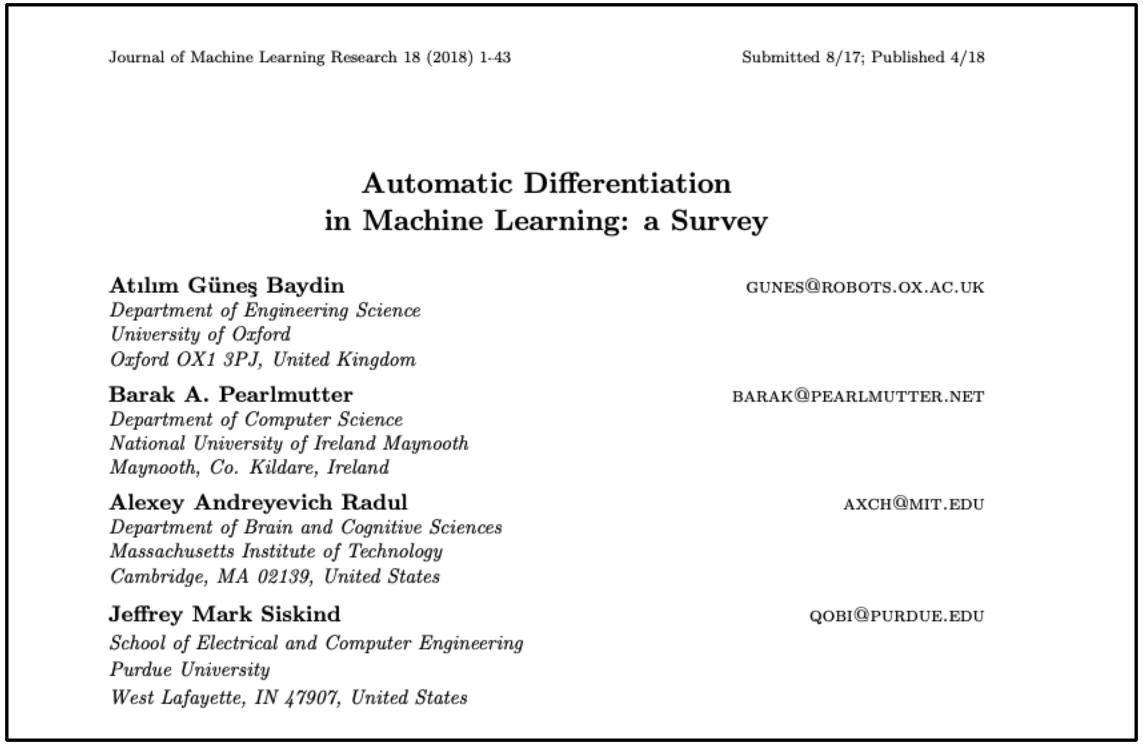
Reverse Adjoint (Derivative) Trace		
$\bar{x}_1 = \bar{v}_{-1}$		$= 5.5$
$\bar{x}_2 = \bar{v}_0$		$= 1.716$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}_1$		$= 1$

$$\bar{v}_5 = \bar{y}_1 = \frac{\partial y_1}{\partial y_1} = 1$$

Reverse Mode AD

- Compute the Jacobian of a function $f\colon \mathbb{R}^n \rightarrow \mathbb{R}^m$ with n independent/input variable x_i and m dependent/output variable y_j.
- An important advantage of the reverse mode is that it is significantly less costly to evaluate (in terms of operation count) than the forward mode for functions with a large number of inputs.
- In the extreme case of $f\colon \mathbb{R}^n \rightarrow \mathbb{R}$ only one application of the reverse mode is sufficient to compute the full gradient.
- Because machine learning practice principally involves the gradient of a scalar-valued objective with respect to a large number of parameters, this establishes the reverse mode as the main technique in ML systems.

Further Reading



Journal of Machine Learning Research 18 (2018) 1-43 Submitted 8/17; Published 4/18

Automatic Differentiation in Machine Learning: a Survey

Atılım Güneş Baydin GUNES@ROBOTS.OX.AC.UK
Department of Engineering Science
University of Oxford
Oxford OX1 3PJ, United Kingdom

Barak A. Pearlmutter BARAK@PEARLMUTTER.NET
Department of Computer Science
National University of Ireland Maynooth
Maynooth, Co. Kildare, Ireland

Alexey Andreyevich Radul AXCH@MIT.EDU
Department of Brain and Cognitive Sciences
Massachusetts Institute of Technology
Cambridge, MA 02139, United States

Jeffrey Mark Siskind QOBI@PURDUE.EDU
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, United States

- Automatic differentiation in machine learning: a survey (https://arxiv.org/abs/1502.05767)

Auto-Diff for a Linear Layer

General Chain Rule

- $y = f(\boldsymbol{x}): \mathbb{R}^n \to \mathbb{R};$
- $\nabla f(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n$

- $\boldsymbol{y} = f(\boldsymbol{x}): \mathbb{R}^n \to \mathbb{R}^m;$
- $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$

- $\boldsymbol{y} = f(\boldsymbol{x}): \mathbb{R}^n \to \mathbb{R}^m;$
- $\boldsymbol{z} = g(\boldsymbol{y}): \mathbb{R}^m \to \mathbb{R}^k;$
- $\boldsymbol{z} = f \circ g(\boldsymbol{x}): \mathbb{R}^n \to \mathbb{R}^k;$
- $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \in \mathbb{R}^{k \times n}$

Linear Layer: Forward

- Forward computation of a linear layer: $\boldsymbol{Y} = \boldsymbol{XW}$
  - Input: $\boldsymbol{X} \in \mathbb{R}^{B \times H_1}$
  - Weight matrix: $\boldsymbol{W} \in \mathbb{R}^{H_1 \times H_2}$
  - Output: $\boldsymbol{Y} \in \mathbb{R}^{B \times H_2}$
- After the forward pass, we assume that the output will be used in other parts of the model, and will eventually be used to compute a scalar loss $L \in \mathbb{R}$.

Linear Layer: Backward

- During the backward pass through the linear layer, we assume that the derivative $\frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{B \times H_2}$ has already been computed and given by:

$$\frac{\partial L}{\partial \boldsymbol{Y}} = \begin{bmatrix} \frac{\partial L}{\partial Y_{1,1}} & \cdots & \frac{\partial L}{\partial Y_{1,H_2}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial Y_{B,1}} & \cdots & \frac{\partial L}{\partial Y_{B,H_2}} \end{bmatrix}$$

- Our goal is to use $\frac{\partial L}{\partial \boldsymbol{Y}}$ to compute $\frac{\partial L}{\partial \boldsymbol{X}}$ and $\frac{\partial L}{\partial \boldsymbol{W}}$.

Linear Layer: Backward

- By the general chain rule, we have:
  - $\frac{\partial L}{\partial \boldsymbol{X}} = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{X}}$
  - $\frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{W}}$
- But, we do not want to explicitly compute $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{X}}$ and $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{W}}$.
- *How can we compute* $\frac{\partial L}{\partial \boldsymbol{X}}$ *and* $\frac{\partial L}{\partial \boldsymbol{W}}$ *without explicitly computing* $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{X}}$ *and* $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{W}}$?

The Jacbian matrices are two large: $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{X}} \in \mathbb{R}^{BH_2 \times BH_1}, \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{W}} \in \mathbb{R}^{BH_2 \times H_1 H_2}$

Linear Layer: Backward

- We know that $\frac{\partial L}{\partial \boldsymbol{X}}$ should have the same shape as $\boldsymbol{X} \in \mathbb{R}^{B \times H_1}$:

$$\frac{\partial L}{\partial \boldsymbol{X}} = \begin{bmatrix} \frac{\partial L}{\partial X_{1,1}} & \cdots & \frac{\partial L}{\partial X_{1,H_1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial X_{B,1}} & \cdots & \frac{\partial L}{\partial X_{B,H_1}} \end{bmatrix}$$

- Let us first try to compute $\frac{\partial L}{\partial X_{1,1}}$, by the chain rule, we have:

$$\frac{\partial L}{\partial X_{1,1}} = \sum_{i=1}^{B} \sum_{j=1}^{H_2} \frac{\partial L}{\partial Y_{i,j}} \frac{\partial Y_{i,j}}{\partial X_{1,1}} = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial X_{1,1}}$$

We have: $\frac{\partial L}{\partial X_{1,1}} \in \mathbb{R}, \frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{B \times H_2}, \frac{\partial \boldsymbol{Y}}{\partial X_{1,1}} \in \mathbb{R}^{B \times H_2}$, so this a <u>inner prodcut</u>.

Linear Layer: Backward

- Since $\frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{B \times H_2}$ has already been given, we only need to compute $\frac{\partial \boldsymbol{Y}}{\partial X_{1,1}}$

- Recall that $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W} = \begin{bmatrix} X_{1,1} & \cdots & X_{1,H_1} \\ \vdots & \ddots & \vdots \\ X_{B,1} & \cdots & X_{B,H_1} \end{bmatrix} \begin{bmatrix} W_{1,1} & \cdots & W_{1,H_2} \\ \vdots & \ddots & \vdots \\ W_{H_1,1} & \cdots & W_{H_1,H_2} \end{bmatrix}$

- $\boldsymbol{Y} = \begin{bmatrix} \sum_{k=1}^{H_1} X_{1k} W_{k1} & \cdots & \sum_{k=1}^{H_1} X_{1k} W_{kH_2} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{H_1} X_{Bk} W_{kH_2} & \cdots & \sum_{k=1}^{H_1} X_{Bk} W_{kH_2} \end{bmatrix}$

- It is easy to check: $\frac{\partial \boldsymbol{Y}}{\partial X_{1,1}} = \begin{bmatrix} W_{11} & \cdots & W_{1H_2} \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$

Linear Layer: Backward

- So the inner prodcut of $\frac{\partial L}{\partial X_{1,1}} = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial X_{1,1}}$ can be computed by:

$$\begin{bmatrix} \frac{\partial L}{\partial Y_{1,1}} & \cdots & \frac{\partial L}{\partial Y_{1,H_2}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial Y_{B,1}} & \cdots & \frac{\partial L}{\partial Y_{B,H_2}} \end{bmatrix} \odot \begin{bmatrix} W_{1,1} & \cdots & W_{1,H_2} \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} = \sum_{k=1}^{H_2} \frac{\partial L}{\partial Y_{1,k}} W_{1k}$$

- Generally, we have $\frac{\partial L}{\partial X_{i,j}} = \sum_{k=1}^{H_2} \frac{\partial L}{\partial Y_{i,k}} W_{jk}$
- Thus, we have $\frac{\partial L}{\partial \boldsymbol{X}} = \frac{\partial L}{\partial \boldsymbol{Y}} \boldsymbol{W}^T$

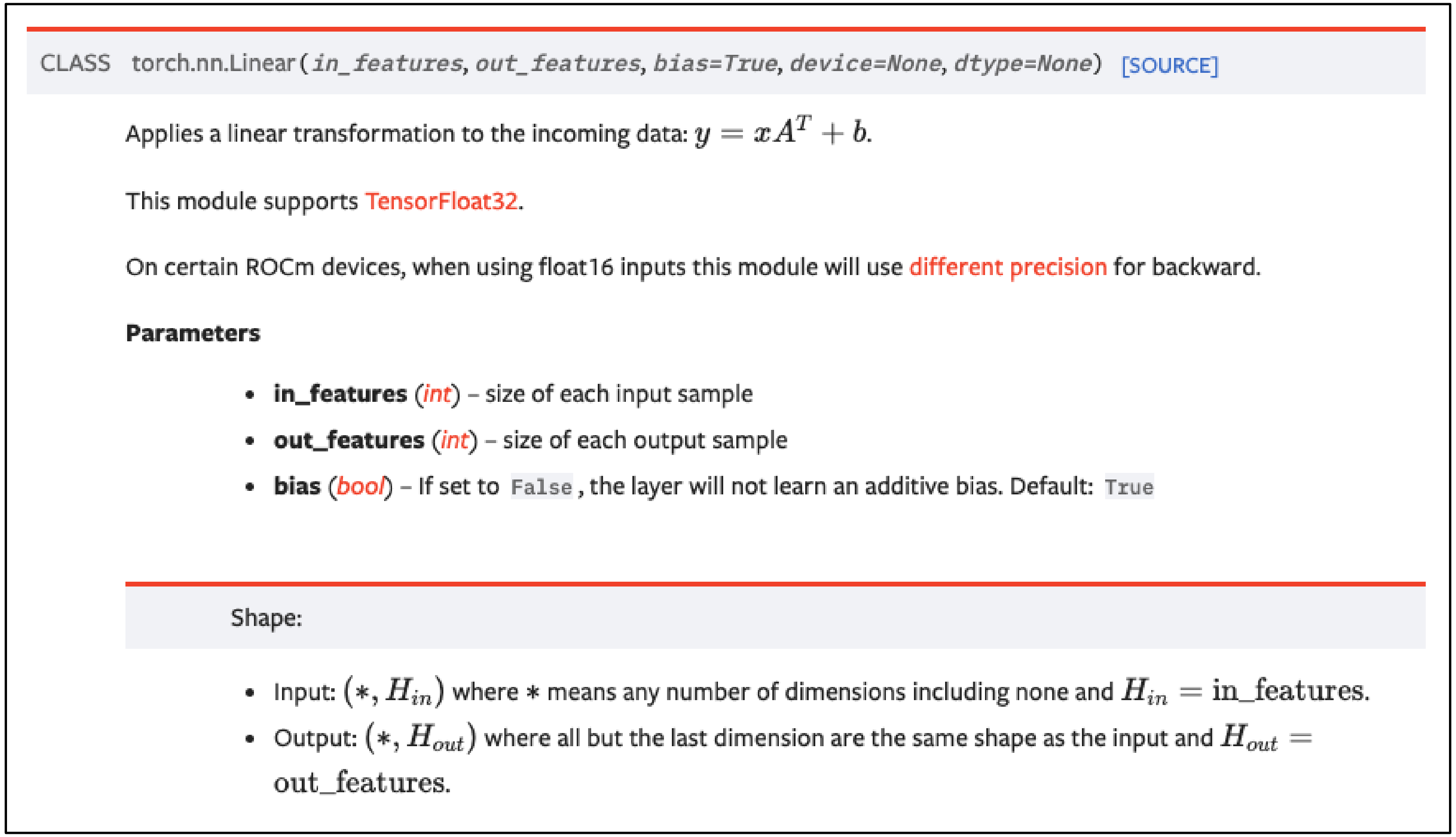
Linear Layer: Backward

- Using the same strategy of thinking about components one at a time, we can derive a similarly simple equation to compute $\frac{\partial L}{\partial \boldsymbol{W}}$ without explicitly forming the Jacobian matrix of $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{W}}$.
- Leave this as a problem in Homework 1.
- Eventually, we will have $\frac{\partial L}{\partial \boldsymbol{W}} = \boldsymbol{X}^{\boldsymbol{T}} \frac{\partial L}{\partial \boldsymbol{Y}}$.

Summary of a Linear Layer Computation

- Forward computation of a linear layer: $\boldsymbol{Y} = \boldsymbol{XW}$
  - Given input: $\boldsymbol{X} \in \mathbb{R}^{B \times D_1}$
  - Given weight matrix: $\boldsymbol{W} \in \mathbb{R}^{D_1 \times D_2}$
  - Compute output: $\boldsymbol{Y} \in \mathbb{R}^{B \times D_2}$
- Backward computation of a linear layer:
  - Given gradients w.r.t output: $\frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{B \times H_2}$
  - Compute gradients w.r.t weight matrix: $\frac{\partial L}{\partial \boldsymbol{W}} = \boldsymbol{X^T} \frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{B \times H_2}$
  - Compute gradients w.r.t input: $\frac{\partial L}{\partial \boldsymbol{X}} = \frac{\partial L}{\partial \boldsymbol{Y}} \boldsymbol{W}^T \in \mathbb{R}^{B \times H_2}$

Linear Layer in PyTorch



CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$.

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

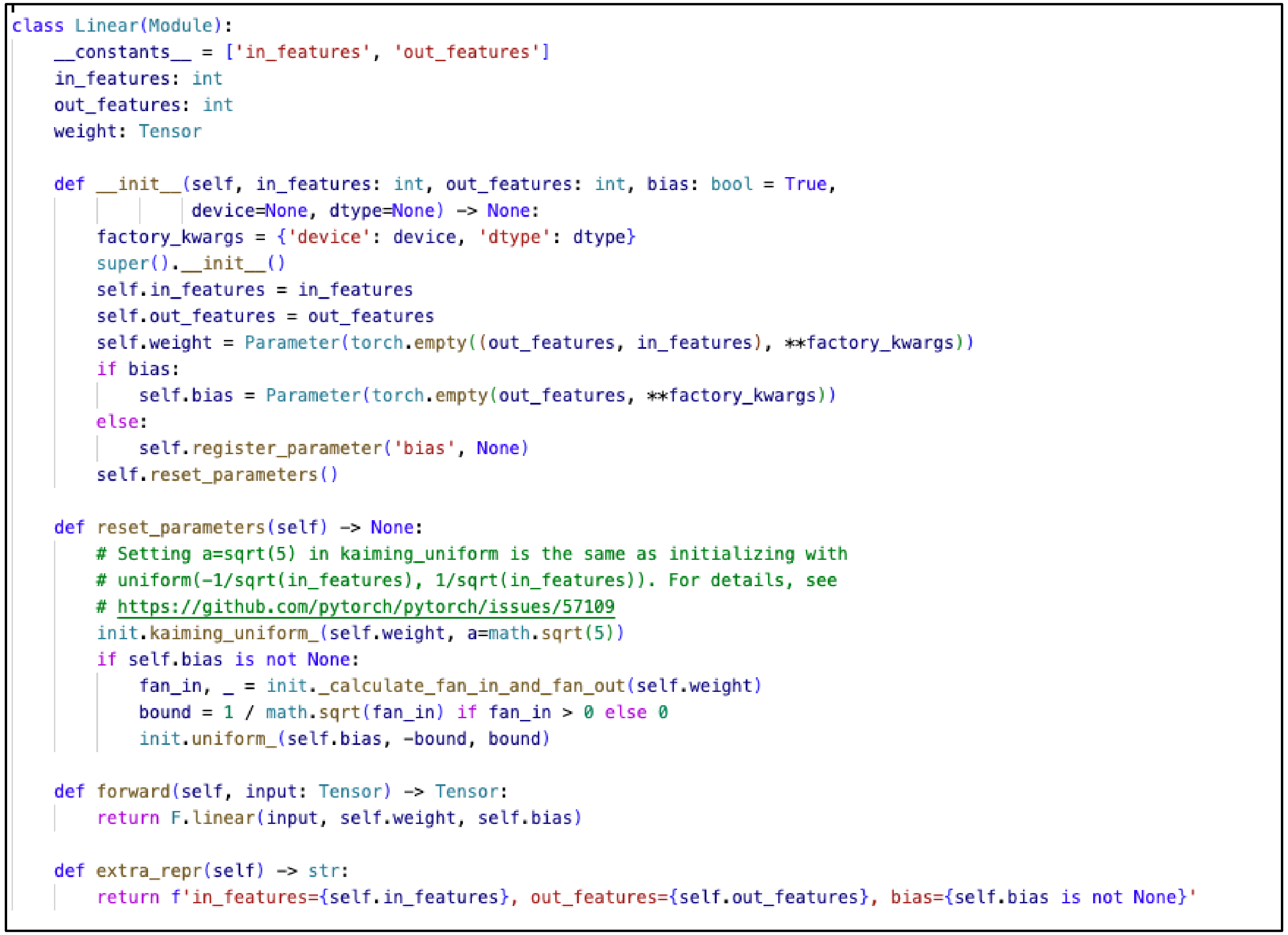
Parameters

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Linear Layer in PyTorch



```
class Linear(Module):
    __constants__ = ['in_features', 'out_features']
    in_features: int
    out_features: int
    weight: Tensor

    def __init__(self, in_features: int, out_features: int, bias: bool = True,
                 device=None, dtype=None) -> None:
        factory_kwargs = {'device': device, 'dtype': dtype}
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.empty((out_features, in_features), **factory_kwargs))
        if bias:
            self.bias = Parameter(torch.empty(out_features, **factory_kwargs))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self) -> None:
        # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
        # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
        # https://github.com/pytorch/pytorch/issues/57109
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
            init.uniform_(self.bias, -bound, bound)

    def forward(self, input: Tensor) -> Tensor:
        return F.linear(input, self.weight, self.bias)

    def extra_repr(self) -> str:
        return f'in_features={self.in_features}, out_features={self.out_features}, bias={self.bias is not None}'
```

Verify what we have calculated.

Define the Model

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28*28, 512, False)
        self.fc2 = nn.Linear(512, 512, False)
        self.fc3 = nn.Linear(512, 10, False)
        self.register_buffer('fc2_input_act', torch.zeros(batch_size, 512))
        self.register_buffer('fc2_output_act', torch.zeros(batch_size, 512))

    def forward(self, x):
        x = self.flatten(x)
        self.fc2_input_act = F.relu(self.fc1(x))
        self.fc2_input_act.retain_grad()
        self.fc2_output_act = self.fc2(self.fc2_input_act)
        self.fc2_output_act.retain_grad()
        x = F.relu(self.fc2_output_act)
        logits = self.fc3(x)
        return logits

model = NeuralNetwork()
```

Verify what we have calculated.

Code

```
loss.backward()

print("Shape of X:", model.fc2_input_act.shape)
print("Shape of dL/dX:", model.fc2_input_act.grad.shape)
print("Shape of W:", model.fc2_input_act.shape)
print("Shape of dL/dW:", model.fc2.weight.grad.shape)
print("Shape of Y:", model.fc2_output_act.shape)
print("Shape of dL/dY:",model.fc2_output_act.grad.shape)

diff1 = torch.sum(torch.abs(model.fc2_input_act.grad -
                            torch.matmul(model.fc2_output_act.grad,
                                         model.fc2.weight)))
print("Check dL/dX = dL/dY W^T, diff1=", diff1.item())

diff2 = torch.sum(torch.abs(torch.transpose(model.fc2.weight.grad,0,1) -
                            torch.matmul(torch.transpose(model.fc2_input_act,0,1),
                                         model.fc2_output_act.grad)))
print("Check dL/dW = X^T dL/dY, diff2=", diff2.item())
```

Output

```
Shape of X: torch.Size([64, 512])
Shape of dL/dX: torch.Size([64, 512])
Shape of W: torch.Size([64, 512])
Shape of dL/dW: torch.Size([512, 512])
Shape of Y: torch.Size([64, 512])
Shape of dL/dY: torch.Size([64, 512])
Check dL/dX = dL/dY W^T, diff1= 0.0
Check dL/dW = X^T dL/dY, diff2= 0.0
```

RELAXED
SYSTEM LAB

References

- Automatic differentiation in machine learning: a survey (https://arxiv.org/abs/1502.05767)
- http://cs231n.stanford.edu/handouts/linear-backprop.pdf
- https://pytorch.org/docs/stable/generated/torch.nn.Linear.html