THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

RELAXED SYSTEM LAB

Automatic Differentiation

COMP4551

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 $f\colon \mathbb{R}^n \to \mathbb{R}$:

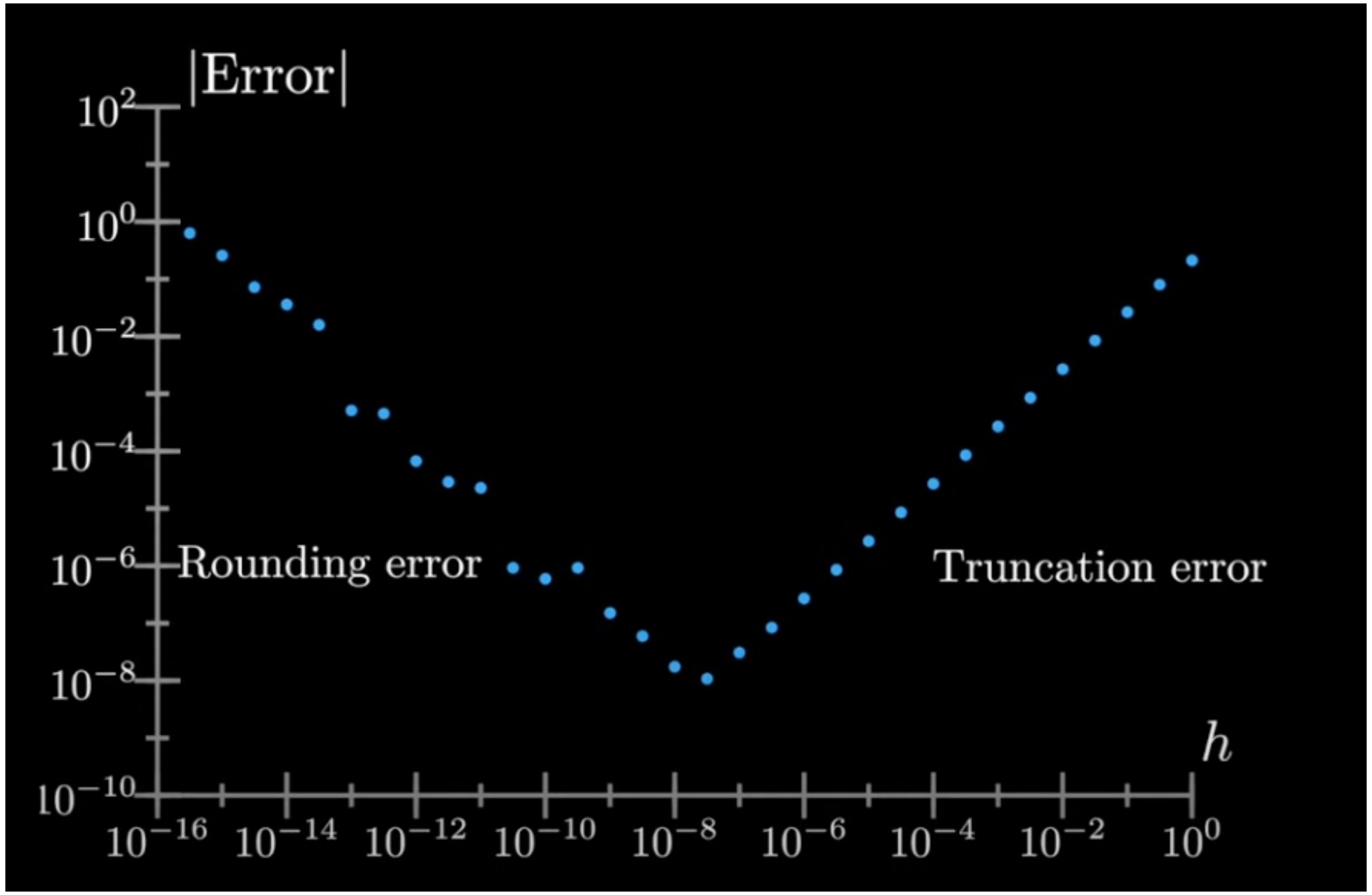
$$\frac{\partial f}{\partial x_i} = \lim_{\epsilon \to 0} \frac{f(\boldsymbol{x} + \epsilon \boldsymbol{e}_i) - f(\boldsymbol{x})}{\epsilon} \approx \frac{f(\boldsymbol{x} + h\boldsymbol{e}_i) - f(\boldsymbol{x})}{h}$$

- $\boldsymbol{e}_i$ 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 $f(x): \mathbb{R} \to \mathbb{R}, \quad g(x): \mathbb{R} \to \mathbb{R}$:
- 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.

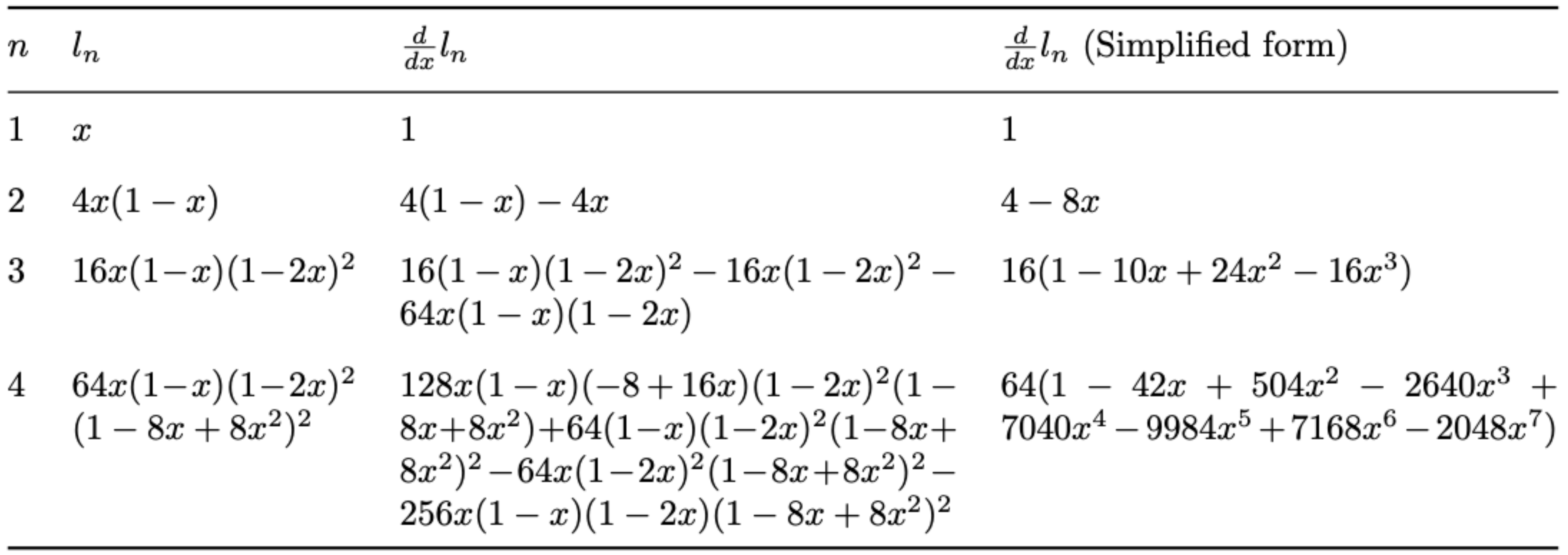
- This is realized in modern computer algebra systems such as Mathematica.

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

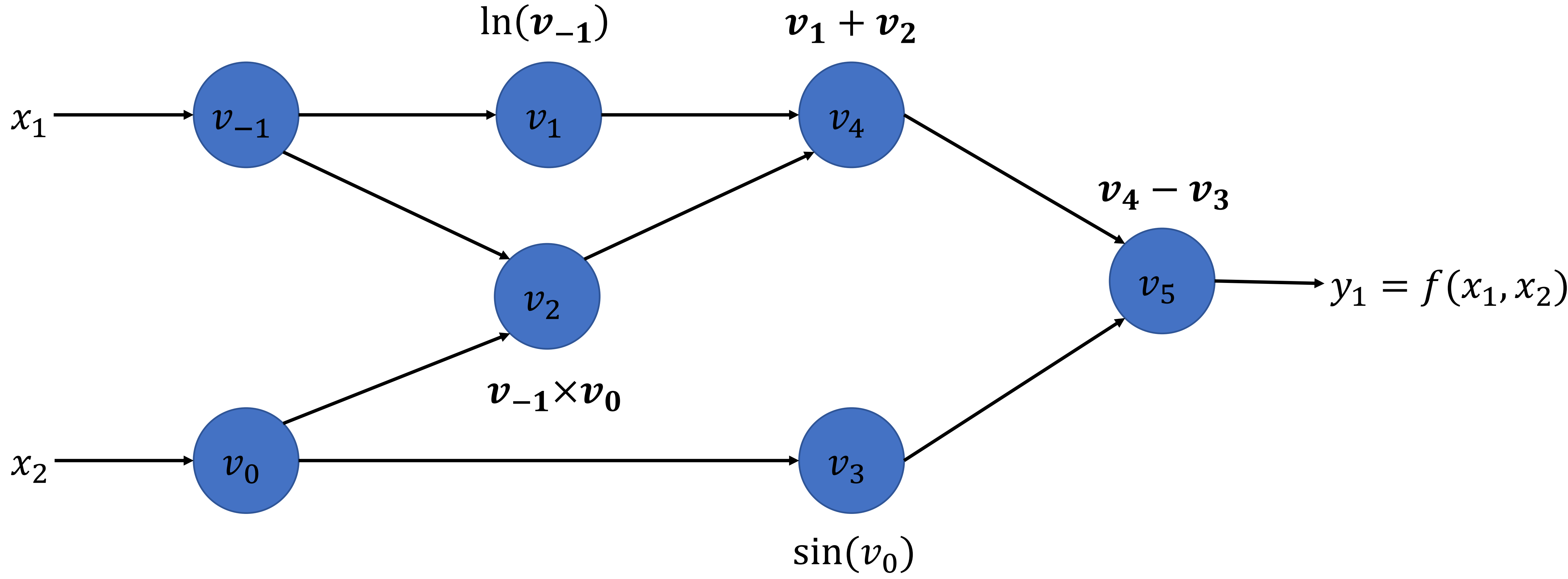
- The <u>*Jacobian matrix*</u> of a function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ is defined by a $m \times n$ matrix noded by $\mathbf{J}$ where $\mathrm{J}_{ij} = \frac{\partial y_i}{\partial x_j}$, or explicitly:

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

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 $y_{m-k} = v_{l-k},\ \ k = 0, \dots, m-1$ are the output variables;

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



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):

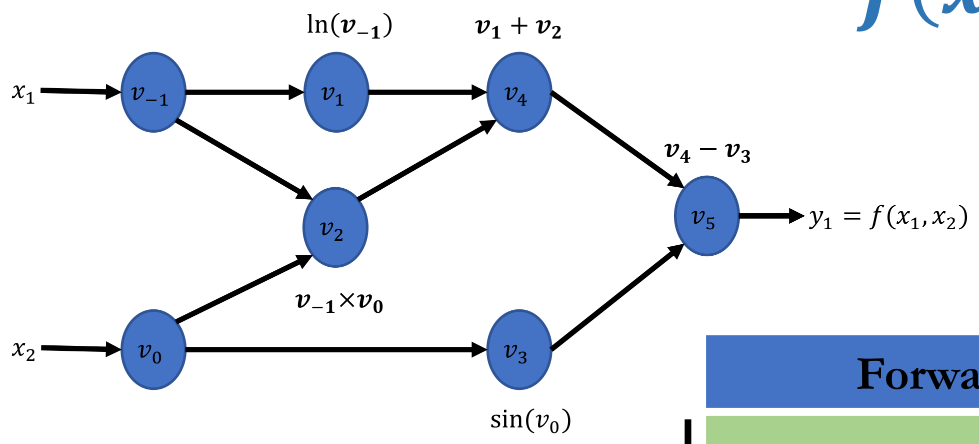
$$\dot{v}_i = \frac{\partial v_i}{\partial x_1}$$

- 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:

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

RELAXED SYSTEM LAB



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 $f\colon \mathbb{R}^n \rightarrow \mathbb{R}^m$ 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:

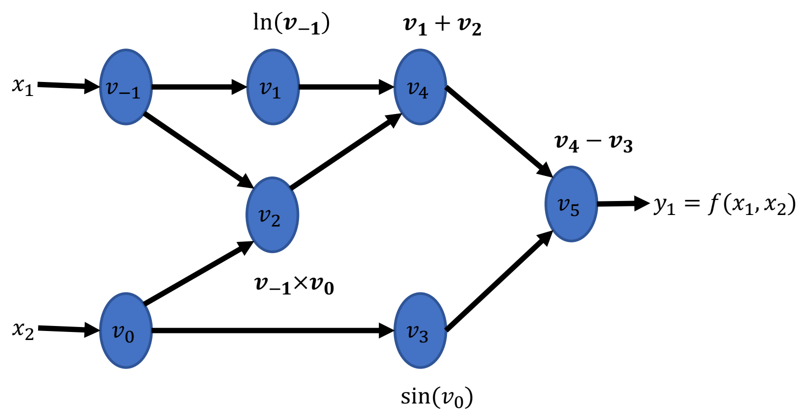
$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$$

- 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:

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

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



RELAXED SYSTEM LAB

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:

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

The way v_0 Influences y is through v_2 and v_3:

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

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

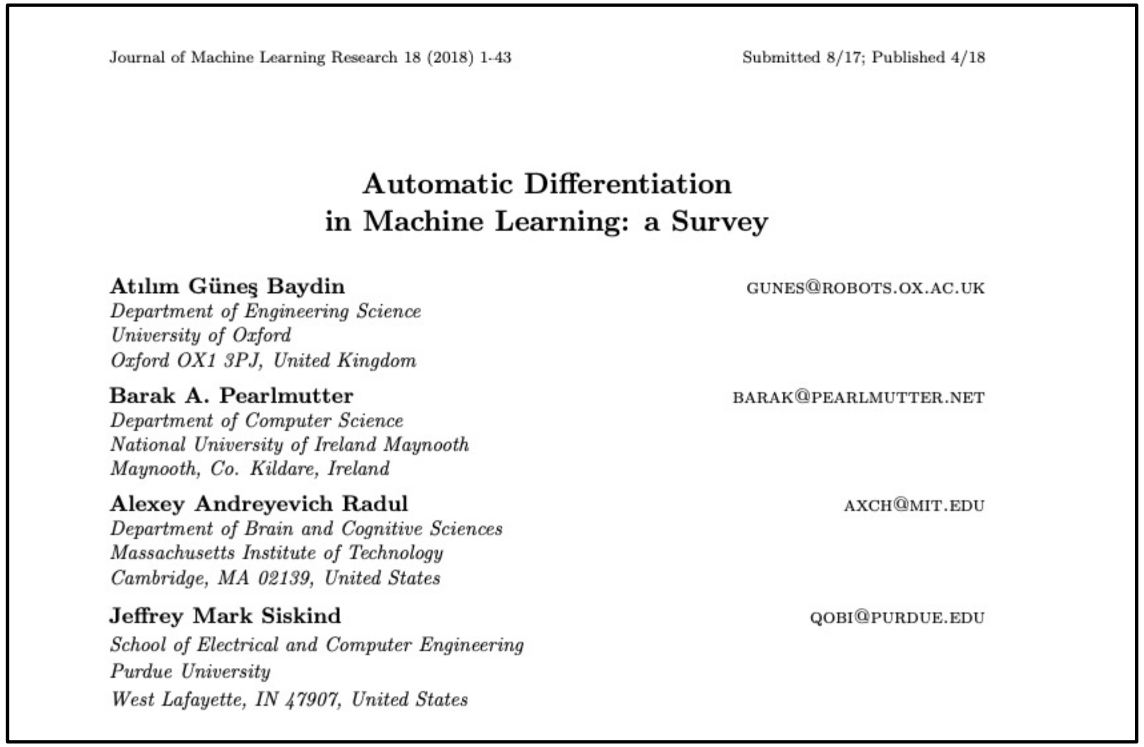
Reverse Adjoint (Derivative) Trace		
$\bar{x}_1 = \bar{v}_{-1}$		$= 5.5$
$\bar{x}_2 = \bar{v}_0$		$= 1.716$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$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

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

Reverse Mode AD

- Compute the Jacobian of a function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ 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 $f\colon \mathbb{R}^n \to \mathbb{R}$ 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

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

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

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

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 $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{W}}$.

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

- *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* $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{W}}$?

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 **inner prodcut**.

Linear Layer: Backward

RELAXED
SYSTEM LAB

- 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 $\boldsymbol{Y} = \boldsymbol{XW} = \begin{bmatrix} X_{1,1} & \cdots & X_{1,H_1} \\ \vdots & \ddots & \vdots \\ X_{B,1} & \cdots & X_{B,H_1} \end{bmatrix} \begin{bmatrix} W_{1,1} & \cdots & W_{1,H_2} \\ \vdots & \ddots & \vdots \\ W_{H_1,1} & \cdots & W_{H_1,H_2} \end{bmatrix}$
- $\boldsymbol{Y} = \begin{bmatrix} \sum_{k=1}^{H_1} X_{1k}W_{k1} & \cdots & \sum_{k=1}^{H_1} X_{1k}W_{kH_2} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{H_1} X_{Bk}W_{kH_2} & \cdots & \sum_{k=1}^{H_1} X_{Bk}W_{kH_2} \end{bmatrix}$
- 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:

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

- 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

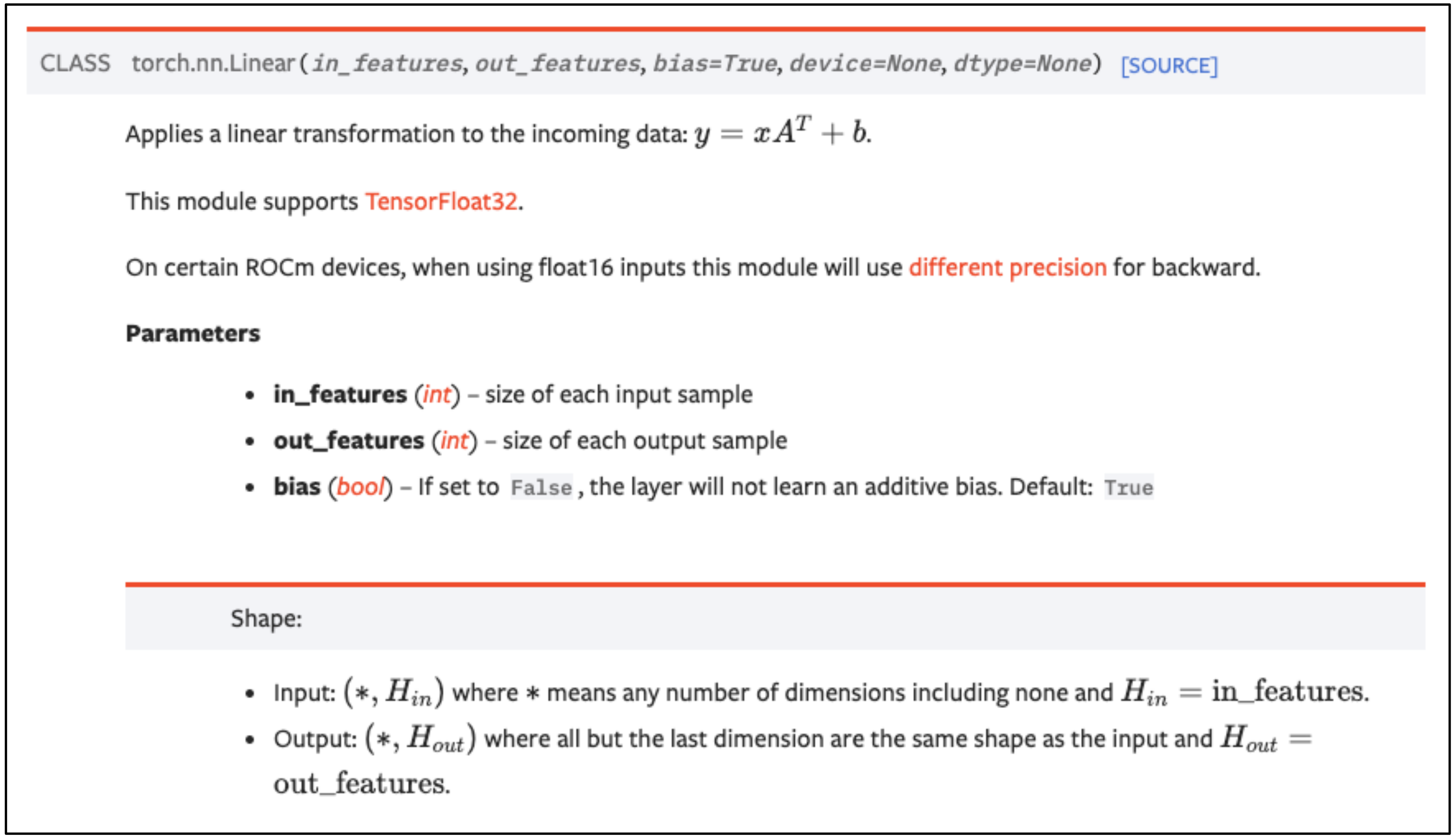
- Using the same strategy of thinking about components one at a time, we can derive a similarly simple equation to compute $\frac{\partial L}{\partial \boldsymbol{W}}$ without explicitly forming the Jacobian matrix of $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{W}}$.
- Leave this as a problem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

RELAXED SYSTEM LAB

- Forward computation of a linear layer: $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}$
  - 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: $\frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{B \times D_2}$
  - Compute gradients w.r.t weight matrix: $\frac{\partial L}{\partial \boldsymbol{W}} = \boldsymbol{X}^{\boldsymbol{T}} \frac{\partial L}{\partial \boldsymbol{Y}} \in \mathbb{R}^{\mathrm{D}_1 \times D_2}$
  - Compute gradients w.r.t input: $\frac{\partial L}{\partial \boldsymbol{X}} = \frac{\partial L}{\partial \boldsymbol{Y}} \boldsymbol{W}^{T} \in \mathbb{R}^{B \times D_1}$

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.

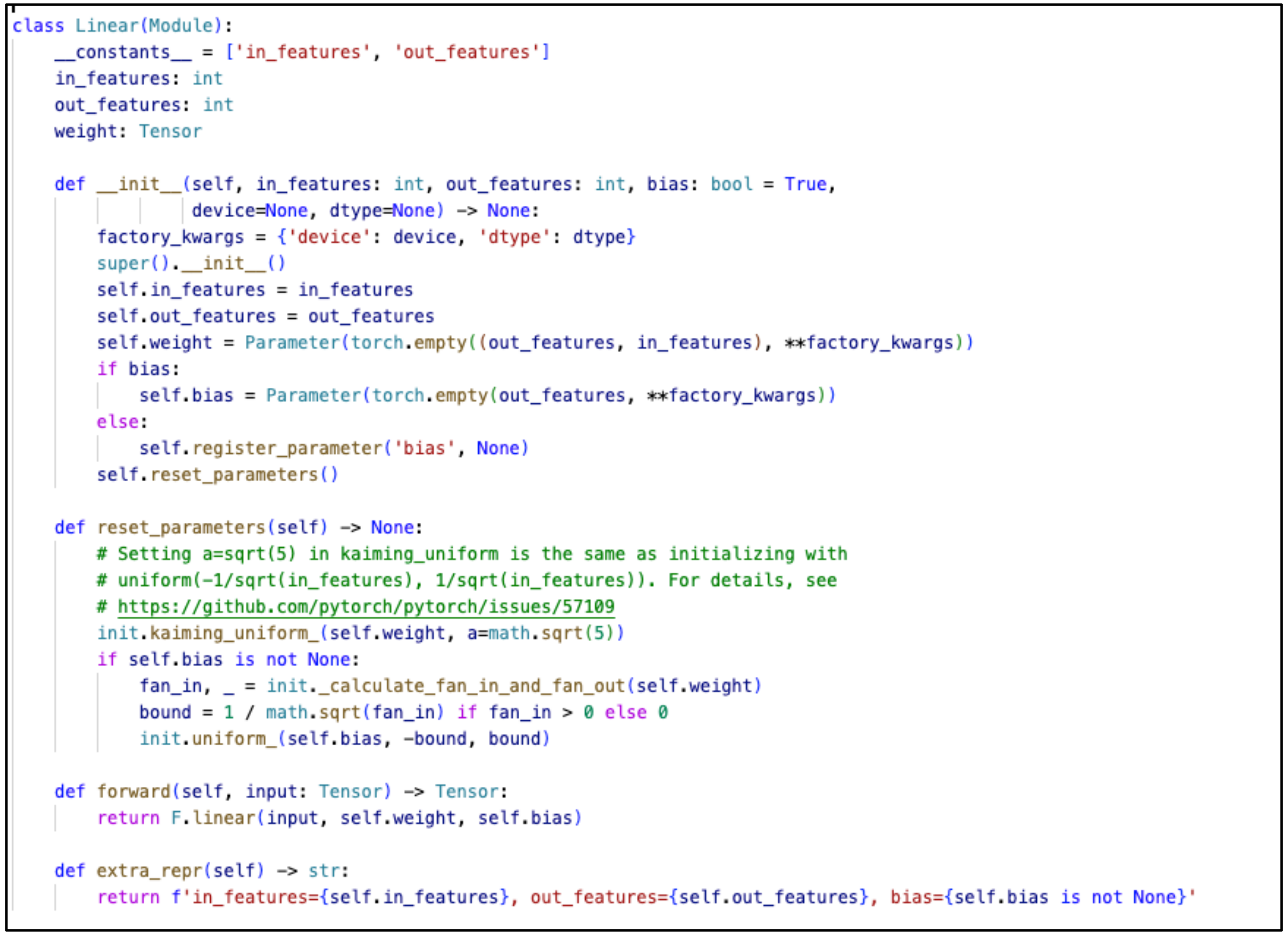
Parameters

- **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.

RELAXED SYSTEM LAB

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

```python
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.weight.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([512, 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

PyTorch Autograd

PyTorch Autograd Overview

- PyTorch has a built-in differentiation engine called **torch.autograd**. It supports the automatic computation of gradients for any computational graph.
- Autograd is a reverse automatic differentiation system.
  - Autograd records a graph of the operations (**Function** object);
  - It is a directed acyclic graph (DAG) :
    - Leaves are the input tensors;
    - Roots are the output tensors.
  - By tracing this graph from roots to leaves, automatically compute the gradients using the chain rule.

PyTorch Autograd Overview

- In a forward pass, **autograd** does two things simultaneously:
  - Run the requested operation to compute a resulting tensor
  - Maintain the operation's gradient function in the DAG.
- The backward pass kicks off when **.backward( )** is called on the DAG root. **autograd** then:
  - Compute the gradients from each **.grad_fn**,
  - Accumulate them in the respective tensor's **.grad** attribute
  - Use the chain rule to propagate all the way to the leaf tensors.

PyTorch Module

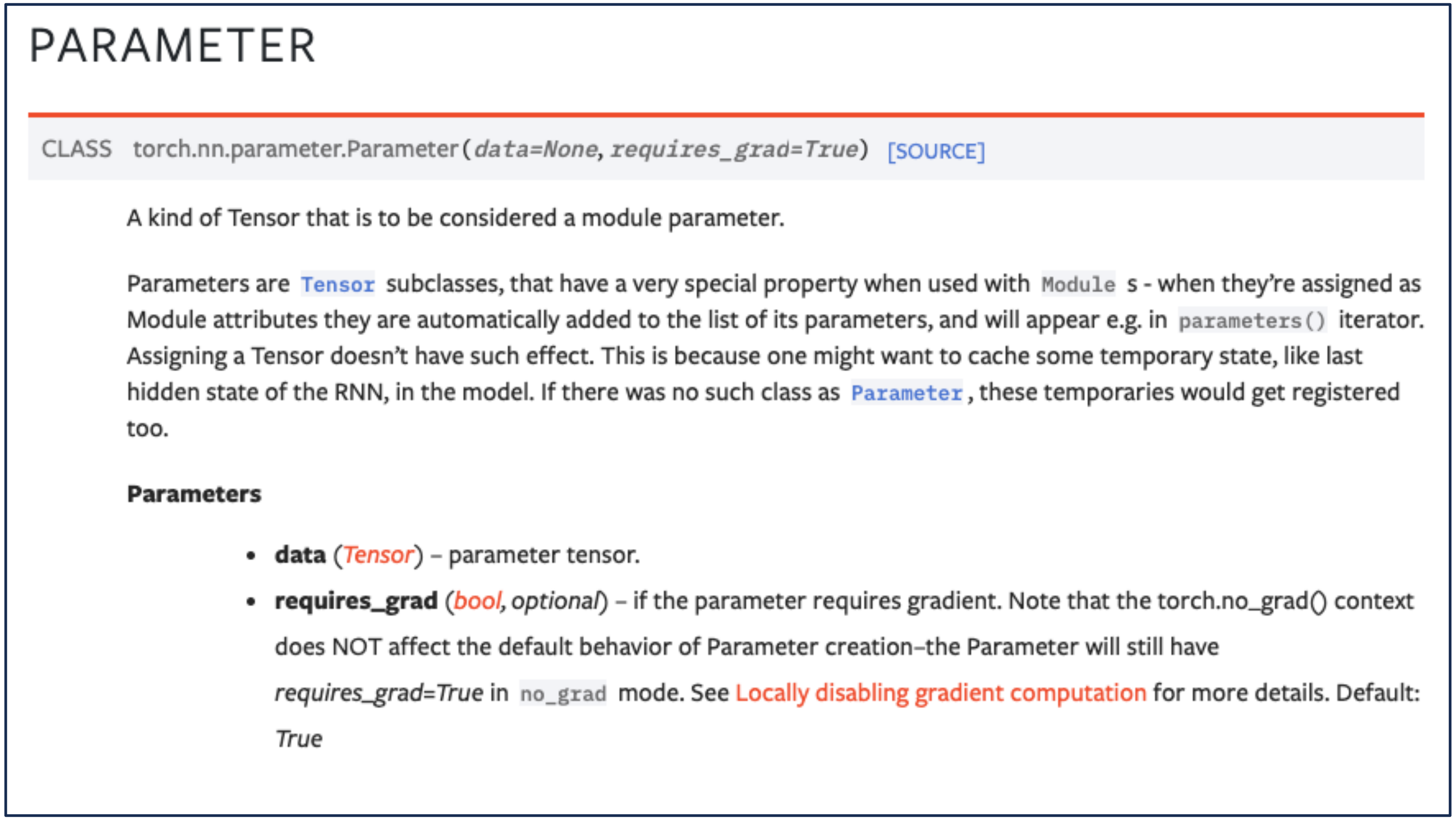
- PyTorch uses modules to represent neural networks. Modules are:
  - Building blocks of stateful computation. PyTorch provides a robust library of modules and makes it simple to define new custom modules, allowing for easy construction of elaborate, multi-layer neural networks.
  - Tightly integrated with PyTorch's autograd system. Modules make it simple to specify learnable parameters for PyTorch's Optimizers to update.
  - Easy to work with and transform. Modules are straightforward to save and restore, transfer between CPU / GPU / TPU devices, prune, quantize, and more.

PyTorch Module

- Import classes：
  - Parameter: A kind of Tensor to be considered a module parameter.
  - ParameterList: holds parameters in a list.
  - ParameterDict: holds parameters in a dictionary.

  - Module: base class for all neural network modules.
  - Sequential: a sequential container.
  - ModuleList: holds submodules in a list.
  - ModuleDict: holds submodules in a dictionary.

nn.Parameter


PARAMETER

CLASS torch.nn.parameter.Parameter(*data=None*, *requires_grad=True*) [SOURCE]

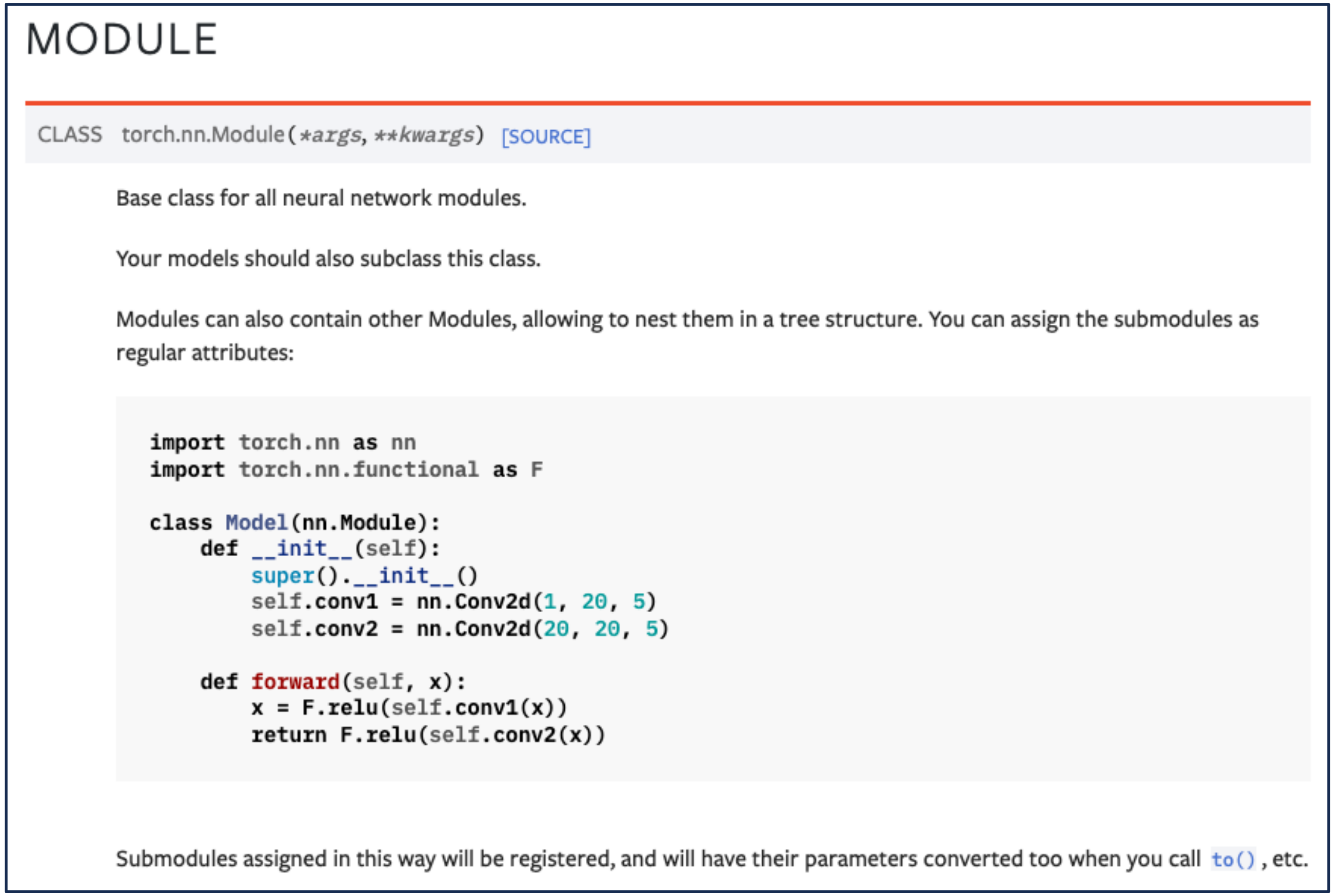
A kind of Tensor that is to be considered a module parameter.

Parameters are `Tensor` subclasses, that have a very special property when used with `Module` s - when they're assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in `parameters()` iterator. Assigning a Tensor doesn't have such effect. This is because one might want to cache some temporary state, like last hidden state of the RNN, in the model. If there was no such class as `Parameter`, these temporaries would get registered too.

Parameters

- **data** (*Tensor*) – parameter tensor.
- **requires_grad** (*bool, optional*) – if the parameter requires gradient. Note that the torch.no_grad() context does NOT affect the default behavior of Parameter creation–the Parameter will still have *requires_grad=True* in `no_grad` mode. See Locally disabling gradient computation for more details. Default: *True*

nn.Module



MODULE

CLASS torch.nn.Module(**args*, ***kwargs*) [SOURCE]

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```python
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Define A Simple Custom Model

Define A Simple Custom Model

```
import torch
from torch import nn

class MyLinear(nn.Module):
  def __init__(self, in_features, out_features):
    super().__init__()
    self.weight = nn.Parameter(torch.randn(in_features, out_features))

  def forward(self, input):
    return torch.matmul(input, self.weight)
```

- Inherits from the base Module class. All modules should subclass **Module** for composability with other modules.
- Defines some "state" that is used in computation. Here, the state consists of randomly-initialized weight tensor that define the computation. Because it is defined as a **Parameter**, it is registered for the module and will automatically be tracked and returned from calls to **parameters( )**. Parameters can be considered the "learnable" aspects of the module's computation. Note that modules are not required to have state and can also be stateless.
- Defines a **forward( )** function that performs the computation. For this MyLinear module, the input is matrix-multiplied with the weight parameter to produce the output. More generally, the **forward( )** implementation for a module can perform arbitrary computation involving any number of inputs and outputs.

Define A Simple Custom Model

RELAXED SYSTEM LAB

Define A Simple Custom Model

```
import torch
from torch import nn

class MyLinear(nn.Module):
  def __init__(self, in_features, out_features):
    super().__init__()
    self.weight = nn.Parameter(torch.randn(in_features, out_features))

  def forward(self, input):
    return torch.matmul(input, self.weight)


m = MyLinear(4, 3)
sample_input = torch.randn(4)
print(m(sample_input))

# tensor([-0.7689,  1.0085,  1.7747], grad_fn=<SqueezeBackward4>)
```

- Note that the module itself is *callable*, and that calling it invokes its **forward( )** function.
- Each module is associated with "forward pass" and "backward pass":
  - The "forward pass" applies the computation defined in the module to the given input(s).
  - The "backward pass" computes gradients of module outputs with respect to its inputs. PyTorch's autograd system automatically takes care of this backward pass computation, so it is *NOT required* to manually implement a **backward( )** function for each module.

Define A Simple Custom Model

- The full set of parameters registered by the module can be iterated through via a call to **parameters( )** or **named_parameters( ).**

Code

```
for parameter in m.named_parameters():
  print(parameter)



for parameter in m.parameters():
  print(parameter)
```

Output

```
tensor([-0.3767,  3.0030,  0.0343], grad_fn=<SqueezeBackward4>)
('weight', Parameter containing:
tensor([[ 0.6101, -0.7031,  1.3140],
        [-0.5459, -0.3645,  0.0464],
        [ 2.9713,  1.0321,  2.4162],
        [ 2.6158, -0.4109,  0.5061]], requires_grad=True))

Parameter containing:
tensor([[-0.6607, -0.3632, -0.7274],
        [-0.8555, -1.9544, -0.3640],
        [-2.1289,  0.0305,  0.0443],
        [-0.6668,  1.7042, -0.0758]], requires_grad=True)
```

Modules as Building Blocks

RELAXED SYSTEM LAB

Modules as Building Blocks

```
net = nn.Sequential(
  MyLinear(4, 3),
  nn.ReLU(),
  MyLinear(3, 1)
)

sample_input = torch.randn(4)
print(net(sample_input))

# tensor([-0.6749], grad_fn=<AddBackward0>)
```

- Modules can contain other modules, making them useful building blocks for developing more elaborate functionality. The simplest way to do this is using the **nn.Sequential** module. It allows us to chain together multiple modules.
- **nn.Sequential** automatically feeds the output of the first **MyLinear** module as input into the **ReLU**, and the output of that as input into the second **MyLinear** module. As shown, it is limited to in-order chaining of modules with a single input and output.

Modules as Building Blocks

RELAXED SYSTEM LAB

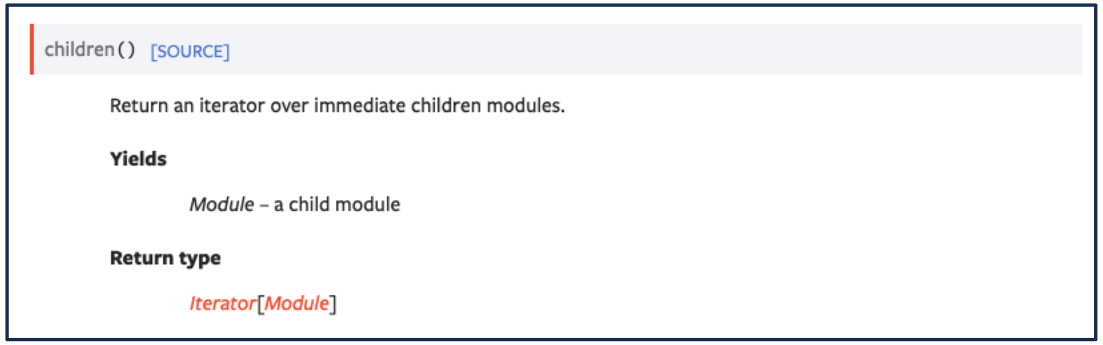
Modules as Building Blocks

```
class Net2(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer0 = MyLinear(4, 3)
    self.layer1 = MyLinear(3, 1)

  def forward(self, x):
    x = self.layer0(x)
    x = F.relu(x)
    x = self.layer1(x)
    return x

net2 = Net2()
sample_input = torch.randn(4)
print(net2(sample_input))

# tensor([-0.2827], grad_fn=<SqueezeBackward4>)
```

- In general, it is recommended to define a custom module for anything beyond the simplest use cases, as this gives full flexibility on how submodules are used for a module's computation.
- This module is composed of two "children" or "submodules" (**layer0** and **layer1**) that define the layers of the neural network and are utilized for computation within the module's **forward( )** method.

Modules as Building Blocks

RELAXED SYSTEM LAB



children() [SOURCE]

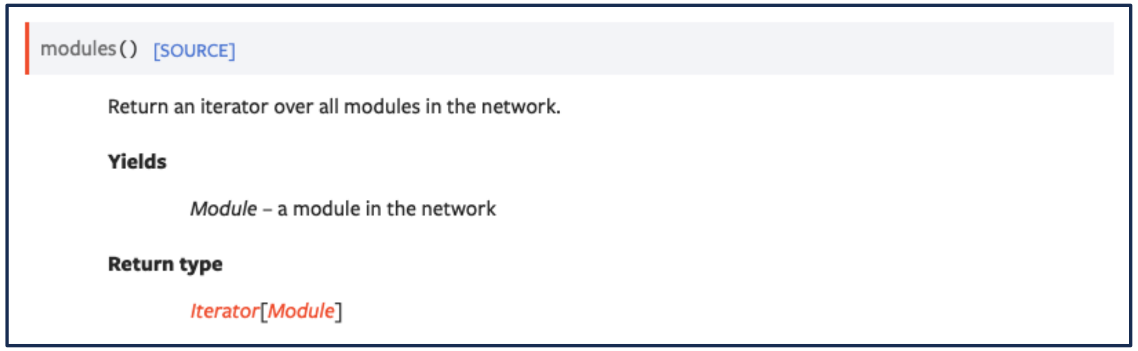
Return an iterator over immediate children modules.

Yields

Module – a child module

Return type

Iterator[*Module*]



modules() [SOURCE]

Return an iterator over all modules in the network.

Yields

Module – a module in the network

Return type

Iterator[*Module*]

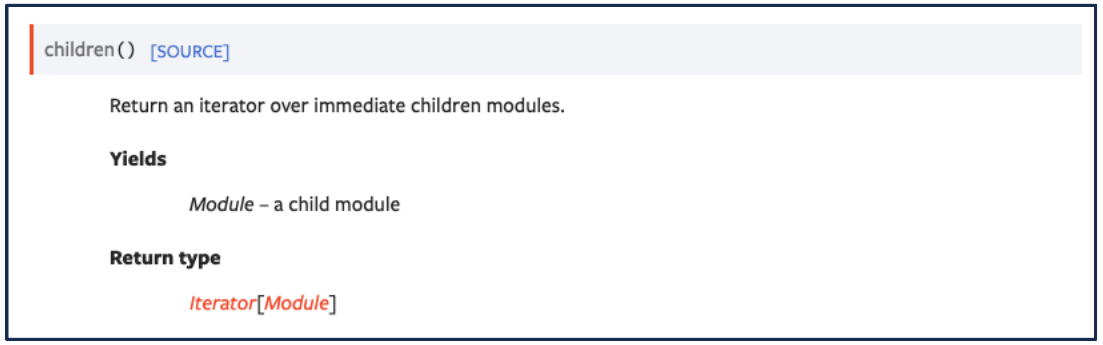
Code

```
print("Check net.children")
for child in net.children():
  print(child)


print("Check net.modules")
for child in net.modules():
  print(child)
```

Output

```
<--Check net.children()-->
MyLinear()
ReLU()
MyLinear()

<--Check net.modules()-->
Sequential(
  (0): MyLinear()
  (1): ReLU()
  (2): MyLinear()
)
MyLinear()
ReLU()
MyLinear()
```

Modules as Building Blocks



children() [SOURCE]

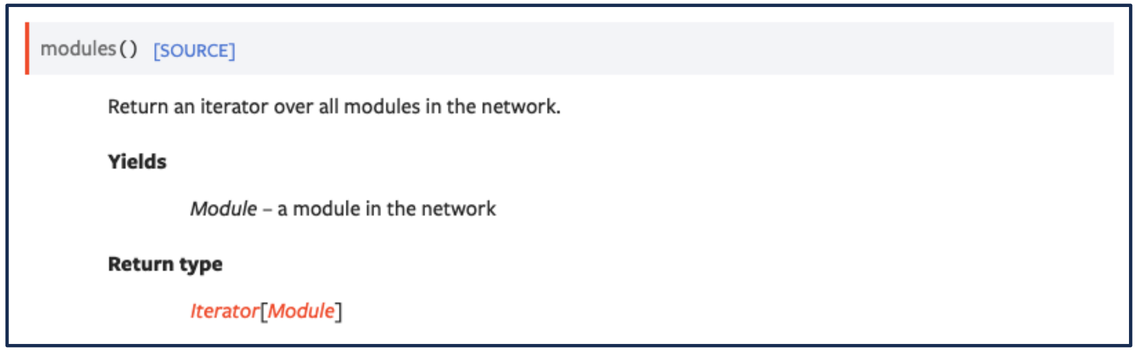
Return an iterator over immediate children modules.

Yields

Module – a child module

Return type

Iterator[*Module*]



modules() [SOURCE]

Return an iterator over all modules in the network.

Yields

Module – a module in the network

Return type

Iterator[*Module*]

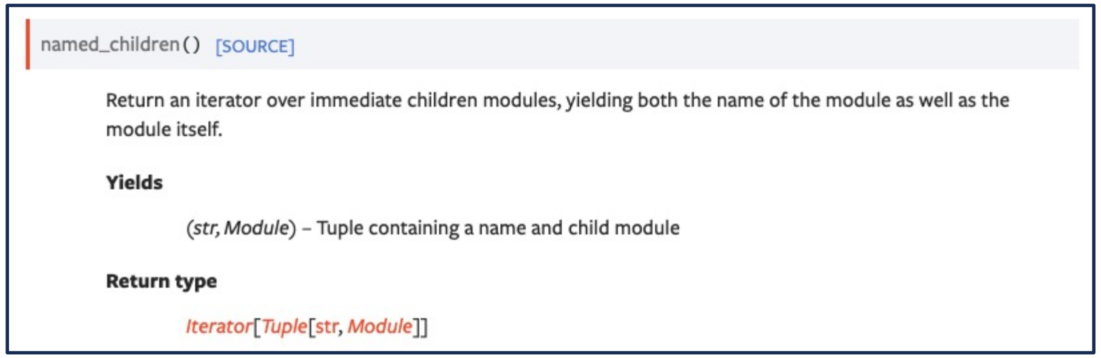
Code

```
print("<--Check net2.children()-->")
for child in net2.children():
  print(child)


print("<--Check net2.modules()-->")
for child in net2.modules():
  print(child)
```

Output

```
<--Check net2.children()-->
MyLinear()
MyLinear()

<--Check net2.modules()-->
Net2(
  (layer0): MyLinear()
  (layer1): MyLinear()
)
MyLinear()
MyLinear()
```

Modules as Building Blocks



named_children() [SOURCE]

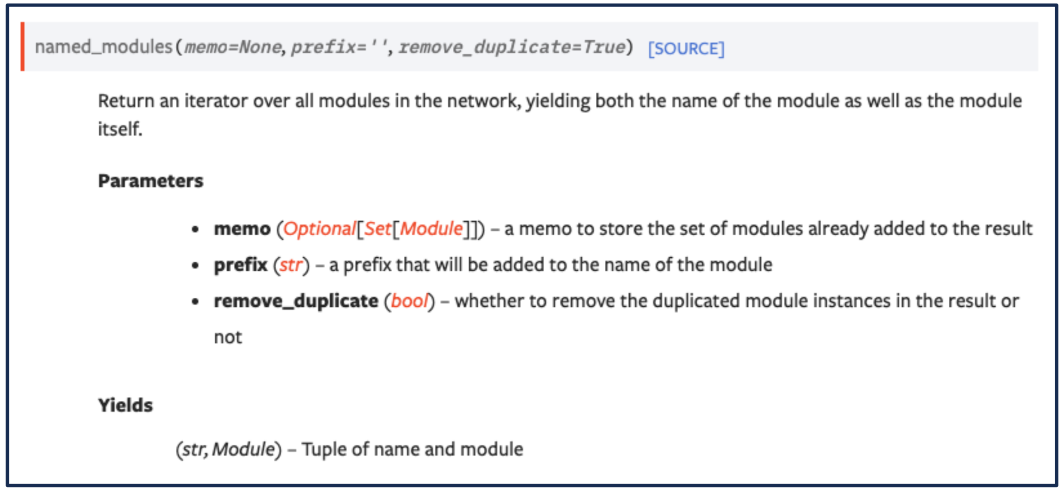
Return an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields

(*str*, *Module*) – Tuple containing a name and child module

Return type

Iterator[*Tuple*[str, *Module*]]



named_modules(*memo=None*, *prefix=''*, *remove_duplicate=True*) [SOURCE]

Return an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** (*Optional*[*Set*[*Module*]]) – a memo to store the set of modules already added to the result
- **prefix** (*str*) – a prefix that will be added to the name of the module
- **remove_duplicate** (*bool*) – whether to remove the duplicated module instances in the result or not

Yields

(*str*, *Module*) – Tuple of name and module

Code	Output

`print("<--Check net2.named_children()-->")`
`for child in net2.named_children():`
`  print(child)`
 	
`print("<--Check net.named_modules()-->")`
`for child in net2.named_modules():`
`  print(child)`
	`<--Check net2.named_children()-->`
	`('layer0', MyLinear())`
	`('layer1', MyLinear())`
	
	`<--Check net.named_modules()-->`
	`('', Net2(`
	`  (layer0): MyLinear()`
	`  (layer1): MyLinear()`
	`))`
	`('layer0', MyLinear())`
	`('layer1', MyLinear())`

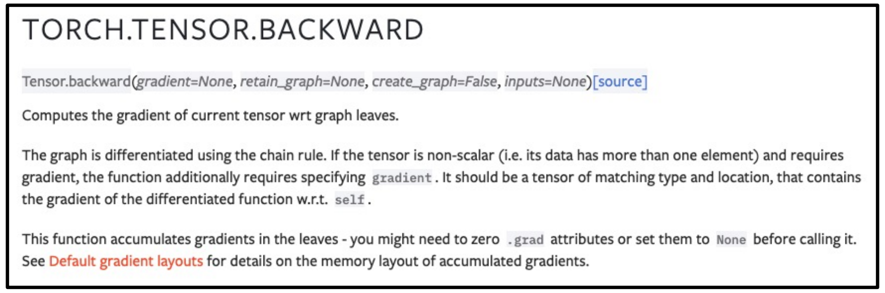
How Does Autograd Execute?

Code

```
x = torch.randn(4)
x.requires_grad_(True)
x.retain_grad()
z = net2(x)
z.retain_grad()

print("x:", x)
print("w0:", net2.layer0.weight)
print("w1:",net2.layer1.weight)
print("z:",z)

z.backward()

print("dz:", z.grad)
print("dw1:",
net2.layer1.weight.grad)
print("dw2:",
net2.layer0.weight.grad)
print("dx:", x.grad)
```

Output

```
x: tensor([ 0.9785,  0.4565, -0.4396, -0.1090],
requires_grad=True)
w0: Parameter containing:
tensor([[-0.4769, -0.9407,  1.1517],
        [-0.4809,  1.2256,  0.9053],
        [ 1.5409, -1.2598, -3.7088],
        [-1.0534, -0.5934,  0.0647]],
requires_grad=True)
w1: Parameter containing:
tensor([[ 1.7888],
        [ 0.1920],
        [-1.8594]], requires_grad=True)
z: tensor([-5.8328], grad_fn=<SqueezeBackward4>)
dz: tensor([1.])
dw1: tensor([[0.0000],
        [0.2575],
        [3.1636]])
dw2: tensor([[ 0.0000,  0.1879, -1.8195],
        [ 0.0000,  0.0877, -0.8488],
        [-0.0000, -0.0844,  0.8174],
        [-0.0000, -0.0209,  0.2027]])
dx: tensor([-2.3221, -1.4479,  6.6540, -0.2343])
```



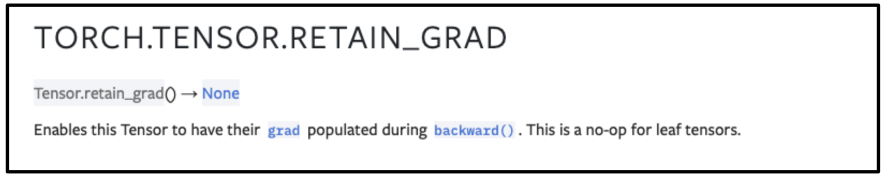
TORCH.TENSOR.BACKWARD

Tensor.backward(*gradient=None*, *retain_graph=None*, *create_graph=False*, *inputs=None*)[source]

Computes the gradient of current tensor wrt graph leaves.

The graph is differentiated using the chain rule. If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally requires specifying `gradient`. It should be a tensor of matching type and location, that contains the gradient of the differentiated function w.r.t. `self`.

This function accumulates gradients in the leaves - you might need to zero `.grad` attributes or set them to `None` before calling it. See Default gradient layouts for details on the memory layout of accumulated gradients.

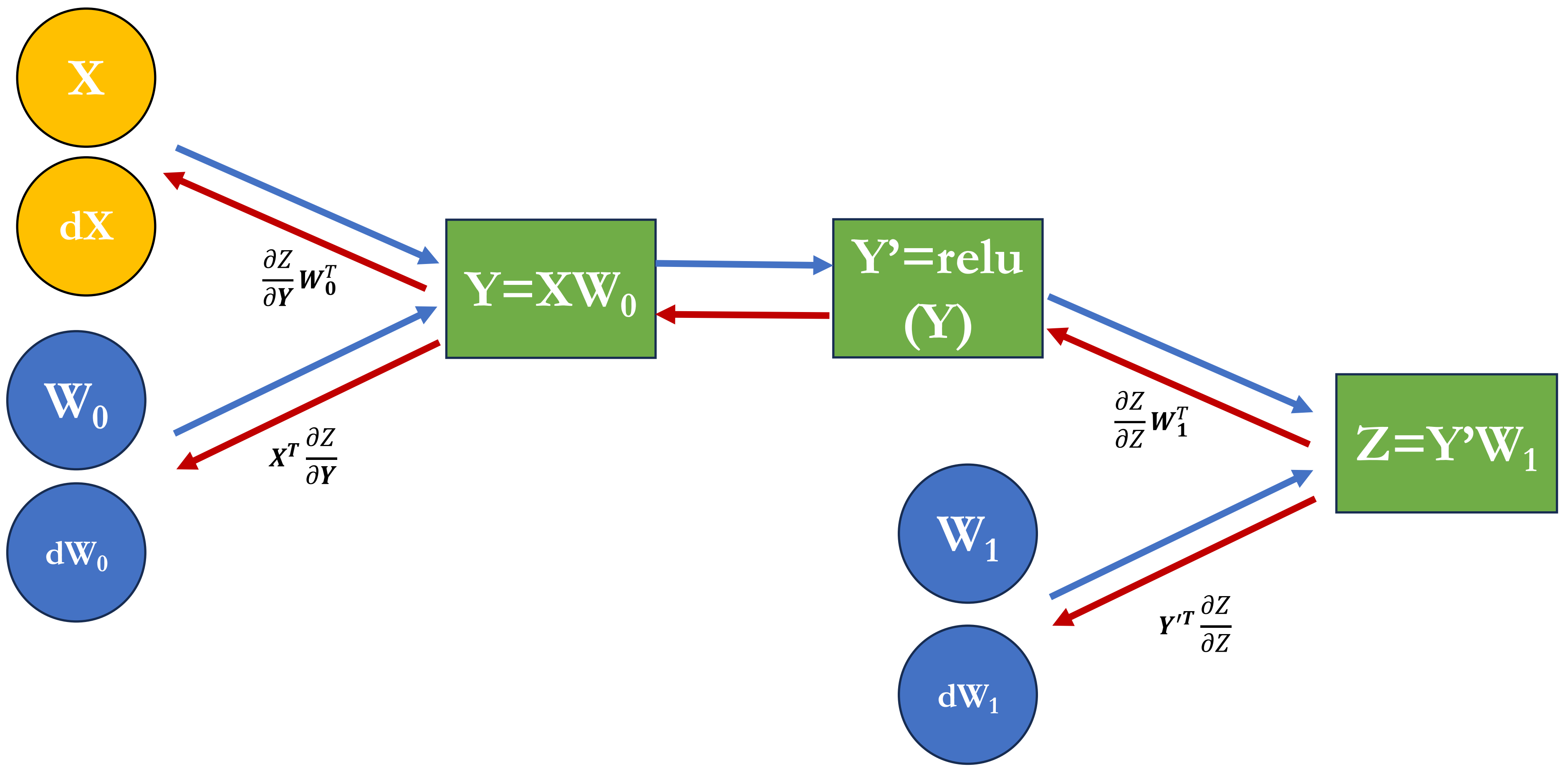


TORCH.TENSOR.RETAIN_GRAD

Tensor.retain_grad() → None

Enables this Tensor to have their `grad` populated during `backward()`. This is a no-op for leaf tensors.

How Does Autograd Execute?



X
dX
W_0
dW_0
$Y=XW_0$
$Y'=\text{relu}(Y)$
$Z=Y'W_1$
W_1
dW_1
$\frac{\partial Z}{\partial Y} W_0^T$
$X^T \frac{\partial Z}{\partial Y}$
$\frac{\partial Z}{\partial Z} W_1^T$
$Y'^T \frac{\partial Z}{\partial Z}$

Locally Disabling Gradient Computation

- The most fine-grained exclusion of subgraphs from gradient computation is setting the **requires_grad** field of a tensor.

- To disable gradients across entire blocks of code, there are also context managers:
    - no-grad mode
    - inference mode.

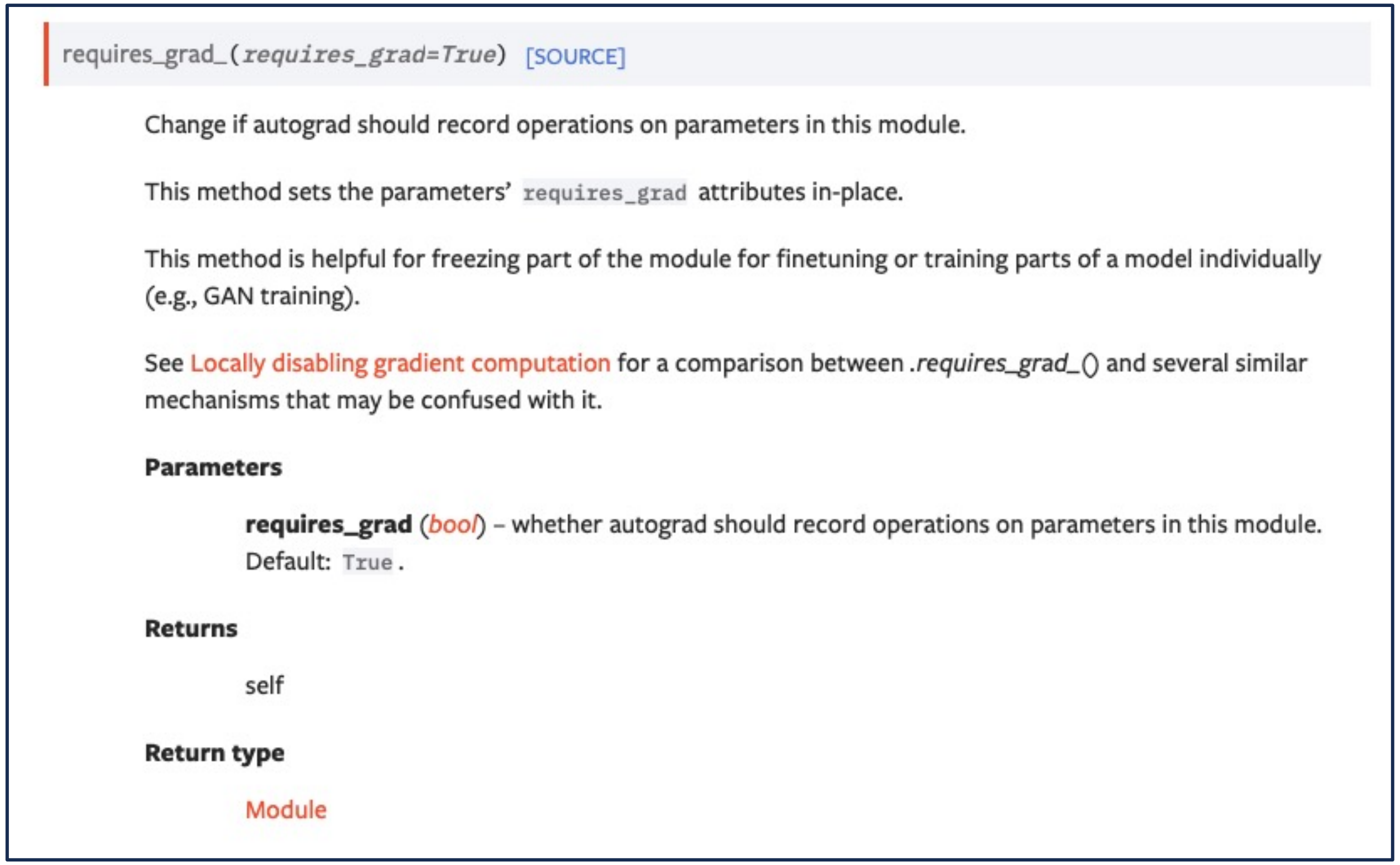
Setting requires_grad

- Recall that “A **torch.Tensor** is a multi-dimensional matrix containing elements of a single data type”.
- The attribute **requires_grad** is a flag, that allows for fine-grained exclusion of subgraphs from gradient computation.
- The attribute **requires_grad** is set to false by default unless wrapped in a **nn.Parameter**.
- It takes effect in both the forward and backward passes:
  - During the forward pass, an operation is only recorded in the backward graph if at least one of its input tensors require grad.
  - During the backward pass (**.backward( )**), only leaf tensors with **requires_grad=True** will have gradients accumulated into their **.grad** fields.

Setting requires_grad

- Although every tensor has this flag, setting it only makes sense for *leaf tensors* (tensors that do not have a **grad_fn**, e.g., a **nn.Module**'s **parameter**s).
- Non-leaf tensors (tensors that do have **grad_fn**) are tensors that have a backward graph associated with them, so their gradients will be needed as an intermediary result to compute the gradient for a leaf tensor that requires grad. Thus, all non-leaf tensors will automatically have **require_grad=True**.
- Setting **requires_grad** should be the main way you control which parts of the model are part of the gradient computation.
  - E.g, if you need to freeze parts of your pretrained model during model fine-tuning.
  - To freeze parts of your model, simply apply **.requires_grad_(False)** to the parameters that you don't want updated.
  - Since computations that use these parameters as inputs would not be recorded in the forward pass, they won't have their **.grad** fields updated in the backward pass because they won't be part of the backward graph in the first place, as desired.
- **requires_grad** can also be set at the module level with **nn.Module.requires_grad_()**. When applied to a module, **.requires_grad_()** takes effect on all of the module's parameters

Setting requires_grad



requires_grad_(*requires_grad=True*) [SOURCE]

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See Locally disabling gradient computation for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters

requires_grad (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns

self

Return type

Module

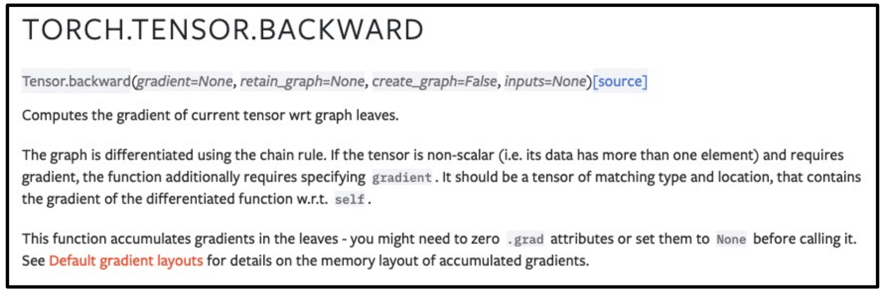
Setting requires_grad

Code

```
net2.layer1.requires_grad_(False)

x = torch.randn(4)
x.requires_grad_(True)
x.retain_grad()
z = net2(x)
z.retain_grad()

print("x:", x)
print("w0:", net2.layer0.weight)
print("w1:",net2.layer1.weight)
print("z:",z)

z.backward()

print("dz:", z.grad)
print("dw1:",
net2.layer1.weight.grad)
print("dw2:",
net2.layer0.weight.grad)
print("dx:", x.grad)
```

Output

```
x: tensor([-0.3019,  0.3072, -0.9097, -2.0537],
requires_grad=True)
w0: Parameter containing:
tensor([[-1.9223, -0.5765, -0.3053],
        [ 1.1695, -0.1050,  1.2679],
        [ 1.0916,  0.5429, -0.6907],
        [-1.3432,  0.0669,  1.4241]],
requires_grad=True)
w1: Parameter containing:
tensor([[0.5816],
        [0.5796],
        [2.0542]])
z: tensor([1.5735], grad_fn=<SqueezeBackward4>)
dz: tensor([1.])
dw1: None
dw2: tensor([[-0.1756, -0.0000, -0.0000],
        [ 0.1787,  0.0000,  0.0000],
        [-0.5291, -0.0000, -0.0000],
        [-1.1945, -0.0000, -0.0000]])
dx: tensor([-1.1181,  0.6803,  0.6349, -0.7813])
```



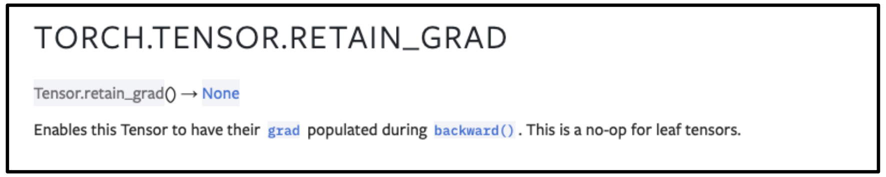
TORCH.TENSOR.BACKWARD

Tensor.backward(*gradient=None*, *retain_graph=None*, *create_graph=False*, *inputs=None*)[source]

Computes the gradient of current tensor wrt graph leaves.

The graph is differentiated using the chain rule. If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally requires specifying `gradient`. It should be a tensor of matching type and location, that contains the gradient of the differentiated function w.r.t. `self`.

This function accumulates gradients in the leaves - you might need to zero `.grad` attributes or set them to `None` before calling it. See Default gradient layouts for details on the memory layout of accumulated gradients.

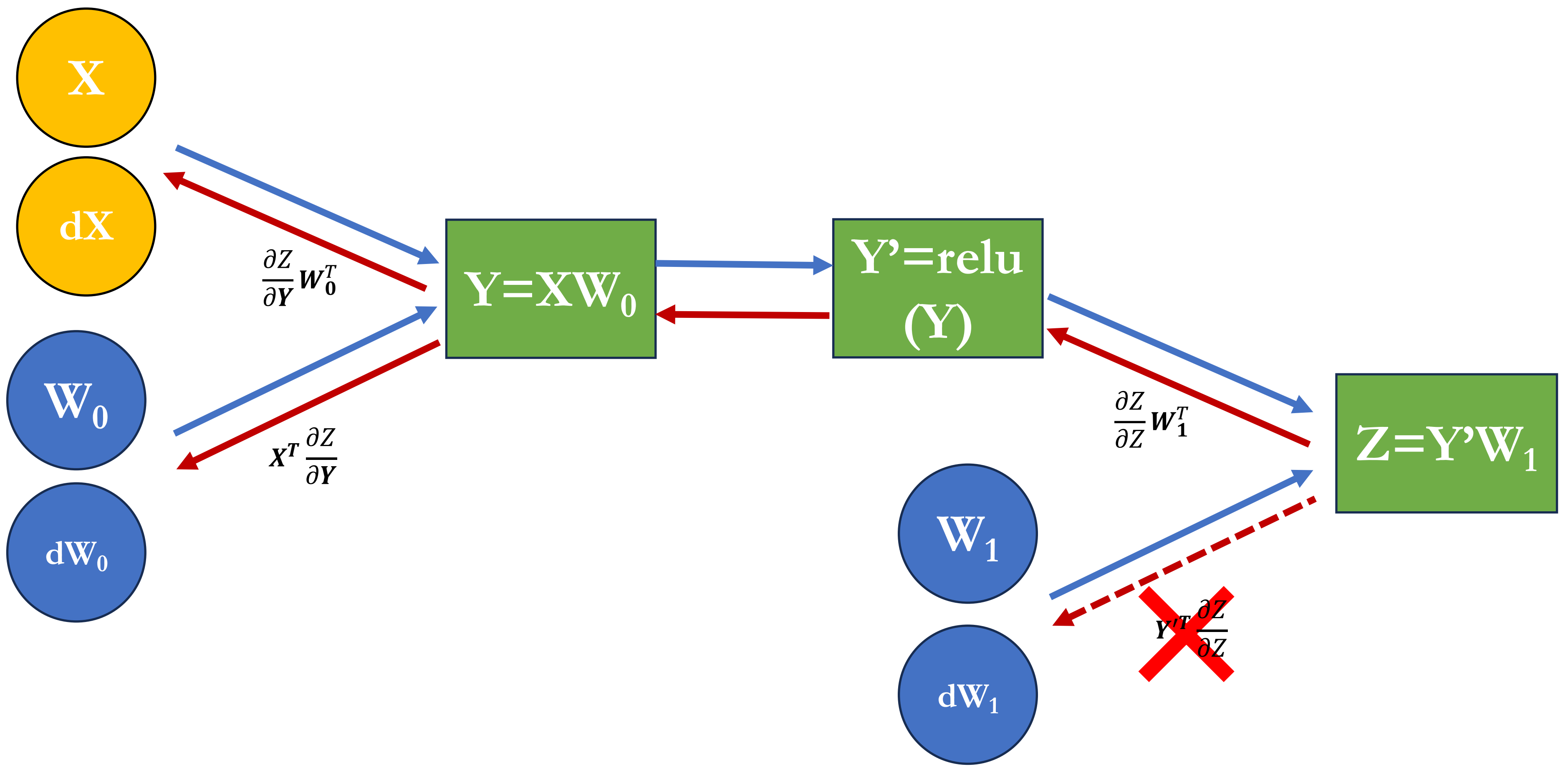


TORCH.TENSOR.RETAIN_GRAD

Tensor.retain_grad() → None

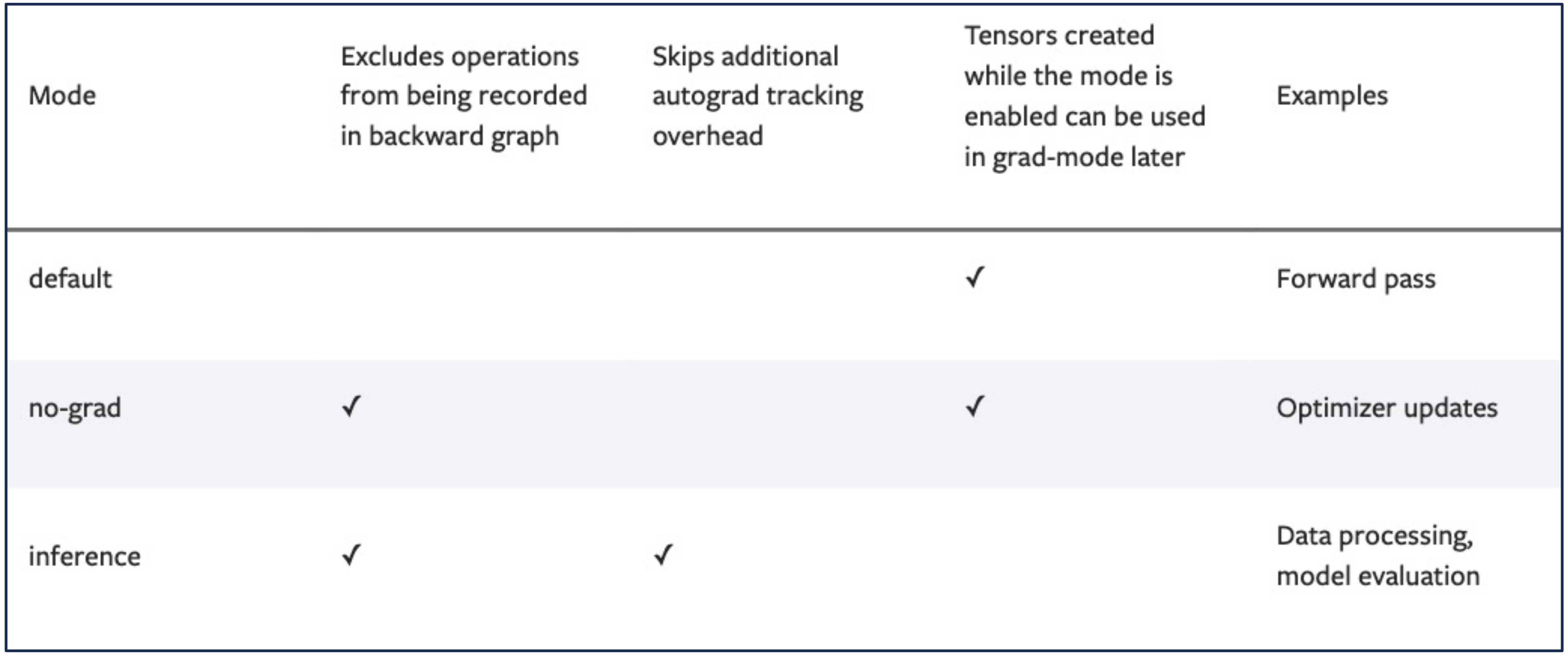
Enables this Tensor to have their `grad` populated during `backward()`. This is a no-op for leaf tensors.

Setting requires_grad



X
dX
W_0
dW_0
$Y=XW_0$
$Y'=relu(Y)$
$Z=Y'W_1$
W_1
dW_1
$\frac{\partial Z}{\partial Y} W_0^T$
$X^T \frac{\partial Z}{\partial Y}$
$\frac{\partial Z}{\partial Z} W_1^T$
$Y'^T \frac{\partial Z}{\partial Z}$

Grad Modes



Mode	Excludes operations from being recorded in backward graph	Skips additional autograd tracking overhead	Tensors created while the mode is enabled can be used in grad-mode later	Examples
default			✓	Forward pass
no-grad	✓		✓	Optimizer updates
inference	✓	✓		Data processing, model evaluation

Default Mode

- The "default mode" is the mode we *are implicitly in* when no other modes like no-grad and inference mode are enabled.

- The most important thing to know about the default mode is that it is the only mode in which **requires_grad** takes effect. **requires_grad** is always overridden to be **False** in the two other modes.

No-grad Mode

- Computations in no-grad mode behave as if none of the inputs require grad.
- Computations in no-grad mode are never recorded in the backward graph even if there are inputs that have **require_grad=True**.
- Enable no-grad mode when you need to perform operations that should not be recorded by **autograd**, but you'd still like to use the outputs of these computations in grad mode later.
- This context manager makes it convenient to disable gradients for a block of code or function without having to temporarily set tensors to have **requires_grad=False**, and then back to **True**.

Recall Our Frist Training Script

RELAXED
SYSTEM LAB

Define the test function

```
def test_loop(dataloader, model, loss_fn):
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Inference Mode

- Inference mode is the extreme version of no-grad mode.
- Computations in inference mode are not recorded in the backward graph -- enabling inference mode will allow PyTorch to speed up your model even more.
- Drawback: tensors created in inference mode will not be able to be used in computations to be recorded by **autograd** after exiting inference mode.
- Enable inference mode when you are performing computations that don’t need to be recorded in the backward graph, AND you don’t plan on using the tensors created in inference mode in any computation that is to be recorded by **autograd** later.

Compare Different Modes

Compare different modes

```
dim1 = 4096
dim2 = 8192

class MyLinear(nn.Module):
  def __init__(self, in_features, out_features):
    super().__init__()
    self.weight = nn.Parameter(torch.randn(in_features, out_features))

  def forward(self, input):
    return torch.matmul(input, self.weight)


class Net3(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer0 = MyLinear(dim1, dim2)
    self.layer1 = MyLinear(dim2, dim2)
    self.layer2 = MyLinear(dim2, dim2)
    self.layer3 = MyLinear(dim2, 1)

  def forward(self, x):
    x = self.layer0(x)
    x = F.relu(x)
    x = self.layer1(x)
    x = F.relu(x)
    x = self.layer2(x)
    x = F.relu(x)
    x = self.layer3(x)
    return x

net3 = Net3()
x = torch.randn(256,dim1)
```

Code	Output

`start_time = time.time()`
`z = net3(x)`
`end_time = time.time()`
`print("Forward computation takes:", end_time-start_time)`
	`Forward computation takes:`
	`0.5174 second`

`start_time = time.time()`
`with torch.no_grad():`
`    z = net3(x)`
`end_time = time.time()`
`print("Forward computation takes: ", end_time-start_time)`
	`Forward computation takes:`
	`0.4768 second`

`start_time = time.time()`
`with torch.inference_mode():`
`  z = net3(x)`
`end_time = time.time()`
`print("Forward computation takes: ", end_time-start_time)`
	`Forward computation takes:`
	`0.4341 second`

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
- https://pytorch.org/docs/stable/notes/autograd.html
- https://pytorch.org/docs/stable/notes/modules.html