“深度学习已死，可微分编程万岁”——最近深度学习三巨头之一的 Yann LeCun 发表了这个惊人的言论。可微编程(Differential Programming, DP)[1]的核心任务是，研究员写少量高层次代码从而实现从数据输入端到输出端的预测。我们知道，传统的编程方法是，程序员写出每一行代码，机器按照给定的代码运行程序，根据输入数据得到运算结果。而可微编程的思想是，程序员可以不写代码或者仅写出少量高层次的代码，但是提供大量输入数据与对应的运算结果，利用可微编程自动地学习从输入数据到最终运算结果的映射（即整个程序），或者结合程序员提供高层次的代码，用神经网络作为中间函数，补全得到整个程序。实际上，现有的深度学习正是通过组装参数化功能模块网络，并用某种基于梯度的优化方法训练它们，来构建一类新算法或者软件。而深度学习虽然在很多领域取得了极大的突破，但是仍存在黑盒智能、可解释性差等问题。而可微编程不同于现有的深度学习，它作为衔接传统算法与深度学习之间的桥梁为深度学习算法提供可解释性，将成为打开深度学习黑盒子的一大利器。

与可微编程直接相关的是自动微分（Automatic Differentiation，AD）[2, 3]技术。自动微分是一种对计算机程序进行高效准确求导的技术，一直被广泛应用于计算流体力学、大气科学、工业设计仿真优化等领域[1]。而近年来，机器学习技术的兴起也驱动着对自动微分技术的研究进入一个新的阶段。常见计算机程序求导或微分的方法包括：（1）手工求导并编写对应的结果程序（Manual Differentiation）；（2）通过有限差分近似方法完成求导，称为数值微分（Numerical Differentiation）[4]；（3）基于数学规则和程序表达式变换完成导，称为符号微分（Symbolic Differentiation）[5]；（4）介于数值微分和符号微分之间的一种求导方法，称为自动微分。其中，手工求导并编写程序费时而且容易出错。数值微分很容易实现，但由于舍入和截断错误，可能得到非常不准确的结果[6]。符号微分法解决了手工方法和数值方法的弱点，但常带来复杂的表达式，并受到“表达式膨胀”问题的困扰[2]。此外，手工和符号方法还要求将模型定义为闭式表达式，排除或严格限制算法控制流和表达式。

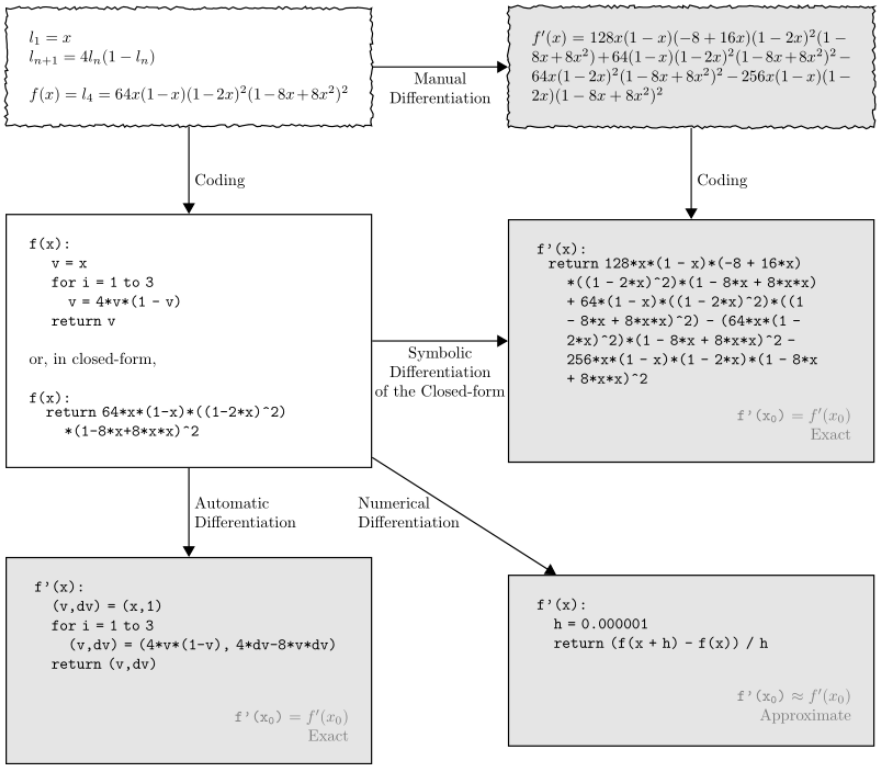

图 1 四种求导方式，手工求导（右上），符号微分（右中），数值微分（右下），自动微分（左下）。

与上述方法不同，AD 将计算机程序中的运算操作分解为一个有限的基本操作集合，且集合中基本操作的求导规则均为已知的[7]。在完成每一个基本操作的求导后，使用链式法则将结果组合得到整体程序的求导结果，从而能保证得到高精度的结果，并且避免符号微分中的表达式膨胀问题。自动微分通常采用三部分表示法，将函数$f: \mathbb{R}^n \to \mathbb{R}^m$的求值轨迹用输入变量、中间变量和输出变量进行表示，其中：

- 变量$v_{i-n} = x_i, i = 1, \dots, n$是输入变量，
- 变量$v_i, i = 1, \dots, l$是中间变量，
- 变量$y_{m-i} = v_{l-i}, i = m-1, \dots, 0$是输出变量。

以函数$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$为例，图 2 展示了对应的计算图[8]，用来表示变量之间的依赖关系。

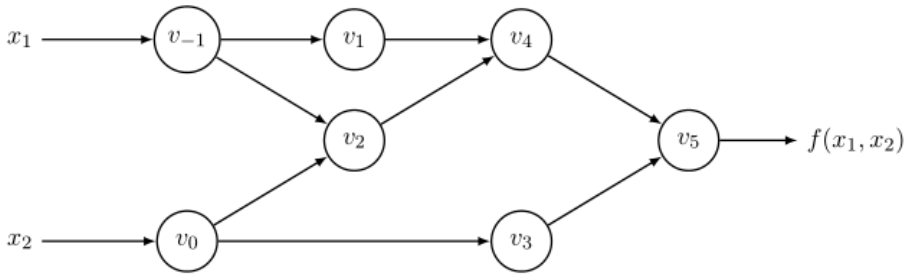

图 2 $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$的计算图，节点的定义见图 3。

AD 一般分两种，一种被称为前向 AD（Forward-Mode AD），另一种被称为后向 AD(Reverse-Mode AD)。在前向 AD 中，为了计算f对x_1的导数，可以计算每个中间变量v_i对输入变量x_1的导数$\dot{v}_i = \frac{\partial v_i}{\partial x_1}$，并将链式法则应用在前向原始轨迹（Forward Primal Trace）上的每个基本运算,得到相应的前向切线/导数轨迹（Forward Tangent/Derivative Trace），如图 3 右半部分所示。通过计算中间变量v_i和其对应的导数$\dot{v}_i$，就能得到最终所要求的导数。后向 AD 是一种广义的

Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= 5.5$

图 3 前向 AD 示例：求函数$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$在$(x_1, x_2) = (2,5)$处对$x_1$的偏导数$\frac{\partial y}{\partial x_1}$，设置$\dot{x}_1 = 1$，$\dot{x}_2 = 0$。

反向传播算法,可以通过计算输出变量y_j对每个中间变量v_i的导数$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$来完成。在后向 AD 中，导数是在两阶段过程中的第二阶段被计算的。在第一阶段，原函数代码向前运行，计算所有中间变量v_i并记录计算图中的依赖关系；在第二阶段，通过从输出到中间变量的反向传播来计算导数。以图 2 中的v_0为例，y对其的导数为$\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0}$或者$\bar{v}_0 = \bar{v}_2\frac{\partial v_2}{\partial v_0} + \bar{v}_3\frac{\partial v_3}{\partial v_0}$，在图 4 中，$\bar{v}_0$通过两步计算得到：$\bar{v}_0 = \bar{v}_3\frac{\partial v_3}{\partial v_0}$和$\bar{v}_0 = \bar{v}_0 + \bar{v}_2\frac{\partial v_2}{\partial v_0}$,进而得到$y$对$x_2$的导数$\frac{\partial y}{\partial x_2} = \bar{x}_2 = \bar{v}_0 = 1.716$。

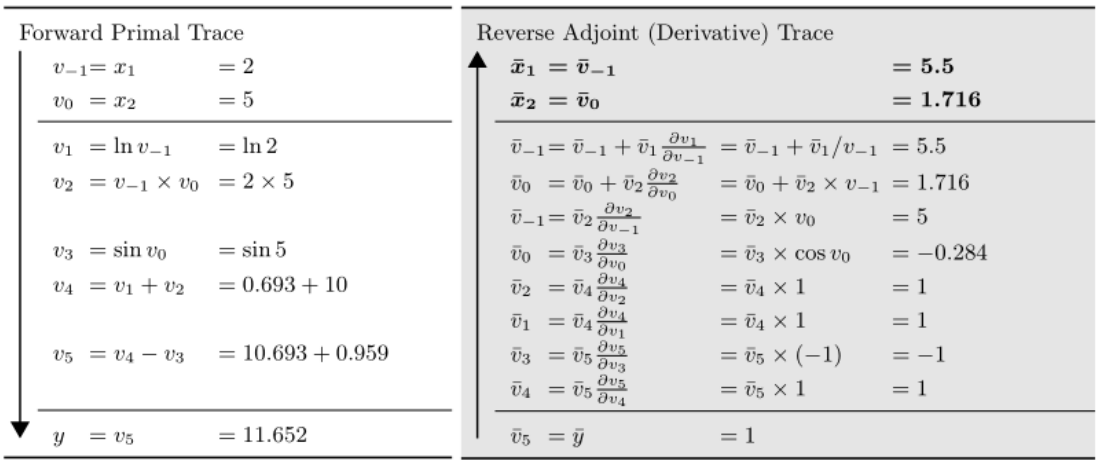

图 4 后向 AD 示例：求函数$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$在$(x_1, x_2) = (2,5)$处对$x_1$和$x_2$的偏导数，设置$\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$。

这两种 AD 适用于不同的问题。当输入参数远大于输出的时候，反向 AD 是效率较高的方法。这也是当今流行的两大机器学习框架 Pytorch[9]和 Tensorflow[10]都采用反向传播机制的原因，因为其所处理的神经网络训练问题往往是成千上万的输入参数对应一个或几个输出。在实现细节上，这两种框架都把计算的表达式展开为“计算图”的形式，图的每个节点代表变量，边代表某种运算，信息和计算就在图的各个节点间流动和传播。

尽管在科学计算、人工智能等领域取得了巨大成功，但是目前的自动微分技术也存在一些限制，如所有的表达式都必须先按规则表达成框架内部的 Tensor 形式，缺少自由度；所有求导规则都必须手动定义并维护等。而随着自动微分技术的发展，业内也逐渐演进出了可微编程的概念。在可微编程中，一个关键的理念是希望将自动微分技术与语言设计、编译器/解释器甚至 IDE 等工具链等深度融合，将微分作为语言中的一等公民特性（first-class feature），即认为反向传播以及自动微分的效用不必拘泥于机器学习领域，而是应该追求广义的函数（比如程序的输入对输出）的微分，从而实现编程易用性、性能、安全性的提升[11]：（1）可以使用更多的语言原生特性，包括各类控制流表达式和更复杂的如 OO 特性等；（2）用户可以更自由的定制微分规则，来进行性能调优或支持对传统意义上不可微的数据类型和操作的“微分”；（3）支持高阶微分和相应的一些性能优化算法；（4）编译器/解释器中提供完善的检查告警系统，对不符合当前系统自动微分规则的代码进行识别和告警；（5）提供更好的自动微分调试能力，如：

中间梯度值输出、单步调试等。

目前，业内有很多在上述方面的探索工作和解决方案。谷歌在 2018 年推出的开源项目 Swift for TensorFlow[12]，旨在将 Swift 语言与 TensorFlow 组合打造为下一代机器学习开发平台。该项目最大的亮点之一即是在 Swift 语言中扩展提供了原生的自动微分特性。该项目对自动微分与语言设计的融合进行了很多探索，产生了非常多优秀的可微编程设计理念。该项目对自动微分中的数学概念进行抽象，并在语言层面将抽象后形成的语言元素通过一系列的接口提供给用户，方便其按需定制相应的自动微分规则。除此之外，业内也有包括 Julia-Zygote[13]、TaiChi[14]等通用或领域专用的可微编程优秀设计方案。此外，可微编程也已经在很多领域得到了应用，例如结合深度学习与物理引擎中的机器人，解决与可微电子结构问题密度泛函理论，可微光线跟踪，图像处理，以及概率性编程等。

图 4 可微编程设计方案

目前可微编程的研究仍有许多问题亟待解决，其挑战主要来自三个方面[11]：（1）易用性：自动微分本身的数学原理相对简单，但是其在可微编程中实现的难点在于对程序表达的分解、微分和组合，而不是简单地对数学表达的分解、微分和组合，需要对计算机程序中的控制流、不同数据类型、非数学表达特性等进行分析，并设计相应的自动微分规则，使其能被自动微分系统进行特殊的处理，以达到可微编程的目的。（2）性能：自动微分的计算过程会涉及到大量的程序的分解、微分和组合，因此如何分解、微分和组合将决定自动微分的性能。这就涉及到原程序和自动微分计算过程的复用、中间结果的保存和重计算等方案设计。（3）安全性：除了易用性和性能外，可微编程的安全性也是一个新的技术挑战。

相比传统程序语言，用户更容易写出不符合自动微分系统规则的可微编程代码，因此自动微分系统需要能够提供足够强大的自动微分规则检查能力，对用户的代码进行检查并给出规则告警，以保证自动微分的安全性。

参考文献

[1] BAYDIN A G, PEARLMUTTER B A, RADUL A A, et al. Automatic differentiation in machine learning: a survey [J]. Journal of machine learning research, 2018, 18.

[2] CORLISS G F. Applications of differentiation arithmetic [M]. Reliability in Computing. Elsevier. 1988: 127-48.

[3] VERMA A. An introduction to automatic differentiation [J]. Current Science, 2000: 804-7.

[4] BURDEN R L. Numerical analysis [Z]. Boston, MA: Brooks/Cole, Cengage Learning. 2011

[5] GRABMEIER J, KALTOFEN E, WEISPFENNING V. Computer algebra handbook: foundations, applications, systems [M]. Springer, 2003.

[6] JERRELL M E. Automatic differentiation and interval arithmetic for estimation of disequilibrium models [J]. Computational Economics, 1997, 10(3): 295-316.

[7] GRIEWANK A, WALTHER A. Evaluating derivatives: principles and techniques of algorithmic differentiation [M]. SIAM, 2008.

[8] BAUER F L. Computational graphs and rounding error [J]. SIAM Journal on Numerical Analysis, 1974, 11(1): 87-96.

[9] PASZKE A, GROSS S, CHINTALA S, et al. Automatic differentiation in pytorch [J]. 2017.

[10] ABADI M, AGARWAL A, BARHAM P, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems [J]. arXiv preprint arXiv:160304467, 2016.

[11] 华为编程语言实验室. 技术分享 | 从自动微分到可微编程语言设计 [Z]. 2021

[12] SAETA B, SHABALIN D. Swift for TensorFlow: A portable, flexible platform for deep learning [J]. Proceedings of Machine Learning and Systems, 2021, 3.

[13] INNES M, EDELMAN A, FISCHER K, et al. A differentiable programming system to bridge machine learning and scientific computing [J]. arXiv preprint arXiv:190707587, 2019.

[14] HU Y, ANDERSON L, LI T-M, et al. Difftaichi: Differentiable programming for physical simulation [J]. arXiv preprint arXiv:191000935, 2019.