arXiv:2303.07030v1 [cs.PL] 13 Mar 2023

∇SD：稀疏张量可微编程

AMIR SHAIKHHA,英国爱丁堡大学MATHIEU HUOT,英国牛津大学SHIDEH HASHEMIAN, 伊朗阿米尔卡比尔理工大学

稀疏张量广泛存在于许多数据密集型应用中，但现有的可微编程框架都是针对稠密张量设计的。这给通过稀疏张量运算高效计算梯度带来了重大挑战，因为其不规则稀疏模式可能导致大量的内存和计算开销。在这项工作中，我们提出了一种新颖的框架，能够对稀疏张量进行高效的自动微分，从而解决这一根本问题。实验表明，所提出的框架在性能和可扩展性方面表现出色，在一系列合成数据集和真实世界数据集上的表现均优于最先进的框架。我们的方法为实现基于稀疏张量的高效、可扩展可微编程提供了一个有前景的方向，这对机器学习、自然语言处理和科学计算等众多应用具有重要意义。

补充关键词　词语和短语：稀疏张量代数、自动微分、半环字典

1 引言

稀疏张量在许多科学与工程应用中至关重要，例如自然语言处理、计算机视觉和图分析。与无论元素值为何都会存储所有元素的稠密张量不同，稀疏张量只存储非零值，因此能够显著节省内存并提高计算效率。稀疏张量还能够高效地表示和操作高维数据结构，而这类结构常见于现代机器学习和科学计算中，例如自然语言处理中表示文档或语料库中词语频率的稀疏张量、网络/关系分析中大型稀疏图的邻接矩阵，以及推荐系统中用于协同过滤的稀疏用户-物品交互矩阵。这激发了近期对更好地支持稀疏张量的兴趣 [Kjolstad 等人 2017年；Strout 等人 2018年；Tang 等人 2020]。

自动微分（AD）是现代机器学习和科学计算中的一项基础技术，可高效计算函数的梯度。这对于优化、参数估计以及许多采用基于梯度的优化方法的其他应用至关重要。尽管稠密张量的自动微分工具丰富且成熟，但缺乏高效的稀疏张量自动微分工具，阻碍了这些技术的更广泛应用，也构成了该领域的一项重大研究挑战。TensorFlow、PyTorch 和 JAX 等库为稠密张量运算提供了高效且可扩展的梯度计算实现，但对稀疏运算的支持有限 [JAX 2023年；PyTorch 2023年；TensorFlow 2023]。因此，人们已尝试为特定的稀疏张量内核手动实现微分 [Nytko 等人 2022]。

稀疏张量代数的自动微分比稠密张量代数更具挑战性，原因有以下几点。首先，稀疏张量的结构比稠密张量更复杂，其稀疏模式往往不规则，并且会因不同的运算而异。这使得在计算图中高效传播梯度，以及确定稀疏张量中的哪些元素与梯度计算相关，都颇具挑战。其次，稀疏张量运算通常需要使用专门的数据结构和算法，例如压缩稀疏

作者地址：Amir Shaikhha，amir.shaikhha@ed.ac.uk，爱丁堡大学，英国；Mathieu Huot，mathieu.huot@stx.ox.ac.uk，牛津大学，英国；Shideh Hashemian，shideh.hashemian@aut.ac.ir，伊朗阿米尔卡比尔理工大学。



图1. ∇SD 中变换的高层概览。

行（CSR）或压缩稀疏列（CSC）格式，而大多数支持自动微分的框架并未原生支持这些格式，或者其自动微分规则并不完整。

本文介绍了 ∇SD，这是一种新颖的可微编程框架，支持对任意稀疏计算进行自动微分。据我们所知，∇SD 是首个具备此能力的框架。与现有框架仅支持对数量有限的稀疏内核进行自动微分不同 [Abadi 等，2015；Bradbury 等，2018；Nytko 等，2022；Paszke 等，2019]，∇SD 支持对可用张量代数表达的任意稀疏计算进行自动微分。

关键洞见是对稀疏张量的逻辑表示进行微分。这意味着，程序微分的语义与优化及数据布局表示彼此明确分离（参见图1）。关于两个向量点积的简单示例，请参见示例 1（续）：§2.2 中微分前的逻辑形式、§4.2 中微分后的形式、§5.3 中的物理坐标格式（COO）表示，以及 §5.5 中的生成代码。

稀疏张量的物理表示（例如 COO/CSR/CSC）涉及多个数组，用于存储矩阵的压缩表示（参见图 3a）。对这类表示进行计算时，需要对这些数组进行命令式嵌套循环。然而，我们的逻辑表示采用嵌套字典，稀疏计算以对这些数组进行嵌套求和的函数式方式表达。之后，逻辑表示可以与具体的物理存储格式融合（参见第 5.3 节）。

具体而言，本文的贡献如下：

- 我们提出了 ∇SD，这是首个系统性支持稀疏张量自动微分的框架。∇SD 基于 SDQLite [Schleich 等人，2022；Shaikhha 等人 2022]，这是一种近期提出的中间语言，可通过将张量计算与存储规范分离来表达稀疏张量工作负载（第 3 节）。
- 我们提出了一种新颖的张量化前向模式自动微分，可批量计算梯度（第 4 节）。我们的自动微分变换作用于语言的逻辑部分（称为 Logical SDQLite），无需考虑物理存储格式。
- 随后，利用以下与自动微分无关的变换来优化微分后的程序：(1) 稀疏性感知重写规则，(2)与物理存储格式相结合，以及 (3)使用基于代价的方式应用代数重写规则，借助等式饱和 [Tate 等，2009；Willsey 等，2021] (第 5 节)。此外，在生成底层 C++代码之前，∇SD 会执行底层变换，以移除嵌套循环中出现的不必要的中间张量。
- 我们证明了所提方法的正确性（第 6 节）。也就是说，自动微分是一种类型正确的变换，能够计算程序的导数，并且我们的优化相对于指称语义是可靠的。
- 我们通过实验将 ∇SD 与最先进的框架进行比较，验证了其有效性（第 7 节）。我们展示了 ∇SD 能够在真实数据集和合成数据集上，将梯度计算扩展到包含大量零元素的大型矩阵。

2 背景

2.1 自动微分

自动微分（AD）是一种强大且广泛应用的技术，在机器学习和科学计算中，它能够高效地计算函数的梯度。梯度是许多优化、参数估计和机器学习算法中的关键量，其计算往往是训练过程中的瓶颈。自动微分通过将函数分解为一系列基本运算，并应用链式法则计算每个运算的导数，提供了一种计算高效且准确的梯度计算方法。由此得到的是精确梯度，其计算成本与原函数相近，无需使用有限差分等近似方法，也无需手动推导。**前向模式**。计算函数梯度的一种方法是前向模式自动微分，它沿着计算图的前向方向计算每个运算的导数。该方法从输入变量开始，逐个运算地沿图传播数值及其导数，直到到达输出变量。在每个运算处，使用链式法则计算输出变量相对于每个输入变量的导数，并将这些导数存储在计算图中，以便用于计算函数的梯度。

示例 1 - 向量点积。考虑以下函数：

$$f([x_1, x_2], [y_1, y_2]) = x_1 y_1 + x_2 y_2$$

它接受两对输入变量 x_1、x_2 和 y_1、y_2，并计算向量 $x = [x_1, x_2]$ 和 $y = [y_1, y_2]$ 的点积。要使用前向模式自动微分计算 f 相对于输入的梯度，我们首先将程序转换为 ANF [Flanagan 等人 1993]:

$$\begin{aligned} f([x_1, x_2], [y_1, y_2]) = \quad & \texttt{let } t_1 = x_1 y_1 \\ & \texttt{let } t_2 = x_2 y_2 \\ & \texttt{let } t_3 = t_1 + t_2 \\ & t_3 \end{aligned}$$

前向模式自动微分通过为每个输入变量和中间变量 v关联一个切变量 v'，将每个变量提升为对偶数。然后，按照链式法则计算每个中间切变量。在前面的示例中，函数 f 被转换如下：

$$\begin{aligned} f'([x_1, x_2], [y_1, y_2], [x_1', x_2'], [y_1', y_2']) = \quad & \texttt{let } t_1 = x_1 y_1 \\ & \texttt{let } t_1' = x_1' y_1 + x_1 y_1' \\ & \texttt{let } t_2 = x_2 y_2 \\ & \texttt{let } t_2' = x_2' y_2 + x_2 y_2' \\ & \texttt{let } t_3 = t_1 + t_2 \\ & \texttt{let } t_3' = t_1' + t_2' \\ & t_3' \end{aligned}$$

要计算 f 对每个输入的偏导数，我们需要将相应的切变量设为 1，并将其他输入的切变量设为 0。例如，f 关于第一个向量的梯度可通过以下偏导数计算得出：

$$\begin{aligned} f'([a_1, a_2], [b_1, b_2], [1, 0], [0, 0]) \to^* \quad b_1 = \quad & \frac{\partial [\![f]\!]}{\partial x_1}(a_1, a_2, b_1, b_2) \\ f'([a_1, a_2], [b_1, b_2], [0, 1], [0, 0]) \to^* \quad b_2 = \quad & \frac{\partial [\![f]\!]}{\partial x_2}(a_1, a_2, b_1, b_2) \end{aligned}$$

反向模式自动微分。对于具有张量输入的标量值函数，前向模式自动微分在计算导数时开销很大。这类函数还常见于通过优化目标函数来训练机器学习模型等场景。这是因为，对表示函数 $\mathbb{R}^n \to \mathbb{R}$ 的程序求导时（在这些上下文中通常如此），需要 n 次运行经前向模式转换后的程序，才能得到完整的梯度。

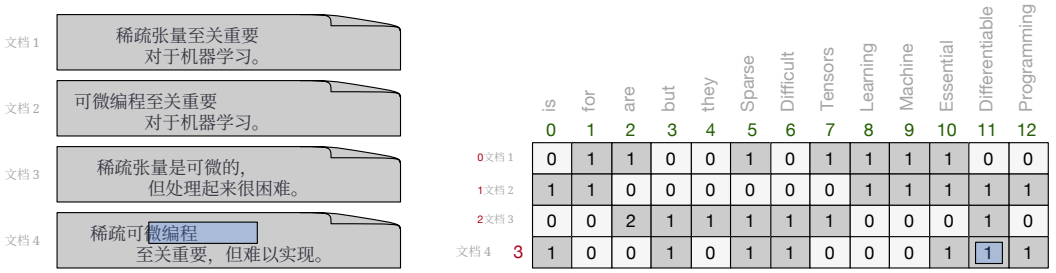


图 2. 使用矩阵表示文档的词袋表示，其中文档为行，词语为列。请注意，矩阵中有许多零元素，也就是说，这是一个稀疏矩阵。

反向模式技术只需运行一次即可计算这类函数的梯度，因此更为合适，并且已在深度学习框架中得到广泛应用 [Abadi 等，2015；Bradbury 等，2018；Paszke 等， 2019]。近期，人们开始关注如何弥合理论正确性保证与更实用、高效且具有复杂度保证的实现之间的差距[Huot 和 Shaikhha，2022；Krawiec 等，2022；Smeding 和 Vákár 2023]。

示例 1（续）。考虑对前一个函数进行推广，其中 · 表示两个向量的点积：

$$f(V_1, V_2) = V_1 \cdot V_2$$

如果每个输入向量有 m 个元素，那么前向模式自动微分的开销为 $O(m^2)$，因为它需要进行 m 次前向传播，每次的开销为 $O(m)$。然而，反向模式自动微分可以通过一次前向传播计算原始值，再通过一次反向传播计算梯度值，从而使总体复杂度为 $O(m)$。

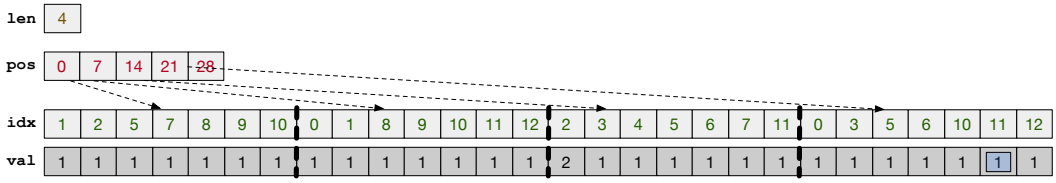
向量前向模式自动微分。受需要计算完整雅可比矩阵的应用场景启发 [Moré 2006]，人们一直在探索前向模式自动微分的批量计算 [Khan 和 Barton，2015；Shaikhha 等人 2019]。实验表明，借助重写规则，甚至可以让向量化前向模式自动微分达到反向模式自动微分的渐近性能 [Shaikhha 等人 2019]。

2.2 稀疏张量与半环字典

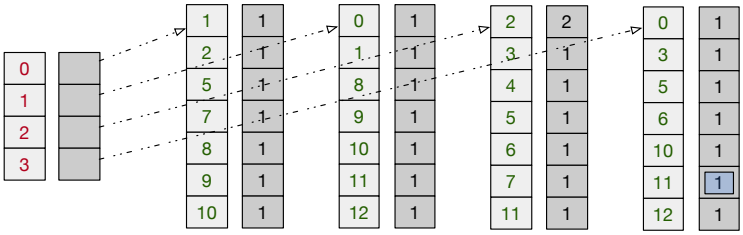
稀疏张量。稀疏张量是一种常用的数据结构，用于表示大多数元素都为零值的高维数据。稀疏张量采用紧凑表示，只存储非零值及其对应的索引，因此在处理大规模数据时，比稠密张量更加节省内存。稀疏张量广泛应用于自然语言处理、计算机视觉和科学计算等领域。例如，在自然语言处理中，稀疏张量可用于将文本数据表示为词袋或词频-逆文档频率（TF-IDF）矩阵，其中行对应文档，列对应词语（图 2）。

可以使用多种专用算法和数据结构来操作稀疏张量，例如压缩稀疏行（CSR）和压缩稀疏列（CSC）格式，这些格式能够高效地执行矩阵-向量乘法及其他运算。然而，稀疏张量不规则的稀疏模式给自动微分带来了重大挑战。

示例 1（续）。在前一个示例中，如果大小为 m 的输入向量中的大多数元素都是零（非零元素的数量记为 nnz，且满足 $nnz << m$），则可以使用图 3a 所示的 CSR 表示。在这种表示中，数组 pos 是行的压缩表示，而 idx 和 val 分别表示非零元素的列和数值



(a) 压缩稀疏行（CSR）表示。



(b) 逻辑半环字典表示。

图 3. 前述词袋矩阵的稀疏表示。

。例如，pos(0)=0、pos(1)=7 表示 idx/val 中索引为 0 到 6 的条目分别对应矩阵第=0 行元素的列和值，而索引为 7 到 14 的条目则对应第=1行的位置。然而，现有的线性代数框架不支持对这种表示求梯度。因此，它们不会在 $O(nnz)$中计算梯度，而是在 $O(m)$的稠密表示上计算梯度。

半环字典。半环字典是一种数据结构，涵盖集合、多重集以及稠密/稀疏张量 [Shaikhha 等人 2022]。半环是一个集合，配备两种满足特定公理的二元运算，例如结合律、分配律和交换律。例如，非负整数集合上的加法和乘法构成一个半环，而布尔值集合上的逻辑 $\vee$ 与 $\wedge$ 运算则构成另一个半环。半环字典旨在将稀疏张量表示为键值对。对于稀疏向量，键对应向量索引，值对应非零元素。对于稀疏矩阵，键对应行索引，值对应与该行关联的稀疏向量。然后根据值上的相应运算（如加法或乘法）来定义半环运算。半环字典中的乘法运算符具有张量外积的语义，这一点可以从下一个示例中看出。

示例 2 - 标量-向量乘积。考虑标量值 s 与向量值 V 之间的标量-向量乘积，其中向量值 V 用半环字典表示。其等价的半环字典表示为 s * V，其中 * 的语义是张量外积。

SDQL。SDQL [Shaikhha et al. 2022] 是一种用于查询半环字典的函数式语言。SDQL 的表达能力足以表示数据库查询和线性代数表达式，因此适合作为混合数据库和机器学习工作负载的中间语言。SDQL 提供以下用于操作半环字典的构造：

(1) `dict(k)` 访问与键 k 关联的 `dict`中的值；如果键不存在，则返回半环的加法单位元（0，对于实数和自然数即为此值）。(2) `{k -> v}` 构造一个只包含一个键值对的字典，其键为k，值为 v。(3) **求和**`(<k,v> 属于 dict) f(k,v)` 遍历 `dict` 中的每个键值对，并计算`f(k,v)` 的求和，起始值为相应半环的加法单位元。

示例 1（　续）。$V1 \cdot V2$ 的等价 SDQL 表达式可以是以下两种之一：

```
sum(<i, a> in V1) a * V2(i)          sum(<i, a> in V2) V1(i) * a
```

首选方案取决于 V1 和 V2 中非零元素的数量。如果 V1（分别是 V2）的非零元素更少，那么左侧（分别是右侧）变体的效率更高。否则，如果两者的非零元素数量相同（例如，两者都是稠密的），则两种变体的性能相同。**SDQLite**。SDQLite [Schleich 等人 2022] 提出，是一种专为稀疏张量处理设计的 SDQL 方言；它将 SDQL 限制为稀疏张量所需的类型，同时扩展了用于处理不同稀疏存储格式（例如 CSR、压缩稀疏列）的构造。以下构造是 SDQLite 的核心：

(1) (st:en) 表示一个稠密数组，其中包含从 st 到 en（不包括 en）的数字范围。(2) arr(st:en) 表示 arr 中从 st 到 en（不包括 en）的子数组。(3) 用于指导重写规则的其他注释，例如 [Schleich 等人 2022]。

3 语言

本节概述我们在 ∇SD 中使用的语言。我们将 SDQLite 分为两部分：较小的片段 Logical SDQLite，在其中执行自动微分；以及完整的片段 Physical SDQLite，它在语言的逻辑子集基础上增加了用于表达不同稀疏存储格式的构造。这些语言的语法和最重要的类型规则见图 4。

逻辑 SDQLite。 最初，程序以 SDQLite 的一个子集表示，该子集足以在逻辑层面表达张量程序，即无需考虑存储格式。因此，在此阶段，我们不要求支持稠密数组。此外，也无需表达元组。不过，为方便起见，我们使用元组式 let 绑定，作为多个 let 绑定的语法糖。

Logical SDQLite 的表达能力足以表示爱因斯坦求和 [Shaikhha 等人 2022]。例如，两个矩阵 M1 和 M2（均表示为嵌套字典）的矩阵乘积可表示为：

```
求和(<i,row> 属于 M1){ i ->求和
(<j,v1> 属于 row)求和(<k, v2> 属
于 M2(j)) {k ->v1* v2} }
```

然而，其表达能力并不局限于爱因斯坦求和。例如，张量 e 的值上的函数 f的映射 可表示为 求和(<k,v> 属于 e) { k -> f(v) }。

物理 SDQLite。 存储规范要求使用与稠密数组相关的构造，例如范围 (st:end) 和子数组 (arr(st:end))。因此，将微分后的程序与存储规范结合后，我们需要在 Physical SDQLite 中包含这些构造。我们将在第 5 节回到这种中间语言。下一节主要关注 Logical SDQLite 及其构造上的微分规则。

4 微分

本节介绍应用于 Logical SDQLite 表达式的微分变换。首先，为便于说明，我们介绍传统前向模式自动微分（FAD）的一种变体。然后，我们展示一种张量化 FAD，它不仅涵盖传统 FAD，还能更高效地计算梯度。最后，我们介绍面向程序员开放的高级 API。

4.1 标量前向模式变换

传统的前向自动微分使用对偶数，在计算实际（原始）结果的同时计算切向（导数）分量。我们称之为标量 FAD，因为对于每个标量表达式，它都会存储一个标量切向分量。

构造	描述	语言
e ::= **求和**(<x,y>**在** e) e | { e -> e } | { } | e(e) | **令** x = e **于** e | x | **非** e | **如果** e **那么** e | e + e | e * e | n | r | **假** | **true** | op(e) | e = e	字典聚合 单元素字典／空字典，查找 变量绑定与访问 否定，条件表达式 加法，乘法 数值和布尔常量 实数一元运算，例如余弦、指数函数、正弦 离散类型相等：布尔值和整数	逻辑 SDQLite
T ::= **实数** | **bool** | D D ::= **整数** | { I -> D } I ::=	标量和张量类型 字典（张量）类型 索引类型（仅整数）	
e ::= ... | e:e | e(e:e) | 参见 *[Schleich 等人 2022]*	源 *SDQLite* 表达式 范围和子数组 其他结构	物理 SDQLite
T ::= ... D ::= ... | **int** I ::= ... | **稠密_整数**	源 *SDQLite* 类型 整数值张量类型 索引类型（+ 稠密整数）	
定义**零**[D]: **零**[**实数**] ≜ 0 **零**[{I -> D}] ≜ { }	定义 ⊗: **实数** ⊗ D ≜ D {I -> D1} ⊗D2 ≜{I -> D1 ⊗ D2}	
定义**let**-配对: **设** <v1 , ..., vn> = <e1 , ..., 其中> **在** 实数 **张量** 0 ≜ **实数** ≜ **令** v1=e1 **属于** ... **令** vn=en **属于** e	定义**张量** n: **张量** n ≜ {**整数** -> **张量** (n-1) }	

$$\frac{\Gamma \vdash e1 : \textbf{张量}\ n{+}1 \qquad \Gamma, k{:}\,\textbf{整数}, v{:}\textbf{张量}\ n \vdash e2 : D}{\Gamma \vdash \textbf{求和}(\langle k,v\rangle\ \textbf{在}\ e1)\ e2 : D} \qquad \frac{\Gamma \vdash e1 : \textbf{张量}\ n{+}1 \qquad \Gamma \vdash e2 : \textbf{整数}}{\Gamma \vdash e1(e2) : \textbf{张量}\ n}$$

$$\frac{\Gamma \vdash k : \textbf{整数} \qquad \Gamma \vdash v : \textbf{张量}\ n}{\Gamma \vdash \{\ k \to v\ \} : \textbf{张量}\ (n{+}1)} \qquad \frac{\Gamma \vdash e1 : D1 \qquad \Gamma \vdash e2 : D2}{\Gamma \vdash e1 * e2 : D1 \otimes D2}$$

图 4. ∇SD 所用语言的语法和部分类型规则。

与其他函数式自动微分框架类似， ∇SD 在进行微分变换之前会先将程序转换为 ANF [Flanagan 等人 1993]。这使得子表达式可以共享，并避免对乘法等非一元构造重复计算。Logical SDQLite 不支持函数定义或高阶函数；所有函数都必须内联 [Paszke 等人 2021]。**标量构造**。 图 5 展示了前向模式变换规则。与现有的函数式自动微分系统不同， ∇SD 不使用显式的配对构造和投影来处理对偶数。相反，$\mathcal{F}[\![\,]\!]$ 构造仅计算微分的切向部分，并引用 ANF 转换后程序中的表达式来获取原始分量（参见 let 绑定规则）。这样就不必在微分的目标语言中扩展配对构造。此外，这也简化了微分规则。对于每个一元实数运算op，我们假设语言中有一个表示其导数的一元实数运算 op' 。最后，所有离散类型（**整数** 和 **布尔值**）的微分结果均为0。

示例 1（续）。再次考虑第 2 节中最初使用的两个长度为二的展开向量的点积。对 SDQLite 中经 ANF 转换的程序应用微分，过程如下：

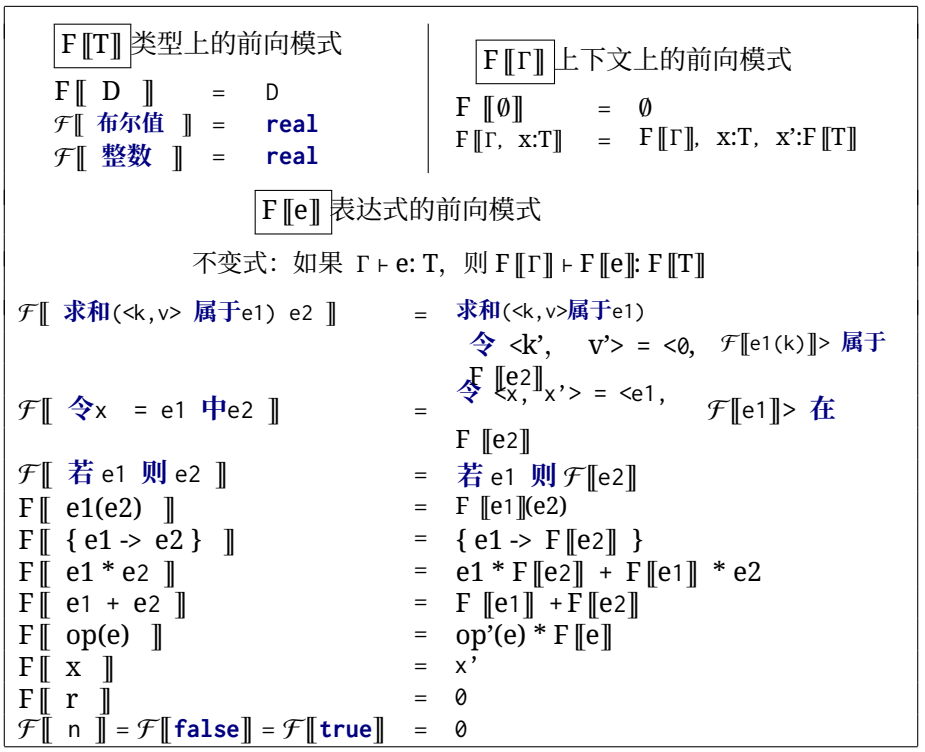


图 5. SDQLite 表达式的前向模式自动微分变换规则。为最大限度地实现共享，所有子表达式都需要绑定到 let 中，即应在微分之前应用 ANF 变换。

$$\mathcal{F}\left[\!\!\left[\begin{array}{l}\texttt{let t1 = x1*y1 in}\\ \texttt{let t2 = x2*y2 in}\\ \texttt{let t3 = t1+t2 in}\\ \texttt{t3}\end{array}\right]\!\!\right]$$

应用图 5 中的规则后，我们得到以下程序：let <t1, t1'> = < x1*y1, x1*y1' + x1'*y1> in let <t2, t2'> = <x2*y2, x2*y2' + x2'*y2> in let <t3, t3'> = <t1+t2, t1'+t2'>in t3'

注意 let 绑定结构只是语法糖；并不会创建成对元素（参见图 4 ）.

张量构造。由于选择不在语言中纳入成对元素，我们便无法将向量微分为成对向量（即结构体数组）。相反，类型为 **张量** n 的表达式会被微分为类型相同的表达式。一个有趣的张量微分规则是我们的求和规则；在该规则中，我们需要访问微分后范围内对应的元素，正如以下示例所示。

示例 1（续）。让我们回到两个向量 V1 和 V2 的点积。微分变换表示如下：

ℱ⟦ 求和(<i,a> 属于 V2) V1(i) * a ⟧

应用微分规则后，得到以下程序：

```
求和(<i,a>属于V2)令 <i',a'
> = <0,V2'(i)>V1(i) * a'  +
V1'(i)* a
```

为了计算该函数相对于某个向量输入（例如 V1）的梯度，我们需要反复将 V1' 设为一个独热向量，使其在索引 j 处为 1、其他位置均为 0，并将 V2' 设为零向量。这需要针对不同的独热向量多次运行前向模式自动微分，计算开销很大。先前的研究 [Shaikhha et al. 2019] 展示了如何通过将向量构造包裹在前向模式自动微分外部并应用循环优化来优化这一过程。

示例 2（续）。考虑标量-向量乘积的情况，在 SDQLite 中表示为 s * V。对该程序应用标量前向自动微分规则后，得到：

```
s * V'  +s' * V
```

对于相对于 V 求微分的情况，与点积示例类似，需要反复将所有不同的独热向量作为 V' 传入。不过，相对于 s 求微分时，只需将 s' 设为 1，并将V' 设为零[张量 1]。

接下来，我们展示一种替代的微分变换，使原生张量化前向模式自动微分成为可能。

4.2 张量化前向模式变换

由于张量是 SDQLite 中的一等公民，我们可以直接表示对类型为 τ（以 $\mathcal{D}_\tau[\![]\!]$ 表示）的张量变量进行微分。这意味着，对于类型为 **张量** n 的表达式，关于类型为 **张量** m的张量求导，结果将是一个 **张量** (n + m)，其类型与外张量积相同（参见图 6 中的类型规则）。

图 6 展示了张量化 FAD 的规则。这些规则推广了标量前向模式自动微分的规则；将 τ 设为 **实数**即可还原后者。主要区别在于实常数和乘法的规则。张量化 FAD 不再仅返回一个实数值 0，而是返回类型为 τ 的零值，表示为 **零**[τ]。对于乘法，如果 τ 是阶数非零的张量类型，第一项仍计算 e1 与 e2的导数的乘积。然而，第二项需要重新排列张量的索引。只允许输入程序中的实数相乘即可避免这一复杂问题。通过应用乘法规范化（参见第 5.4 节）即可实现这一点。

示例 1（续）。在我们的贯穿示例中，张量化前向自动微分变换表示为$\mathcal{D}_\tau[\![$**求和**(<i,a> **属于**V2) V1(i) * a$]\!]$。

应用图 6 中的规则后，我们得到：**求和**

```
(<i,a>属于V2)令 <i',a'> = <0,
V2'(i)>V1(i) * a'  + V1'(i)* a
```

如你所见，这段程序看起来与标量 FAD 生成的版本完全相同。区别在于 V1' 和 V2' 的类型。在标量 FAD 中，它们的类型与 V1 和 V2 相同，也就是 **张量** 1。在张量化 FAD 中，它们的类型是 **张量** 2。因此，最后一个表达式中的乘法现在是标量-向量乘法。

与标量 FAD 不同，我们需要将所有迭代中的零向量和独热向量组合起来；不再逐个传递向量，而是将它们作为一个整体矩阵传递，其中每一行代表一个独热向量。图 7 中 **独热**[**张量** 1] 的定义说明了如何为变量 x 构建这样的矩阵。

示例 3. 考虑计算矩阵 M 的迹。SDQLite 中该表达式的张量化 FAD 表示为 $\mathcal{D}_\tau[\![$ **求和**(<i,r> $\in$ M) r(i)$]\!]$。

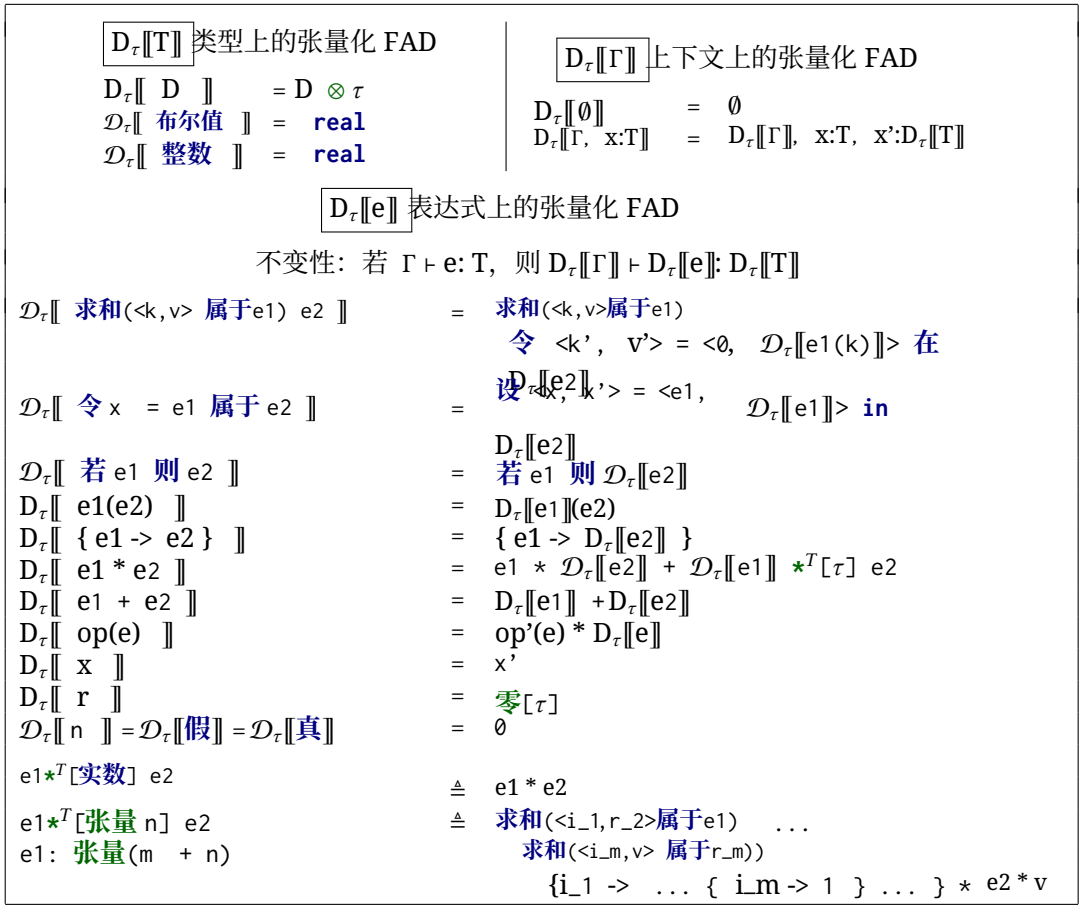


图 6. SDQLite 表达式的张量化前向自动微分（FAD）规则。类型τ 需要是张量类型，即遵循 D 的语法。图 5 的规则是$\mathcal{F}[\![]\!]$=$\mathcal{D}$实数$[\![]\!]$。

对于矩阵 M 中索引为 i 的每一行 r，我们计算由 r(i) 指定的对角元素的求和。经过张量化 FAD 变换后，我们得到以下程序：

```
求和(<i,r>属于M)
  令 <i',r'> = <0,M'(i)>
     r'(i)
```

要计算矩阵的独热输入，我们需要将向量的情况推广。在对向量（张量 1）求导时，我们传入一个矩阵（张量 2）作为独热输入。而在对矩阵（张量 2）求导时，我们需要传入一个四阶张量（张量 4）。独热独热[张量 2] 的定义也可以在图 7 中找到。

4.3 将所有内容整合起来

使用机器学习框架的程序员无需关注后台采用的源到源转换。为此， ∇SD 通过以下方式提供高级应用程序接口：梯度宏，它接受两个输入。第一个输入是 SDQLite

```
                                 让 <v1',...,
梯度 e (x: τ)          ≜                  vn'> = <ingrad v1 x,...,ingrad vn x> in
                                 D_τ⟦e⟧
                                 {v1, ...,      vn} = FVS(e)
                 其中
   输入梯度 (v:D)x      ≜  零[D]
   输入梯度 (x:D)x      ≜  独热[D] x
    独热[实数] x        ≜  1
独热[张量 1] x         ≜  求和(<i,_>属于x) { i -> {i -> 1} }
独热[张量 2] x         ≜  求和(<i,v>属于x) { i ->
                            求和(<j,_>属于v) {j ->
                              {i -> {j -> 1} } } }
独热[张量n] x          ≜  求和(<i_1,v_2>在 x) { i_1 ->
                 如果 n > 2   求和(<i_2,v_3> 属于v_2) { i_2->
                              ... 求和(<i_n,_> 在 v_n)) {i_n ->
                                    {i_1 -> ... { i_n -> 1 } ... }
```

图 7. 梯度API 由 ∇向程序员公开。

表达式，第二个输入则是我们要针对其执行微分的自由变量。

考虑计算表达式 e 关于 x 的梯度。这表示为 gradient e x。梯度宏会为表达式 e 的所有自由变量 (FVS(e)) 的微分分量生成 let 绑定。自由变量 v 的 let 绑定的右侧是 ingrad v x。ingrad 宏负责计算输入零值或独热值，具体取决于变量 v 是否与 x 不同。前一种情况下，会传入 v 的类型来使用 zero 宏；后一种情况下，则使用 onehot 宏。对于类型为 tensor n 的输入变量，onehot 会组装一个稀疏张量 (2*n)，其中输入变量 x 的 n 个索引各重复两次，从而将超立方体的对角线设为 1。

示例 1（续）。我们的贯穿示例指定为 b 通过高级 API，如下所示：

梯度（sum(<i, a> ∈ V1) a * V2(i)）V2 展开后得到以下表达式：let V1' = 零[{int -> 实数}] let V2' = 独热[{int -> 实数}] V2 sum(<i, a>∈ V1) let <i', a'> = <{}, V1'(i)>in a * V2'(i) + a' * V2(i) 接着又展开为：let V1' = {} let V2' = sum(<i,_> ∈ V2) {i -> {i -> 1}} sum(<i, a>∈ V1) let <i', a'> = <{}, V1'(i)> in a * V2'(i) + a' * V2(i)

在下一节中，我们将看到如何优化这个表达式。请注意，在这个示例中，仅通过简单的常量传播，渐近复杂度就已经是 $O(n)$。事实上，我们只会进行通用优化，这些优化并非自动微分专用。

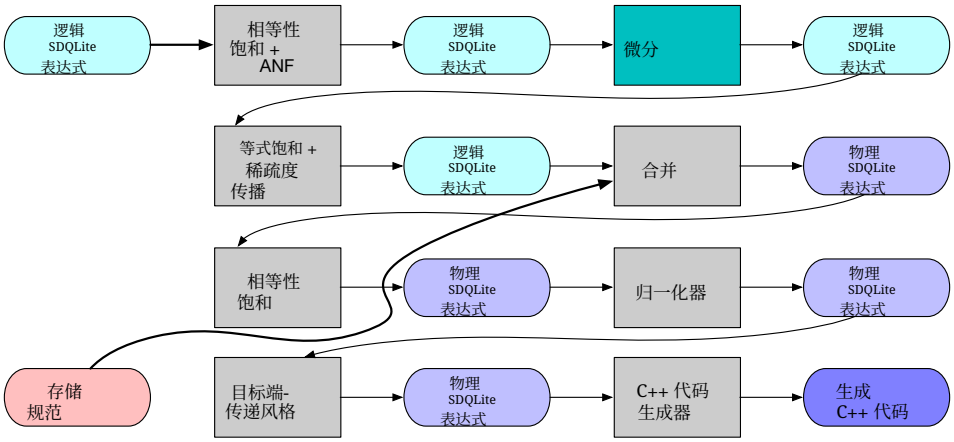


图 8. ∇SD 中的变换。

5 性能

本节将回顾我们用于提升性能的技术。图1展示了我们方法的简化概览，而图8则更详细地呈现了应用于 ∇SD 的一系列变换。

5.1 微分前处理

等式饱和。 首先， ∇SD 对输入程序应用代数重写规则，以利用因式分解、循环融合等优化 [Schleich 等人 2022]。我们采用等式饱和，避免阶段排序问题，并确保重写规则在全局范围内应用 [Tate 等人 2009]。我们使用 EGG [Willsey 等人 2021]，这是一种先进的等式饱和实现，已成功用于 SDQLite 中面向张量程序的灵活存储规范 [Schleich 等人 2022]。我们依赖于 [Schleich 等人 2022]所指定的成本模型和代数重写规则；无需指定任何 AD 专用成本模型或重写规则。

ANF。然后， ∇SD 应用 ANF 变换 [Flanagan 等人 1993], 提出的方法，确保子表达式都是简单表达式，即常量值或变量引用。若子表达式还不是简单表达式，则通过 let 绑定来实现这一点。随后，将经过 ANF 变换的程序输入第 4 节介绍的微分变换。

5.2 微分后处理

稀疏性传播。微分后，会构造出许多中间的高度稀疏值（例如，零张量）。虽然可以编写局部重写规则来简化这些值，但这类程序规模很大，使用等式饱和对其进行优化需要很大的搜索空间。因此，我们将这些稀疏性传播规则作为一个独立的处理步骤应用。图 9 中第一组重写规则对应于稀疏性传播规则。最后，我们再次将程序交由等式饱和处理。

示例 1（续）。在点积示例中，假设采用第二种变体（遍历第二个向量），则相对于第一个向量的微分后程序如下：

```
令 V1' = 求和(<i, a> 属于 V1) {i -> {i -> 1}}
令 V2' = {}
```

重写规则			条件
零[D1] *e2	⇝	零[D1 ⊗ D2]	如果 e2：D2
e1* 零[D2]	⇝	零[D1 ⊗ D2]	若 e1：D1
e1 + 零[D1]	⇝	e1	必须满足 e1：D1
零[D1] +e2	⇝	e2	必须有 e2: D1
零[张量 n](e2)	⇝	零[张量 (n-1)]	必须有 n > 0
令x=为零[D]中e2	⇝	e2[x→零[D]]	
求和(<k,v>属于零[D])e2	⇝	零[D2]	如果 e2:D2
e1 * e2	⇝	求和(<i_1,v_2>属于e1) ... 求和(<i_n,v> 属于v_n)) {i_1->...{i_n-> v*e1}...}	如果e1：张量 n 和 n > 0
e1 * e2	⇝	求和(<i_1,v_2>属于 e2) ... 求和(<i_n,v> 属于v_n)) {i_1->...{i_n-> e1*v}...}	如果e1：实数 且e2：张量 n 且 n > 0

图9。∇SD 中使用的变换。第一组重写规则与稀疏性传播有关。第二组对应于乘法规范化。

```
求和(<i, a>属于 V2)令 <i', a'> = <
{}, V2'(i)> 属于V1(i) * a'  +
V1'(i)* a
```

稀疏性传播优化会传播 V2' 并简化相关表达式（例如 V1(i) * a'），如下所示：

```
let V1' = 求和(<i, a> 属于 V1) {i -> {i -
> 1}}求和(<i, a>属于 V2)V1'(i) *a
```

应用等式饱和后，如果 V1 是稠密向量，程序会得到优化，并简化为 V2；一个优化程度稍低的版本如下：

```
求和(<i, a>属于
V2){i -> a}
```

5.3 存储组合

优化求导后的表达式后， ∇SD 会使用存储规范来定义每个输入张量。图10 展示了 SDQLite 中几种存储格式的规范。 ∇SD 会生成 let 绑定，将每个输入张量绑定到用于指定其物理存储格式的表达式。然而，由于不必要的中间张量分配，这反而会使性能变得更差。幸运的是，先前的研究 [Schleich 等人 2022] 展示了等式饱和不仅可以移除这些中间张量，还能恢复稀疏张量计算中广为人知的算法。

示例1（续）。在前面的示例中，假设 V1 是一个稠密向量，而 V2 是一个使用 COO表示的稀疏向量，将存储规范与微分后的程序结合，结果如下：

```
let V1 = 求和(<_, i>in (0:V1_len)) { unique(i) -> V1_V(i) } in let V2 = 求和(<_,
i>in (0:V2_len)) { unique(V2_row(i)) -> V2_val(i) } in 求和(<i, a>in V2)

  { i -> a }
```

存储格式（输入）	**物理 SDQLite 定义**
COO 向量 （len,row,val）	**求和**(<_, i> **在**(0:长度)) {**唯一**(row(i)) -> val(i) }
稠密向量 (len,V)	**求和**(<_, i> **在**(0:长度)) {**唯一**(i) -> V(i) }
CSR 矩阵 (len,pos,idx,val)	**求和**(<_, i> **在**(0:len)) {**唯一**(i) ->**求和**(<p, j> **在**idx(pos(i):pos(i+1))) {**唯一**(j) -> val(p) } }
行主序稠密矩阵 (行,列,M)	**求和**(<_, i> **在**(0:行)) {**唯一**(i) ->**求和**(<_, j> **在**(0:列)) {**唯一**(j) -> M(i*cols+j) } }

图10. Physical SDQLite 中各种稀疏/稠密格式的函数式实现。CSR 矩阵和稠密行主序的表达方式分别与压缩稀疏列和稠密列主序类似。

通过应用等式饱和， ∇SD 返回了以下程序，不会引入不必要的中间张量，正如预期的那样 [Schleich 等人所述 2022]:

```
求和(<_, i>属于(0:V2_len)){V
2_row(i) -> V2_val(i) }
```

5.4 规范化

∇SD 会对优化后的存储格式感知程序执行额外的底层变换。首先，乘法规范化会将张量外积重写为包含标量乘法的求和表达式。图9第二组规则展示了乘法规范化的重写规则。与循环融合规则结合后，乘法规范化会移除不必要的中间张量。第二种规范化涉及 ANF 变换，这些变换此前已被等式饱和和其他变换抵消。

示例 5 - BATAX。BATAX 内核 [Nelson 等人 2015] 将其表示为 $\beta A^T A x$，其中 β 是一个标量值， A 是一个矩阵， x 是一个向量。该内核在 SDQLite 中的表达如下:

```
求和(<i, r>属于A)求和(<j,v1> 属于r)求和(<
k,v2> 属于r){ j -> ((beta * v1) * v2)*
(x(k)) }
```

之后 对 x 求微分后，经过微分后处理优化， ∇SD 得到:

```
β * (求和(<i,    r> 在 A) r* r)
```

考虑 A 的 CSR 表示和 x 的稠密表示，在应用存储组合和等式饱和后，我们推导出:

```
beta *求和(<_, i> 中(0:A_长度))
  令 r  = 求和(<p, j>属于 A_VCol(A_VRow(i):A_VRow((i  + 1))))
    {j ->A_VVal(p) } 在
  r * r
```

β 与求和结果的乘法是一个标量-向量乘积。此外，最后一个表达式 r * r 对应于一个向量外积。乘法规范化会重写这两个表达式，结果如下：

```
求和(<_, i>in (0:A_len)) let r = 求和(<p, j>in A_VCol(A_
VRow(i):A_VRow((i + 1)))) { j ->A_VVal(p) } in 求和(<i1, v1>
in r) { i1 ->求和(<i2, v2> inr) { i2 -> beta * v1 * v2 } }
```

5.5 目标传递风格与 C++代码生成

最后一步， ∇SD 会生成 C++ 代码。索引和标量类型 int 与 real 分别转换为 size_t 和 double。类型为 { int -> real } 的表达式会转换为 dict_type<size_t, double> 类型的对象。嵌套字典会递归转换，例如，{ int -> { int -> real } } ⇝ dict_type<size_t, dict_type<size_t, double>>。C++ 运行时使用 Robin Hood 字典 [rbd 2023] 。键类型为 dense_int 的字典会转换为 arr_type，而不是 dict_type。例如，{ dense_int -> real } 会转换为 arr_type<double> ，而 { dense_int -> int } 则会转换为 arr_type<size_t>。

SDQLite 大多数构造的 C++代码生成都很直接。构造`dict(key)` 会被翻译为 `dict[key]`, ，这在 C 中对应字典或数组查找++。加法和乘法构造会转换为 C 中相同的基本操作++。这需要支持字典类型上的 `+ + *` ，我们的 C++ 运行时提供了这种支持。构造 **sum** 会被翻译为 for 循环。同样，嵌套求和会被翻译为嵌套 for 循环。如果求和的范围表达式是范围构造 `(st:en)` 或子数组 `arr(st:en)`，则会生成标准的 for 循环。不过，如果它是字典，则会生成遍历该字典的 for-each 构造。另一个有趣的例子是单例字典构造的代码生成。该构造主要用于求和内部。在这种情况下，它会被翻译为如下的字典更新：

```
令 res = 字典_类型<大小_t, double>res;求和(<k,v> 属于 dict) { f(k,v) -> g(k,v) }
⇝ for(auto&kv:dict){ res[f(kv.first,kv.second)] +=g(kv.first,kv.second); }
```

请注意，如果 g(k,v) 输出字典，上述翻译就能正常工作。这是因为 += 也针对字典进行了重载。

示例 1（续）。 微分后的程序生成的 C++ 代码如下：

```
void VVD_关于_V1(
    // 稠密向量 V1 的输入 arr_type<double>& V1_V, size_t V1_len, // 稀疏向
量 V2 的输入 arr_type<size_t>& V2_VRow, arr_type<double>& V2_VVal,
size_t V2_len, // 目标字典 dict_type<size_t, double>& result) { for(size_t i = 0;
i < V2_len; i++) { result[V2_VRow[i]] += V2_VVal[i]; } }
```

为了生成高效的 C++, ∇SD， ∇SD 运用了目标传递风格（DPS）技术 [Minamide 1998] ，方式有两种。首先，为生成的函数提供用于存储最终结果的目标

对象 [Shaikhha 等人 2017]。其次，目标传递风格变换会移除内层循环中创建的中间张量。具体做法是将所有单例字典构造推入内层循环，以避免不必要的中间字典构造 [Shaikhha 2022]。

示例 4（续）。在 BATAX 内核中，C++代码生成会生成以下程序：

```
void BATAX_stg_wrt_X( // 输入标量 double& beta_S, // CSR 矩阵 A 的输入 arr_type<size_t>& A_VRow, arr_type<size_t>& A_VCol, arr_type<double>& A_VVal, size_t A_len, // 稠密向量 X 的输入 arr_type<double>& X_V, size_t X_len, // 目标字典 dict_type<size_t, dict_type<size_t, double>>& result) { for(size_t i = 0; i < A_len; i++) { dict_type<size_t, double>r; for(size_t p = A_VRow[i]; p < A_VRow[(i + 1)]; p++) { size_t j = A_VCol[p]; r[j] += A_VVal[p]; } dict_type<size_t, dict_type<size_t, double>>tmp1; for(auto& iv1 : r) { dict_type<size_t, double>tmp2; for(auto& iv2: r) { tmp2[iv2.first] += beta_S * (iv1.second * iv2.second); } tmp1[iv1.first] += tmp2; } result += tmp1; } }
```

注意，最后一个内层循环（已突出显示）需要构造两个中间字典 tmp1 和 tmp2。可以利用半环字典的结合律重新排列插入操作，使所有插入操作都在最内层循环中直接写入 result：

```
对于每个(自动& iv1 : r) {对于每个(自动& iv2: r) {结果[iv1.first][
```

这样就不必分配中间字典 tmp1 和 tmp2。我们将在第 7 节看到这一优化的影响。

6 证明

在本节中，我们首先证明自动微分的类型可靠性。然后，我们说明 Logical SDQLite 中的程序表示可微函数，其中可微性的推广保守地扩展了欧几里得空间之间的可微性。最后，我们证明标量和张量化前向模式自动微分的正确性。

6.1 微分的类型可靠性

L引理 6.1（$D_\tau[\![\,]\!]$ 的类型正确性）。若 $\Gamma \vdash e{:}T$ 则 $D_\tau[\![\Gamma]\!] \vdash D_\tau[\![e]\!] {:} D_\tau[\![T]\!]$。

Pʀ哎。对 e 的类型推导树进行归纳，并按情况分析。有趣的情况是 e:

情况*： 考虑e1 * e2：**张量**m和τ =**张量**n。我们需要证明$\mathcal{D}_\tau[\![$e1 * e2$]\!]$：**张量**(m+n)。根据归纳假设和类型规则，我们有如下结论：

- e1：**张量** m1 ⇒ $\mathcal{D}_\tau[\![$e1$]\!]$：**张量** (m1+n)
- e2：**张量** m2 ⇒ $\mathcal{D}_\tau[\![$e2$]\!]$：**张量** (m2+n)
- e1：**张量** m1, e2：**张量** m2 ⇒ e1 * e2：**张量** (m1+m2) ~~where~~ m1+m2 = m

通过考察 $D_\tau[\![$e1 * e2$]\!]$ 的子表达式，我们有：

- e1 * $\mathcal{D}_\tau[\![$e2$]\!]$：**张量** (m1+(m2+n)) ⇒ e1 * $\mathcal{D}_\tau[\![$e2$]\!]$：**张量** ((m1+m2)+n)
- $\mathcal{D}_\tau[\![$e1$]\!]$ $*^T[\tau]$ e2：**张量**((m1+m2)+n)

因此，这两个子表达式相加后也具有相同的类型，即 **张量**((m1+m2)+n)。由于 m1+m2 = m，输出类型为 **张量**(m+n)。

情形 求和： 考虑 Γ ⊢ **求和**(<k, v> **属于** e1) e2：**张量**m 和 τ = **张量**n。我们需要证明 $\mathcal{D}_\tau[\![\Gamma]\!] \vdash \mathcal{D}_\tau[\![$**求和**(<k, v>属于 e1) e2$]\!]$：**张量**(m+n)。根据归纳假设和类型规则，我们有如下结论：

- Γ ⊢ e1：**张量**(m1+1) ⇒ $\mathcal{D}_\tau[\![\Gamma]\!] \vdash \mathcal{D}_\tau[\![$e1$]\!]$：**张量**(m1+1+n)
- Γ′ = Γ,k：**整数**,v：**张量** m1
- Γ″ = $\mathcal{D}_\tau[\![\Gamma]\!]$,k：**整数**,v：**张量** m1,k'：**实数**,v'：**张量** (n+m1)
- Γ′ ⊢ e2：**张量** m ⇒ Γ″ ⊢ $\mathcal{D}_\tau[\![$e2$]\!]$：**张量** (m+n)

将**求和** 中经过微分后的主体记为 e'：

- $\mathcal{D}_\tau[\![$**sum**(<k, v> in e1) e2$]\!]$ =
  sum(<k,v> **in** e1) **let** <k',v'> = <0,$\mathcal{D}_\tau[\![$e1$]\!]$(k)> **in** $\mathcal{D}_\tau[\![e2]\!]$ =
  sum(<k,v> **in** e1) e' ⇒
  e' = **let** <k',v'> = <0,$\mathcal{D}_\tau[\![$e1$]\!]$(k)> **in** $\mathcal{D}_\tau[\![e2]\!]$

根据归纳假设，我们知道 Γ′ ⊢ e'：**张量** (m+n)。

因此，我们推导出 Γ ⊢**求和**(<k,v> **属于** e1) e'：**张量** (m+n)，这正是我们需要证明的。 □

本节其余部分将深入探讨更多数学细节，非专业读者可以跳过。本节旨在将张量化 FAD 能计算梯度这一论证形式化。我们首先将 Logical SDQLite 程序解释为函数，并定义这类函数如何计算导数和梯度。然后，我们将可微性和微分的定义扩展到通常针对笛卡尔空间之间函数所采用的定义之外。最后，我们使用逻辑关系论证，证明 $D_\tau[\![$e$]\!]$ 表示 e 的微分，因此能够产生正确的梯度。

6.2 语义与可微性

非光滑语义。我们可以将 Logical SDQLite 的指称语义视为 [Shaikhha 等人 2022]所给语义的一个特例。类型 real 被解释为实数 $\mathbb{R}$，而程序 x: real ⊢ e: real 被解释为函数 $\mathbb{R} \to \mathbb{R}$。我们用语义括号 $[\![-]\!]$ 表示类型和项的指称语义。我们将 bool 解释为布尔值 $\mathbb{B}$，将 int 解释为自然数$\mathbb{N}$。tensor n 的语义通过归纳定义，其中 $[\![${ int -> D }$]\!]$ 由可数集 $\mathbb{N} = [\![$int$]\!]$ 上的自由 $\mathbb{R}$ 向量空间与实向量空间 $[\![$D$]\!]$的张量积给出。这个空间只在实向量空间同构的意义下定义，因此我们可以自由地将$[\![$tensor n $]\!]$ 选为 I×...×I 上的自由实向量空间，其中 I n 是 I 与自身的 n 重笛卡尔积。将上下文 $[\![\Gamma]\!]$ = {x1: T1, . . ., xn:Tn} 解释为 $[\![\Gamma]\!] = [\![T1]\!] \times \ldots \times [\![Tn]\!]$后，程序Γ ⊢e: T 表示函数 $[\![e]\!] : [\![\Gamma]\!] \to [\![T]\!]$。

图 9 中重写规则的正确性显而易见，因为它们都可视为相关工作中所给优化的特例 [Schleich 等人，2022年；Shaikhha 等人 2022]，而且它们依赖于我们字典的半环结构。

光滑语义。非光滑语义的局限在于，它无法告知我们程序是否可微。函数 $f\colon \mathbb{R}^n \to \mathbb{R}^m$ 若可微且其导数光滑，则称为光滑函数。我们可以通过将 Logical SDQLite 的类型解释为笛卡尔空间的一种推广——称为微分空间 [Iglesias-Zemmour 2013],——并将函数解释为微分空间的态射，来细化其语义。微分空间的态射可以视为这些空间之间的广义光滑函数。这里我们紧随 [Huot 等人 2020]的做法。更准确地说，微分空间是一个二元组 (X, P^X)，其中 X是一个集合，而对于欧几里得空间 $\mathbb{R}^n$的所有子集 U，都有相应的函数集合 $\mathrm{P}^X_U \subseteq X^U$，这些函数称为绘图。每个这样的绘图集合 P^X_U都必须包含所有常值函数，在光滑函数的前复合下封闭，并且在可数粘合下封闭。最后一项的含义是：若 $(f_i\colon U_i \subseteq \mathbb{R}^n \to X)_{i\in\mathbb{N}}$是一族可数的绘图，并且对于每个$i$, $j \in \mathbb{N}$、 f_i和 f_j在 $U_i \cap U_j$上相等，那么 $f\colon \bigcup_i U_i \to X$， $x \in U_i \mapsto f_i(x)$就是一个绘图。微分空间 $(X, \mathrm{P}^X) \to (Y\mathrm{P}^Y)$之间的态射 f是满足以下条件的函数 $f\colon X \to Y$：对于所有 $p \in \mathrm{P}^X_U$，都有 $f \circ p \in \mathrm{P}^Y_U$。 $\mathbb{N}$或 $\mathbb{B}$ 若以常值函数为绘图，便构成微分空间。 $\mathbb{R}$若以光滑函数 $U \to \mathbb{R}$ 作为绘图 $\mathrm{P}^{\mathbb{R}}_U$，便是一个微分空间。微分空间 $(\mathbb{R}, \mathrm{P}^{\mathbb{R}}) \to (\mathbb{R}, \mathrm{P}^{\mathbb{R}})$之间的态射恰好就是光滑函数 $\mathbb{R} \to \mathbb{R}$。微分空间对乘积封闭，其中绘图 $f\colon U \to X \times Y$由一对绘图 (f_1, f_2)给出，且 $f_1 \in \mathrm{P}^X_U$、 $f_2 \in \mathrm{P}^Y_U$。现在，我们可以通过将类型解释为微分空间、将项解释为态射，来修订指称语义。对于类型的解释，唯一剩下的情形是字典类型。其底层集合与集合解释相同，而绘图定义如下

$$\mathcal{P}_U^{[\![\{I\to D\}]\!]} = \{f : U \to [\![\{I \to D\}]\!] \mid \exists g \in \mathcal{P}_U^{[\![D]\!]^n}, i : [\![D]\!]^n \to [\![\{I \to D\}]\!] \text{ linear injective}, f = i \circ g\}$$

很容易证明，语言中所有构造的语义都保留绘图。因此，我们得到以下推论：

命题 6.2。 类型正确的程序 x1： **实数**， ...， xn： **实数** ⊢ e： **实数** 表示光滑函数 $\mathbb{R}^n \to \mathbb{R}$。

微分。对于函数 $f\colon \mathbb{R} \to \mathbb{R}$，可微函数 f 的微分是一个函数$df\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$，满足对于任意 $x \in \mathbb{R}$， $df(x, -)\colon \mathbb{R} \to \mathbb{R}$ 是 f 在 x 处的最佳线性近似，其特征由斜率 $f'(x)$ 确定。因此， $df(x, v) = f'(x) \cdot v$。更一般地，可微函数 $f\colon \mathbb{R}^n \to \mathbb{R}^m$ 的微分是一个函数 $df\colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m$，满足$df(x, v) = J_x f \cdot v$，其中 $\cdot$ 现在表示矩阵向量乘积。 $J_x f$ 是 f 在每一点处的雅可比矩阵，即 f 在 x 处偏导数构成的$n \times m$矩阵： $J_x f = \left(\frac{\partial}{\partial x_j} f_i(x)\right)_{1\le j\le n, 1\le i\le m}$，其中 f_i 是 f 在其第 i 个分量上的投影。

6.3 标量前向模式的正确性

一阶子语言上的正确性。考虑 Logi- cal SDQLite 中不涉及字典的简单一阶子语言。在这个子语言中，从语义上说，前向模式算法会将 f映射为其微分 df，输入的顺序可能有所调整。更精确地说，若 x1: 实数, . . ., xn: 实数 ⊢ e: 实数，且 x1: 实数, x1': 实数, . . ., xn: 实数, xn': 实数 ⊢ F ⟦e⟧: 实数，则 $[\![\mathrm{F}\ [\![\mathrm{e}]\!]]\!](x_1, x'1, \ldots, x_n, x'n) = J_{(x_1,\ldots,x_n)}[\![\mathrm{e}]\!] \cdot (x'1, \ldots, x'n)^T$，其中 $(-)^T$表示转置。通过对项的结构进行常规归纳即可证明这一点。

超越微分。由于张量的存在，我们程序的语义并不表示函数$\mathbb{R}^n \to \mathbb{R}^m$，因此缺少微分的定义。即使我们只关注形式如下的程序x1： **实数**， ...， xn： **实数** ⊢ e： **实数**，e 内部也可能使用张量，而通过简单归纳进行正确性证明会失效。因此，我们需要定义输入或输出可以是张量的函数的微分。我们称 π： $[\![$**张量** m$]\!] \to \mathbb{R}^k$ 为典范投影，若它是投影到由 $\mathbb{R}^k$ 个元素构成的子集 k 所生成的自由向量空间上的投影，其中该子集属于I。其伴随线性

映射 $i:\ \mathbb{R}^k \to [\![\textbf{张量}\ \mathtt{m}]\!]$ 称为典范嵌入。典范嵌入体现了固定形状张量的直观概念，也就是说，如果将它们存储为稠密张量，就能保证其大小固定。当我们对张量求导时，默认它具有固定形状。同样，在我们的语言中，程序输出的张量大小不能取决于实数输入，因为我们没有从实数到整数的非连续原语。因此，如果程序的所有整数输入都固定，那么它的输出张量就具有固定形状，而这可以用典范投影来刻画。换句话说，对于程序$\mathtt{x}:\ \textbf{张量}\ \mathtt{m} \vdash \mathtt{e}:\ \textbf{张量}\ \mathtt{n}$，存在一个典范投影，它在$[\![\mathtt{e}]\!]$的像上是单射。因此，我们将张量输出函数的微分定义如下：

定义 *6.3.* 函数 $dg: \mathbb{R}^n \times \mathbb{R}^n \to [\![\textbf{张量}\ \mathtt{m}]\!]$ 是 $f: \mathbb{R}^n \to [\![\textbf{张量}\ \mathtt{m}]\!]$ 的微分，当且仅当对于所有典范投影 $\pi: [\![\textbf{张量}\ \mathtt{m}]\!] \to \mathbb{R}^k$, $\pi \circ dg$ 是 $\pi \circ f$ 的微分。

我们可以将此定义扩展到以张量为输入的函数：

定义 6.4. 函数 $df: [\![\text{张量 n}]\!] \times [\![\text{张量 n}]\!] \to [\![\text{张量 m}]\!]$ 是 $f: [\![\text{张量 n}]\!] \to [\![\text{张量 m}]\!]$ 的微分，当且仅当对于所有典范嵌入 $i: \mathbb{R}^k \to [\![\text{张量 n}]\!]$，$dg \circ (i \times i)$ 是 $f \circ i$ 的微分，其中 $i \times i: (x, y) \mapsto (i(x)\ i(y))$。

与通常的可微性情形类似，我们可以将定义扩展到具有多个输入和输出的函数，并采用逐点方式。也就是说：

定义 *6.5.* 函数 $dg: [\![T]\!] \times [\![T]\!] \to [\![T1]\!] \times [\![T2]\!]$ 是函数 $f: [\![T]\!] \to [\![T1]\!] \times [\![T2]\!]$ 的微分，当且仅当 $\pi_i \circ dg$ 是 $\pi_i \circ f$ 的微分。函数 $dg: [\![T1]\!] \times [\![T1]\!] \times [\![T2]\!] \times [\![T2]\!]$ 是函数 $f: [\![T1]\!] \times [\![T2]\!] \to [\![T]\!]$ 的微分，当且仅当对于所有 $x \in [\![T1]\!]$, $dg(x, 0, -, -)$ 都是 $f(x, -): [\![T2]\!] \to [\![T]\!]$ 的微分，并且对于所有 $y \in [\![T2]\!]$, $dg(-, -, y, 0)$ 都是 $f(-, y): [\![T1]\!] \to [\![T]\!]$ 的微分。

现在我们可以表述前向自动微分计算的是微分：

定理 6.6. 设 $\Gamma \vdash \mathtt{e}: \mathtt{D}$，其中 Γ 仅包含张量类型。那么，$[\![\mathcal{F}[\![\mathtt{e}]\!]]\!]$ 是 $[\![\mathtt{e}]\!]$ 的微分。

证明现在可直接由逻辑关系论证得出，如 [Huot 等人 2020]所示。逻辑谓词定义如下：

$$
\begin{aligned}
\mathcal{R}_{\mathtt{real}} &= \{(f: \mathbb{R} \to \mathbb{R}, g: \mathbb{R} \to \mathbb{R}) \mid f \in \mathcal{P}^{\mathbb{R}}_{\mathbb{R}}, g = f'\} \\
\mathcal{R}_{\mathtt{int}} &= \{(f: \mathbb{R} \to \mathbb{N}, g: \mathbb{R} \to \mathbb{R}) \mid f \text{ constant}, g = \lambda x.0\} \\
\mathcal{R}_{\mathtt{bool}} &= \{(f: \mathbb{R} \to \mathbb{B}, g: \mathbb{R} \to \mathbb{R}) \mid f \text{ constant}, g = \lambda x.0\} \\
\mathcal{R}_{\mathtt{\{I\ ->\ D\}}} &= \{(f: \mathbb{R} \to [\![\mathtt{\{I\ ->\ D\}}]\!], g: \mathbb{R} \to [\![\mathtt{\{I\ ->\ D\}}]\!] \mid f \in \mathcal{P}^{[\![\{I \to D\}]\!]}_{\mathbb{R}}, g = df(-, 1), \\
&\qquad df \text{ differential of } f\}
\end{aligned}
$$

根据逻辑关系的基本引理，证明语言中每个构造的前向微分解释都保持逻辑谓词（通过后复合）之后，我们便可得出每个程序的解释都保持逻辑谓词。与 [Huot 等人 2020]的方法相同，我们将该定理作为基本引理的推论得出。更准确地说，对于谓词 $\mathrm{R}_{\mathtt{real}}$，我们选择 $f = id$。对于谓词 $\mathrm{R}_{\mathtt{\{I\ ->\ D\}}}$，对于任意线性单射 $\mathbb{R}^k \to [\![\mathtt{\{I\ ->\ D\}}]\!]$以及任意 $x_1, \ldots, x_{k-1} \in \mathbb{R}^{k-1}$，我们对 $g_i = \lambda x.(x_1, \ldots, x, \ldots, x_{k-1})$取 $f_i = i \circ g_i$，其中 x位于位置 i。

6.4 张量化前向模式自动微分的正确性

视角的转换：τ-微分。给定一个可微函数 $f: \mathbb{R}^n \to \mathbb{R}^m$，我们也可以将其微分视为高阶函数 $df: \mathbb{R}^n \times (\mathbb{R} \multimap \mathbb{R}^n) \to (\mathbb{R} \multimap \mathbb{R}^m)$，其中$A \multimap B$是从 A 到 B的 $\mathbb{R}$-线性函数所构成的实向量空间。给定空间 A, B 的基底后，我们

可以将线性函数 $A \multimap B$对应为矩阵。这样一来，我们有 $df(x,v) = Jf_x \cdot v$，其中 v是一个 $n\times1$ 矩阵，而 $\cdot$ 现在表示矩阵乘法。令 $\tau = \mathbb{R}^k$。我们可以将 $f\colon \mathbb{R}^n \to \mathbb{R}^m$的微分视为函数 $df\colon \mathbb{R}^n \times (\tau \multimap \mathbb{R}^n) \to (\tau \multimap \mathbb{R}^m)$，从而推广前面的定义，该函数仍由 $df(x,v) = Jf_x \cdot v$定义。我们称这些函数为 τ-微分。函数 $f \in \tau \multimap \mathbb{R}^n$可视为从 [Betancourt 2018]出发的 $(k,1)$ 速度。对于有限维向量空间A, B，向量空间 $A \multimap B$与 $A \otimes B$同构，其中 $A \otimes B$ 是向量空间的张量积。因此，定义 $f\colon \mathbb{R}^n \to \mathbb{R}^m$ 的 τ- 微分的另一种等价方式，是将其定义为函数 $df\colon \mathbb{R}^n \times (\mathbb{R}^n \otimes \tau) \to (\mathbb{R}^m \otimes \tau)$。本质上，这正是我们的张量化前向模式自动微分所计算的内容。例如，若 $\tau = \mathbb{R}^n$，通过令 v为单位矩阵，我们可得 $df(x,v) = J_x f$是 f的完整雅可比矩阵；而常规前向模式自动微分通过令 v 为热向量，只计算雅可比矩阵的一列。微分的这一视角也可推广至输入或输出为张量的函数 f，如下所示：

定义 6.7。若对于所有典范投影 $\pi\colon [\![\text{tensor m}]\!] \to \mathbb{R}^k$ 和典范嵌入 $i\colon \mathbb{R}^p \to \tau$，$\lambda(x,l).\pi \circ df(x, l \circ i)$ 是 $\pi \circ f$ 的 $\mathbb{R}^p$- 微分，则函数 $df\colon \mathbb{R}^n \times (\tau \multimap \mathbb{R}^n) \to (\tau \multimap [\![\text{tensor m}]\!])$ 是 $f\colon \mathbb{R}^n \to [\![\text{tensor m}]\!]$ 的 τ-微分。

同理，若对于所有典范投影 $\pi\colon [\![\text{tensor m}]\!] \to \mathbb{R}^k$ 和典范嵌入 $i\colon \mathbb{R}^p \to \tau, j\colon \mathbb{R}^q \to [\![\text{tensor n}]\!]$，$\lambda(x,l).\pi \circ df(\ j(x)\ i^* \circ l \circ i)$ 是 $\pi \circ f \circ j$ 的 $\mathbb{R}^p$-微分，其中 i^* 是 i 的线性伴随，则函数 $df\colon [\![\text{tensor n}]\!] \times (\tau \multimap [\![\text{tensor n}]\!]) \to (\tau \multimap [\![\text{tensor m}]\!])$ 是 $f\colon [\![\text{tensor n}]\!] \to [\![\text{tensor m}]\!]$ 的τ-微分。

然后，按照定义 6.5，它进一步扩展到具有多个输入或输出的函数。现在我们可以表述，张量化前向模式自动微分能够计算微分：

定理 6.8. 设 τ 为张量类型。设 $\Gamma \vdash \mathtt{e}\colon \mathtt{D}$，其中 Γ 只包含张量类型。那么，$[\![\mathcal{D}_\tau[\![\mathtt{e}]\!]]\!]$ 是 $[\![\mathtt{e}]\!]$ 的 τ-微分。

证明沿用了定理 6.6 中类似的逻辑关系论证，只是对逻辑谓词稍作修改，以考虑类型 τ。例如

$$\mathcal{R}^\tau_{\mathtt{real}} = \{f : [\![\tau]\!] \to \mathbb{R}, g : \mathbb{R} \times ([\![\tau]\!] \multimap [\![\tau]\!]) \to ([\![\tau]\!] \multimap \mathbb{R}) \mid f \text{ morphism of diffeological spaces}, \ g\ \tau\text{-differential of } f\}$$

7 实验结果

在本节中，我们通过真实世界和合成张量来考察 ∇SD在实践中的有效性。我们将回答以下研究问题：

- ∇SD在真实世界的稀疏矩阵上执行张量内核时，与最先进的自动微分框架相比表现如何？ ∇SD能否扩展到大型稀疏矩阵?
- 不同的物理存储格式对 ∇SD 的性能有何影响?
- 不同的优化对 ∇SD 的性能有何影响?
- 在哪些密度下，使用稀疏表示并通过 ∇SD 计算梯度才有意义?

7.1 实验设置

实验在一台 MacBook Pro 上进行，该设备搭载 3.5 GHz 双核 Intel Core i7 CPU 和 16 GB LPDDR3 内存（2133 MHz），运行 macOS Big Sur 11.5.2。我们使用 CLang 1205.0.22.11 编译生成的 C++ 代码，并使用 Python 3.7.4 执行 Python 代码。我们使用了 TensorFlow 2.11.0（启用 XLA）和 PyTorch 1.13.1。

我们考虑以下稀疏矩阵内核：

表 1. 端到端实验中使用的真实世界矩阵。

矩阵	维度	密度	# 非零元	描述
bcspwr10	5.3K × 5.3K	$2^{-9.3}$	22K	电力网络
nopoly	10K × 10K	$2^{-9.5}$	70K	无向加权图
pdb1HYS	36K × 36K	$2^{-8.2}$	2.19M	无向加权图
rma10	46K × 46K	$2^{-9.8}$	2.37M	计算流体动力学
cant	62K × 62K	$2^{-9.9}$	2.03M	有限元法
consph	83K × 83K	$2^{-10.1}$	3.05M	有限元法
cop20k_A	121K × 121K	$2^{-12.4}$	1.36M	有限元法

• **BATAX.** 根据以下公式计算的 BATAX 内核 [Nelson 等人 2015]：$f(j) = \Sigma_{i,k} \beta \cdot A(i,j) \cdot A(i,k) \cdot X(k)$。我们考虑其关于向量的梯度：$\frac{\partial f}{\partial X}$。

• **SMMM.** 矩阵乘矩阵运算各元素的求和 [Schleich 等人 2022]：$f = \Sigma_{i,j,k} A(i,k) \cdot B(k,j)$。我们考虑其相对于第二个矩阵的梯度：$\frac{\partial f}{\partial B}$。

• **SMVM.** 矩阵向量乘法结果各元素的求和 [Smeding 和 Vákár 2023]：$f = \Sigma_{i,j} A(i,j) \cdot X(j)$。我们考虑其相对于向量的梯度：$\frac{\partial f}{\partial X}$。

前两个内核生成矩阵结果，而最后一个生成向量。我们还考虑以下来自 d⁻f（[Shaikhha 等人 2019]）的向量内核：

• **VVA.** 两个向量的加法：$f(i) = V_1(i) + V_2(i)$。我们考虑其相对于第一个向量的梯度：$\frac{\partial f}{\partial V_1}$。

• **VVD.** 两个向量的点积：$f = \Sigma_i V_1(i) \cdot V_2(i)$。我们考察其相对于第一个向量的梯度：$\frac{\partial f}{\partial V_1}$。

• **VSM.** 向量-标量乘法：$f(i) = V(i) \cdot s \cdot s$。标量值与自身相乘，使微分过程更加复杂。我们考察其相对于标量值的梯度：$\frac{\partial f}{\partial s}$。

第一个内核产生矩阵结果，而最后两个产生向量。作为矩阵核的竞品，我们考虑以下系统：

• **∇SD（字典）**：由 ∇SD 根据使用嵌套字典的张量化前向模式自动微分生成的 C++ 代码。

• **∇SD(arr)**：由 ∇SD 根据张量化前向模式自动微分生成的 C++ 代码，该方法还利用了稀疏表示指定的物理存储。

• **TensorFlow**：TensorFlow 框架的反向模式自动微分，使用其梯度和雅可比矩阵 API。

• **PyTorch**：PyTorch 框架的反向模式自动微分，使用其雅可比矩阵 API。

对于向量核，除了上述竞争方案外，我们还考虑以下系统：[1]

• **Tapenade (R)**：由 Tapenade 使用反向模式自动微分生成的 C 代码。

• **Tapenade (F)**：由 Tapenade 使用前向模式自动微分生成的 C 代码。

• **d⁻ғ**：由 d⁻f 使用其前向模式自动微分生成的 C 代码。

• **d⁻ғ + DPS**：由 d⁻f 使用其前向模式自动微分及 DPS 优化生成的 C 代码。

我们的研究使用了真实世界数据集和合成数据集。为获取前者，我们从 SuiteSparse 矩阵集合中收集了七个稀疏矩阵 [Davis 和 Hu 2011]。表 1 总结了这些数据集。对于合成数据，我们创建了密度和维度配置各不相同的随机矩阵和向量。

[1]T生成代码可在此处的 d⁻f 获取：https://github.com/amirsh/autodiff2/tree/master/icfp19

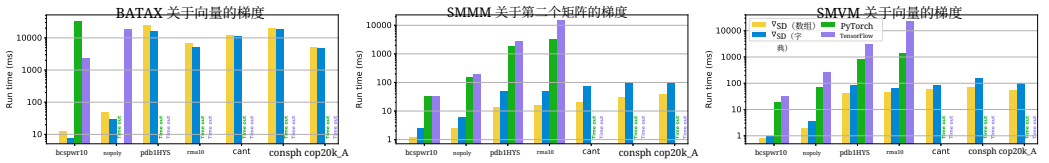


图 11. 对真实世界稀疏矩阵上的内核进行微分的性能结果。 ∇SD 在稀疏矩阵上使用张量化前向模式，而 TensorFlow 和 PyTorch 在稠密矩阵上使用反向模式。

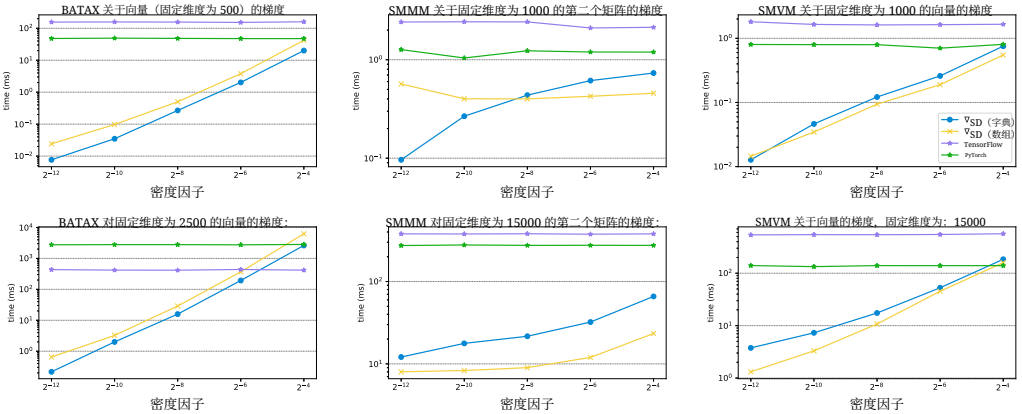


图 12. 通过改变稀疏度，对不同稀疏矩阵内核进行微分的性能结果。

在所有这些基准测试中，我们考虑相对于一个稠密变量的梯度。所有实验均使用单核。我们取五次运行的平均时间。

7.2 真实世界数据集上的基准测试

在本节中，我们考察真实世界的稀疏矩阵，并将 ∇SD 与 TensorFlow 和 PyTorch 进行比较。尽管这两个系统都支持稀疏运算，但都不支持对稀疏矩阵进行自动微分。

图11展示了真实世界稀疏矩阵的结果。我们得出以下观察。首先，对于所有内核， ∇SD 都凭借利用稀疏表示而优于 TensorFlow 和 PyTorch。其次， ∇SD 生成的、利用基于数组的物理表示的代码，在 SMVM 和 SMMM 中比使用嵌套字典的版本运行得更快。最后，与 TensorFlow 和 PyTorch 不同， ∇*SD* 可以扩展到大型稀疏矩阵；由于需要存储整个矩阵（包括零元素），TensorFlow 和 PyTorch 无法处理这类矩阵。

7.3 合成数据集上的基准测试

在本节中，我们使用合成数据集，以便更好地分析稀疏度和维度的影响。我们考察两种情形：（1）固定维度并改变稀疏度；（2）固定稀疏度并改变维度。

矩阵核。 图12展示了针对小型和大型矩阵，通过改变稀疏度得到的矩阵核结果。我们有以下观察。首先，对于低密度（其范围与真实世界数据集相同）， ∇SD 相较于 TensorFlow 和 PyTorch 有明显优势。其次，正如预期，TensorFlow 和 PyTorch 的性能不受稀疏度影响。随着矩阵变得更加稠密，这两个框架与 ∇SD 之间的差距会缩小。最后，对于 BATAX 内核，基于字典的表示优于基于数组的表示

∇SD：稀疏张量的可微编程

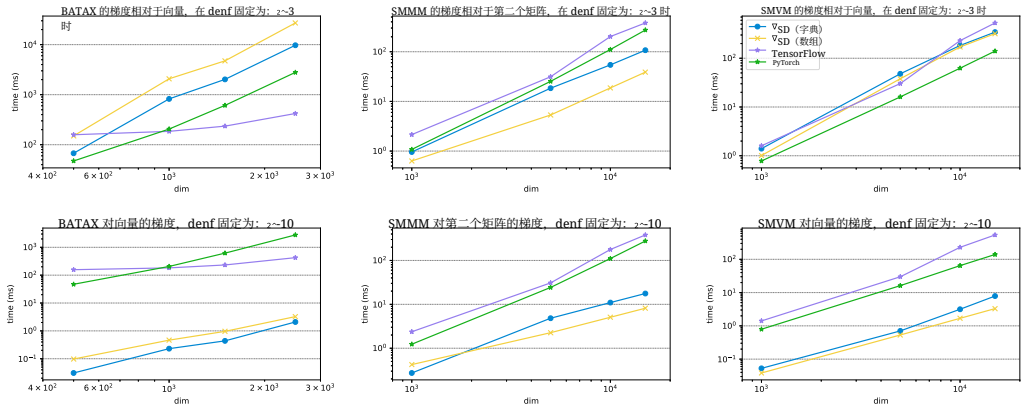


图 13。 性能结果　　不同稀疏矩阵内核的微分　　通过改变维度。

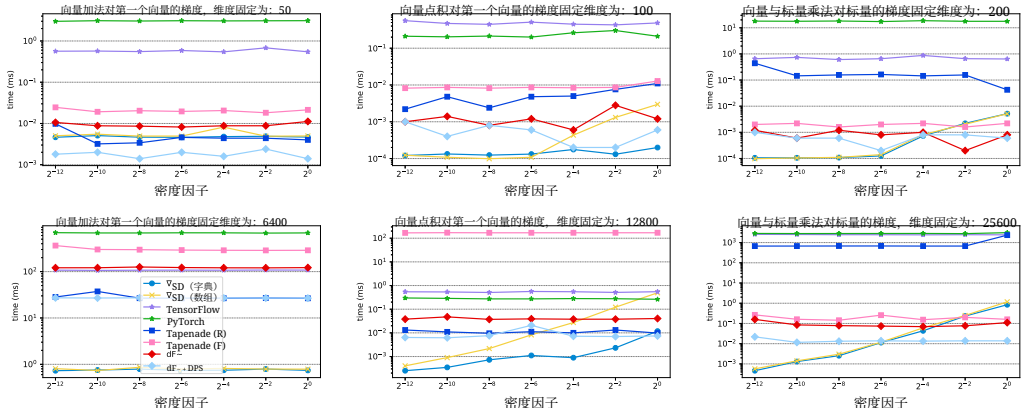


图 1　4. 不同稀疏向量内核在稀疏度变化时的微分性能结果　　ity.

不过，对于另外两个内核，尤其是处理较大矩阵时，基于数组的表示表现更好。

图13展示了在两种不同稀疏度下，改变维度所得到的结果。对于高密度矩阵，TensorFlow 和 PyTorch 的性能优于 ∇SD，且维度越大，差距越明显。然而，对于低密度矩阵，我们观察到了相反的影响。

向量核。 图 14 和图 15 展示了改变稀疏度和密度时向量核的结果。对于较小的维度， ∇SD 并无明显优势；对于 VVA， d⌒f 的表现优于 ∇SD。然而，对于较大的维度，我们观察到 ∇SD 具有明显优势，尤其是在低密度矩阵上。对于 VSM，前向模式系统明显优于反向模式系统；在低密度情况下， ∇SD 的表现优于其他系统，而在高密度情况下，d⌒f 是性能最好的系统。

7.4 优化的影响

本节探讨优化对 ∇SD 性能的影响。对于每种数据表示，我们考虑生成的 C++代码的以下几种情况：（1）不进行低级转换；（2）进行乘法规范化；（3）进行乘法规范化，并使用 DPS 移除嵌套循环中的中间张量。

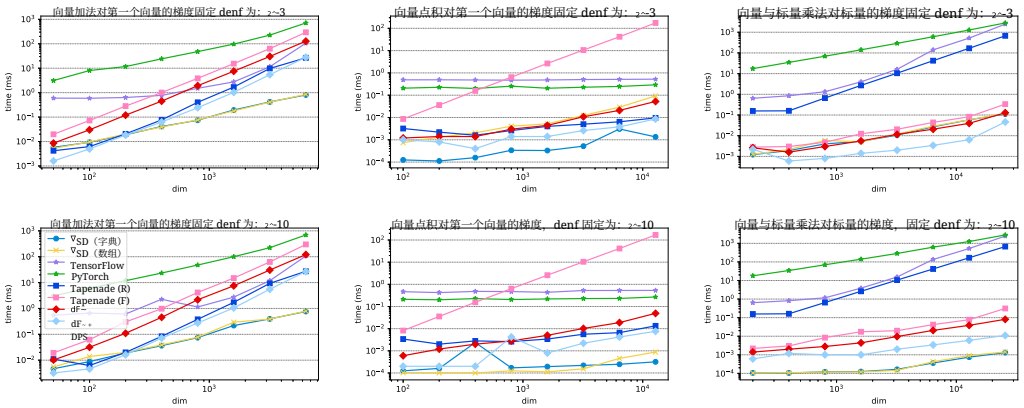


图15。 性能结果　　不同稀疏向量内核在不同维度下的微分　　通过改变维度来处理通道。

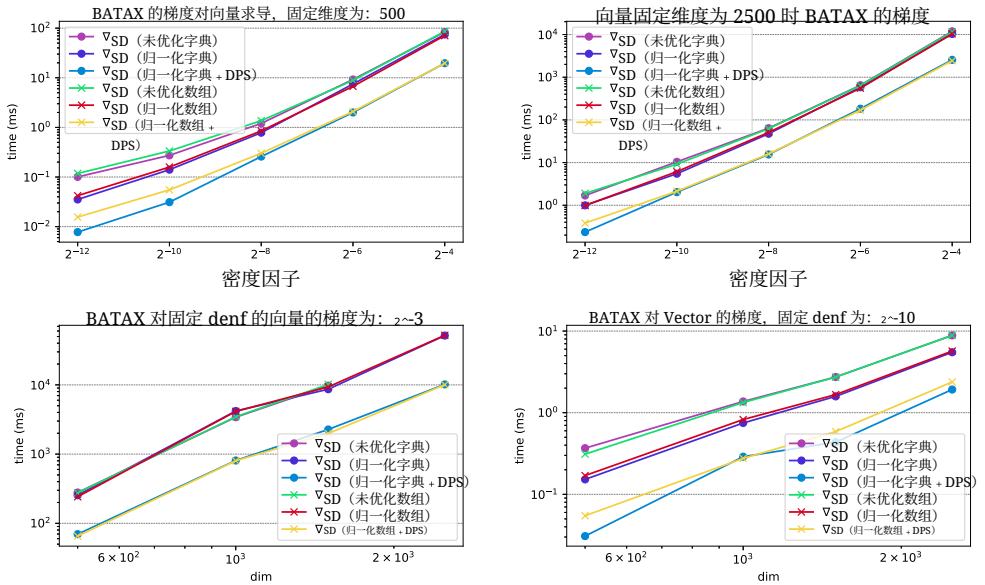


图 16. 规范化和 DPS 对 BATAX 内核上 ∇SD 性能的影响。

图16展示了 BATAX 内核的结果。与此前几组实验类似，我们发现，基于字典的表示略优于基于数组的表示。这是因为基于数组的版本会创建一个中间字典（参见第 5.5 节）。此外，每项优化都能提升性能。尤其是在密度较低时，优化带来的影响更加显著。

8 相关工作

自动微分。 目前已有多种面向命令式和函数式编程语言的自动微分（AD）框架和库。ADIFOR [Bischof 等人 1992]和 Tapenade [Hascoët 和 Pascual 2013] 分别为 Fortran 和 C 程序执行自动微分；Adept [Hogan 2014] 和 ADIC [Narayanan 等人 2010] 则使用表达式模板为 C++ 执行自动微分。ForwardDiff [Revels 等人 2016] 使用向量前向模式自动微分对 Julia 程序进行微分，而 DiffSharp [Baydin 等人 2015] 是一个以 F# 实现的自动微分库，支持

前向模式和反向模式。Stalingrad [Pearlmutter 和 Siskind 2008] 是面向 Scheme 方言的优化编译器，具有一等自动微分算子，并且同时支持自动微分的前向模式和反向模式。类似地，Karczmarczuk [Karczmarczuk 1998]介绍了一种同时支持前向模式和反向模式自动微分的 Haskell 实现；Elliott [Elliott 2018] 则基于范畴论对自动微分进行了推广，以实现前向模式和反向模式自动微分。近年来，研究者致力于为函数式语言提供正确且渐近高效的反向模式自动微分 [Krawiec 等，2022；Radul 等，2023；Smeding 和 Vákár 2023], ，其相关思想已在 JAX [Bradbury 等，2018；Frostig 等人 2018]和 Dex [Paszke 等人 2021]中实现。

TensorFlow 和 PyTorch 等机器学习库基于张量抽象实现。这些系统提供一组预定义的高效内核，用于操作张量，并且可以利用编译后端进一步优化。Lantern [Wang 等人 2019] 使用多阶段编程来执行反向模式自动微分。然而，前述框架都不支持对采用不规则存储格式的稀疏数据结构进行自动微分 [JAX 2023年；PyTorch 2023年；TensorFlow2023]。已有一些工作尝试静态地纳入稀疏性，但这需要程序员手动指定稀疏模式 [Peng 和 Dubach 2023]，且不适用于具有任意模式的大型稀疏矩阵 [Ghorbani 等人 2022]。

稀疏张量代数。近年来，稀疏张量代数一直是众多研究与开发工作的重点，并促使多个旨在支持该领域的框架和系统相继问世。TACO [Chou 等人 2018年；Kjolstad 等人 2017]是一种能够处理张量代数中稀疏和稠密计算的系统。另一个值得关注的框架是稀疏多面体框架 [Strout 等人 2018], ，它扩展了多面体编译的能力，以支持稀疏张量代数。此外，SciPy [Virtanen 等人 2019], 、TensorFlow、PyTorch 和 MATLAB 张量工具箱 [Bader 和 Kolda 2007] 等软件包支持多种稀疏矩阵表示，从而能够对科学与工程应用中常见的稀疏张量数据结构进行高效计算。

尽管该领域已经取得进展，稀疏张量的自动微分仍未得到广泛支持。近期为数不多的相关工作之一 [Nytko 等人 2022] 为有限种内核提供了手动梯度。对稀疏张量求导的主要挑战在于其不规则的数据表示，这使微分过程变得复杂。为了解决这一问题，我们提出将逻辑稀疏张量表示与其物理存储格式分离，从而实现更高效、更有效的微分。

9 结论与展望

本文介绍了 ∇SD，这是首个支持稀疏张量自动微分的框架。我们的核心见解是将逻辑问题与物理数据存储表示分离。我们针对 SDQLite 的逻辑片段提供了一种张量化前向模式变换。SDQLite 是一种近期提出的语言，它将稀疏张量的物理存储与内核的逻辑规范融合在一起。借助通过等式饱和全局应用的代数优化，我们进一步提升了求导程序的性能，并融合其物理表示。实验结果表明，在真实世界和合成数据集上，我们的框架均优于最先进的可微张量框架。

未来，我们计划增加对反向模式自动微分的支持（借鉴 [Berg 等人 2022年；Radul 等人 2023] 的思路）。此外，我们计划支持调度变换，以添加 GPU 后端 [Senanayake 等人 2020]。这样，我们就能利用我们的框架训练深度但稀疏的学习模型，例如图神经网络（GNNs） [Nytko 等人 2022]。最后，我们计划支持完整的 SDQL [Shaikhha 等人 2022] （包括集合与多重集构造所需的布尔值半环和整数半环，以及元组构造），以便为关系数据库以及混合关系代数与线性代数工作负载提供自动微分。

致谢

第一作者感谢华为对爱丁堡大学分布式数据管理与处理实验室的支持。第二作者获得英国皇家学会大学研究奖学金的资助。

参考文献

2023年。基于 Robin Hood 哈希的快速且内存高效的 C++ 哈希表。https://github.com/martinus/robin-hood-hashing。

Martín Abadi、Ashish Agarwal、Paul Barham、Eugene Brevdo、Zhifeng Chen、Craig Citro、Greg S. Corrado、Andy Davis、Jeffrey Dean、Matthieu Devin、Sanjay Ghemawat、Ian Goodfellow、Andrew Harp、Geoffrey Irving、Michael Isard、Yangqing Jia、Rafal Jozefowicz、Lukasz Kaiser、Manjunath Kudlur、Josh Levenberg、Dandelion Mané、Rajat Monga、Sherry Moore、Derek Murray、Chris Olah、Mike Schuster、Jonathon Shlens、Benoit Steiner、Ilya Sutskever、Kunal Talwar、Paul Tucker、Vincent Vanhoucke、Vijay Vasudevan、Fernanda Viégas、Oriol Vinyals、Pete Warden、Martin Wattenberg、Martin Wicke、Yuan Yu 和 Xiaoqiang Zheng。2015年。TensorFlow：异构系统上的大规模机器学习。https: //www.tensorflow.org/ 可从 tensorflow.org 获取软件。

Brett W. Bader 和 Tamara G. Kolda。2007年。使用稀疏和因子化张量进行高效 MATLAB 计算。*SIAM J. Sci. Comput.* 30, 1 (2007), 205–231。https://doi.org/10.1137/060676489 Atilim Gunes Baydin、Barak A. Pearlmutter 和 Jeffrey Mark Siskind。2015年。DiffSharp：自动微分库。*CoRR* abs/1511.07727 (2015)。arXiv:1511.07727 http://arxiv.org/abs/1511.07727 Birthe van den Berg、Tom Schrijvers、James McKinna 和 Alexander Vandenbroucke。2022年。前向模式还是反向模式自动微分：有什么区别？*arXiv*预印本 *arXiv:2212.11088* (2022)。Michael Betancourt。2018年。高阶自动微分的几何理论。*arXiv*预印本 *arXiv:1812.11592* (2018)。Christian H. Bischof、Alan Carle、George F. Corliss、Andreas Griewank 和 Paul D. Hovland。1992年。ADIFOR——为 Fortran 程序生成导数代码。*Sci. Program.* 1, 1 (1992), 11–29。https://doi.org/10.1155/1992/717832 James Bradbury、Roy Frostig、Peter Hawkins、Matthew James Johnson、Chris Leary、Dougal Maclaurin、George Necula、Adam Paszke、Jake VanderPlas、Skye Wanderman-Milne 和 Qiao Zhang。2018年。*JAX*：*Python* 的可组合变换*+NumPy* 程序。http://github.com/google/jax Stephen Chou、Fredrik Kjolstad 和 Saman P. Amarasinghe。2018年。稀疏张量代数编译器的格式抽象。*ACM* 编程语言 2, OOPSLA (2018), 123:1–123:30。https://doi.org/10.1145/3276493 Timothy A. Davis 和 Yifan Hu。2011年。佛罗里达大学稀疏矩阵集合。*ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25。https://doi.org/10.1145/2049662.2049663 Conal Elliott。2018年。自动微分的简单本质。*ACM* 编程语言 2, ICFP (2018), 70:1–70:29。https://doi.org/10.1145/3236765 Cormac Flanagan、Amr Sabry、Bruce F. Duba 和 Matthias Felleisen。1993年。使用延续进行编译的本质。载于*ACM SIGPLAN'93* 编程语言设计与实现会议（*PLDI*），美国新墨西哥州阿尔伯克基，*1993*年*6*月*23*日至*25*日论文集，Robert Cartwright（编）。ACM，237–247。https://doi.org/10.1145/155090.155113 Roy Frostig、Matthew James Johnson 和 Chris Leary。2018年。通过高级跟踪编译机器学习程序。机器学习系统 4, 9 (2018)。Mahdi Ghorbani、Mathieu Huot、Shideh Hashemian 和 Amir Shaikhha。2022年。编译结构化张量代数。*arXiv*预印本 *arXiv:2211.10482* (2022)。Laurent Hascoët 和 Valérie Pascual。2013年。Tapenade 自动微分工具：原理、模型与规范。*ACM Trans. Math. Softw.* 39, 3 (2013), 20:1–20:43。https://doi.org/10.1145/2450153.2450158 Robin J. Hogan。2014年。使用 C++ 中的表达式模板进行快速反向模式自动微分。*ACM Trans. Math. Softw.* 40, 4 (2014), 26:1–26:16。https://doi.org/10.1145/2560359 Mathieu Huot 和 Amir Shaikhha。2022年。语义正确、纯函数式且高效的反向模式自动微分。*CoRR* abs/2212.09801 (2022)。https://doi.org/10.48550/arXiv.2212.09801 arXiv:2212.09801 Mathieu Huot、Sam Staton 和 Matthijs Vákár。2020年。通过微分学与范畴粘合实现自动微分的正确性……载于*FoSSaCS*。319–338。Patrick Iglesias-Zemmour。2013年。微分学。第185卷。美国数学学会。JAX。2023年。GitHub Issue #13118。https://github.com/google/jax/issues/13118。访问日期：2023-01-30。Jerzy Karczmarczuk。1998年。计算机程序的函数式微分。载于第三届 *ACM SIGPLAN* 函数式编程国际会议（*ICFP '98*），美国马里兰州巴尔的摩，*1998*年*9*月*27*日至*29*日论文集，Matthias Felleisen、Paul Hudak 和 Christian Queinnec（编）。ACM，195–203。https://doi.org/10.1145/289423.289442 Kamil A. Khan 和 Paul I. Barton。2015年。用于广义导数求值的向量前向模式自动微分。*Optim. Methods Softw.* 30, 6 (2015), 1185–1212。https://doi.org/10.1080/10556788.2015.1025400

∇SD：稀疏张量的可微编程

Fredrik Kjolstad、Shoaib Kamil、Stephen Chou、David Lugato 和 Saman Amarasinghe。2017年。张量代数编译器。*ACM* 编程语言 1, OOPSLA，文章77（2017年10月），29页。https://doi.org/10.1145/3133901 Faustyna Krawiec、Simon Peyton Jones、Neel Krishnaswami、Tom Ellis、Richard A. Eisenberg 和 Andrew W. Fitzgibbon。2022年。可证明正确、渐近高效的高阶反向模式自动微分。*ACM* 编程语言 6, POPL (2022), 1–30。https://doi.org/10.1145/3498710 Yasuhiko Minamide。1998年。带有空洞的数据结构的函数式表示。载于第*25*届 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集。75–84。Jorge J Moré。2006年。Levenberg-Marquardt算法：实现与理论。载于数值分析：*1977*年*6*月*28*日至*7*月*1*日在邓迪举行的双年度会议论文集。Springer，105–116。Sri Hari Krishna Narayanan、Boyana Norris 和 Beata Winnicka。2010年。ADIC2：用于微分 C 和 C++ 的组件源代码变换系统开发。载于国际计算科学会议论文集，*ICCS 2010*，荷兰阿姆斯特丹大学，*2010*年*5*月*31*日至*6*月*2*日（*Procedia Computer Science*，第*1*卷），Peter M. A. Sloot、G. Dick van Albada 和 Jack J. Dongarra（编）。Elsevier，1845–1853。https://doi.org/10.1016/j.procs. 2010.04.206 Thomas Nelson、Geoffrey Belter、Jeremy G. Siek、Elizabeth R. Jessup 和 Boyana Norris。2015年。可靠地生成高性能矩阵代数。*ACM Trans. Math. Softw.* 41, 3 (2015), 18:1–18:27。https://doi.org/10.1145/2629698 Nicolas Nytko、Ali Taghibakhshi、Tareq Uz Zaman、Scott MacLachlan、Luke N Olson 和 Matt West。2022年。用于反向模式自动微分的优化稀疏矩阵运算。*arXiv*预印本 *arXiv:2212.05159* (2022)。Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Kopf、Edward Yang、Zachary DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。2019年。PyTorch：一种命令式风格的高性能深度学习库。载于神经信息处理系统进展 *32*。Curran Associates, Inc.，8024–8035。Adam Paszke、Daniel D Johnson、David Duvenaud、Dimitrios Vytiniotis、Alexey Radul、Matthew J Johnson、Jonathan Ragan-Kelley 和 Dougal Maclaurin。2021年。直奔主题：点式数组编程中的索引集与保持并行性的自动微分。*ACM* 编程语言论文集 5, ICFP (2021), 1–29。Barak A. Pearlmutter 和 Jeffrey Mark Siskind。2008年。函数式框架中的反向模式自动微分：Lambda：终极反向传播器。*ACM Trans. Program. Lang. Syst.* 30, 2 (2008), 7:1–7:36。https://doi.org/10.1145/1330017.1330018 Mai Jacob Peng 和 Christophe Dubach。2023年。LAGrad：MLIR 中静态优化的可微编程。载于第*32*届 *ACM SIGPLAN*编译器构造国际会议论文集（加拿大魁北克省蒙特利尔）（*CC 2023*）。计算机协会，纽约，美国，228–238。https://doi.org/10.1145/3578360.3580259 PyTorch。2023年。GitHub问题 # 12498。https://github.com/pytorch/pytorch/issues/12498。访问日期：2023-01-30。Alexey Radul、Adam Paszke、Roy Frostig、Matthew J. Johnson 和 Dougal Maclaurin。2023年。只需线性化一次：切向量转置为梯度。*ACM* 编程语言 7, POPL (2023), 1246–1274。https://doi.org/10.1145/3571236 Jarrett Revels、Miles Lubin 和 Theodore Papamarkou。2016年。Julia 中的前向模式自动微分。*CoRR* abs/1607.07892 (2016)。arXiv:1607.07892 http://arxiv.org/abs/1607.07892 Maximilian Schleich、Amir Shaikhha 和 Dan Suciu。2022年。在灵活存储上优化张量程序。*arXiv*预印本 *arXiv:2210.06267* (2022)。Ryan Senanayake、Changwan Hong、Ziheng Wang、Amalee Wilson、Stephen Chou、Shoaib Kamil、Saman Amarasinghe 和 Fredrik Kjolstad。2020年。稀疏张量代数的稀疏迭代空间变换框架。*ACM* 编程语言 4, OOPSLA，文章158（2020年11月），30页。https://doi.org/10.1145/3428226 Amir Shaikhha。2022年。通过深度融合实现高效的嵌套递归计算。载于第*21*届 *ACM SIGPLAN*生成式编程国际会议：概念与经验论文集。33–44。Amir Shaikhha、Andrew Fitzgibbon、Simon Peyton Jones 和 Dimitrios Vytiniotis。2017年。用于高效内存管理的目标传递风格。载于第*6*届 *ACM SIGPLAN*函数式高性能计算国际研讨会论文集。12–23。Amir Shaikhha、Andrew W. Fitzgibbon、Dimitrios Vytiniotis 和 Simon Peyton Jones。2019年。函数式数组处理语言中的高效可微编程。*ACM* 编程语言 3, ICFP (2019), 97:1–97:30。https: //doi.org/10.1145/3341701 Amir Shaikhha、Mathieu Huot、Jaclyn Smith 和 Dan Olteanu。2022年。使用半环字典进行函数式集合编程。*ACM* 编程语言 6, OOPSLA1 (2022), 1–33。https://doi.org/10.1145/3527333 Tom Smeding 和 Matthijs Vákár。2023年。通过经典程序变换实现高效对偶数反向自动微分。*ACM* 编程语言 7, POPL (2023), 1573–1600。https://doi.org/10.1145/3571247 Michelle Mills Strout、Mary W. Hall 和 Catherine Olschanowsky。2018年。稀疏多面体框架：组合编译器生成的检查器-执行器代码。*Proc. IEEE* 106, 11 (2018), 1921–1934。https://doi.org/10.1109/JPROC.2018. 2857721

Xuan 特定代码生成。计算机图形论坛 39, 5 (2020 P), 209–219. https://doi.org/10.1111/cgf.14080 Ross Tate - Michael Stepp、Zachary Tatlock 和 Sorin Lerner。2009。等式饱和：一种新的优化方法。载于第 *36* 届 *ACM SIGPLAN-SIGACT* 编程语言原理年度研讨会论文集。264–276。TensorFlow。2023年。GitHub 问题 #43497。https://github.com/tensorflow/tensorflow/issues/43497。访问日期：2023-01-30。Pauli Virtanen、Ralf Gommers、Travis E. Oliphant、Matt Haberland、Tyler Reddy、David Cournapeau、Evgeni Burovski、Pearu Peterson、Warren Weckesser、Jonathan Bright、Stéfan van der Walt、Matthew Brett、Joshua Wilson、K. Jarrod Millman、Nikolay Mayorov、Andrew R. J. Nelson、Eric Jones、Robert Kern、Eric Larson、C. J. Carey、Ilhan Polat、Yu Feng、Eric W. Moore、Jake VanderPlas、Denis Laxalde、Josef Perktold、Robert Cimrman、Ian Henriksen、E. A. Quintero、Charles R. Harris、Anne M. Archibald、Antônio H. Ribeiro、Fabian Pedregosa、Paul van Mulbregt 和 SciPy。2019。SciPy 1.0——Python 科学计算的基础算法。*CoRR* abs/1907.10121 (2019)。arXiv:1907.10121 http://arxiv.org/abs/1907.10121 Fei Wang、Daniel Zheng、James M. Decker、Xilun Wu、Grégory M. Essertel 和 Tiark Rompf。2019。揭开可微编程的奥秘：shift/reset 这一终极反向传播器。*ACM* 编程语言论文集 3, ICFP (2019), 96:1–96:31. https: //doi.org/10.1145/3341700 Max Willsey、Chandrakana Nandi、Yisu Remy Wang、Oliver Flatt、Zachary Tatlock 和 Pavel Panchekha。2021。egg：快速且可扩展的等式饱和。*ACM* 编程语言论文集 5, POPL (2021), 1–29. https://doi.org/10.1145/3434304