arXiv:2305.06692v1 [cs.PL] 11 May 2023

可微编程：控制流引发的不连续性的高效平滑处理

塞巴斯蒂安·克里斯托杜卢和乌韦·瑙曼，德国亚琛工业大学（RWTH Aachen University）计算工程软件工具研究组。

y

文章历史记录
编译于 2023 年 5 月 12 日

摘要
我们希望在不连续的程序代码中求得导数，而默认的算法微分可能无法取得良好的效果。具体而言，我们考虑由控制流语句引起的不连续性，在这种情况下，有意义的导数理想情况下应能够表示轨迹中由此产生的“跳跃”。为实现这一目标，可以在求导之前，对控制流语句处的轨迹进行插值。我们提出了一种方法，能够高效地对程序代码中由控制流引起的所有边界进行插值。从理论上讲，代码可以被视为一系列依次应用的分段连续函数。这些函数彼此嵌套，最终形成包含多个情形的函数复合。我们将这种函数复合理解为一棵树，并设计了一种启发式方法，用于识别与插值相关的路径。这使我们能够构思一种语言，自动且高效地对控制流语句进行平滑插值，从而使其完全可微。

关键词
算法微分；控制流；不连续；组合数学；平滑；导数

1. 引言

经微分的程序代码有广泛的应用，从求解器的稳定性估计到计算金融中的敏感性分析 [5]。我们关注的是分段连续且可表示为计算机程序的函数。例如，具有不连续收益或成本的函数 [15]，以及分段连续模型。原则上，可以对这些程序应用算法微分（AD），但在不连续点处可能无法得到有意义的导数：生成的导数只会像原始程序一样，按照控制流中的相同规则切换。这限制了 AD 在不连续情形下的效用 [13]。我们希望生成的导数能够考虑控制流造成的“突然跳变”及其之后的轨迹。我们开发了一种语言，使控制流语句完全可微。

我们的方法受到这样一种思想的启发：可以通过对分段定义函数的各种情形进行插值来替代不连续点。将插值用于这种方式以获得有意义的导数，这一想法可追溯至 Bertsekas [3]（针对不可微情形）以及 Zang [18]（针对分段连续函数）。此外，

联系人：S. Christodoulou。电子邮箱：christodoulou@stce.rwth-aachen.de

近期在这一方向开展的类似研究 [6, 10],其方法通常依赖于对分段连续函数的所有情形进行加权，以便在它们之间进行插值：针对给定的输入，分别评估每种情形，并通过权重确定其适用程度。

然而，用这种视角来看待程序代码并非总是有益。尽管程序沿控制流经过的路径可以映射到各种情形，但决策与计算通常交织在控制流之中。因此，要确定某个给定情形是否适用，通常必须预先执行大部分相关程序。

对于任意不连续或含噪函数，一种更通用的获取平滑导数的方法，是根据可用的函数求值对微分过程进行正则化 [7]。基于采样的方法在非光滑优化中也很常见：在关注点周围对梯度进行采样，然后进行插值 [4]。这类方法适用于被求值函数是黑盒的情形。而在我们的可微编程语言中，我们利用已知程序控制逻辑这一事实，在执行期间 计算插值。由于不连续性是由控制流求值过程中的比较语句引起的，我们可以根据每条控制流语句求值时出现的不连续性，推导出距离度量 。正如我们将在全文中看到的，这种方法的优势在于，它能够实现与基于情形的视角完全相同的插值，同时避免评估控制流中的所有路径。

在对控制流导致的不连续性进行平滑插值后，我们可以使用自动微分的反向模式 [8] 来计算导数，同时保留有限差分独有的优势：提供一种能体现不连续性的导数轨迹表示 [13]。这项工作面临两个主要挑战：第一，函数复合需要特殊处理。第二，对于给定程序的控制流路径，只有其中一部分与计算平滑插值相关。识别这些路径可转化为一个组合问题，我们试图解决这一问题。

本文结构如下：在第2—4节中，我们利用不连续函数的概念构建一些理论，并介绍平滑处理及高效评估这类函数复合所面临的挑战。第5—6节介绍一种用于高效平滑评估的算法，并展示其在实际场景中的实现。最后，本文通过一个案例研究，展示平滑自动微分在不连续优化中的应用。

2. 形式化与基本方法

我们考虑一个分段连续函数 f，它根据边界 ξ 在两个子函数之间切换。该边界（通常是一个流形）用于将 n 维空间划分为任意两个子空间 $S_1,\ S_2 \subseteq \mathbb{R}^n$, $S_1 \cap S_2 = \emptyset$, $S_1 \cup S_2 = \mathbb{R}^n$。我们定义 $c_\xi(x)$，使其当且仅当 $x \in S_1$ 时求值为真 。

$$f(x) = \begin{cases} f_1(x), & c_\xi(x) \\ f_2(x), & \bar{c}_\xi(x) \end{cases} := \begin{cases} f_1(x), & d_\xi(x) \le 0 \\ f_2(x), & d_\xi(x) > 0 \end{cases} \tag{1}$$

其中，函数 d_ξ 描述了到边界 x 的距离。因此， $\xi := \{x \mid d_\xi(x) = 0\}$ ，并描述了到该边界的距离。此外，我们将 ξ 定义为具有如下方向，使得仅当 $d_\xi \le 0$ 时，c_ξ 才成立。

我们插值背后的原理可以理解为：给定一个随机扰动，选择另一个子函数的概率为 d_ξ。扰动将从一个以 0 为中心、且其概率随距离增加而衰减的分布中抽取。将该分布中的一个样本加到距离函数 d_ξ 上，最有可能只会使距离发生轻微变化。为了将这一点形式化，设随机变量 $X \in \mathbb{R}$ 服从概率分布 $P[X = x] = \sigma'(x)$。对于 σ'，我们要求其关于 0 对称、连续可微，且满足 $\sigma' > 0$，$\frac{d}{dx}(\sigma'(x)) \geq 0$ 对所有 $x \leq 0$ 均成立，并且积分 $\sigma(x) \equiv \int_{-\infty}^{x} \sigma'(x)$ 是一个概率密度函数，即 $\int_{-\infty}^{\infty} \sigma' = 1$。

为了表示扰动，我们绘制一个 $q \sim X$，并将其加到情形选择器中进行比较的距离上。通过这种方式，各种情形将以概率方式被选中。

$$\hat{f}(x) = \begin{cases} f_1(x), & d_\xi(x) + q \leq 0 \\ f_2(x), & d_\xi(x) + q > 0 \end{cases} \tag{2}$$

情形 f_1 和 f_2 分别以概率 $p(d_\xi(x+q) \leq 0)$ 和 $p(d_\xi(x+q) > 0)$˜ ˆ 发生。对于期望值 $f(x) := E[f(x)]$，这意味着

$$\begin{aligned} \tilde{f}(x) = &\quad E[p(d_\xi(x)+q) \leq 0] \cdot f_1(x) + \quad E[p(d_\xi(x)+q) > 0] \cdot f_2(x) \\ = &\quad \sigma(d_\xi(x)) \cdot f_1(x) + \quad \overline{\sigma}(d_\xi(x)) \cdot f_2(x) \\ \equiv &\quad \sigma_\xi(x) \cdot f_1(x) + \quad \overline{\sigma}_\xi(x) \cdot f_2(x) \end{aligned} \tag{3}$$

其中，σ 是 σ'的积分。上文中，我们使用了定义 $\sigma \equiv 1 - \sigma$。¯这很有用，因为 $\sigma \in [0, 1]$ 表示一个概率密度函数，而 σ 是其复合形式。将方程（2）¯中每种情况对应的概率期望值归入某种情况的贡献中。贡献由 f_1 和 f_2之前的因子表示。请注意，这些贡献之和为 1。

在本研究中，我们将仅使用具有两个情形的分段定义函数。具有任意数量情形的函数可以通过对这些函数进行嵌套来构造。此外，二元分段定义函数模拟了编程语言中任意超平面的定义方式，并且还能保证所有生成的情形构成输入空间的一个划分。

3. 复合函数需要特定的平滑方法

在一篇较早的论文中，Zang [18] 以不同的方式提出了类似的思想。在他的研究中，不连续性被表示为每个空间细分区域不同边界处的阶跃函数。随后，根据这些边界，在所有 情形之间进行插值。

这一计算可能会变得十分复杂。我们提出了一种方法，仅在评估过程中确有需要时才跟踪边界距离。这显著降低了插值所需的计算量。更重要的是，我们将平滑插值的思想推广到了嵌套函数。这使我们能够将该方法应用于程序代码。

在嵌套分段连续函数时，它们的不连续性也会随着嵌套层次的增加而不断增多。因此，所产生的边界点不再能通过输入空间直接确定，而是隐式确定的，

$$f(x) = \begin{cases} f_1 & c_{\xi_f}(x) \\ f_2 & \overline{c}_{\xi_f}(x) \end{cases}, \quad g(x) = \begin{cases} g_1 & c_{\xi_g}(x) \\ g_2 & \overline{c}_{\xi_g}(x) \end{cases}$$

$$f \circ g = \begin{cases} f_1 \circ g_1 & c_{\xi_g}(x) \wedge c_{\xi_f}(g_1(x)) \\ f_1 \circ g_2 & \overline{c}_{\xi_g}(x) \wedge c_{\xi_f}(g_2(x)) \\ f_2 \circ g_1 & c_{\xi_g}(x) \wedge \overline{c}_{\xi_f}(g_1(x)) \\ f_2 \circ g_2 & \overline{c}_{\xi_g}(x) \wedge \overline{c}_{\xi_f}(g_1(x)) \end{cases} \tag{4}$$

因为 c_ξ 在 $f \circ g$ 情形中的条件包含了来自更上游函数的结果。若要显式地表示方程 (4) 中的情形（以输入空间的形式），则需要求函数的逆。由于逆函数仅在某些条件下存在（例如参见 [2]），因此，更实际的做法是在不了解输入空间中边界点的情况下进行平滑插值。

3.1. 先平滑，再组合

在介绍我们的方法之前，我们先指出，直接组合经过平滑处理的函数，会产生具有不理想轨迹的平滑插值。考虑将 f 和 $\tilde{g}$ 组合为 $f \circ \tilde{g}$。按照公式 (3) 中的方案，我们得到

$$\begin{aligned} \tilde{f} \circ \tilde{g}\,(x) = \quad & \sigma_{\xi_f}(\tilde{g}(x)) \cdot f_1(\tilde{g}(x)) \\ + \quad & \overline{\sigma}_{\xi_f}(\tilde{g}(x)) \cdot f_2(\tilde{g}(x)) \end{aligned}$$

其中

$$\tilde{g}(x) = \sigma_{\xi_g}(x) \cdot g_1(x) + \overline{\sigma}_{\xi_g}(x) \cdot g_2(x)$$

我们可以找到各个情形的贡献，以描述离散˜版本（由公式 (4) 给出）中的每种情形对插值 $f \circ \tilde{g}$的取值所占的贡献。注意，$\sigma_{\xi_g}(x)$ 决定了 g_1 的局部贡献。在组合过程的更下游，f_1 的˜局部贡献由 $\sigma_{\xi_f}(\tilde{g}(x))$ 决定。总体而言，在 $f \circ \tilde{g}$ 中，情形 $f_1 \circ g_1$ 的贡献为

$$E\left[p(\tilde{f} \circ \tilde{g} = f_1 \circ g_1)\right] = \sigma_{\xi_g}(x) \cdot \sigma_{\xi_f}(\tilde{g}(x))$$

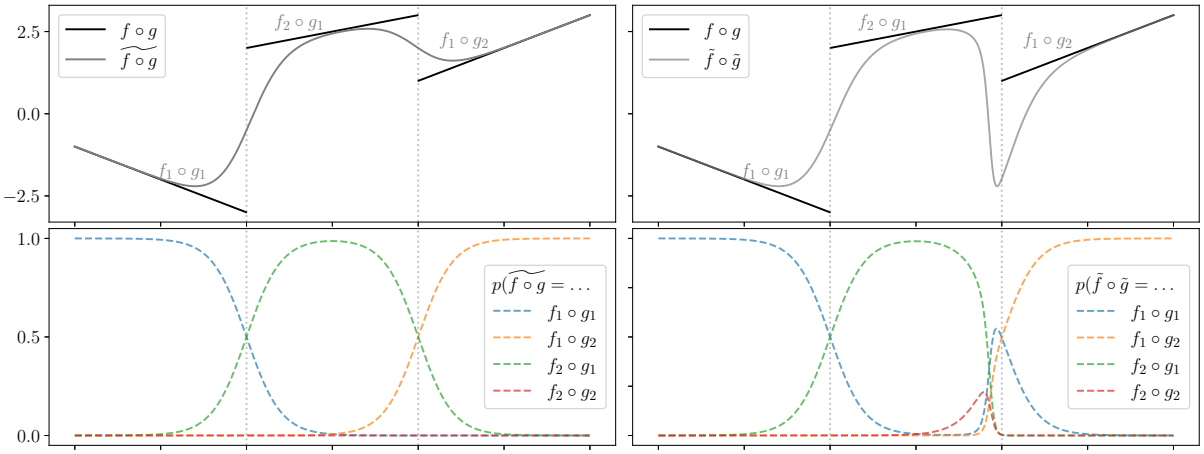
这不符合第 2 节中所引入的 $f \circ g$ 的概率解释。概率解释要求通过扰动距离函数，对式 4 中的每个选择器（c_{ξ_f} 和 c_{ξ_g}）施加扰动。从随机变量 X （定义于第 2 节）中抽取一个 $q \sim X$ 和一个 $r \sim X$，以扰动每个选择器，由此得到第一种情况 $f_1 \circ g_1$，其概率为

$$\begin{aligned} p(\widetilde{f \circ g} = f_1 \circ g_1) &= p(d_{\xi_g}(x) + q \leq 0 \wedge d_{\xi_f}(g_1(x)) + r > 0) \\ &= p(d_{\xi_g}(x) + q \leq 0) \cdot p(d_{\xi_f}(g_1(x)) + r > 0) \end{aligned} \tag{5}$$

由此，我们再次取期望，得到贡献 $\kappa_{f_1\circ g_1} := E[p(\widetilde{f\circ g} = f_1 \circ g_1)]$, 含义

$$\kappa_{f_1\circ g_1}(x) = \sigma_{\xi_g}(x) \cdot \overline{\sigma}_{\xi_f}(g_1(x))$$

以及所有其他贡献也以类似方式处理。两种平滑方法的比较如图1所示。与 $f \circ g$相比，$f \circ \tilde{g}$ 在从情形 $f_2 \circ g_1$ 过渡到情形 $f_1 \circ g_2$时的轨迹出现了偏差。这是因为并非源自这两个过渡情形中任何一个的贡献会短暂地达到峰值$\widetilde{\ }$。在 $f \circ g$中，某一情形的贡献会随着与该情形边界的距离增加而严格递减。



$\widetilde{\ }$**图1.** $f \circ g$ 和 $f \circ \tilde{g}$的平滑方法与贡献

3.2. 先组合，再平滑

我们的观点使我们能够通过对情形选择施加扰动，简单地从概率角度解释每种情形。按照公式 (5)，对扰动后的情形选择取期望值，即可得到所有贡献 $\{\kappa_{f_i\circ g_i}\}_{i,\ j\in\{1,2\}}$ 。将公式 (4) 中的所有情形按其贡献加权，即可得到插值。

$$\widetilde{f \circ g} := \sum_{i,j\in\{1,2\}} f_i \circ g_j \cdot \kappa_{f_i\circ g_j}$$

其中，将每种情况的权重 $\kappa_{f_i\circ g_j}$ 称为其贡献。

我们现在将这一想法形式化。我们使用算子 $\bigcirc$ 来定义函数的顺序复合，顺序从最外层函数到最内层函数。我们考虑k 分段定义函数 $Q = \bigcirc_{i\in 1\ldots k} q^{(i)}$，其中每个 $q^{(i)}$ 都会在边界 $\xi_{q^{(i)}}$处于 $q_1^{(i)}$ 和 $q_2^{(i)}$ 之间切换。 Q 中的所有可能情形都由穿过嵌套的路径$\mathcal{P} = \{p \mid p \in \{1,2\}^{|p|}\}$ 表示。我们用 $|p|$ 表示一条

路径的长度。
对于

$$q^{(i)} = \begin{cases} {q^{(i)}}_1, & c_{\xi_{q^{(i)}}} \\ {q^{(i)}}_2, & \overline{c}_{\xi_{q^{(i)}}} \end{cases} \qquad \text{we obtain} \qquad \widetilde{Q} = \sum_{p \in \mathcal{P}} \kappa_p \cdot \bigcirc_{i \in 1 \dots |p|} q_{p_i}^{(i)} \tag{6}$$

其中，路径的贡献 κ_p 是该路径上每个开关的局部贡献的乘积。

$$\kappa_p = \prod_{j \in \{1 \dots |p|\}} \kappa_{p_j} \tag{7}$$

并且局部贡献取决于控制流语句中求值的输入

$$\kappa_{p_j} = \begin{cases} \sigma_{\xi_{q^{(j)}}}(\bigcirc_{j<l\leq|p|} q_{p_l}^{(l)}) & \text{if } c_{\xi_j} \\ \overline{\sigma}_{\xi_{q^{(j)}}}(\bigcirc_{j<l\leq|p|} q_{p_l}^{(l)}) & \text{otherwise} \end{cases}$$

我们注意到，对于任意路径 p，它都是由组合过程中计算出的 κ_{p_j} 构成的。从计算角度来看，将组合应用到每个 $q^{(j)}$，并计算 κ_{p_j} 是很有用的。这样，所有 κ_{p_j} 都可以在路径计算过程中预先累积起来。这一点在图2中更加清晰，我们将输入的下游流表示为一棵树，其中展示了所有控制流路径。由于路径会在每个条件处分支，因此可以使用树来可视化组合的输入在嵌套函数和条件中的下游流动。

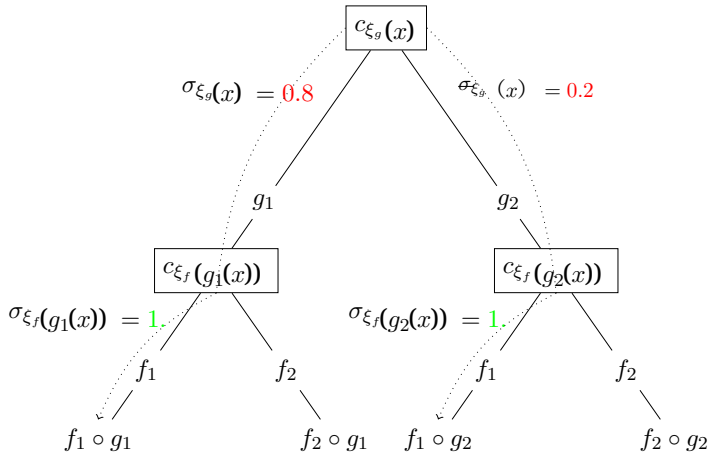


图2. 通过剪枝，$\widetilde{f \circ g}$ 避免计算具有 $\kappa_p < \epsilon$的路径。虽然此处 ϵ 为机器精度，但其他取值将在第（4.2）节中讨论。

组合得到的树是二叉树，但不一定对称：在某些分支处，一个分支可能对应一个分段定义的函数，而另一个分支则可能对应一个定义在整个定义域上的函数。此外，嵌套中的路径 p 不一定都具有相同的长度。第 5.2 节将通过展示如何将通用控制流代码映射为一棵树，进一步阐释这些问题。

4. 组合问题与剪枝

$\widetilde{\quad}$与 $f \circ g$ 的路径相对应的树如图2所示，其中每个节点代表一个决策点，每条边代表一个子函数。这体现了以下组合问题：每当遇到一个分支时，都必须计算两个新的子树。计算平滑插值（公式（6））意味着必须同时计算树的所有叶节点及其对应的贡献，因为从数学上讲，每条路径都以 $\kappa_p > 0$作为其贡献。

然而，对所有叶节点进行求值也意味着需要执行贡献为 $\kappa_p \approx 0$ 的路径。因此，直接实现方程（6）—（7）将包含 κ_p 在数值上为零的路径。这意味着，尽管这些路径不会影响结果，它们仍然会被执行。我们希望识别此类情形，以便剪除其求值不会产生贡献的子树。这类路径的特征在于，在路径 $\ell \leq |p|$ 结束之前，方程（7）中局部贡献的连续乘积在到达第 ℓ 条边时，会使乘积 κ_p 在数值上变为零。形式化地说

$$\kappa_p = \prod_{j \in \{1 \ldots \ell\}} \kappa_{p_j} < \epsilon \tag{8}$$

其中 ϵ 取为机器精度。我们可以将公式 (8) 作为一种不那么严格但更简单的启发式方法的指导。我们不再追踪已执行的分支最终是否会被剪除，而是在执行期间直接识别无关子树：在每个满足 $\kappa_{p_j} > 1 - \epsilon$ 的树节点处，另一条分支的贡献为 $< \epsilon$，因此可以将其剪除。未被此启发式方法剪除的分支会被添加到结果中，并称为相关路径。

4.1. 平滑锐度

到目前为止，我们假设扰动的大小是固定的。我们引入参数 $h \in \mathbb{R}$ 来描述扰动的范围。从数学上讲，我们扩展并重新利用第 2 节中给出的函数 σ'，以描述由概率分布 $P[Y = x] = \frac{1}{h}\sigma'(h\, x)$ 给出的随机变量 Y，其中 h 描述扰动的强度。同样，我们对 $q \sim Y$ 和 $r \sim Y$进行采样。类似于公式 (3)，我们得到

$$\begin{aligned} \tilde{f}(x) = &\quad E[p(d_\xi(x) + q \leq 0)] \cdot f_1(x) + E[p(d_\xi(x) + r > 0)] \cdot f_2(x) \\ = &\quad \sigma\left(d_\xi(x) \cdot h\right) \cdot f_1(x) + \overline{\sigma}\left(d_\xi(x) \cdot h\right) \cdot f_2(x) \\ \equiv &\quad \sigma_{\xi,h}(x) \cdot f_1(x) + \overline{\sigma}_{\xi,h}(x) \cdot f_2(x) \end{aligned} \tag{9}$$

这里，减小 h 会降低过渡的锐度，而增大 h 则会使过渡更加陡峭。在极限情况下，$h \to \infty$ 会得到方程 (1) 的离散形式，因为 $\sigma(h \cdot x)_h$ 根据 x 的符号，其值为 0 或 1。

此外，h 会对组合问题产生影响：它会改变具有 $\kappa_p \geq \epsilon$的相关路径数量。这是因为对于方程（8）中的每个因子$\sigma_{\xi_{q^{(i)}}}$，如果 $h_2 > h_1$，则对于所有 x，都有$\sigma_{\xi_{q^{(i)}},\ h_1}(x) \geq \sigma_{\xi_{q^{(i)}},\ h_2}(x)$。总的来说，$h$越高，对于给定的任何输入，我们可能需要评估的分支就越多。

4.2. 精度与组合问题

不同数据类型的机器精度有所不同。因此，数据类型的精度为 ϵ （在公式 (8) 中）提供了自然的选择。直观而言，距离函数定义了一个范围 $\{x \mid 1-\epsilon \geq \sigma(d_\xi(x)) \geq \epsilon\}$ ，该范围以 ξ 为中心，在此范围内，插值会同时计算条件 c_ξ 的两个子树。超出该范围时，则只计算其中一个子树。只要将 ϵ 设为机器精度（或更低），所得插值就是平滑的，因为从数值角度来看，在机器上测试 $< \epsilon$ 等价于与零进行比较。该方法计算了所计算的树中的所有相关路径。

相反，将 ϵ 设置为高于机器精度，意味着某些节点处的子树可能会被剪枝，即使其中包含一条与方程 6 中的求和相关的路径。当输入的微小变化导致相关路径集合发生变化时，无法保证所得函数在这些点上连续。这是因为求和式（方程 6）中可能缺少一些具有 $\kappa_p > 0$ 的路径。

不过，这也伴随着权衡：剪去贡献较小的分支$\kappa_p < \epsilon$ 可以缩小组合问题的规模。这样便能控制尖锐度参数 h 对问题规模的影响：可以任意降低插值的锐度 h ，同时通过 ϵ 单独控制评估路径的数量。

5. 实现说明

5.1. 条件的概率解释

要使我们的方法适用于代码，其中定义的概率必须映射为布尔逻辑。在布尔逻辑中，可以通过逻辑连接词将条件汇总为子句。为了定义子句的概率解释，我们考虑编程语言中常见的连接词 $\{\wedge, \vee, \ \}$。考虑条件 $c_{\xi_1} :\ = d_{\xi_1} \leq 0$ 和 $c_{\xi_2} :\ = d_{\xi_2} \leq 0$。它们在概率逻辑中的对应表达式，可通过计算在扰动样本 $q,\ r$ 下的期望概率得到，如第 2 节所述。表 1 中的定义也与 Zadeh 模糊

表 1. 给定扰动时布尔条件的概率等价形式

布尔	概率
$c_{\xi_1} \wedge c_{\xi_2}$	$E[p(d_{\xi_1}(x)+q\leq 0 \wedge d_{\xi_2}(x)\ +r\leq 0)]$ $= E[p(d_{\xi_1}(x)+q\leq 0)]\cdot E[p(d_{\xi_2}(x)\ +r\leq 0)]$ $= \sigma_{\xi_1}(x)\cdot\sigma_{\xi_2}(x)$
$c_{\xi_1} \vee c_{\xi_2}$	$E[p(d_{\xi_1}(x)+q\leq 0 \vee d_{\xi_2}(x)\ +r\leq 0)]$ $= E[p(d_{\xi_1}(x)+q\leq 0)]\cdot E[p(d_{\xi_2}(x)\ +r\leq 0)] - E[p(d_{\xi_1}(x)\ +q\leq 0)]\cdot E[p(d_{\xi_2}(x)\ +r\leq 0)]$ $= \sigma_{\xi_1}(x)+\sigma_{\xi_2}(x)] - \sigma_{\xi_1}(x)\ \cdot\sigma_{\xi_2}(x)$
$\bar{c}_{\xi_1}$	$E[1-p(d_{\xi_1}(x)+q\leq 0)] = 1-\sigma_{\xi_1}(x)$

逻辑 [17]。

接下来我们转向比较算子 $\{<, \leq, \geq, >\}$。注意，$\{\leq, >\}$ 的概率已在方程 1-3 中定义为 $\sigma(d_\xi(x))$ 和 $\sigma(d_\xi(x))$。由于 σ 的点对称性，我们可以看到，

$$\begin{aligned} \sigma(d_\xi(x)) &= \quad \overline{\sigma}(-d_\xi(x)) \\ \Leftrightarrow \quad E[p(d_\xi(x)\leq 0)] &= \ E[p(-d_\xi(x)>0)] \ = E[p(d_\xi(x)<0)] \end{aligned} \tag{10}$$

换言之，在我们的概率逻辑中，$\{<, \leq\}$ 的贡献由同一个函数表示。类似地，$\{>, \geq\}$ 的贡献由前者的复合形式表示。更具体地说，这意味着，任何在 $d_\xi \leq 0$处为真的c_ξ，其期望值都由函数 $\sigma(d_\xi(x))$表示，该函数也表示 $d_\xi < 0$。我们尚未处理概率逻辑中的等于（$=$）算子。它可以由先前定义的算子构造而成。

$$\begin{aligned} p(x = \xi) &= p(d_\xi(x) \leq \xi \wedge d_\xi(x) \geq \xi) = p(d_\xi(x) \leq \xi) \cdot p(d_\xi(x) \geq \xi) \\ \Leftrightarrow\ E[p(x = \xi)] &= \sigma_\xi(x) \cdot \overline{\sigma}_\xi(x) \end{aligned}$$

这样，所有比较算子和逻辑连接词都以类比布尔逻辑的方式在概率逻辑中得到了定义。

5.2. 控制流

在本文的其余部分中，我们通过使用分支的父条件来标识分支上的函数，从而简化控制流树的表示法：条件c_i 处的分支会根据条件求值为真或假，分别选择指令序列 $f_{i,T}$ 和 $f_{i,F}$。图 3 展示了这种方案，它更适合表示任何类型的控制流，并且通过不显式命名控制流内部的指令序列，简化了表示法。

The fo 第 5.3 节中的以下算法采用了标记树节点的思想，其中的

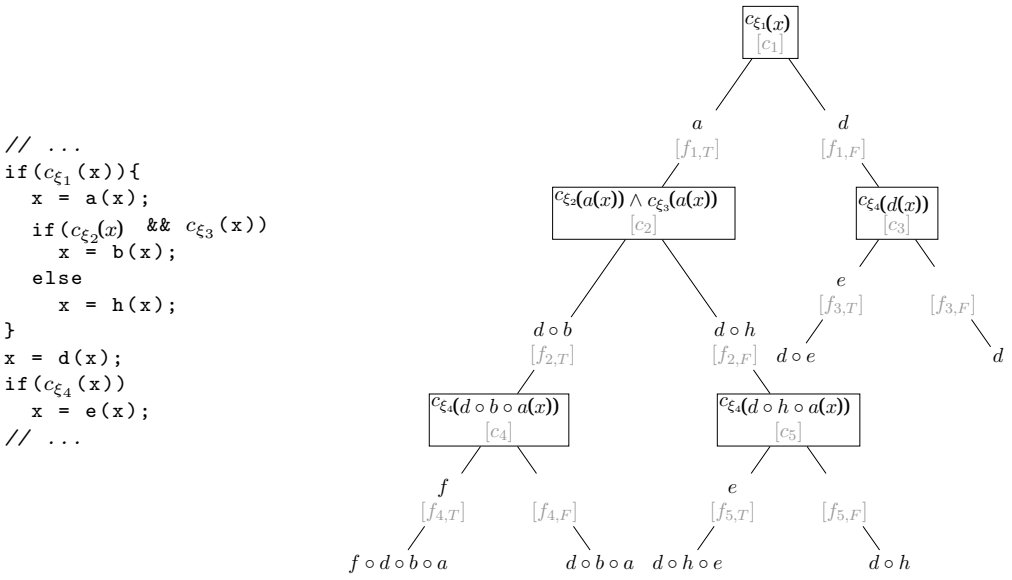


图 3。 代码（左）及其对应的控制流树（右）。空边表示没有函数改变输入。控制流很容易变得复杂。为清晰起见，我们根据函数的条件（灰色）来标识函数。

插值要求对两个分支都进行求值。请注意，算法无需显式构建控制流树，甚至无需了解其结构：标记会在运行时间根据遇到的特定条件设置。后续路径执行依赖这些标记所提供的信息，以确定下一次运行时如何对每个条件求值。这使得算法能够进行深度优先遍历，并将范围限定在相关路径集合内。

5.3. 树追踪算法

我们将函数执行视为沿其控制流所形成的执行树中的路径（参见图 3）。执行期间，某些条件 c_{ξ_i} 会计算为真，其局部贡献为 $\sigma_{\xi_i} \geq \epsilon$；或计算为假，其局部贡献为 $\sigma_{\xi_i} \geq \epsilon$。这些情形表明，在节点 c_{ξ_i} 处，与当前路径上计算结果相反的布尔值对应的子树中包含相关路径。我们通过在执行期间为此类条件设置标记来表示这一点。

每条路径 p 都有一个贡献 κ_p，该贡献通过将路径上的所有局部贡献相乘来计算。评估单条路径的过程如算法 1 所示。

算法 1 使用标记评估控制流和局部贡献

函数 EVALUATE PATH(标记)
  j $\leftarrow 0$
  $\kappa_p \leftarrow 1$
  当 j 存在 **时执行**
    离散 $\leftarrow c_{\xi_j}(x)$ ▷ 离散值
    连续 $\leftarrow \sigma_{\xi_j}(x)$ ▷ 概率值
    如果 $j \in$ 标记 $\wedge$ 标记$[j] =$ 为假**则**
      离散 $\leftarrow$离散
      con $\leftarrow 1-$con
    else if $con \geq \epsilon$ **then** 标记$[j] = \text{true}$
    结束 if
    $\kappa_p \leftarrow \kappa_p$ * con ▷累积贡献
    如果 dis $=$ true**则** ▷ 选择已执行的分支
      $\boldsymbol{f} \leftarrow \boldsymbol{f}_{j,T}$
    else
      $\boldsymbol{f} \leftarrow \boldsymbol{f}_{j,F}$
    结束 if
    x $\leftarrow \boldsymbol{f}$(x)
    $j \leftarrow$ 下一个条件(j, dis) ▷ 根据离散布尔值选择下一条边
  结束循环
  返回 x , κ_p, 标记
结束函数

执行一条路径后，会根据其贡献对结果加权，并将其加入总结果。随后，最近设置的标记（最深标记）会通过将其值设为 false 来取反，除非它已经被取反。每当路径执行经过一个已取反的标记时，它都会强制分支的布尔值和局部贡献取相反值。换句话说，已取反的标记会强制后续执行的控制流转向相反子树。

如果树标记中的最深标记是一个否定标记，并且执行过程经过该标记而没有添加更深层的标记，则可以安全地将其移除：这意味着其两个子树中的所有相关分支均已执行。对于以该最深标记结尾的连续负标记序列，情况也是如此。总体而言，该算法类似于对相关路径进行深度优先搜索。移除所有标记后，所有相关路径均已完成评估，并根据各自的贡献加权后添加到 y 中。至此，相关路径的公式（6）计算完成。

算法 2 树追踪

$y \leftarrow 0$
标记 $\leftarrow$ map{int, bool}
当true **时执行**
  y_p, κ_p, 标记 $\leftarrow$ EVALUATE PATH(markers)
  $y \leftarrow y + y_p \cdot \kappa_p$
  当 markers.size $> 0 \wedge$ markers.last $=$ false **执行**
    markers.remove last() ▷ 移除最深层的否定标记序列
  结束 while 循环
  如果 标记.size > 0 **则**
    标记.end $\leftarrow$false
  否则 跳出
  结束 if
结束循环

6. 平滑算法微分

为了获得平滑导数，将平滑插值与自动微分（AD）结合使用。我们选择通过 C++ 中的运算符重载来实现树追踪（算法1–2）。重载后的数值数据类型 SType 还会在逻辑比较运算符和逻辑连接词处计算局部贡献（参见第5.1节）。随后，对这些运算符和连接词的求值会在树节点必须同时求值两个子树的条件处触发标记过程。代码清单1给出了平滑自动微分的一种实现。我们的

```
#include <ad_tool>#include "smoothing.h"template<typename T> f(T x1, T x2){
T r = 2;if (x1*x1 + x2*x2 < 2.) r = r - 1;if (x1 < x2) r = r - 1;return r;}
int main(){SType<ADType<double>> x = 2;
ADType<double> y = run_all_paths(f, x); // run paths until markers consumed
y.derivative = 1;                       // seed derivative on smoothed result
interpret_adjoint(y);                   // adjoint tape accumulation
double dy_dx = x.derivative;            // harvest smooth derivative}
```

代码清单1 平滑类型中嵌套 AD 类型的算子重载示意图

GitHub 上提供的实现[1] 使用 dco/c++ [11] 作为重载的 ADType。请注意，用于平滑过程的重载 SType 可以轻松地与任何 ADType 结合。这样便可在平滑逻辑之上运行导数逻辑，从而得到平滑插值函数的导数。值得注意的是，我们也可以通过反向嵌套类型，得到导数的平滑插值（而不是平滑插值的导数）。

[1]https://github.com/sebastianfchr/Smoothing Essential

参见代码清单1。尽管插值导数是非光滑优化中的一种常见方法 [4], ，我们在此没有进一步探讨平滑算法微分的这一方面，因为插值后的导数会丢失函数中不连续性的信息。

7. 优化案例研究

为了将我们的方法应用于一个切实的用例，我们在不可微函数上运行一种基于梯度的下降方法。基于梯度的优化器从任意点 x_0 出发，寻找函数 f 的局部最小值，其基本原理是沿负梯度方向前进

$$x_{i+1} = x_i - \ell \cdot \nabla f(x_i)$$

其中 ℓ 是步长，也称为学习率。我们考虑一个来自非光滑优化问题集合 [12] 的函数：新月函数，定义如下

$$cr(x_1, x_2) = max\{x_1^2 + (x_2 - 1)^2 + x_2 - 1, -x_1^2 - (x_2 - 1)^2 + x_2 + 1\}$$

可以通过一条 if 语句轻松编写成代码，然后在重载的 SType 上运行以进行平滑处理。对于上述函数，优化器面临的挑战是

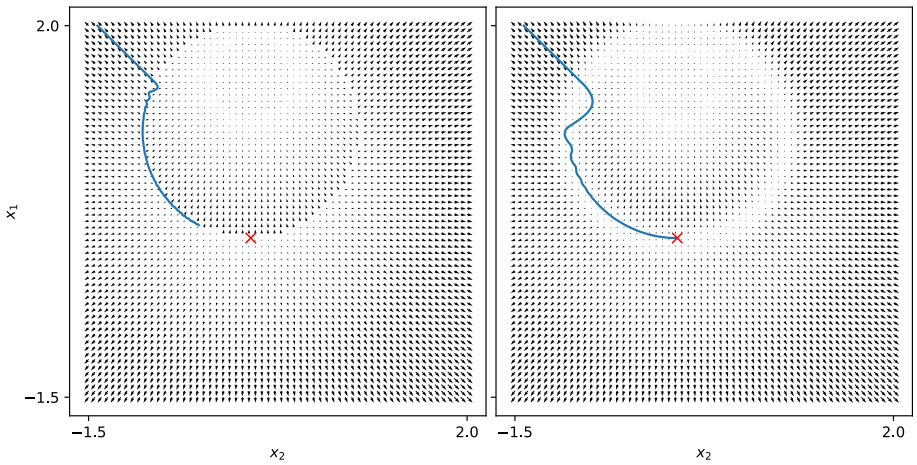


图 4. *Crescent* 函数与梯度场（左列为非光滑，右列为平滑后）。采用平滑梯度后，优化器的前进速度更快。该线表示锐度为 $h = 100$ 时优化器在第 300 次迭代的轨迹。为便于区分图中的曲线，所绘制的锐度 $h \ll 100$。

以便沿着 max 算子引入的不可微性进行优化，而梯度会导致优化器来回振荡，从而减缓其优化进程。我们假设，平滑处理能够提高最小值的质量，或减少优化器所需的迭代步数。我们将平滑后的函数集成到 TensorFlow [1] 中，以便能够直接使用现成的优化器，并选择了 TensorFlow 对 ADAM [14] 优化器的实现。采用不同锐度因子 h 进行优化的结果如表 2 所示。请注意， $h = \infty$ 对应于原始的非平滑版本。我们使用了 [12] 推荐的起始点来优化新月形，并针对不同的迭代步数运行优化器。

对于任意迭代次数，优化器几乎总能达到更好的最小值。

表2. Crescent 按每 h 步及迭代次数统计的优化结果。（初始学习率 $= 0.02$）

h \ 步	200	300	400	500	750	1000	1500	2000
10	0.7116	0.0164	0.013	0.013	0.013	0.013	0.013	0.013
50	0.0026	0.0026	0.0026	0.0026	0.0026	0.0026	0.0026	0.0026
100	0.0013	0.0013	0.0013	0.0013	0.0013	0.0013	0.0013	0.0013
500	0.0083	0.0068	0.011	0.0012	0.0006	0.0005	0.0006	0.0005
1000	0.0054	0.0109	0.0051	0.0029	0.0005	0.0025	0.0003	0.0011
∞	0.0075	0.007	0.0086	0.0055	0.0061	0.0035	0.0016	0.0056

与非平滑情形相比，经过平滑处理的情形（$h \neq \infty$）。图 4 对比了平滑情形与非平滑情形下优化器的轨迹。

对于我们的方法，一个更有力的应用案例是优化分段不连续函数。在标准算法微分中，这类函数的不连续性不会体现在梯度场中。这样的例子可以是

$$g(x_1, x_2) = \begin{cases} 0 & \text{if } {x_1}^2 + {x_2}^2 < 2 \wedge x_1 < x_2 \\ 2 & \text{if } {x_1}^2 + {x_2}^2 \geq 2 \wedge x_1 \geq x_2 \\ 1 & \text{else} \end{cases} \tag{11}$$

编程语言可以显式表达上述方程（11）中的每种情形，也可以像我们在代码清单1中那样采用更简洁的形式。在那里，函数 f 表达的行为完全相同。借助我们的方法，我们可以对其程序代码的平滑版本求导。从平滑版本得到的导数能够体现函数值的突变，如图5所示。因此，优化器可以找到该函数的最小值。

我们的示例结合了平滑处理与优化，但将平滑处理的锐度 h 设为固定值。尽管这种做法有助于说明问题，却绝非在优化器中运用平滑处理的理想方式。许多优化器会在其参数中记录近期的下降历史，并据此进行调整 [14]。一种更有前景的平滑与优化结合方式，是将锐度视为一个参数，并与动量、学习率等参数同步调整。

8. 展望

我们提出了一种方法，无需显式提供输入空间中不连续性的信息，即可对由控制流引起的不连续性进行插值。与现有方法相比，我们的方法支持函数嵌套，并且仅通过求值那些贡献相关的情形来进行插值。这使得我们能够使用一种语言进行编程，在这种语言中，控制流不连续性会被自动插值。

所提出的方法旨在集成到不同的应用场景中。我们利用这一点，在不连续的景观及其平滑版本中运行开箱即用的优化器。尽管我们有限的案例研究将优化器的功能与平滑处理分开，但仍然取得了更好的结果。

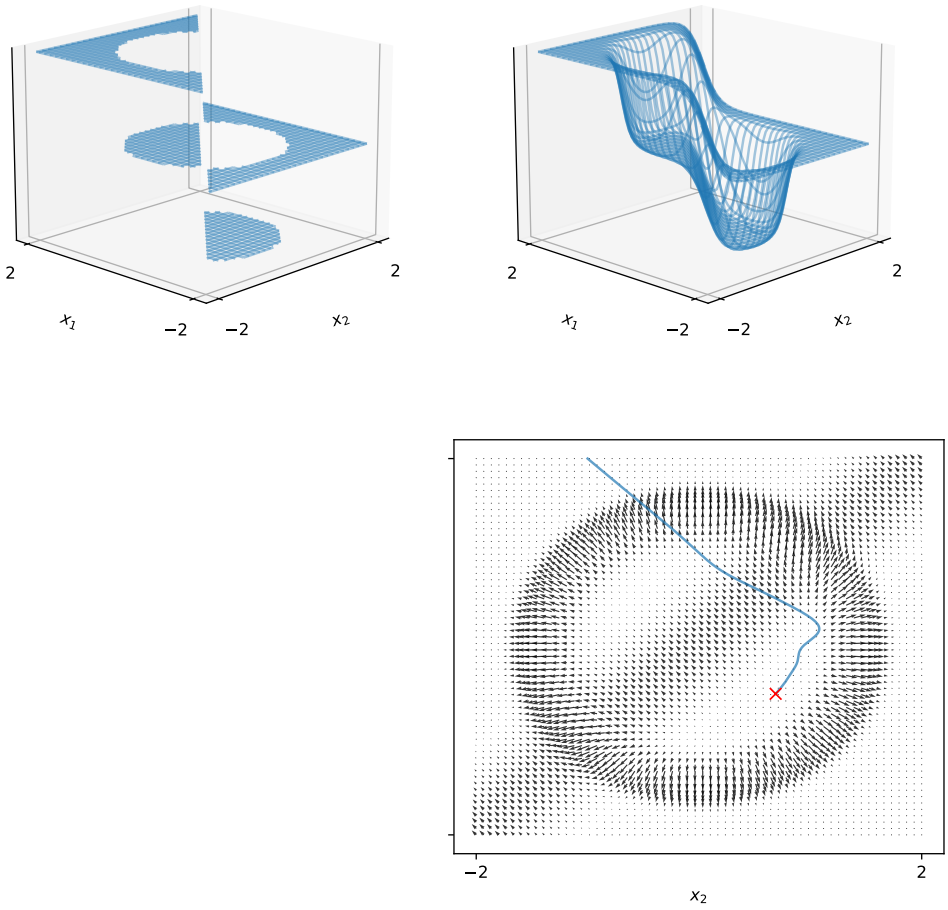


图5。 *Gradients in Discontinuous Functions*。左上图展示了方程（11），该方程也以代码形式呈现在代码清单1中。使用平滑版本（右上图）求导数时，原先的不连续性得以表示。梯度场（右下图）包含优化器的轨迹。

平滑方法的执行时间受到平滑程序的控制流、核（σ）以及精度（ϵ）和锐度（h）参数的影响，因为这些因素会影响评估路径的数量。除此之外，控制流中各条路径的执行时间也各不相同。因此，运行时间是一个高度个性化的问题，有待未来研究进一步探讨。我们这一思想的核心贡献在于，通过仅评估控制流中必要的路径，以高效的方式对程序代码进行平滑插值。沿着这一思路，还值得注意的是，输入和输出的数量（或维度）不会影响算法的复杂度。

我们将平滑程序的执行与反向模式自动微分相结合，从而得到平滑导数。目前，人们正致力于在能够实现大规模并行的系统中高效实现反向模式自动微分 [9, 16], ，例如机器学习和科学计算领域。未来，还需要实现平滑的并行化实现，以便将其集成到这些领域中。

参考文献

[1] M. Abadi、P. Barham、J. Chen、Z. Chen、A. Davis、J. Dean、M. Devin、S. Ghemawat、G. Irving、M. Isard 等，《Tensorflow：一种大规模机器学习系统》，载于 OSDI 会议论文集，第 16 卷，2016 年 11 月 2–4 日，美国佐治亚州萨凡纳。Usenix，2016 年，第 265–283 页。[2] A.V. Arutyunov 和 S.E. Zhukovskiy，《逆映射的存在性与性质》，《Steklov 数学研究所论文集》271（2010），第 12–22 页。[3] D.P. Bertsekas，《通过近似进行非可微优化》，《数学规划》9（1975），第 1–25 页。[4] J.V. Burke、F.E. Curtis、A.S. Lewis、M.L. Overton 和 L.E.A. Sim˜oes，《非光滑优化的梯度采样方法》，《数值非光滑优化》（2020），第 201–225 页。[5] L. Capriotti，《利用算法微分快速计算希腊值》，SSRN 1619626（2010）。[6] A. Chakraborty、A.S. Roy 和 B. Dasgupta，《非可微优化问题中梯度方法的非参数平滑》，载于 2016 年 IEEE 系统、人类与控制论国际会议（SMC）。IEEE，2016 年，第 3759–3764 页。[7] R. Chartrand，《含噪声非光滑数据的数值微分》，《国际学术研究通报》第 2011 卷（2011）。[8] A. Griewank，《自动微分的数学视角》，《数值数学学报》12（2003），第 321–398 页。[9] T. Kaler、T.B. Schardl、B. Xie、C.E. Leiserson、J. Chen、A. Pareja 和 G. Kollias，《PARAD：一种高效并行的反向模式自动微分算法》，载于算法计算机系统原理研讨会（APOCS），2021 年 1 月 13 日，线上举行。SIAM，2021 年，第 144–158 页。[10] B. Khajwal、C.H.L. Ong 和 D. Wagner，《通过平滑实现概率编程中快速且正确的基于梯度的优化》（2023）。doi: X10.48550/arXiv.2301.03415。[11] J. Lotz，《利用 dco/c++ 进行伴随代码生成的混合方法》，博士论文，亚琛工业大学，2016 年。[12] L. Luk˘san 和 J. Vlcek，《非光滑无约束与线性约束优化的测试问题》，技术报告 798，捷克科学院计算机科学研究所，捷克共和国布拉格，2000 年。[13] J.J. Mor´e 和 S.M. Wild，《你相信导数还是差分？》，《计算物理学杂志》273（2014），第 268–277 页。[14] S.J. Reddi、S. Kale 和 S. Kumar，《论 Adam 及其后续算法的收敛性》，arXiv 预印本 arXiv:1904.09237（2019）。[15] R. Siegel，《具有先发优势和非单调成本的不对称竞赛》，《美国经济学杂志：微观经济学》6（2014），第 59–105 页。[16] B. Van Merri¨enboer、O. Breuleux、A. Bergeron 和 P. Lamblin，《机器学习中的自动微分：现状与未来方向》，第 31 卷，2018 年 12 月 2–8 日，加拿大蒙特´利尔。Curran Associates，2018 年，第 8757–8767 页。[17] L.A. Zadeh，《模糊逻辑》，《计算机》21（1988），第 83–93 页。[18] I. Zang，《通过平滑进行不连续优化》，《运筹学数学》6（1981），第 140–152 页。

致谢

T本研究部分由英国牛津的 Numerical Algorithms Group Ltd. 资助。 .