深度学习库中的自动微分模糊测试

Chenyuan Yang 伊利诺伊大学厄巴纳-香槟分校
cy54@illinois.edu

Yinlin Deng 伊利诺伊大学厄巴纳-香槟分校
yinlind2@illinois.edu

Jiayi Yao 香港中文大学（深圳）
jiayiyao@link.cuhk.edu.cn

Yuxing Tu 华中科技大学
yxtu@hust.edu.cn

Hanchi Li 中国科学技术大学
slxiaochi@mail.ustc.edu.cn

Lingming Zhang 伊利诺伊大学厄巴纳-香槟分校
lingming@illinois.edu

arXiv:2302.04351v1 [cs.SE] 8 Feb 2023

摘要—近年来，深度学习（DL）受到广泛关注，并得到广泛应用。因此，越来越多的研究致力于测试深度学习库和框架。然而，现有研究在很大程度上忽视了深度学习系统的一个关键组件——自动微分（AD），而它正是近年来深度学习发展的基础。为此，我们提出了 ∇Fuzz，这是首个专门针对深度学习库中关键自动微分组件的通用实用方法。我们的核心洞见是，每个深度学习库 API 都可以抽象为一个处理张量/向量的函数，并可在各种执行场景下进行差分测试（使用不同实现计算输出/梯度）。我们将 ∇Fuzz 实现为一个完全自动化的 API级模糊测试器，专门用于测试深度学习库中的自动微分。它利用不同执行场景下的差分测试来测试一阶和高阶梯度，并纳入自动化过滤策略，以去除由数值不稳定性导致的假阳性。我们对四个最受欢迎且维护最活跃的深度学习库——PyTorch、TensorFlow、JAX 和 OneFlow——进行了广泛研究。结果表明，在代码覆盖率和缺陷检测方面，∇Fuzz 均大幅优于最先进的模糊测试器。截至目前，∇Fuzz 已在所研究的深度学习库中检测到 173 个程序错误，其中 144 个已由开发者确认（117 个此前未知，107 个与自动微分相关）。值得注意的是，在两个月内，∇Fuzz 发现了 PyTorch 和 JAX 所有高优先级自动微分缺陷的 58.3%（7/12）。现有模糊测试器均未检测到这些已确认的自动微分缺陷。

一、引言

近年来，深度学习（DL）研究取得了快速发展，DL 解决方案和技术也在各种应用定义域中得到广泛采用，例如自然语言处理 [1], 医疗保健 [2], 科学发现 [3], 和软件工程 [4]–[9]。因此，人们越来越关注此类系统的正确性和可靠性。例如，对于自动驾驶这类安全关键应用定义域，DL 系统中的程序错误可能导致严重后果，甚至造成人员死亡 [10]。

由于确保日益具有影响力的 DL 系统的质量至关重要，许多研究工作都集中于测试/验证 DL 模型 [11]–[21] 或应用程序 [22]–[24]。最近，针对底层 DL 库的测试库/框架（例如 PyTorch/TensorFlow）也受到了广泛关注，因为深度学习库是所有深度学习应用的核心基础设施。CRADLE [25] 是首批使用各种深度学习模型对 Keras 的多个后端进行差分测试的工作之一。AUDEE [27] 和 LEMON [28]进一步在现有模型上应用基于搜索的变异策略，以生成更多样化的测试输入。虽然这些基于变异的技术严重依赖种子模型，但 Muffin [29] 通过自顶向下生成方法，直接根据深度学习 API 合成深度学习模型。此外，Muffin 还能检测 Keras 不同后端在模型训练和推理阶段的不一致。与上述模型级测试技术不同，近期的 FreeFuzz 工作 [30]提出了一种全自动 API 级模糊测试技术，通过从开源项目中挖掘 API 输入来实现。类似地，另一项近期工作 DocTer [31], 根据从深度学习 API 文档中提取的深度学习特定输入约束（辅以人工标注），直接为每个 API 生成输入。尽管深度学习库测试近期取得了进展，现有技术仍存在一个主要局限：深度学习模型的推理阶段或深度学习 API 的直接执行获得了最多关注，而任何深度学习系统的关键组成部分——自动微分（*AD*） *[32]* 仍未得到充分研究。许多深度学习算法，尤其是反向传播 [33],（训练前馈神经网络的关键算法之一），都严重依赖自动微分来计算任意数值函数的导数。自动微分使复杂深度学习模型和算法的开发成为可能，因为如果没有自动微分，人们就必须手动或通过符号计算大型深度学习模型中数十亿个参数的导数 [34]。为自动获得导数，需要显式触发深度学习库中的特殊梯度/雅可比计算操作（例如，`tf.GradientTape.gradient()`（TensorFlow 中））。值得注意的是，自动微分中的程序错误可能导致深度学习模型无法收敛，和/或在实际部署中表现不佳，这对安全关键型应用而言是致命的。例如，静默的自动微分计算错误可能导致已部署深度学习模型的输出与训练阶段的输出显著偏离

```
输入 = 张量（形状=[5, 5, 5]）目标 = 张量（形状=[5]）
RevGrad(KLDivLoss, (输入, 目标)) # 崩溃
```

图 1：自动微分中的崩溃程序错误

此外，自动微分错误还可能直接导致整个训练过程崩溃；在训练近期流行的大型模型时，这会浪费大量计算资源，和/或引发潜在的拒绝服务（DoS）攻击 [35]。图1展示了一个危险的崩溃漏洞：广泛使用的 PyTorch API `KLDivLoss` [36] 在特殊输入形状下进行自动微分计算时会崩溃。然而，现有研究尚未对这类自动微分引擎进行充分测试。

尽管近期的 Muffin 工作 [29] 有望测试深度学习模型的训练阶段，但距离实用仍有很大差距。首先，Muffin 需要手动标注所考虑的深度学习 API 的输入约束，并使用重塑操作来确保生成模型的有效性。因此，Muffin 只能覆盖少量 API（已在 §VI-A中确认）。其次，完整模型测试效率低下，尤其是对于大型模型/数据集。第三，假阳性可能源于随机性、精度损失和数值不稳定性，而这些问题在模型训练场景中会进一步加剧。第四，训练阶段的差分测试要求不同深度学习库具有相同的 API 接口，这进一步限制了其应用。Muffin 使用 Keras 及其支持的后端 TensorFlow、Theano 和 CNTK。2019年发布的 Keras 2.3.0 是最后一个支持 TensorFlow 以外后端的版本 [37]。最后，Muffin 无法全面测试深度学习库中的自动微分引擎，因为它只覆盖了部分反向模式自动微分，并忽略了前向模式自动微分。事实上，Muffin 论文并未报告任何已确认的自动微分漏洞（已在 §VI-A中确认）。

为了全面、自动地测试深度学习库中的自动微分引擎，我们提出了 ∇Fuzz，这是首个专门针对关键自动微分组件的通用且实用的框架。我们的关键洞见是，深度学习库中的每个 API 都可以抽象为一个处理张量/向量的函数，并可在各种执行场景下进行差分测试（采用不同实现计算输出/梯度）。例如，同一个深度学习 API 可以在不使用自动微分或使用不同自动微分模式的情况下执行，但在不同执行场景下，API 输出和梯度都应保持一致，这自然可以作为差分测试的预言机。此外，由于我们的测试预言机适用于函数层面，我们还可以将每个 API 转换为其梯度函数，以测试高阶梯度计算的正确性。

∇Fuzz 有望解决 Muffin 的所有上述局限。通过 API 级测试，∇Fuzz 不再受模型级测试中严格输入限制的影响，并且可以完全自动化。此外，API 级变异效率更高，因为它避免了在大型模型和大型数据集上进行大量计算。再者，由于 API 级测试不会累积浮点精度损失，因此可以显著减少假阳性。最后，与 Muffin 相比，我们的方法也更通用，因为我们利用了任何深度学习库中都可用的自然自动微分预言机。

```
# Model building
input = Input(shape=(784,))
hidden = Dense(64, activation="relu")(input)
output = Dense(10,
activation="softmax")(hidden)
model = Model(inputs=input, outputs=output)
# Dataset loading
x_train, y_train, x_test, y_test = load_data()
# Training
model.compile(
      optimizer=SGD(),
      loss=MeanAbsoluteError())
model.fit(x_train, y_train)
# Inference
predictions = model.predict(x_test)
```

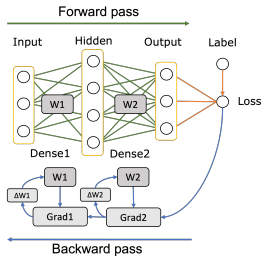


图 2：深度学习模型训练与推理示例

我们已将 ∇Fuzz 实现为一种完全自动化的 API 级模糊测试技术，并配备专门针对深度学习库中自动微分的测试预言机。具体而言，虽然我们的方法具有通用性，并且可以利用任何现有的 API 级深度学习库模糊测试工具生成输入，但我们基于最先进的 FreeFuzz ∇构建了 Fuzz [38]，因为它完全自动化且公开可用。对于测试预言机，∇Fuzz 会在底层深度学习库提供的不同执行场景下，自动对每个深度学习库 API（及其高阶梯度）进行差分测试。∇Fuzz 还采用自动化过滤策略，进一步减少由数值不稳定问题引起的假阳性。我们在四种使用最广泛且仍在积极维护的深度学习库上，对 ∇Fuzz 进行了广泛研究：PyTorch、TensorFlow、JAX 和 OneFlow。结果表明，在代码覆盖率和缺陷检测方面，∇Fuzz 的表现显著优于最先进的深度学习库模糊测试工具（包括 FreeFuzz 和 Muffin）。事实上，图 1 中的程序错误可由 ∇Fuzz 检测到，而以往任何技术都无法检测到。总之，本文的贡献如下：

1）据我们所知，这是首项专门针对深度学习库中关键自动微分组件进行模糊测试、并采用实用且通用的测试预言机的工作。我们提出的自动微分预言机有望增强并影响未来所有针对深度学习库/系统的模糊测试工作。2）我们已将 ∇Fuzz 实现为一种全自动的深度学习库自动微分测试技术。∇Fuzz 基于最先进的 FreeFuzz 构建，并通过针对各种微分场景进行差分测试，解决了测试预言机这一难题；∇Fuzz 还可以测试任意阶梯度计算的正确性。此外，我们还设计了新颖的策略来过滤假阳性。3）我们对热门深度学习库（PyTorch、TensorFlow、JAX 和 OneFlow）进行了全面研究。∇Fuzz 总共检测出 173 个程序错误，其中 144 个已获开发者确认（117 个此前未知，107 个与自动微分相关），38 个已修复。值得注意的是，在两个月内，∇Fuzz 找出了 PyTorch 和 JAX 的全部高优先级自动微分缺陷中的 58.3%（7/12）。现有工作均无法检测出这 107 个与自动微分相关的程序错误。

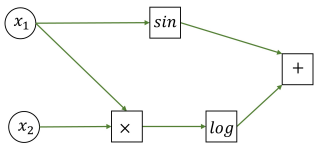
二、背景

A. 深度学习库基础

深度学习模型和深度学习 API。为了开发深度学习流水线，用户通常会在深度学习程序中调用深度学习 API 来完成以下工作：

```
定义 f(x1, x2):
    v1 = x1 * x2
    v2 = log(v1)
    v3 = sin(x1)
    v4 = v2 + v3
    返回 v4
```

(a) 定义



(b) 计算图

图 3：函数 $f(x_1,\ x_2) = \log(x_1 \cdot x_2) + \sin(x_1)$

构建深度学习模型、加载数据集、使用带标签的训练数据训练深度学习模型，并使用评估数据对其进行测试。图2 左侧所示的 TensorFlow 程序示例构建了一个密集神经网络，其中包含两个 `Dense` 层，激活函数分别为 ReLU 和 Softmax 。从输入层 `input`开始，通过依次调用深度学习 API 得到张量输出。随后，通过定义输入和输出构建模型，并使用指定的优化器（随机梯度下降）和损失函数（平均绝对误差）进行编译。接下来，我们使用高级 API`model.fit` 训练模型，并使用`model.predict`进行预测。

训练阶段。 为了便于使用，深度学习库通常会提供高级 API（例如，TensorFlow 中的`model.fit()` ）。然而，实际的训练阶段较为复杂，由三个阶段组成：前向传播、损失计算和反向传播。这些训练步骤会反复执行，直到收敛。图 2 展示了示例程序中一个训练步骤的示意图，其中`W1` 和`W2` 代表`Dense` 层的权重张量。在前向传播过程中，利用输入张量和权重张量计算输出张量时，每个执行的操作都会被自动记录，以用于自动微分（AD）。请注意，为简洁起见，图2 中省略了额外的迹信息。损失计算完成后（需要标签），记录的自动微分上下文将用于计算损失相对于权重张量的梯度（`Grad1`, `Grad2`）。最后，优化器会将梯度应用于权重更新：将 `ΔW1` 和 `ΔW2` 分别加到`W1` 和`W2` 上（这些增量根据梯度计算得出）。

推理阶段。 在推理阶段，深度学习 API 会像前向传播中那样执行，以计算输出张量，但通常会禁用自动微分以提高效率。

B. 自动微分

自动微分（AD）是深度学习框架的核心组成部分之一，对深度学习的成功作出了重要贡献。自动微分将函数/模型分解为一组已知导数的基本运算，并利用链式法则组合这些运算的导数 [39]。它使我们无需大量手动操作即可计算任意函数/模型的导数。自动微分通常有两种不同模式：反向模式（或反向累积）和前向模式（或前向累积）。

反向模式。 反向模式是深度学习库中最常见的自动微分模式。它从输出到输入应用链式法则，这与原始函数/模型的顺序相反。反向模式分两个阶段计算导数：前向阶段和反向阶段。在

表 I：反向模式自动微分计算迹

前向阶段	反向阶段
$x_1 = 1$	$\bar{x}_1 = 1.54$
$x_2 = 2$	$\bar{x}_2 = 0.5$
$v_1 = x_1 \cdot x_2 = 2$	$\bar{x}_1 = \bar{x}_1 + \bar{v}_1 \cdot \frac{\partial v_1}{\partial x_1} = 0.54 + 0.5 \cdot 2 = 1.54$
	$\bar{x}_2 = \bar{v}_1 \cdot \frac{\partial v_1}{\partial x_2} = 0.5$
$v_2 = \log(v_1) = 0.69$	$\bar{v}_1 = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} = 1/v_1 = 0.5$
$v_3 = \sin(x_1) = 0.84$	$\bar{x}_1 = \bar{v}_3 \cdot \frac{\partial v_3}{\partial x_1} = \cos(1) = 0.54$
$v_4 = v_2 + v_3 = 1.53$	$\bar{v}_2 = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_2} = 1$
	$\bar{v}_3 = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_3} = 1$
$f = v_4 = 1.53$	$\bar{v}_4 = \bar{f} = \partial f / \partial v_4 = 1$

前向阶段，我们会得到原始函数的输出，并计算输出值和所有中间变量，这些值会存储在内存中。在反向阶段，通过利用链式法则和中间值计算导数，从而将导数从输出传播回输入。

图3展示了函数 $f(x_1, x_2)$ 及其计算图示例。当 (x_1, x_2) 为 (1,2) 时，反向模式自动微分的计算迹如表 I 所示。为简化表示，我们用 $\bar{v} = \frac{\partial f}{\partial v}$ 表示 f 关于变量 v的偏导数。首先，函数在前向阶段计算输出值 ($f = 1.53$)，并存储所有中间值，如表 I 左侧所示。由于基本运算的导数已知，例如 $\sin(x_1)$ 的导数为 $\cos(x_1)$，反向模式自动微分便可利用链式法则和存储的值，将导数从 f 自动反向传播至输入 x_1, x_2，如表 I 右侧所示。

值得注意的是，由于该函数的输出值是标量，导数 $\bar{x_1} = 1.54$ 和 $\bar{x_2} = 0.5$只需通过一次反向遍历即可计算得出。如果函数更一般（例如，$\mathbf{f}: \mathbb{R}^n \to \mathbb{R}^m$），反向模式自动微分需要 m 次反向遍历才能计算梯度。由于在实践中，深度学习通常需要计算低维张量（例如标量损失值）相对于大量参数的梯度，因此反向模式是深度学习库中主要使用的自动微分模式。

前向模式。 与反向模式不同，前向模式自动微分在计算原始函数/模型输出的同时计算导数，也就是说，它按照从输入到输出的方向求值链式法则。因此，它无需像反向模式那样存储中间值。前向模式自动微分也分为两个阶段：前向原始阶段 和前向切向阶段。前向原始阶段 用于获取原始函数的输出，同时，前向切向阶段 则通过沿输入到输出的方向应用链式法则来计算梯度。

返回 f 图中所示3，其前向模式计算轨迹如表 II所示。前向模式自动微分在前向原始阶段沿途对每个基本运算应用链式法则来计算导数。然而，它每次只能计算一个输入的梯度。在本例中，我们计算f 相对于 x_1的偏梯度。为简单起见，我们将$\dot{v} = \frac{\partial v}{\partial x_1}$ 定义为 v 相对于输入 x_1的偏梯度。因此，我们在开始时设定 $\dot{x_1} = 1$， $\dot{x_2} = 0$因为 x_1 相对于自身的梯度为 1，并且

表 II：前向模式自动微分计算迹

前向原始阶段	前向切向阶段
$x_1 = 1$ $x_2 = 2$	$\dot{x}_1 = 1$ $\dot{x}_2 = 0$
$v_1 = x_1 \cdot x_2 = 2$ $v_2 = \log(v_1) = 0.69$ $v_3 = \text{正弦}(x_1) = 0.84$ $v_4 = v_2 + v_3 = 1.53$	$\dot{v}_1 = \dot{x}_1 \cdot x_2 = 2$ $\dot{v}_2 = \dot{v}_1 / v_1 = 1$ $\dot{v}_3 = \dot{x}_1 \cdot \text{余弦}(x_1) = 0.54$ $\dot{v}_4 = \dot{v}_2 + \dot{v}_3 = 1.54$
$f = v_4 = 1.53$	$\dot{f} = \dot{v}_4 = 1.54$

x_2 不会影响 x_1。沿着表II右侧所示的前向切向阶段，最终偏梯度 $\dot{f} = \frac{\partial f}{\partial x_1}$ 为 1.54，与反向模式计算的结果相同。

一般而言，对于函数 $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$，前向模式自动微分需要进行 n 次求值才能计算梯度：对每个输入，分别将 $\dot{x}_i =$ 设为 1，其余设为 0。因此，当 $n \leq m$ 时，前向模式自动微分在时间和内存方面可能比反向模式更高效，也适用于高效计算海森矩阵 [40] 等情况。

尽管近年来深度学习库测试取得了进展 [25], [27]–[31], [41], ，但目前仍鲜有工作能够有效测试深度学习库中至关重要的自动微分组件。因此，本文旨在构建首个专门针对深度学习库中自动微分的实用模糊测试技术。

III. 预备知识

本节将介绍微分计算的预备知识，这些知识对于理解深度学习库中的自动微分实现以及我们的方法至关重要。

A. 自动微分背后的数学

微分是计算给定函数在给定点处梯度的过程。特别地，对于标量值函数，梯度定义如下：

定义 1。 梯度。 对于标量值函数 $f : \mathbb{R}^n \to \mathbb{R}$ 以及点 $\boldsymbol{x}$，其在 $\boldsymbol{x}$ 处的梯度定义如下：

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}^T \tag{1}$$

梯度还可以进一步推广到返回非标量值的函数，即雅可比矩阵 [42]：

定义 2。 雅可比矩阵。 向量值函数 $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ 的雅可比矩阵定义为一个 $m \times n$ 矩阵：

$$\mathbf{J}(\mathbf{f}) = \begin{bmatrix} \nabla^T f_1 \\ \vdots \\ \nabla^T f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{2}$$

其中 $\mathbf{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}) \ f_2(\boldsymbol{x}) \ldots\ldots, \ f_m(\boldsymbol{x}))$。

本文为简洁起见，用“梯度”表示向量值函数的雅可比矩阵。$\mathbf{f}$ 的梯度函数定义为 $\mathbf{f}'$: $\mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^n$，其中 $\mathbf{f}'(\boldsymbol{x})$ 是 $\mathbf{f}$ 在点 $\boldsymbol{x}$ 处的雅可比矩阵。

梯度 $\mathbf{f}'(\boldsymbol{x}) \in \mathbb{R}^m \times \mathbb{R}^n$ 也可以视为一个线性映射，它将定义域中 $\mathbf{f}$ 在点 $\boldsymbol{x}$ 处的切空间 [43] 映射到陪域的切空间 $\mathbf{f}$ 在点 $\mathbf{f}(\boldsymbol{x})$ 处。给定 $\mathbf{f}'(\boldsymbol{x})$: $\mathbb{R}^n \to \mathbb{R}^m$ 这一映射，我们现在可以定义雅可比向量积（JVP）和向量雅可比积（VJP），它们已被用作高效深度学习训练的理论基础，并通过深度学习库中的前向／反向模式自动微分实现：

定义 3。 雅可比向量积。 给定输入点 $\boldsymbol{x} \in \mathbb{R}^n$ 以及切向量 $\mathbf{u} \in \mathbb{R}^n$，其中该向量来自 $\mathbf{f}$ 在 $\boldsymbol{x}$ 处的切空间，雅可比向量积定义如下：

$$\text{JVP}(\boldsymbol{x}, \mathbf{u}) = \mathbf{f}'(\boldsymbol{x}) \cdot \mathbf{u} \tag{3}$$

JVP 计算方向梯度，方向为 $\mathbf{u} \in \mathbb{R}^n$，对于函数 $\mathbf{f}$: $\mathbb{R}^n \to \mathbb{R}^m$ 在点 $\boldsymbol{x} \in \mathbb{R}^n$ 处，并在深度学习库中通过前向模式自动微分实现。回到图 3所示的示例，前向模式自动微分迹（表 II）中的 $(\dot{x}_1, \dot{x}_2)$ 是定义 3中的切向量 $\mathbf{u}$ 。因此，对于输入 $\boldsymbol{x} \in \mathbb{R}^n$ 和切向量 $\mathbf{u} \in \mathbb{R}^n$，前向模式自动微分只需通过设置 $\dot{\boldsymbol{x}} = \mathbf{u}$ 进行一次计算即可求得 JVP。与此同时，使用前向模式自动微分计算完整的雅可比矩阵需要 n 次计算。

定义 4。 向量雅可比积。 给定输入点 $\boldsymbol{x} \in \mathbb{R}^n$ 和余切向量 $\mathbf{v} \in \mathbb{R}^m$，向量雅可比积定义为如下映射：

$$\text{VJP}(\boldsymbol{x}, \mathbf{v}) = \mathbf{v} \cdot \mathbf{f}'(\boldsymbol{x}) \tag{4}$$

方向为 $\mathbf{v} \in \mathbb{R}^m$ 时，VJP 计算函数 $\mathbf{f}$: $\mathbb{R}^n \to \mathbb{R}^m$ 在点 $\boldsymbol{x} \in \mathbb{R}^n$ 处的伴随方向梯度。同样，深度学习库通过反向模式自动微分计算 VJP。反向模式自动微分迹（表 I）中的 $f = 1$ 是输出为标量时余切向量 $\mathbf{v}$ 的特殊情况。一般而言，对于输入 $\boldsymbol{x} \in \mathbb{R}^n$ 和余切向量 $\mathbf{v} \in \mathbb{R}^m$，反向模式自动微分能够通过初始化并设置 $\mathbf{f} = \mathbf{v}$，仅需一次计算即可求得 m VJP。相比之下，反向模式计算完整的雅可比矩阵需要

B. 数值微分

数值微分（ND） [44]是另一种通过使用原函数在若干采样点处的值来估算函数导数的方法。最常见的方法是使用有限差分近似。例如，对于标量值函数 $f : \mathbb{R}^n \to \mathbb{R}$，在点 $\boldsymbol{x}$ 处，我们可以使用数值微分计算 x_i 的偏导数：

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \frac{f(\boldsymbol{x} + \epsilon \boldsymbol{e}_i) - f(\boldsymbol{x} - \epsilon \boldsymbol{e}_i)}{2\epsilon} \tag{5}$$

其中，$\boldsymbol{e}_i$ 是第 i 个单位向量，$\epsilon > 0$ 是一个较小的步长。

然而，数值微分（ND）可能因截断误差和舍入误差而不准确 [39], ，尤其是在使用低精度数据类型时。此外，对于一个 n 维的梯度，数值微分的耗时为 $O(n)$ ；这是其在深度学习库中应用的主要障碍，因为深度学习模型中的 n 可能高达数十亿 [34]。因此，深度学习库不会将数值微分作为计算梯度的主要方法。相反，大多数深度学习库会在开发者测试期间利用数值微分交叉检查自身的梯度计算实现。

在本研究中，我们进一步扩展了 ∇模糊测试预言机，使其结合数值微分。这是因为两种自动微分模式可能返回相同的错误梯度，而通过比较反向模式和前向模式无法检测到这一问题（详见§IV-B2）。据我们所知，我们也是首个将数值微分用于深度学习库自动化模糊测试的研究。

IV. 方法

图4展示了我们用于测试深度学习库自动微分机制的 ∇Fuzz 方法概览。请注意，为便于说明，我们将每个待测深度学习库 API 抽象为一个函数 f：$\mathbb{R}^n \to \mathbb{R}^m$。 ∇Fuzz 首先调用现成的 API级模糊测试器来生成函数的输入 $\boldsymbol{x} \in \mathbb{R}^n$ （第 IV-A 节）。然后， ∇Fuzz 会在不同执行场景下交叉检查其输出和梯度 $\boldsymbol{x}$ （第 IV-B 节）。如果 f 在给定 $\boldsymbol{x}$ 的测试中通过（不存在任何不一致）， ∇Fuzz 将继续测试 f 的高阶梯度： ∇Fuzz 会将 f 包装为其梯度函数 f′：$\mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^n$，并重新运行测试判定器（第 IV-B3 节）。如果存在任何不一致（发生在一阶或高阶梯度计算期间）， ∇Fuzz 将过滤掉由数值不稳定性导致的假阳性（第 IV-C 节）。最后， ∇Fuzz 会返回候选程序错误。以下各小节将详细介绍每个组件。

A. API级模糊测试器

∇Fuzz 的第一个组件是一个 API 级模糊测试器，用于生成输入以调用每个深度学习 API。我们的方法具有通用性，可以利用任何现成的 API 级深度学习库模糊测试器 [30], [31]。在这项工作中，我们使用 FreeFuzz [30] 为函数/API 创建输入，因为它完全自动化且达到了最先进的水平。深度学习库的 API 通常以 Python 形式提供，而 Python 是一种动态类型语言，因此确定每个深度学习 API 的输入类型更加困难。为解决这一问题，FreeFuzz 会在执行从各种来源挖掘的代码时自动跟踪 API 输入，这些来源包括深度学习模型、开发者测试以及深度学习文档中的代码片段。FreeFuzz 还包含变异策略，可根据跟踪到的种子 API 输入生成更多输入。

深度学习库的 API 可能带有配置参数（除了输入张量之外）。例如，`torch.sum(input,dim)`会返回`input` 张量沿`dim`维度逐行求和的结果，其中 dim 是一个配置参数。FreeFuzz 可以为输入张量和配置参数生成输入。对于 FreeFuzz 生成的每次成功 API 调用， ∇Fuzz 都会自动创建一个包装函数，将 API 调用转换为一个从输入映射到输出张量的函数。此外，深度学习库的 API 可能会将多个多维张量作为输入或输出，例如`tf.add(x,y)`，它会对两个多维张量`x` 和`y` 进行逐元素相加。在我们的实现中，虽然 ∇Fuzz 直接应用于这类输入和输出均为张量的 API，但为便于说明，我们将每个 API 抽象为$\mathbf{f}$ ： $\mathbb{R}^n \to \mathbb{R}^m$这种抽象可以看作是将多维张量展平为向量（如果有多个输入或输出张量，则将它们拼接起来）。

算法 1： ∇Fuzz 预言机算法

```
1  函数 ∇Fuzz-Oracle(fn，input，order)：
       输入： 被测函数 fn，函数输入
              输入，以及要测试的梯度阶数
       输出：预言机结果
2      当前阶数 ← 1
3      当当前阶数 ≤ 阶 执行
4          输出 ←DirectInv (函数, 输入, REP=10)
5          如果不IsOutputConsistent(输出) 则
6              返回 RANDOM
7          反向输出, 反向梯度 ←反向逆变换 (函数, 输入)
8          前向输出, 前向梯度 ← 前向逆变换 (函数, 输入)
9          如果 不 IsOutputConsistent(输出,
             反向输出，前向输出)则
10             返回输出不一致 _
11         ndGrad ← NDGrad（fn，input）
12         如果 不 IsGradientConsistent(revGrad,
             fwdGrad，ndGrad）那么
13             返回梯度不一致    _
14         fn ← Grad (fn)
15         curOrder ← curOrder + 1
16     返回 通过
```

B. 测试预言机

如算法1所示， ∇Fuzz 预言机算法的输入包括被测函数、函数的一个输入，以及要测试的最高阶梯度。我们从一阶梯度测试开始（第2行）。 ∇Fuzz 首先通过多次（默认 10 次）使用给定输入直接调用函数，检查函数的确定性（第4行）。如果输出不一致， ∇Fuzz 将返回 RANDOM，并终止该函数的模糊测试过程（第5-6行）。否则，它会继续使用反向模式自动微分和前向模式自动微分调用该函数（第7-8行）。随后，它会比较直接调用和使用自动微分调用所返回的输出。如果检测到任何不一致（第9行）， ∇Fuzz 将跳过梯度检查，并返回这一输出不一致（第10行）。否则， ∇Fuzz 将通过比较反向模式自动微分、前向模式自动微分和数值微分计算出的梯度，继续检查梯度计算的正确性（第12行）。如果这些梯度不同，它将返回这一不一致（第13行）。如果函数通过了上述所有检查，并且我们希望继续测试高阶梯度计算（第3行）， ∇Fuzz 将该函数转换为其梯度函数（第14行）。主循环将继续测试这个新函数，直到满足终止条件，例如检测到不一致，或通过最高阶梯度计算测试。接下来，我们将更详细地介绍输出和梯度检查。

1) 输出检查： 在使用反向模式或前向模式自动微分计算梯度时，始终会产生一些额外操作，例如跟踪或形状检查。因此，使用自动微分的调用所得输出可能与直接调用不同。然而，不同执行场景下的输出不应存在差异，这意味着任何不一致都可能是程序错误。因此， ∇Fuzz 会比较直接调用的输出，以及使用

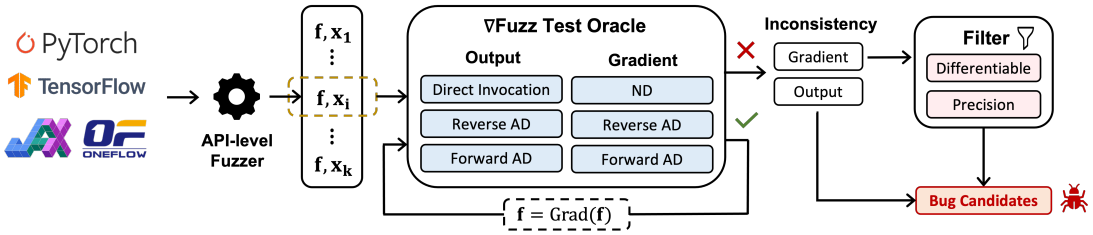


图 4： ∇Fuzz 概览

反向模式和前向模式自动微分的调用所得输出。

采用 JAX API `jax.lax.dynamic_索引_在_维度`例如，该函数对输入数组执行整数索引操作 [45]。图 5 展示了一个示例，其中直接调用和反向模式自动微分得到的输出值不同。其根本原因是，反向模式自动微分会导致索引被多次规范化，从而改变越界负索引的结果。该程序错误已被 ∇Fuzz 检测到，并已由 JAX 开发者确认和修复。

```
定义 函数(输入):
    返回 动态_索引_在_维度(输入, 索引=-7, 轴=1)
输入 = 数组([[1., 2., 3., 4., 5.]])
直接值(fn, 输入) # [[1. 0.]]
反向值(fn, 输入)              # [[4.0]]
```

图 5：使用和不使用自动微分时输出不一致

2）梯度检查：由于反向模式和前向模式采用不同的方法计算梯度，因此对于相同的函数和输入，它们可能产生不同的梯度。此外，由于反向模式和前向模式可能采用同一个不正确的公式，导致得到相同的错误梯度值，因此可以使用数值微分来测试自动微分的梯度计算。因此，∇Fuzz 比较反向模式自动微分、前向模式自动微分和数值微分计算出的梯度，以检测缺陷。

例如，PyTorch API `torch.trace` [46] 会返回输入二维矩阵的对角元素之和。显然，形状为 `(4,2)` 的输入，其对角线上有两个元素，因此该 API 会返回这两个元素之和。然而，反向模式下计算出的梯度中有三个 元素的梯度为 1，而前向模式下只有两个 元素的梯度为 1。这种不一致是由反向模式自动微分中用于`torch.trace`的错误公式导致的。∇Fuzz 发现的这个程序错误已得到开发者确认。

下面是另一个例子，展示了进一步利用数值微分的价值。PyTorch API `hardshrink(x,lambd)` [47]在 `|x|>lambd`时返回 `x`；否则，它只返回 0。也就是说，当 `lambd` 为 0 时，这个 API 等价于线性函数 $y = x$。然而，对于输入 0，在自动微分和数值微分中，该 API 的梯度会因`lambd=0`而有所不同。反向模式和前向模式自动微分都返回 0 作为梯度，而数值微分返回 1。显然，梯度应为 1。∇Fuzz 检测到的这个程序错误也已在 PyTorch 中得到确认。

然而，由于数值微分（ND）的缺点（例如截断和舍入误差），它可能会产生不一致的梯度。因此，我们只对精度较高的输入使用数值微分，例如双精度浮点数，以尽可能减小截断和舍入误差的影响。此外，我们设计了策略来缓解数值微分在 §IV-C 中不稳定所导致的假阳性。

3）高阶梯度：除了基本的一阶梯度外，∇Fuzz 还能测试高阶梯度计算的正确性。更确切地说，∇Fuzz 可以将当前被测函数 $\mathbf{f}: \mathbb{R}^n \to \mathbb{R}^m$ 的梯度函数 $\mathbf{f}'$：$\mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^n$ 作为输入，因为我们设计的测试预言机适用于梯度函数。随后，∇Fuzz 可以对 $\mathbf{f}'$同时进行输出检查和梯度检查。但是，$\mathbf{f}'$的梯度（即$\mathbf{f}$的二阶梯度）是什么形式呢？为说明这一点，我们先介绍标量值函数的二阶梯度——海森矩阵 [40],:

定义 5。海森矩阵。标量值函数 f：$\mathbb{R}^n \to \mathbb{R}$ 的海森矩阵定义为如下 $n \times n$ 矩阵：

$$\mathbf{H}(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{6}$$

由此，更一般的向量值函数$\mathbf{f}$：$\mathbb{R}^n \to \mathbb{R}^m$ 的二阶梯度可以定义为一个$m \times n \times n$ 矩阵，它可以看作由多个 m 海森矩阵组成的数组。形式上，对于函数 $\mathbf{f}(\boldsymbol{x}) = (f_i(\boldsymbol{x}))_{i=1}^m$，二阶梯度可定义为：

$$\mathbf{H}(\mathbf{f}) = (\mathbf{H}(f_1), \mathbf{H}(f_2), \ldots, \mathbf{H}(f_m)) \tag{7}$$

值得注意的是，当前受测函数可以是其他函数的梯度。因此，从理论上讲，∇Fuzz 可以测试任意阶的梯度计算。当 α 阶梯度函数通过上述所有测试，而我们希望测试其高阶梯度计算的正确性时，∇Fuzz 会将 α 阶梯度函数转换为其梯度函数，并重新运行测试。在这项工作中，我们关注一阶和二阶梯度计算的正确性，因为它们最常用。此外，据我们所知，现有的深度学习库中，只有极少数提供了计算三阶以上梯度的 API。

以 JAX API `jax.lax.pow(a,b)` [48]为例，该 API 返回 a^b。当输入 `(a,b)` 为`(2,0)`时，该 API 可以通过一阶梯度测试。随后，∇Fuzz 将测试在该输入下计算二阶梯度的正确性，如图 6所示。结果表明，反向模式自动微分中计算出的二阶梯度与数值微分中的结果不同，而后者是正确的。这是因为梯度 $\frac{\partial^2 f}{\partial a \partial b}$ 应当与其完全相同

$\frac{\partial^2 f}{\partial b \partial a}$ 对于 API `jax.lax.pow` 而言，由 ∇Fuzz 检测到的这一不一致已得到确认，甚至被 JAX 开发者标记为“紧急”，并在我们的报告发布后立即修复。

```
a, b = 数组(2.0), 数组(0.0)RevGrad(Grad(jax.lax.pow),(a, b)) #[[0.0, 2.0],[0.5, 0.48]]NDGrad(Grad(jax.lax.pow), (a, b)) #[[0.0, 0.5],[0.5, 0.48]]
```

图 6：自动微分和数值微分中的二阶梯度不一致

C. 过滤策略

针对 ∇Fuzz 的梯度检查会检查由完全不同模式/实现计算出的梯度，因此，与输出检查（检查主要由共享实现返回的值）相比，可能会产生更多假阳性；§VI-C中的结果分析也证实了这一点。因此， ∇Fuzz 还采用另外两种过滤策略，以处理由数值不稳定性问题引起的梯度不一致。

1）可微性：对于由不可微点引起的梯度不一致，我们以绝对值函数为例，如图7所示。JAX 中由自动微分计算出的梯度在输入为 0 时为 1，但数值微分会计算

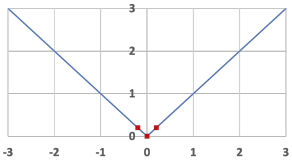


图 7：绝对值函数

梯度为 0。不过，这种不一致是可以接受的，因为绝对值函数在点 0 处不可微。不可微点处的梯度未定义，因此自动微分可以返回任意值作为梯度。因此，我们需要过滤掉由不可微点导致的此类情况。为此，我们首先定义可微性这一性质 [49]:

定义 6. 可微性。函数f 在 x 处可微，当且仅当：*1*）f 在 x处连续；*2*）f 的所有偏导数在 x 的邻域内存在，并且在 x 处连续。

根据定义 6，要测试函数 f 在点 x处的可微性，我们可以对x的一些邻居进行采样。采样后，计算这些点处的输出和梯度，并与x处的输出和梯度进行比较。请注意，我们选择数值微分作为可微性检查方法，因为自动微分机制是主要测试目标，我们不希望误将自动微分错误当作不稳定性，从而漏掉这些错误。

我们将采样 N （默认值为 5）个随机邻居来检查可微性。更具体地说，我们将随机邻居定义为随机(x) $= x +$ 均匀分布($-\delta$, $+\delta$)，其中采样距离 δ 是一个超参数（默认值为 10^{-4}）。如果任一邻居的输出或梯度不同于该点x， ∇Fuzz 将认为f 在该点x处不可微。回到绝对值函数的例子。对于点 0，很明显，其左邻居的梯度为 -1，右邻居的梯度为 1。二者都不同于 0，而 0 是 ND 在点 0 处计算出的梯度。如

一个结果， ∇Fuzz 将过滤掉这个假阳性案例。

2）精度转换：图8 展示了一个`fn`函数，该函数将输入张量转换为半精度浮点数数据类型，并返回其所有元素的求和 [50]。尽管输入具有

```
定义 fn(input)：返回torch.sum(input，dtype=float16)input = torch.tensor([16。0], dtype=float64)fn(input)          # 张量(16.，dtype=float16)fn(input + 1e-4) # 张量(16.，dtype=float16)fn(input - 1e-4) # 张量(16.，dtype=float16)
```

图 8：精度损失示例 表III：所研究的深度学习库详情

	GitHub 星标数	公司	# API总数	# 已覆盖API	版本
PyTorch	57.7K	Meta	1592	1071	1.11
TensorFlow	167K	Google	6381	1902	2.9
JAX	19.7K	Google	791	634	0.3.14
OneFlow	3.6K	OneFlow	409	299	0.7.0

数据类型双精度浮点数，对其施加扰动`1e-4` 也无法改变输出，这是由于舍入导致的精度损失，如图8所示。因此，ND 将返回 0 作为梯度，这显然是错误的。此外，精度转换还可能导致反向模式和前向模式自动微分中的梯度不一致。在大多数深度学习库中，反向模式自动微分计算出的梯度与输入具有相同的数据类型，而前向模式返回的梯度与输出具有相同的数据类型。因此，前向模式和反向模式自动微分可能会因精度损失而产生轻微的不一致。为排除这类不一致，我们排除 API 输入和输出具有不同精度的所有情况。

V. 实验设置

本研究探讨以下研究问题：

- **研究问题 1：** ∇Fuzz 在检测现实世界漏洞和提高代码覆盖率方面是否有效?
- **研究问题 2：** ∇Fuzz 预言机的不同组件如何影响其性能?
- **研究问题 3：** 过滤策略如何帮助降低 ∇Fuzz 的假阳性率?

为回答这些研究问题，我们对 PyTorch、TensorFlow、JAX 和 OneFlow 进行了广泛研究，详情见表III。PyTorch 和 TensorFlow 在 GitHub 上分别拥有 57.7K 和 167K 个星标，是最受欢迎的两个深度学习库，也在以往的深度学习库测试研究中得到了广泛探讨 [30], [31], [41]。此外，JAX [51] 和 OneFlow [52]是两个新兴的深度学习库，在 GitHub 上分别拥有 19.7K 和 3.6K 个星标。JAX 提供简单而强大的应用程序接口，用于编写加速数值代码，以支持高性能机器学习研究。随着在分布式设备上训练大型模型的研究不断增加，OneFlow 采用了简洁明了的重新设计，与现有框架相比，可更轻松地编程实现各种并行范式。

我们将 ∇Fuzz 与最先进的模型级深度学习库模糊测试工具（Muffin [29]）和 API 级深度学习库模糊测试工具（FreeFuzz [30]）进行比较。所有实验均在一台配备 32 核 AMD CPU（3.5GHz）、256GB 内存并运行 Ubuntu 20.04 的机器上进行。

A. 实现

1) 输入生成器：我们利用 FreeFuzz [38] 的输入数据库和模糊测试策略来生成 API 输入。虽然我们的方法具有通用性，但我们选择 FreeFuzz，因为它是最先进且完全自动化的。我们采用其默认

设置，为每个 API 通过变异生成 1000 个输入。由于 FreeFuzz 仅针对 PyTorch 和 TensorFlow 实现，我们还自行实现了适用于 JAX 和 OneFlow 的 FreeFuzz 类模糊测试引擎。我们沿用 FreeFuzz 的做法，从开源项目中收集 API 输入，并实现相应的模糊测试策略。对于 JAX 的输入收集，我们仅追踪开发者测试（83 个测试文件），因为这些测试已覆盖 80.2% 的 JAX API（634/791）。对于 OneFlow，我们从全部三个来源收集输入：文档、519 个开发者测试和 51 个深度学习模型，覆盖了 73.1% 的 OneFlow API（299/409）。

2) 执行场景： 表 IV 展示了我们的工具在每种执行场景中使用的示例微分 API。请注意，受篇幅限制，表中未列出我们使用的所有与自动微分相关的 API。表中的“N/A”表示该深度学习库支持或提供该场景对应的 API。只有 OneFlow 尚未实现前向模式自动微分，因此我们跳过了针对 OneFlow 的前向模式自动微分测试。

对于提供了可比较数值微分和自动微分梯度的 API（见“ND”列）的深度学习库， ∇Fuzz 会直接使用此类 API。结果表明，只有 OneFlow 不提供此类 API，因此我们自行实现了其数值微分。

3) 筛选： 对于可微性检查中的邻域采样，我们将采样次数 N 设为 5，并将距离δ 默认设为 10^{-4} 。我们还在 §VI-C 中探讨了它们的影响。

B. 指标

检测到的错误数量。 遵循以往关于深度学习库测试或模糊测试的工作 [25], [28]–[31], [41], [53], ，我们报告 ∇Fuzz 和对比基线方法检测到的错误数量。

假阳性率。 过滤掉由不稳定性导致的不一致情况后，我们得到错误候选项。然而，并非所有候选项都是真正的程序错误。假阳性率（FPR）用于计算候选项中误报所占的比例，并广泛用于先前的测试／模糊测试研究 [53]–[56]。按照 Muffin [29], 的做法，对于∇Fuzz 报告的每个不一致情况，三位作者分别独立检查，判断其是否属于程序错误，然后共同讨论并达成共识。此外，与 Muffin 不同，我们还利用开发者反馈来校准检查结果。也就是说，如果作者达成共识，认为某个不一致情况并非程序错误，或者开发者驳回了我们的报告，我们就会将其报告为假阳性。

代码覆盖率。 代码覆盖率是软件测试的主要标准之一，最近也被用于测试深度学习库／编译器 [29], [30], [57]。最先进的 FreeFuzz [30] 和 Muffin [29] 仅采用 C++或 Python 覆盖率，而我们同时采用这两种代码覆盖率标准，以进行更全面的评估。按照 FreeFuzz 和 Muffin 的做法，我们采用行覆盖率，并分别通过 GCOV [58] 和`Coverage.py` [59], 跟踪 C++ 和 Python 的行覆盖率。

执行时间。 由于 ∇Fuzz 会利用额外的预言机来检测自动微分错误，因此其耗时会比现有的 API 级模糊测试工具（如 FreeFuzz）更长。因此，我们参照先前的工作 [28], [30], [57], 将执行时间纳入考量。

VI. 结果分析

A. RQ1：检测到的错误与覆盖率

1）检测到的错误： 表 V 展示了 ∇Fuzz 在所有研究库中检测到的真实缺陷汇总。“总计”列显示检测到的错误总数。“已确认（已修复）”列显示经开发者确认并修复的错误数量。我们进一步将已确认的错误分为此前未知和已知两类。此外，“已驳回”列显示开发者驳回的错误数量。最后，“待处理”列显示开发者尚未分类处理的错误数量。

我们可以看到， ∇Fuzz 能够在所研究的四个深度学习库中总共检测到 173 个缺陷，其中 144 个已由开发者确认，38 个已修复，这凸显了 ∇Fuzz 的有效性。值得注意的是，开发者确认其中 117 个是此前未知的错误，只有 6 个被驳回。在已确认的 144 个缺陷中，最先进的 FreeFuzz 和 Muffin 只能检测到 21 个非自动微分错误（全部由 FreeFuzz 检测到，Muffin 检测到 0 个）。在 FreeFuzz 检测到的这些缺陷中，15 个是未知错误，6 个此前已为人所知。在这些未知错误中，10 个来自 JAX 和 OneFlow（原版 FreeFuzz 不支持这两个库），5 个来自 PyTorch 和 TensorFlow。6 个缺陷因以下原因被驳回：3 个是使用低精度数据类型导致精度损失，2 个是为确保数值稳定性而有意实现的，1 个则源于不可微点处的未定义行为。

值得注意的是，我们检测到的 PyTorch 错误中有 6 个被标记为“高优先级”，JAX 的 1 个程序错误被标记为“P0（紧急）”（这 7 个错误都与自动微分有关），因为它们至关重要，应当立即处理。另两个库（TensorFlow 和 OneFlow）没有此类标签，因此本文不予讨论。图 9 展示了我们在 `rrelu` [60]中检测到的一个错误梯度错误，该错误由 PyTorch

开发者将其视为一个重大程序错误，标记为“高优先级”，并立即修复。对于 PyTorch，在我们报告问题的两个月（2022 年 5 月和 6 月）内，其整个问题跟踪系统中共有 78 个高优先级错误，其中 11 个与自动微分有关。也就是说， ∇*Fuzz* 贡献了 *7.7%* 的高优先级错误，以及 *55.5%* 的高优先级自动微分缺陷，这表明我们的方法十分有效。JAX 的问题跟踪系统中有 22 个“紧急”错误（涵盖所有时间），其中只有 1 个与自动微分有关，该错误由我们报告。

```
定义 函数(input): 返回rrelu(input, -2.9, -2.7, True)input =
PyTorch.tensor([0.1250, 0.4313])RevGrad(fn, input) # 张量([[0.,
0.],[0.,0.])NDGrad(fn, input) # 张量([[1.,1.],[1.,1.])
```

图 9：PyTorch 中的高优先级崩溃错误

鉴于 §IV 中已讨论过多个已确认/已修复的程序错误，这里我们将讨论一个已驳回的程序错误示例。图 10 展示了 JAX API`jax.numpy.sinc(x)` [61], 的一个实例，该函数计算 $\sin(\pi x)/(\pi x)$。当输入 `x` 采用精度最低的浮点数据类型 `bfloat16` [62], 时，此 API 在前向模式和反向模式自动微分中计算出的梯度会有所不同。我们向 JAX 开发者报告了这一不一致，但该问题遭到驳回：

表 IV：所研究的深度学习库中与自动微分相关的 API 示例

	反向模式自动微分	前向模式自动微分	ND
PyTorch	torch.autograd.grad	torch.autograd.forward_ad	torch.autograd.gradcheck
TensorFlow	tf.GradientTape.gradient	tf.autodiff.ForwardAccumulator	tf.test.compute_梯度
JAX	jax.jacrev	jax.jacfwd	jax.test_util.check_grads
OneFlow	oneflow.autograd.grad	N/A	N/A

表 V：检测到的错误汇总

	总计	已确认（已修复）		已驳回	待处理
		未知	已知		
PyTorch	80	62 (10)	15 (9)	3	0
TensorFlow	29	18 (0)	5 (2)	2	4
JAX	34	20 (5)	3 (2)	1	10
OneFlow	30	17 (6)	4 (4)	0	9
总计	173	117 (21)	27 (17)	6	23

“*bfloat16* 是有意设计的结果。在深度学习场景中，这是以速度换取精度的一个值得的权衡……”。

```
x = 数组(-0.125, dtype=
bfloat16)RevGrad(jax.numpy.sinc, x) #
0.34375FwdGrad(jax.numpy.sinc, x) # 0.375
```

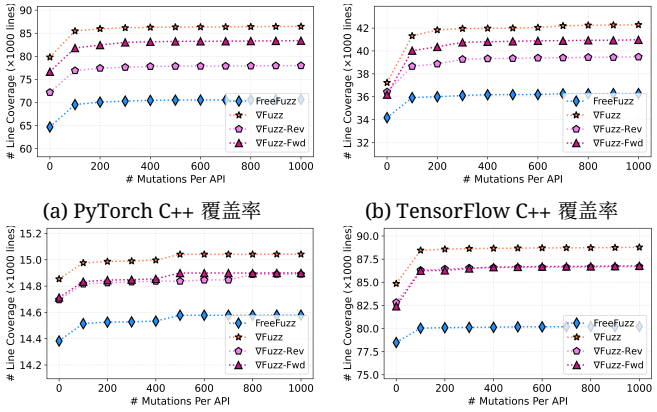
图 10：反向/前向模式下不一致的梯度

2）覆盖率：我们展示了∇Fuzz 和最先进的 FreeFuzz 在默认实验对象 PyTorch 和 TensorFlow 上实现的代码覆盖率，因为它们不仅是最受欢迎的深度学习库，也是 FreeFuzz 研究过的仅有的两个库。在 JAX/OneFlow 上的对比结果类似，因篇幅限制而省略。我们遵循 FreeFuzz 的默认设置 [30]，，即在运行数据库中的种子输入后，为每个 API 执行 1000 个变异输入。

图 11展示了覆盖率结果，其中 x 轴表示为每个 API 生成的变异体数量（从 100 到 1000，间隔为 100），而 y 轴表示实现的总体行覆盖率。请注意，运行 FreeFuzz 数据库中的种子输入（不进行任何变异）所实现的代码覆盖率，是每条线的起点。对于 C++ 覆盖率，我们可以观察到，∇Fuzz 在 PyTorch 和 TensorFlow 上都显著优于 FreeFuzz，提升幅度分别为 22.4% 和 16.6%。请注意，这种提升非常有价值，因为额外覆盖的代码主要涉及关键的自动微分机制。对于 Python 覆盖率，∇Fuzz 仍然优于 FreeFuzz，但提升幅度小于 C++。可能的原因是，深度学习库的关键自动微分功能主要以 C++实现，例如，PyTorch 的官方资料称：“*Autograd* 是 *PyTorch* 性能的关键瓶颈，因此大部分繁重工作都以 *C*++实现。” [63]。

表VI 进一步展示了 FreeFuzz 和 ∇Fuzz 的耗时和整体系统覆盖率。由于需要额外进行梯度计算（主要是开销较大的二阶梯度计算），∇Fuzz 的耗时高于 FreeFuzz。同时可以发现，∇Fuzz 仅运行数据库中的种子输入（行“∇Fuzz（仅种子输入）”）时，虽然耗时更少，代码覆盖率仍优于 FreeFuzz。此外，∇Fuzz 实现了可观的系统覆盖率，例如，整个 PyTorch C++ 代码库的覆盖率为 25.8%，整个 TensorFlow Python 代码库的覆盖率为 33.3%。

我们还将 ∇Fuzz与Muffin进行了比较。由于Muffin不支持PyTorch、JAX或OneFlow，因此我们仅在TensorFlow上进行比较。我们使用Muffin的默认设置运行它（耗时6.8小时）。为确保公平比较，我们运行 ∇Fuzz，并将每个API的变异体数量设置为150，使其能够在6.8小时内完成。如表 VII所示，在执行时间略短（6.1小时）的情况下，∇Fuzz在代码覆盖率和API覆盖率方面都已显著优于Muffin。事实上，即使仅使用我们的自动微分预言机运行种子API输入而不进行变异（仅耗时2.9小时），也足以在C++和Python覆盖率方面超过Muffin。这是因为∇Fuzz能够覆盖更多API和更多自动微分模式，而Muffin仅在少量API上考虑反向模式自动微分。更具体地说，Muffin只能覆盖79个TensorFlow API，而 ∇Fuzz可以覆盖1902个。这是因为Muffin仅考虑一组预定义的高级层 API [29]来进行模型生成。与此同时，Muffin已经能够实现相当不错的代码覆盖率（尽管低于 ∇Fuzz），因为这些高级 API 会使用各种底层操作。



(a) PyTorch C++ 覆盖率　(b) TensorFlow C++ 覆盖率

(c) PyTorch Python 覆盖率　(d) TensorFlow Python 覆盖率

图 11：覆盖率趋势分析

表 VI：与 FreeFuzz 的比较

	PyTorch			TensorFlow		
	C++ Cov	Python 覆盖率	Time	C++ Cov	Python 覆盖率	Time
FreeFuzz	70639 (21.1%)	14579 (13.9%)	3.1h	36279 (9.77%)	80220 (30.1%)	3.9h
∇模糊测试	86459 (25.8%)	15042 (14.3%)	25.7小时	42284 (11.4%)	88783 (33.3%)	24.3 小时
∇模糊测试（仅种子）	79808 (23.4%)	14854 (14.1%)	1.4h	37233 (10.0%)	84848 (31.9%)	2.9h

表 VII：与 Muffin 的比较

	C++ 覆盖率	Python 覆盖率	# 已覆盖 API	Time
∇模糊测试	41625（11.21%）	88524（33.24%）	1902	6.1h
Muffin	36884（9.94%）	78754（29.57%）	79	6.8h

B. 研究问题 *2*：测试预言机的不同组成部分

1）对缺陷检测的影响：在表VIII中，我们根据已确认缺陷所在的执行场景对其进行分类，场景包括直接调用、自动微分和数值微分。

表 VIII：已确认的缺陷的场景分布

	直接	AD			ND
	调用	全部仅反向		仅前向	
PyTorch	11	64	33	9	2
TensorFlow	3	18	5	4	2
JAX	3	20	3	1	0
OneFlow	16	5	5	N/A	N/A
总计	33	107	46	14	4

表 IX：已确认缺陷的症状

	输出	梯度总数	一阶	二阶
PyTorch	31	46	44	2
TensorFlow	4	19	17	2
JAX	14	9	8	1
OneFlow	16	5	2	3
总计	65	79	71	8

这些缺陷所在的场景。在自动微分相关缺陷中，“全部”列显示位于自动微分场景中的缺陷总数，而“仅反向”/“仅前向”列分别显示仅出现在反向模式/前向模式中的缺陷数量。也就是说，有 47 个（107-46-14）缺陷同时存在于反向和前向自动微分模式中。从该表可以看出，Fuzz 检测到的大多数缺陷都与我们的主要目标自动微分有关，这体现了 Fuzz 对深度学习库进行自动微分模糊测试的能力。有一个有趣的发现：在 OneFlow 中，我们通过直接调用检测到的缺陷多于通过自动微分检测到的缺陷。这可能是因为原始 FreeFuzz 工作未测试 OneFlow，且 OneFlow 仅支持反向模式自动微分。此外，由于反向模式在深度学习库中的实现更为普遍，我们检测到的反向模式特有缺陷多于前向模式特有缺陷。如 §V-A2所述，我们直接使用 PyTorch、TensorFlow 和 JAX 中的数值微分计算/比较 API。结果发现，我们甚至能在这类 API 中检测到 4 个缺陷。

我们还在表 IX 中按照检测方式对所有已确认的缺陷进行分类，例如，缺陷是通过不一致的输出（“输出”列）还是梯度（“梯度总数”列）发现的。我们还将通过梯度检测到的缺陷进一步划分为一阶梯度检查（“一阶”列）和二阶梯度检查（“二阶”列）。我们可以看到，超过半数的缺陷是通过不一致的梯度检测到的，这表明梯度预言机十分重要。此外，大多数与梯度相关的缺陷都涉及一阶梯度。这是因为一阶和二阶梯度计算通常共享部分实现，而前者中的任何程序错误都会阻止 ∇Fuzz 测试后者。值得注意的是，∇Fuzz 仍可通过二阶梯度检查检测到 8 个缺陷，这表明我们的方法具有通用性。更有意思的是，不一致的输出揭示了 65 个缺陷，这说明自动微分机制甚至可能影响正常深度学习 API 的前向计算！

2）对代码覆盖率的影响： 为研究执行场景对代码覆盖率的影响，我们设置了两种 ∇模糊测试变体：∇Fuzz-Rev（禁用反向模式自动微分）和 ∇Fuzz-Fwd（禁用前向模式自动微分）。由于 ND 通常基于基本的直接 API 调用实现，且几乎无法覆盖新代码，因此我们跳过了对 ND 的覆盖率分析。图 11还展示了针对所研究变体的结果，其中每个 API 采用了不同数量的变异。我们可以观察到，反向模式自动微分在深度学习库实现中所占的比重高于前向模式，因为在代码覆盖率方面，∇Fuzz-Fwd 优于 ∇Fuzz-Rev。这与反向模式自动微分是深度学习系统中所采用的主要技术这一事实相符。更重要的是，我们还可以观察到，反向模式自动微分和前向模式自动微分的代码覆盖率都不可忽略，这表明在进行深度学习库模糊测试时，有必要同时考虑二者。

C. *RQ3*：过滤策略的假阳性率与有效性

表X 展示了假阳性率（FPR）的结果，并根据检查类型进行了分类。“输出”/“梯度”列分别展示输出检查/梯度检查的假阳性率。“总计”列是我们技术的总体假阳性率。在“梯度”列下，“全部”表示使用两种过滤策略时的假阳性率，“不适用”表示未使用任何策略时的假阳性率。“可微性”/“精度”列分别展示仅使用可微性策略/精度策略时的假阳性率。从表中可以看出，∇Fuzz 的总体假阳性率仅约为 20%，这表明 ∇Fuzz 行之有效。值得注意的是，我们的过滤策略没有误删任何真实缺陷；我们对报告的所有不一致都进行了人工检查，证实了这一点。此外，经过过滤后，梯度预言机的假阳性率（22.4%）远低于未过滤时（66.7%），这表明我们的过滤策略有效。我们还可以看到，两种过滤策略都能有效降低假阳性率。具体而言，可微性策略比精度策略更有帮助，尤其是在 OneFlow 中，后者无法减少任何假阳性。此外，输出预言机的假阳性率低于梯度预言机（即使经过过滤也是如此）。主要原因是，不同自动微分模式不应影响 API 输出，而由于底层实现不同，反向模式自动微分、正向模式自动微分和数值微分得到的计算梯度可能略有不同。

我们进一步评估超参数采样次数 N （默认值为 5）和距离 δ （默认值为 10^{-4}）的影响。受篇幅限制，本研究仅针对默认测试对象 PyTorch 和 TensorFlow 开展。为评估采样次数 N的影响，我们使用不同的 N 取值（1、2、5、10）进行实验，如表XI所示。 N 的选择对梯度预言机的假阳性率影响很小，因为这些取值下的结果都很接近。此外，随着 N 增大，假阳性率会降低，因为需要比较的邻居越多，过滤掉的假阳性就越多。然而，随着 N 增大，耗时也会增加（例如， $N = 10$的耗时约为 $N = 5$的 1.5 倍）。因此，$N = 5$ 是在假阳性率和耗时之间的折中。

关于距离的影响 δ，我们使用不同的 δ取值进行实验 10^{-1}， 10^{-2}， 10^{-3}， 10^{-4}， 10^{-5}。结果如表XII所示。首先，随着 δ 增加，假阳性率会降低。这是因为距离更远的邻居更可能具有不同的梯度，从而使一些假阳性被过滤掉。然而，这也可能排除真正的程序错误，因为较大的距离可能导致即使在可微点处梯度也发生剧烈变化。例如，在 PyTorch 中， δ =

表 X：假阳性率 (FPR)

	输出	梯度				总计
		All	Diff	精度	N/A	
PyTorch	19.3%	21.2%	25.5%	57.3%	61.9%	20.7%
TensorFlow	8.3%	21.1%	34.8%	46.4%	53.1%	16.1%
JAX	11.1%	21.0%	58.1%	68.6%	78.2%	17.3%
OneFlow	12.5%	25.0%	25.0%	64.0%	64.0%	20.0%
总计	15.0%	22.4%	37.7%	59.8%	66.7%	19.3%

表 XI：梯度预言机相对于 N的假阳性率

采样次数 N	1	2	5	10
PyTorch	23.6%	22.6%	21.2%	21.2%
TensorFlow	21.1%	21.1%	21.1%	21.1%

表 XII：梯度预言机的假阳性率（相对于 δ）

采样距离 δ	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
PyTorch	17.2%	19.3%	20.4%	21.2%	22.9%
TensorFlow	7.1%	18.8%	18.8%	21.1%	21.1%

10^{-3} 可能会过滤掉 2 个真正阳性样本。我们选择 $\delta = 10^{-4}$ 作为默认设置，因为它不会过滤掉任何真正阳性样本。

D. 有效性威胁

对内部效度的主要威胁在于 ∇Fuzz 的实现。作者对 ∇Fuzz 的代码进行了全面测试和审查，以减轻这一威胁。对外部效度的威胁主要在于所使用的评估基准。我们在四个应用最广泛且持续积极维护的深度学习库上评估了 ∇Fuzz，以验证我们方法的普适性。此外，我们采用检测到的错误、代码覆盖率、误报分析和执行时间，来降低所用指标面临的构念效度威胁。

VII. 相关工作

CRADLE [25]是深度学习库模糊测试领域的一项开创性工作，它利用差分测试，在不同的 Keras 底层深度学习库上运行现有深度学习模型来检测程序错误。 [26]AUDEE [27] 和 LEMON [28] 进一步扩展了 CRADLE：通过对现有深度学习模型应用基于搜索的变异策略，覆盖更多库代码。虽然 LEMON 采用了先进的变异规则（例如添加层），但由于其变异规则严格，例如除非模型中某个 API 的输入和输出张量形状完全相同，否则不能添加或删除该 API，因此仍只能覆盖少量 API。 [30] 更重要的是，这些技术着眼于深度学习模型的推理阶段，因此无法检测任何自动微分漏洞。为解决这一问题，近期提出的 Muffin [29] 通过自顶向下方法生成深度学习模型，以检测推理和训练阶段中的程序错误。虽然 Muffin 有望覆盖反向模式自动微分，但它只能覆盖特定库中的少量 API，且无法检测到任何已确认的自动微分漏洞（详细讨论请参见 §I ）。近期，NNSmith [64]利用轻量级形式化规约对各个算子进行建模，并通过符号约束求解生成多样且有效的模型。虽然已有研究证明，NNSmith 是最先进的模型级深度学习库模糊测试工具，但它仍只针对推理阶段；而我们的工作与之正交，并可用于进一步扩展 NNSmith。

除了利用深度学习模型测试深度学习库，研究人员还探索了直接对深度学习库 API 进行模糊测试。同时，深度学习库 API 通常以 Python 的形式提供，而 Python 是一种动态类型语言，因此甚至难以确定深度学习 API 的输入类型。为克服这一问题，Predoo [65]要求手动设置深度学习库模糊测试的 API 参数，因此仅在 7 个 TensorFlow API 上进行了评估（这是受人工工作量所限）。近期，DocTer [31] 通过合成规则，从深度学习库文档中提取 API 输入约束，然后依据这些约束生成 API 输入。不过，它仍需要人工标注 30% 的 API 参数。与上述工作不同，FreeFuzz [30] 是一种全自动的深度学习库 API 模糊测试技术。具体而言，FreeFuzz 会在运行从开源代码中挖掘出的代码时自动跟踪 API 输入；随后根据跟踪到的种子 API 输入进行变异，以生成更多输入。近期还有一类工作为深度学习库设计了其他测试预言机。例如，DeepREL [53] 会自动推断关系应用程序接口（即在给定相同输入时应返回相同值或状态的 API），并将其作为预言机来检测深度学习库中的不一致性漏洞；EAGLE [41]则利用等价图对深度学习 API 进行差分测试。尽管这些技术有效，现有应用程序接口级技术都未针对深度学习库中至关重要的自动微分引擎。

与上述模型级和 API 级模糊测试器不同，后者难以覆盖大量 API 的有效 API 序列（因为输入和形状约束复杂），最新的 LLMFuzz 工作 [66] 提出直接利用现代大语言模型（LLM） [67] 生成多样化的深度学习 API 序列。其核心洞见是，大语言模型能够从海量训练语料库中的深度学习程序隐式学习复杂的深度学习 API 约束。LLMFuzz 首次证明，现代大语言模型（例如 Codex [67]）可直接用于真实系统的端到端模糊测试。 ∇Fuzz 与 LLMFuzz 相互正交，还可进一步用于增强其预言机支持。

VIII. 结论

∇Fuzz 是首个专门针对深度学习库中自动微分引擎的方法，而自动微分引擎是任何深度学习系统的关键组件。该方法利用不同的执行场景作为测试预言机，测试一阶和高阶梯度，并引入自动化过滤器，以减少数值不稳定性导致的假阳性。在 PyTorch、TensorFlow、JAX 和 OneFlow 上对 ∇Fuzz 的评估表明， ∇Fuzz 总共检测出 173 个程序错误，其中 144 个已获开发者确认（117 个此前未知），38 个已修复。值得注意的是，在两个月内，∇Fuzz 发现了 PyTorch 和 JAX 全部高优先级自动微分缺陷的 58.3%（7/12）。

数据可用性：我们的代码和数据可在 [68] 获取。

致谢

本研究部分得到了美国国家科学基金会（NSF）项目 CCF-2131943 和 CCF-2141474 的资助。我们也感谢 Google 和 Meta 提供的支持。

参考文献

[1] T. Young、D. Hazarika、S. Poria 和 E. Cambria，“基于深度学习的自然语言处理最新趋势”，*IEEE*计算智能杂志，第 13 卷，第 3 期，第 55–75 页，2018 年。[2] A. Esteva、A. Robicquet、B. Ramsundar、V. Kuleshov、M. DePristo、K. Chou、C. Cui、G. Corrado、S. Thrun 和 J. Dean，“深度学习在医疗保健领域的指南”，《自然·医学》，第 25 卷，第 1 期，第 24–29 页，2019年。[3] J. Jumper、R. Evans、A. Pritzel、T. Green、M. Figurnov、O. Ronneberger、K. Tunyasuvunakool、R. Bates、A. ˇZ´ıdek、A. Potapenko、A. Bridgland、C. Meyer、S. A. A. Kohl、A. J. Ballard、A. Cowie、B. Romera- Paredes、S. Nikolov、R. Jain、J. Adler、T. Back、S. Petersen、D. Reiman、E. Clancy、M. Zielinski、M. Steinegger、M. Pacholska、T. Berghammer、S. Bodenstein、D. Silver、O. Vinyals、A. W. Senior、K. Kavukcuoglu、P. Kohli 和 D. Hassabis，“利用 AlphaFold 高精度预测蛋白质结构”，《自然》，第 596 卷，第 7873 期，第 583–589 页，2021 年。[4] A. Goffi、A. Gorla、M. D. Ernst 和 M. Pezz`e，“自动生成异常行为的预言机”，载于第 *25* 届国际软件测试与分析研讨会论文集，2016 年，第 213–224 页。[5] X. Gu、H. Zhang 和 S. Kim，“深度代码搜索”，载于*2018 IEEE/ACM* 第 *40* 届国际软件工程大会（*ICSE*）。IEEE，2018 年，第 933–944 页。[6] X. Li、W. Li、Y. Zhang 和 L. Zhang，“DeepFL：整合多个故障诊断维度，实现深度故障定位”，载于第 *28* 届 *ACM SIGSOFT* 国际软件测试与分析研讨会论文集，2019 年，第 169–180 页。[7] Y. Li、S. Wang 和 T. N. Nguyen，“DLFix：面向自动化程序修复的上下文代码转换学习”，载于*ACM/IEEE* 第 *42* 届国际软件工程大会论文集，2020 年，第 602–614 页。[8] C. S. Xia、Y. Wei 和 L. Zhang，“大型预训练语言模型时代的自动化程序修复”，载于 *ICSE*，2023 年，即将发表。[9] C. S. Xia 和 L. Zhang，“少训练，多修复：通过零样本学习重新审视自动化程序修复”，载于*FSE*，2022 年。[10] Garcia、Joshua、Feng、Yang、Shen、Junjie、Almanee、Sumaya、Xia、Yuan、Chen 和 Qi Alfred，“自动驾驶车辆程序错误的综合研究”，载于*ACM/IEEE* 第 *42* 届国际软件工程大会论文集，2020 年，第 385–396 页。[11] D. Shriver、S. Elbaum 和 M. Dwyer，“Artifact：简化深度神经网络属性，以便通过对抗攻击进行证伪”，载于*2021 IEEE/ACM* 第 *43* 届国际软件工程大会：配套论文集（*ICSE-Companion*）。IEEE，2021 年，第 162–163 页。[12] N. Akhtar 和 A. Mian，“计算机视觉中深度学习遭受对抗攻击的威胁：综述”，*IEEE Access*，第 6 卷，14 430，2018 年 14 410pp. – 。[13] N. Carlini、A. Athalye、N. Papernot、W. Brendel、J. Rauber、D. Tsipras、I. Goodfellow、A. Madry 和 A. Kurakin，“论对抗鲁棒性的评估”，*arXiv* 预印本 *arXiv:1902.06705*，2019 年。[14] D. Gopinath、M. Zhang、K. Wang、I. B. Kadron、C. Pasareanu 和 S. Khurshid，“用于神经网络重要性分析与对抗样本生成的符号执行”，载于*2019 IEEE* 第 *30* 届软件可靠性工程国际研讨会（*ISSRE*）。IEEE，2019 年，第 313–322 页。[15] A. Madry、A. Makelov、L. Schmidt、D. Tsipras 和 A. Vladu，“迈向能够抵御对抗攻击的深度学习模型”，*arXiv* 预印本 *arXiv:1706.06083*，2017 年。[16] S.-M. Moosavi-Dezfooli、A. Fawzi 和 P. Frossard，“DeepFool：一种欺骗深度神经网络的简单而准确的方法”，载于*IEEE*计算机视觉与模式识别大会论文集，2016 年，第 2574–2582 页。[17] N. Papernot、P. McDaniel、S. Jha、M. Fredrikson、Z. B. Celik 和 A. Swami，“对抗环境下深度学习的局限性”，载于*2016 IEEE* 欧洲安全与隐私研讨会（*EuroS&P*）。IEEE，2016 年，第 372–387 页。[18] K. Pei、Y. Cao、J. Yang 和 S. Jana，“DeepXplore：深度学习系统的自动化白盒测试”，载于第 *26* 届操作系统原理研讨会论文集，2017 年，第 1–18 页。[19] M. Yan、J. Chen、X. Zhang、L. Tan、G. Wang 和 Z. Wang，“通过梯度反向传播揭示深度学习中的数值缺陷”，载于第 *29* 届 *ACM* 欧洲软件工程大会与软件工程基础研讨会联合会议论文集，2021 年，第 627–638 页。

[20] H. Zhang、Z. Fu、G. Li、L. Ma、Z. Zhao、H. Yang、Y. Sun、Y. Liu 和 Z. Jin，“迈向深度程序处理模型的鲁棒性——检测、估计与增强”，ACM软件工程与方法学汇刊（TOSEM），第 31 卷第 3 期，第 1–40 页，2022年。[21] Z. Yang、J. Shi、J. He 和 D. Lo，“面向代码预训练模型的自然攻击”，载于《第 44 届国际软件工程大会论文集》。美国纽约州纽约市：美国计算机协会，2022年，第 1482–1493 页。 [在线]。可访问：https://doi.org/ 10 1145 3510003 3510146 。/。J. Cao、B. Chen、C. Sun、[22] L. Hu 和 X. Peng，“深度学习系统中的性能缺陷特征分析”，arXiv 预印本 arXiv:2112 01771，2021年。。[23] Y. Zhang、L. Ren、L. Chen、Y. Xiong、S.-C. Cheung 和 T. Xie，“检测神经网络架构中的数值缺陷”，载于《第 28 届 ACM 欧洲软件工程大会与软件工程基础研讨会联合会议论文集》，2020年，第 826–837 页。[24] S. Lagouvardos、J. Dolby、N. Grech、A. Antoniadis 和 Y. Smaragdakis，“TensorFlow程序中的形状静态分析”，载于《第 34 届欧洲面向对象编程大会》（ECOOP 2020）。Schloss Dagstuhl-Leibniz-Zentrum f¨ur Informatik，2020年。[25] H. V. Pham、T. Lutellier、W. Qi 和 L. Tan，“CRADLE：跨后端验证，用于检测和定位深度学习库中的缺陷”，载于《2019 IEEE/ACM 第 41 届国际软件工程大会》（ICSE），2019年，第 1027–1038 页。[26] “Keras”，https://keras.io，2015年。[27] Q. Guo、X. Xie、Y. Li、X. Zhang、Y. Liu、X. Li 和 C. Shen，“AUDEE：深度学习框架的自动化测试”，载于《2020 年第 35 届 IEEE/ACM 自动化软件工程国际会议》（ASE）。IEEE，2020年，第 486–498 页。[28] Z. Wang、M. Yan、J. Chen、S. Liu 和 D. Zhang，“通过有效模型生成测试深度学习库”，载于《第 28 届 ACM 欧洲软件工程大会与软件工程基础研讨会联合会议论文集》，2020年，第 788–799 页。[29]J. Gu、X. Luo、Y. Zhou 和 X. Wang，“Muffin：通过神经架构模糊测试测试深度学习库”，载于《2022 IEEE/ACM 第 44 届国际软件工程大会》（ICSE）。美国加利福尼亚州洛斯阿拉米托斯：IEEE计算机学会，2022年 5 月，第 1418–1430 页。[在线]。可访问：https://doi.ieeecomputersociety.org/ 10 1145 3510003 3510092 。/。A. Wei、[30] Y. Deng、C. Yang 和 L. Zhang，“测试的免费午餐：从开源项目对深度学习库进行模糊测试”，载于《2022 IEEE/ACM 第 44 届国际软件工程大会》（ICSE），2022年，第 995–1007 页。[31] D. Xie、Y. Li、M. Kim、H. V. Pham、L. Tan、X. Zhang 和 M. W. Godfrey，“DocTer：用于测试深度学习API函数的文档引导模糊测试”，载于《第 31 届 ACM SIGSOFT国际软件测试与分析研讨会论文集》，2022年，待发表。[32] B. van Merrienboer、O. Breuleux、A. Bergeron 和 P. Lamblin，“机器学习中的自动微分：现状与未来方向”，载于 NeurIPS，2018年。[33] 维基百科贡献者，“反向传播——维基百科，自由的百科全书”，https://en.wikipedia.org/w/index.php?title=Backpropagation& oldid =1104872812，2022年。[34] S. Zhang、S. Roller、N. Goyal、M. Artetxe、M. Chen、S. Chen、C. Dewan、M. Diab、X. Li、X. V. Lin 等，“OPT：开放式预训练Transformer语言模型”，arXiv 预印本 arXiv:2205.01068，2022年。[35] P. Gasti、G. Tsudik、E. Uzun 和 L. Zhang，“命名数据网络中的 DoS 和分布式拒绝服务攻击”，载于《2013 年第 22 届国际计算机通信与网络大会》（ICCCN），2013年，第 1–7 页。[36] “PyTorch官方文档中的 Xlogy 定义”，https:// pytorch.org/docs/stable/generated/torch.nn.KLDivLoss.html，2022年。[37] “Keras 2.3.0 发布”，https://github.com/keras-team/keras/releases/tag/2. 3.0，2019年。[38] “FreeFuzz代码仓库”，https://github.com/ise-uiuc/FreeFuzz，2022年。[39] A. G. Baydin、B. A. Pearlmutter、A. Radul 和 J. M. Siskind，“机器学习中的自动微分：综述”，《机器学习研究期刊》，第 18 卷，第 153:1–153:43 页，2017年。[40] 维基百科贡献者，“海森矩阵——维基百科，自由的百科全书”，https://en.wikipedia.org/w/index.php?title=Hessian matrix& oldid=1107031412，2022年。

[41] J. Wang、T. Lutellier、S. Qian、H. V. Pham 和 L. Tan，“EAGLE：创建等价图以测试深度学习库”，2022年。[42] 维基百科贡献者，“雅可比矩阵与行列式——维基百科，自由的百科全书”，https://en.wikipedia.org/w/index.php?title=Jacobian matrix and determinant&oldid=1104898576，2022年。[43] 维基百科贡献者，“切空间——维基百科，自由的百科学全书”，https://en.wikipedia.org/w/index.php?title=Tangent space& oldid =1091055882，2022年。[44] R. L. Burden、J. D. Faires 和 A. M. Burden，《数值分析》。Cengage Learning，2015年。[45] “JAX官方文档中的 dim 中的动态索引定义”，https://jax.readthedocs.io/en/latest/ autosummary/jax.lax.dynamic index in dim.html，2022年。[46] “PyTorch官方文档中的迹定义”，https:// pytorch.org/docs/stable/generated/torch.trace.html，2022年。[47] “PyTorch官方文档中的硬收缩定义”，https: // pytorch.org/docs/stable/generated/torch.nn.Hardshrink.html，2022年。[48] “JAX官方文档中的幂运算定义”，https:// jax. readthedocs.io/en/latest/ autosummary/jax.lax.pow.html，2022年。[49] 维基百科贡献者，“可微函数——维基百科，自由的百科全书”，https://en.wikipedia.org/w/index.php?title=Differentiable function&oldid=1101867284，2022年。[50] “PyTorch官方文档中的求和定义”，https:// pytorch.org/docs/stable/generated/torch.sum.html，2022年。[51] J. Bradbury、R. Frostig、P. Hawkins、M. J. Johnson、C. Leary、D. Maclaurin、G. Necula、A. Paszke、J. VanderPlas、S. Wanderman-Milne 和 Q. Zhang，“JAX：可组合变换的 Python+NumPy程序”，http://github.com/google/jax，2018年。[52] J. Yuan、X. Li、C. Cheng、J. Liu、R. Guo、S. Cai、C. Yao、F. Yang、X. Yi、C. Wu、H. Zhang 和 J. Zhao，“OneFlow：从头重新设计分布式深度学习框架”，2021年。[53] Y. Deng、C. Yang、A. Wei 和 L. Zhang，“通过自动化关系型 API 推断对深度学习库进行模糊测试”，载于《第 30 届 ACM 欧洲软件工程大会与软件工程基础研讨会联合会议》（ESEC/FSE），2022年。[54] T. Su、Y. Yan、J. Wang、J. Sun、Y. Xiong、G. Pu、K. Wang 和 Z. Su，“用于检测不崩溃逻辑错误的 Android 应用全自动功能模糊测试”，《ACM编程语言汇刊》，第 5 卷，OOPSLA 期，第 1–31 页，2021年。[55] W. You、X. Wang、S. Ma、J. Huang、X. Zhang、X. Wang 和 B. Liang，“Profuzzer：即时输入类型探测，以更好地发现零日漏洞性漏洞发现”，载于《2019年 IEEE安全与隐私研讨会（SP）》。IEEE，2019年，第769–786页。[56] A. F. Donaldson、H. Evrard、A. Lascu 和 P. Thomson，《图形着色器编译器的自动化测试》，《美国计算机协会编程语言汇刊》，第1卷，OOPSLA期，第1–29页，2017年。[57] J. Liu、Y. Wei、S. Yang、Y. Deng 和 L. Zhang，《通过联合IR通道变异进行覆盖率引导的张量编译器模糊测试》，《ACM编程语言汇刊》，第6卷，OOPSLA1期，2022年4月。 [在线]。网址：https://doi.org/10 1145 3527317 。/ “GCOV”，[58] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html，2022年。[59] “Coverage.py”，https://github.com/nedbat/coveragepy，2022年。[60] “PyTorch官方文档中RReLU的定义”，https: // pytorch.org/docs/stable/generated/torch.nn.RReLU.html，2022年。[61] “JAX官方文档中Sinc函数的定义”，https:// jax. readthedocs.io/en/latest/ autosummary/jax.numpy.sinc.html，2022年。[62] 维基百科贡献者，《Bfloat16浮点格式——维基百科，自由的百科全书》，https://en.wikipedia.org/w/index.php?title= Bfloat16 floating-point format&oldid=1041556217，2021年。[63] “PyTorch官方资料中的Autograd”，https://github.com/pytorch/ pytorch/blob/master/torch/csrc/autograd/README.md，2022年。[64] J. Liu、J. Lin、F. Ruffy、C. Tan、J. Li、A. Panda 和 L. Zhang，《NNSmith：为深度学习编译器生成多样且有效的测试用例》，载于《第28届ACM编程语言与操作系统架构支持国际会议论文集》，第2卷，ASPLOS 2023系列。美国纽约州纽约：美国计算机协会，2023年，第530–543页。 [在线]。网址：https://doi.org/10 1145 3575693 3575707 。/ X. Zhang、N. Sun、C. Fang、J. Liu、J. Liu、D. Chai、J. Wang 和 Z. Chen，[65] 《Predoo：深度学习算子的精度测试》，载于《第30届ACM SIGSOFT国际软件测试与分析研讨会论文集》，2021年，第400–412页。[66] Y. Deng、C. S. Xia、H. Peng、C. Yang 和 L. Zhang，《利用大语言模型对深度学习库进行模糊测试》，载于《第32届ACM SIGSOFT国际软件测试与分析研讨会论文集》，ISSTA 2023系列，2023年。[67] M. Chen、J. Tworek、H. Jun、Q. Yuan、H. P. d. O. Pinto、J. Kaplan、H. Edwards、Y. Burda、N. Joseph、G. Brockman等，《评估基于代码训练的大语言模型》，arXiv预印本，arXiv:2107 03374，2021年。。[68] “∇Fuzz仓库”，https://github.com/ise-uiuc/NablaFuzz，2022年。