基于元算子的深度学习框架缺陷检测方法

谷典典[1] 石屹宁[1] 刘譞哲[1] 吴 格[2),3)] 姜海鸥[4] 赵耀帅[2),3)] 马 郓[1),5)]

[1)](高可信软件技术教育部重点实验室(北京大学) 北京 100871)

[2)](中国民航信息网络股份有限公司 北京 101318)

[3)](中国民用航空局民航旅客服务智能化应用技术重点实验室 北京 101318)

[4)](北京大学(天津滨海)新一代信息技术研究院 天津 300452)

[5)](北京大学人工智能研究院 北京 100871)

摘 要 在用于构建深度学习模型的深度学习框架中,算子的正确计算对于深度学习模型的正确预测至关重要.然而,已有的深度学习框架缺陷检测方法只能通过比较和推测的方式找到不同深度学习框架之间计算结果相差较大的算子,而且无法检测深度学习模型在训练过程中产生的计算错误,具有很大的局限性.针对此问题,本文设计并实现了基于元算子的深度学习框架缺陷检测方法,通过将不同深度学习框架中算子的共性计算逻辑抽象为"元算子",支持在不改变模型代码的前提下绑定元算子的具体实现,从而可以细粒度地对比同一模型使用不同深度学习框架的运算结果,进而发现缺陷.本文的方法同时支持训练过程和推断过程的缺陷检测,还可以对计算错误的定位进行验证.本文验证了元算子计算的准确性,并评估其运算性能;收集了深度学习框架中已知有错误计算的算子,并将本文方法应用在包含这些算子的深度学习模型上,验证了本文缺陷检测方法的有效性.

关键词 深度学习框架;元算子;缺陷检测;深度学习;软件测试

中图法分类号 TP391 **DOI 号** 10.11897/SP.J.1016.2022.00240

Defect Detection for Deep Learning Frameworks Based on Meta Operators

GU Dian-Dian[1] SHI Yi-Ning[1] LIU Xuan-Zhe[1] WU Ge[2),3)] JIANG Hai-Ou[4]
ZHAO Yao-Shuai[2),3)] MA Yun[1),5)]

[1)](*Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing* 100871)

[2)](*TravelSky Technology Limited, Beijing* 101318)

[3)](*Key Laboratory of Intelligent Passenger Service of Civil Aviation, CAAC, Beijing* 101318)

[4)](*Peking University Information Technology Institute (Tianjin Binhai), Tianjin* 300452)

[5)](*Institute for Artificial Intelligence, Peking University, Beijing* 100871)

Abstract Deep Learning (DL) has been widely adopted in various fields such as image recognition, machine translation, and autonomous driving. In order to better support deep learning tasks and promote the application of DL, more and more platforms and frameworks have emerged, such as TensorFlow, PyTorch, and Keras. These platforms and frameworks are known as deep learning frameworks. Using the programming interfaces provided by these deep learning frameworks, developers can easily design, train, and test the deep learning models. Deep learning frameworks usually take "operator" as the unit of calculation, and different operators define different types of numerical calculation. In deep learning frameworks, the correct calculation of operators is critical

收稿日期:2020-09-01;在线发布日期:2021-04-13.本课题得到国家重点研发计划(2018YFB1004400)、北京高等学校卓越青年科学家项目(BJJWZYJH01201910001004)资助.**谷典典**,博士研究生,中国计算机学会(CCF)会员,主要研究方向为系统软件、机器学习系统.E-mail:gudiandian1998@pku.edu.cn.**石屹宁**,硕士研究生,主要研究方向为深度学习、高性能计算.**刘譞哲**,博士,副教授,中国计算机学会(CCF)杰出会员,主要研究方向为服务计算、系统软件.**吴 格**,本科,主要研究方向为大数据及人工智能.**姜海鸥**,博士,助理研究员,中国计算机学会(CCF)会员,主要研究方向为云计算、大数据、机器学习.**赵耀帅**(通信作者),硕士,高级软件工程师,主要研究方向为大数据及人工智能.E-mail:yszhao@travelsky.com.**马 郓**(通信作者),博士,助理教授,中国计算机学会(CCF)会员,主要研究方向为系统软件、Web 计算.E-mail:mayun@pku.edu.cn.

to the correctness of deep learning models. These calculation errors could affect the accuracy of the prediction results of the deep learning models, or even result in serious consequences such as traffic accidents in automatic driving. In recent years, attention has been paid on testing and diagnosis of deep learning frameworks, but existing defect detection methods have great limitations. On the one hand, existing defect detection methods for deep learning frameworks can detect only large calculation differences of operators between different deep learning frameworks through comparison and speculation. On the other hand, existing methods can diagnose only calculation errors of deep learning models in the inference process, and cannot diagnose calculation errors in the training process. To address the issue, we expect to detect errors of deep learning models due to the defects of deep learning frameworks automatically in the process of training or inference and verify the accuracy of detection results. There are many challenges in implementing such a defect detection method. First, the deep learning model usually consists of a complex network structure. For a deep learning model, given any input instance, it is very difficult to determine the correct output. Second, a deep learning model usually consists a large number of operators and their relationship in the model is very complex, making locating defective operators difficult. In addition, verifying the correctness of defect location in a large and complicated deep learning model is challenging. In response to the above challenges, in this paper, we design and implement a defect detection method for deep learning frameworks based on meta operators. We abstract common computing logic of operators such as forward computation and gradient computation of operators in different deep learning frameworks as "meta operators". We bind the specific implementation of operators without changing the code of deep learning models. In this way, users can make fine-grained replacements of operators in deep learning models. Through fine-grained operator replacement, not only can the calculation errors of the deep learning frameworks during the inference process be found, but also the calculation errors during the training process and the localization of these errors can be verified by recording the meta operator's running time and memory consumption. We verify the accuracy of the meta operator calculation and evaluate its performance. We collect the known operators with calculation errors in the deep learning frameworks and apply the defect detection method on deep learning models containing these operators, showing the effectiveness of the defect detection method.

Keywords　deep learning frameworks; meta operator; defect detection; deep learning; software testing

1　引　言

深度学习(Deep Learning)在许多领域都有广泛的应用，例如图像识别[1]、语音识别[2]、机器翻译[3]、自动驾驶[4]等等. 为了更好地实现深度学习任务、推动深度学习的发展，近年来涌现出越来越多的用于构建深度学习模型的框架和平台，例如TensorFlow[5]、PyTorch[6]和Keras[7]等. 使用深度学习框架提供的编程接口，开发者可以轻松地设计、训练以及测试深度学习模型. 深度学习框架通常以"算子"作为计算的单位，不同的算子定义了不同类型的数值计算. 然而，各个深度学习框架在这些算子计算的实现上可能会存在错误，这些计算错误会影响深度学习模型预测结果的准确性，甚至会因此产生严重的后果，例如：Uber 深度学习框架中的一处错误曾导致一辆自动驾驶汽车撞人致死[8]. 因此，对深度学习框架的缺陷检测工作十分重要.

深度学习框架主要存在三类计算错误：一是因为机器表示而产生的错误，由于浮点数能表示的精度有限，这类计算错误无论在软件中还是在深度学习框架中都十分常见；二是由于数学估算产生的错误，在深度学习框架中，一些特定的函数计算是通过估算或近似算法实现的，比如三角函数，如果估算或

近似算法不够准确，就会产生计算错误；三是深度学习框架的实现有误，这一类错误往往是深度学习框架开发者所实现的算子存在错误[9]. 本文关注第三类错误的检测. Jia 等人[10]对 TensorFlow 框架中的实现错误进行了实证研究，发现在 TensorFlow 框架本身的缺陷中，算法或接口实现上的错误最为常见，占全部缺陷的 38.21%.

目前对深度学习框架进行测试和缺陷检测的相关工作仍然较少. 其中，CRADLE 是一个自动诊断并定位深度学习框架中的错误的工具[11]. 给定一个深度学习模型，CRADLE 使用距离度量比较模型在不同深度学习框架作为后端上的输出，以此检测是否存在不一致的计算结果，并且通过跟踪异常数据传播来确定在模型中产生这种不一致的位置. 然而，目前以 CRADLE 为代表的深度学习框架缺陷检测方法存在以下的局限性：这些方法只能通过比较和推测的方式找到不同深度学习框架之间计算结果相差较大的算子，无法验证检测结果的准确性. 此外，现有方法只能检测深度学习框架在执行模型推断过程中的计算错误，无法检测训练过程中产生的计算错误.

针对此问题，本文期望实现自动化地检测深度学习模型由于深度学习框架缺陷而导致的错误，并支持在定位计算错误的算子之后验证定位的准确性. 然而，实现这样的缺陷检测方法面临许多挑战. 首先，基于深度学习框架实现的深度学习模型通常包含复杂的网络结构. 对于一个深度学习模型，给定任意一个输入实例，确定该网络正确的输出结果是十分困难的. 例如，对于一个分类模型而言，模型的输出结果通常不是一个简单的类别，而是当前的输入数据属于不同类别的概率，将这样的输出结果与输入数据的标签相比较无法验证模型计算的准确性. 在模型的训练阶段，我们同样很难确定每一个算子的梯度的正确数值并验证每一个算子的梯度计算的正确性. 其次，深度学习模型中通常包含大量的、多样化的算子. 深度学习模型中算子的数量以及模型本身的复杂性大大提高了错误算子定位的难度. 另外，同样由于深度学习模型的复杂性，在定位了某一计算错误的算子之后，如何在包含大量复杂算子的深度学习模型中验证错误定位的准确性也是本文工作中的一个难点.

针对上述挑战，本文设计了一个基于元算子的深度学习框架缺陷检测方法. 在该方法中，不同深度学习框架中算子的正向计算、梯度计算等共性计算逻辑被抽象为“元算子”，用户无需修改模型底层代码即可改变指定算子的底层实现. 本文的主要贡献包括：

(1) 提出“元算子”的概念，将不同深度学习框架中算子的共性计算逻辑抽象为“元算子”，支持在不改变模型代码的前提下绑定元算子的具体实现，从而高效实现算子的细粒度替换.

(2) 设计并实现了基于元算子的深度学习框架缺陷检测方法. 通过算子的细粒度替换，可以对比同一模型使用不同深度学习框架的运算结果，实现深度学习框架中计算错误的检测、定位以及对错误定位的检验.

(3) 对本文提出的方法进行可行性评估和性能评估，并通过实例分析对本文方法的有效性进行了验证. 实验结果显示，虽然元算子为实现缺陷检测方法的功能牺牲了一定的性能，但是其计算结果可以准确反映深度学习框架中算子的计算结果，且本文工具可以有效地检测和定位不同深度学习框架中算子计算的差异.

本文第 2 节介绍与本文有关的研究工作；第 3 节介绍本文检测方法的详细设计；第 4 节介绍本文检测方法的实现细节；第 5 节对本文提出的方法进行实验评估；第 6 节总结全文.

2 相关工作

本节将对本文的相关工作进行介绍，包括软件调试、深度学习模型测试和深度学习框架测试. 其中，深度学习框架测试与本文工作关系最紧密. 深度学习框架测试方面的相关工作较少，但在近年来逐渐受到关注.

2.1 软件调试

软件调试指的是寻找软件出错的原因并修复错误的过程，包括发现错误、定位错误、在错误与代码的实现逻辑相关联、修改相关代码片段、进行回归测试等步骤. 其中，软件错误定位是软件调试过程中最复杂、最困难的步骤之一.

软件调试是保障软件正确运行的重要环节，因此长期以来工业界对软件调试有很高的需求[12]. 其中一种常用的方法是通过将程序代码划分为与数据流相关的语句集进行调试这种方法被称之为切片(slicing)[13-14]. 同一切片中的语句不一定在程序代码中是连续的，而可以是分散的. 切片可分为静态切片和动态切片两种类型，每一种切片类型都有许多

不同的算法.

另一种常用的软件调试方法是基于模型的方法.在这种方法中,源程序被自动转换成逻辑表示形式,称之为模型(model).给定一个模型和特定的测试用例,基于模型的软件调试方法通过搜索算法确定出最小语句集,这些语句的错误可以解释程序执行中的错误.Abreu 等人[15]设计了低复杂性的近似最小碰集算法,可用于基于模型的软件调试.Wotawa 等人[16]在原有的基于模型的软件调试方法上进行创新,将源程序的模型转变成为无循环的形式,从而提供了有效的搜索和诊断能力.

除此之外,还有基于频谱(spectrum-based)的方法[17]和基于突变(mutation-based)的方法等也被应用于软件调试[18-19].其中,本文思路与基于突变的软件调试方法类似:传统的软件调试中,基于突变的方法对程序语句进行修改从而进行调试,而本文则是对算子的底层实现方式进行修改来进行缺陷检测.

2.2　深度学习应用错误的经验研究

为了更好地理解深度学习应用的缺陷,近年来涌现出越来越多的对这些缺陷进行了实证研究.Zhang 等人[20]进行了一项实证研究以了解 TensorFlow 应用中的缺陷.Islam 等人[21]以及 Humbatova 等人[22]更深入地分析了多种使用了深度学习框架的深度学习应用中的缺陷.与这一类工作不同的是,本文主要关注于深度学习框架本身所存在的缺陷,而非深度学习应用开发者在实现深度学习应用时带来的错误.

2.3　深度学习模型测试

近些年来,随着深度学习的快速发展,为了测试深度学习模型的正确性,深度学习模型的测试方面也有了越来越多的工作[23].人们提出了自动化测试技术以改进深度学习模型的测试.Pei 等人[24]提出了对深度学习模型的第一个白盒测试 DeepXplore,并引入了神经元覆盖率指标,以指导白盒模糊测试.DLFuzz[25]将模糊测试引入图像分类任务深度学习模型测试,在 DeepXplore 的基础上进一步提高了神经元覆盖率.Tian 等人[26]提出了一种用于 DNN 驱动的自动驾驶汽车的自动化测试的工具 DeepTest,可以使用通过不同的变换生成的测试图像来最大化深度神经网络的神经元覆盖范围,并利用特定于域的变态关系来发现深度神经网络的错误行为.

对深度学习模型的测试方法也有不断的创新.生成对抗性样本(adversarial example)测试深度学习模型的方法较为常见,例如 Papernot 等人[27]的工作专门为 RNN 生成对抗输入序列,以供测试.Srisakaokul 等人[28]提出了一种针对有监督的学习软件的多重实施测试的方法:k-最近邻居和朴素贝叶斯.此外,Ma 等人[29]提出使用突变测试来评估对深度学习模型的测试的有效性.

针对深度学习模型,人们也提出了新的衡量指标.近期,Ma 等人[29]提出的 Deepgauge 和 Du 等人[30]提出的 Deepstellar,为深度学习模型的测试提供了更高级的覆盖率指标,很好地改善现有的对深度学习模型的测试.

2.4　深度学习框架测试

深度学习框架的自动化测试在近些年来逐渐受到关注.例如,Nejadgholi 等人[9]从结果参照物近似(oracle approximation)的具体实践的角度,研究了深度学习框架中已有的各个级别的、对数值计算的测试.

对深度学习框架的测试是软件工程领域中深度学习框架开发的一个步骤,通常是在有已知的条件和可预知的结果的情况下检测系统的实现是否符合预期.而我们做的缺陷检测是推理出整个系统的问题出在哪里的过程.而与对深度学习框架的测试不同,本文致力于对深度学习框架的缺陷检测工作,即对深度学习框架中各环节的计算进行排查,从而推理出整个系统中隐藏的计算缺陷.在对深度学习框架中算子的单元测试方法已经很成熟的情况下,在深度学习框架中仍然存在缺陷.因此,在单元测试的基础上对深度学习框架进行缺陷检测是十分必要的.Srisakaokul 等人[28]比较同一机器学习算法用不同方法实现后的计算结果,并使用这样的方法对三种深度学习框架进行了诊断.然而这种方法需要假设对每种算法的多数实现方式是正确无误的,且该方法无法定位错误.Pham 等人[11]提出 CRADLE,通过不同深度学习框架之间计算结果一致性的检查,检测深度学习框架中的错误,并通过跟踪异常数据传播定位错误,解决了深度学习框架中错误算子定位的难题.Wang 等人[31]基于 CRADLE 的工作,提出了基于突变的生成有效测试模型的方法 LEMON,可用于对深度学习框架的测试.

3　方法设计

本节主要介绍基于元算子的深度学习框架缺陷检测方法的设计方案与设计细节,包括整体流程和

流程中每一步骤的设计.

3.1 总体流程

本文方法借助用户搭建的深度学习模型,通过对该模型中的算子在不同框架下计算结果进行逐一比对,来检测深度学习框架中算子的计算错误,具体流程如图 1 所示.

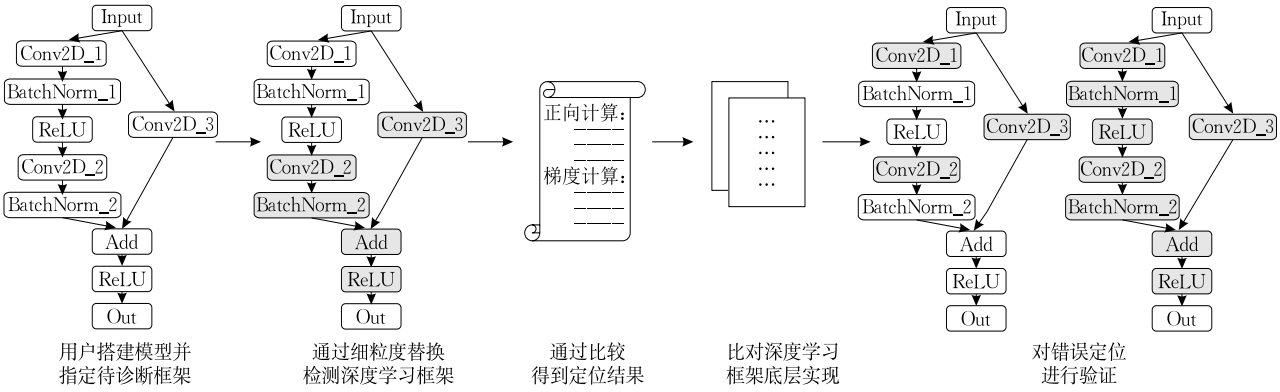

图 1 深度学习框架缺陷检测方法总体流程

方法主要包括 5 个步骤. 第 1 步,用户使用本文检测工具提供的接口搭建深度学习模型,并指定两个不同的深度学习框架用于比较计算结果. 本文方法通过比较这两种深度学习框架的计算结果是否存在较大差异,来判断是否存在有实现错误的算子;第 2 步,通过算子细粒度替换的方式,对模型中的每个算子逐一检测,并记录作出每一次算子替换之后模型的推断结果和模型参数梯度;第 3 步,将每次改变算子底层实现前后模型计算结果的平均绝对离差(Mean Absolute Distance)与阈值比较并排序,找出可能存在计算错误的算子,并实现对错误算子的定位;第 4 步,用户对比定位得到的可能存在错误的算子在两种深度学习框架中的底层实现,查看这些算子在深度学习框架中是否有实现上的错误;第 5 步,对错误定位结果的验证.

3.2 深度学习框架缺陷检测方法设计

本文提出了“元算子”的概念. 在不同的深度学习框架中,存在大量“语义”上相同的算子,例如不同的框架中都有进行加法、减法、卷积等计算的算子. 这些算子的计算由不同的框架实现,但是这些算子所做的计算在数学上是等价的是相同的. 元算子是本文检测方法中进行计算的单位,每一种元算子抽取了不同深度学习框架中语义相同的算子中的共同特征,实现了不同深度学习框架中算子共有的属性和方法,既可以完成深度学习框架中算子的正向推断计算,也可以完成梯度计算. 每一个元算子的计算都采用已有深度学习框架的算子计算方法实现,用户可以指定或修改每一个元算子的计算由哪一种深度学习框架的计算方法来实现.

为了使用本文缺陷检测方法,用户首先需要使用本文检测工具的元算子构建一个机器学习模型作为待测对象,并指定两种深度学习框架 A、B. 该模型的在缺陷检测过程中,通过比较这两种深度学习框架中算子的计算是否有较大差别,来判断这些深度学习框架中是否可能存在计算错误. 根据所搭建的模型的类型,本文方法的输入数据可以是图像数据、文本序列或其他类型的数据. 输入数据的取值可以是人工输入的定值、任意随机赋值的 tensor,也可以由对深度学习模型的模糊测试、对抗测试中采用的取值方法生成. 同样,模型中的参数(weights 和 bias)也可以通过加载在不同框架中训练好的模型参数或自由赋值等方式来取值. 采用不同的输入数据和参数的取值策略时,检测的效率不同. 已有的深度学习测试工作对测试时的取值策略已有很多的探讨,我们将这些取值策略作为本文工具的外部构件,而非本文关注的重点. 在完成了对模型的搭建以及对模型中参数的赋值后,可以得到一个确定的机器学习模型,本文工具可以自动实现模型中元算子实现方式的计算,用户也可以通过工具提供的接口改变任意元算子的底层实现方式.

算法 1 展示了本文方法检测过程中记录算子计算结果过程的伪代码. 在检测过程中,首先用深度学习框架 A 实现模型中所有算子(第 2 行~第 4 行),然后按照模型计算图拓扑排序的反序,从底向上的顺序逐个、自动化地将算子的底层实现替换成 B(第 9 行~第 15 行). 这样的替换顺序可以保证每一次替换算子底层实现的前后,该算子的输入数据是相同的. 每次替换算子的底层实现框架之后,该检测方法会以用户指定的数据作为输入,自动对模型运行正向计算和反向梯度传播,并记录作出这一次算子替换之后新的推断结果和模型参数的梯度(第 11 行~第 14 行). 这样的记录会作为后续错误定位的依据.

算法 1. 算子计算结果记录.

输入：深度学习模型 *model*

模型输入数据 *feed_dict*

待检测深度学习框架 *frameworks*

输出：模型推断结果集合 *output_values*

参数梯度计算结果集合 *grad_tables*

```
1. FUNCTION run(model, feed_dict, frameworks)
2. FOREACH op IN model.ops() DO
       //所有元算子都由第一个深度学习框架实现
3.     op.framework←frameworks[0]
4. END
   //计算模型正向推断结果
5. output←model.compute_outputs()
   //计算模型中每个参数的梯度
6. grad_table←model.compute_gradients()
7. output_values.add(output)
8. grad_tables.add(grad_table)
9. FOREACH op IN reverse(model.ops) DO
       //将元算子的实现替换为第二个深度学习框架
10.    op.framework←frameworks[1]
       //计算模型正向推断结果
11.    output←model.compute_outputs()
12.    //计算模型中每个参数的梯度
13.    grad_table←model.compute_gradients()
14.    output_values.add(output)
15.    grad_tables.add(grad_table)
16. END
17. RETURN output_values, grad_tables
```

3.3　深度学习框架错误定位方法设计

在使用本文检测工具的元算子构建一个深度学习模型，并指定两种深度学习框架后，由算法 1 的算法可以得到一组推断结果集合和梯度计算结果集合. 集合中相邻的两次计算结果为替换了某元算子的底层实现后，以相同的输入数据作为输入得到的两次计算结果. 因此，分别比较两个集合中两两相邻的计算结果即可推断出某算子在不同深度学习框架中的实现是否有明显差异.

无论是分类任务还是回归模型，平均绝对离差(Mean Absolute Distance)都可以用来衡量深度学习模型计算的结果之差. 因此，参考该领域已有的工作[11,31]，本文工具采用平均绝对离差衡量模型每次计算的结果之差. 对于正向推断过程来说，若在替换元算子的底层实现 O 之前模型的正向推断结果为 Y_O，将元算子的底层实现 O 替换为 O' 之后的正向推断结果为的为 $Y_{O'}$，两个正向推断结果向量均含有 n 个元素，则两个结果之差可被计算为

$$\delta_{C,O,O'}=\frac{1}{n}\sum_{i=1}^{n}|Y_{O,i}-Y_{O',i}| \tag{1}$$

若将某元算子的底层实现 O 替换为 O' 后，模型计算的结果之差 $\delta_{C,O,O'}$ 大于用户指定的阈值 T_C，则两种深度学习框架中的实现方式 O 或 O' 之一可能存在计算错误，从而实现了对模型中存在计算错误的算子的定位.

对于梯度计算结果的集合来说，集合中每一个参数在相邻两次计算中的梯度之差同样可用平均绝对离差衡量. 在本文检测方法中，替换一次元算子的底层实现之后，首先计算得到每一个梯度在该次替换前后的梯度之差，然后计算深度学习模型中所有参数梯度之差的平均值，用来衡量这次替换对参数梯度的整体影响大小. 若深度学习模型中有 m 个可求梯度的参数，则在将某元算子的底层实现从 O 替换为 O' 前后的整体梯度之差为

$$\delta_{G,O,O'}=\frac{1}{m}\sum_{i=1}^{m}\frac{1}{n_i}\sum_{j=1}^{n_i}|G_{O,j}-G_{O',j}| \tag{2}$$

同样，若将某元算子的底层实现 O 替换为 O' 后，模型的整体梯度之差 $\delta_{G,O,O'}$ 大于用户指定的阈值 T_G，则两种深度学习框架中的实现方式 O 或 O' 之一的梯度计算方面可能存在错误，从而实现了对模型中梯度计算错误的算子的定位.

本文检测方法会将某次元算子实现替换前后正向推断结果之差和整体梯度之差分别从大至小排序，排名越靠前的那次替换涉及到的算子越有可能存在错误. 之后，对于排名较高且计算结果之差 $\delta_{C,O,O'}$ 或整体梯度之差 $\delta_{G,O,O'}$ 明显大于其他算子的算子，用户可手动对比算子两个排序中在两种深度学习框架中的底层实现代码，进行进一步确认.

3.4　错误定位的验证方法设计

若用户通过本文检测方法得到的错误定位结果中发现某算子在其中一种深度学习框架中的实现有错误，本文检测方法支持对该定位结果进行验证. 采用的验证方法为：将模型分别用两种深度学习框架实现，然后将其中有错误的实现方法替换为另一种深度学习框架，即正确的实现方法. 比较两次模型的计算结果. 若两次计算结果相近，则对算子计算错误的定位准确.

例如，在如图 2 所示是一个深度学习模型的计算图，每个节点代表一个元算子，节点的不同颜色代表使用不同的深度学习框架作为元算子的底层实现. 若本文方法在错误定位阶段给出的结果表明模型中 Conv2D 元算子的两种实现方法会导致最终结

果有较大差异.经过对两种深度学习框架代码的人工比对,确认白色代表的深度学习框架的 Conv2D 算子的实现方法存在错误,而灰色代表深度学习框架的 Conv2D 算子正确.则用户可使用本文工具按照图 2 先后计算两次模型的计算结果.若两次计算结果之差在阈值范围内,则说明用两种深度学习框架实现该模型计算结果差异较大,主要是由 Conv2D 算子的计算导致的.如果对于模型给定的输入数据,用户恰好有相应的预期输出结果,那么此时用户可以比较两个模型的输出结果是否符合预期,并以此验证人工比对后的判断结果.

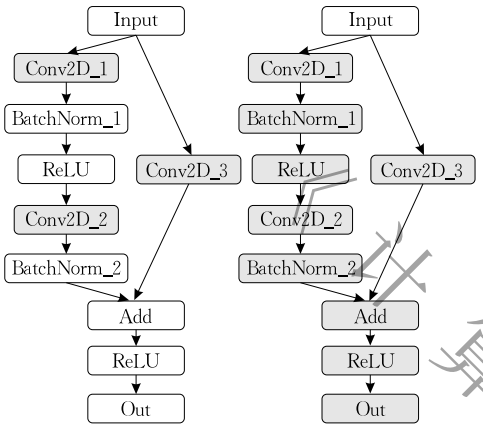

图 2 缺陷检测方法对错误定位的验证

4 方法实现

本节首先介绍缺陷检测工具的整体架构,然后介绍算子细粒度替换实现细节,最后分别介绍检测工具架构中每一组成部分的实现细节.

4.1 深度学习框架缺陷检测工具整体架构

本文缺陷检测工具的设计分为六层,其系统架构图示意如图 3 所示.

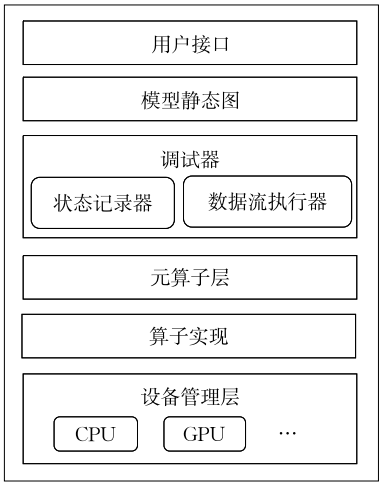

图 3 深度学习框架缺陷检测工具系统架构图

本文检测工具系统架构的最顶层为用户 Python 接口,其负责为用户提供搭建深度学习模型并检测模型中的算子的编程接口.模型静态图层根据用户的代码,构建模型静态计算图的结构.调试器由状态记录器和数据流执行器两部分组成,其中,状态记录器负责记录每次算子替换后计算图执行的计算结果;数据流执行器根据静态计算图的结构,执行正向计算和反向梯度计算操作.元算子层实现了本文检测工具执行各类型计算的基本单位.算子实现层由不同的深度学习框架实现,包括数值计算和多维数组操作等等.最底层的设备层包括深度学习框架分别在 CPU、GPU 等设备上的实现,对上层提供了一个统一的接口,使上层只需要算子计算等逻辑,而不需要关心在硬件上具体的计算的实现过程.其中,算子实现层和设备层在各个深度学习框架已经得到实现,在使用本文深度学习框架缺陷检测方法时,只需要调用由深度学习框架提供的接口即可,无需再次实现.

4.2 元算子

深度学习框架以“算子”(operator)为计算的单元.算子通过运行深度学习框架中的算子底层实现代码,来完成数值计算或者多维数组操作等不同的操作.训练和使用深度学习模型需要模型中的每一个算子进行正向计算以得到推断结果,或进行梯度计算以更新模型权重.

无论是由哪一种深度学习框架实现的算子,除了逻辑判断等类型算子以外,大多都具有正向计算和梯度计算这两种计算逻辑.考虑到这两种共性计算逻辑,本文将不同深度学习框架中语义上相同的算子抽象为同一个元算子.元算子中实现了不同深度学习框架中算子共有的属性和方法,也提供了调用不同深度学习框架中算子计算实现的接口.用户仅通过参数指定该算子的底层实现框架,元算子类通过接口绑定该算子计算时采用的底层框架实现,从而实现了算子底层实现的细粒度替换.本文检测方法以元算子作为计算的单元.

以加法计算为例的加法元算子类的类图如图 4 所示.包括加法在内的元算子均继承自元算子基类.在元算子基类中,定义了各类元算子都拥有的属性和方法.*framework* 属性表示实现算子底层计算的深度学习框架.*name* 属性为算子的名字,以区分模型中不同的算子.*input_nodes* 为一个节点列表,列表中的节点为在深度学习模型整体的静态图中该算子所在节点的输入节点.同理,*output_nodes* 也是一个节点列表,列表中的节点均以该算子所在节点为

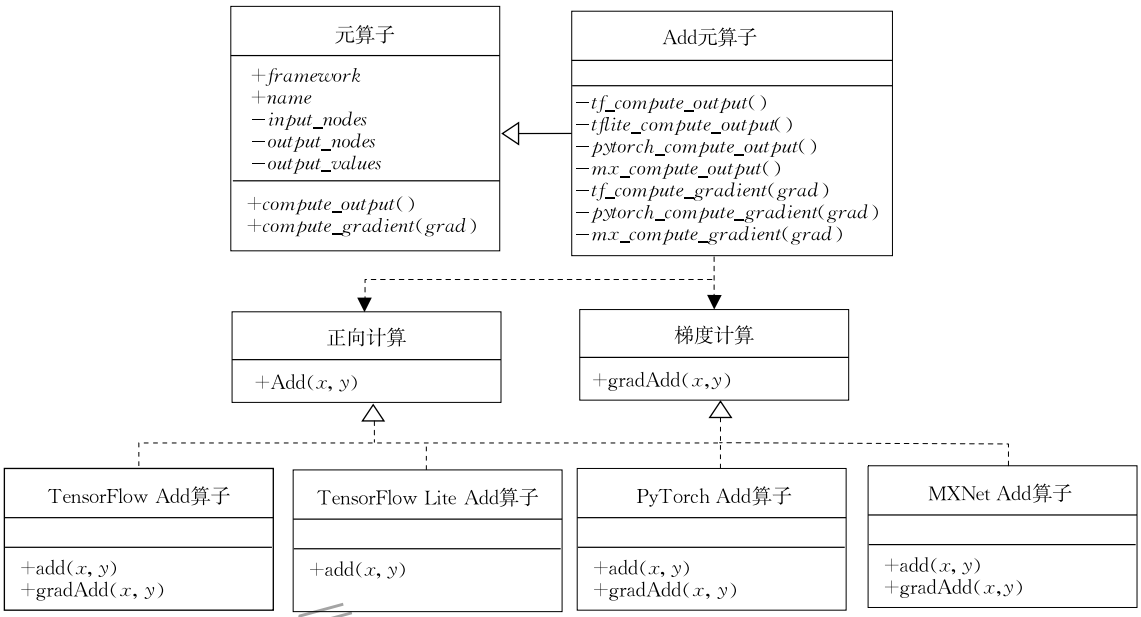

图 4　加法元算子类类图

输入节点. *output_values* 为该节点的正向计算结果，在该算子进行过第一次正向计算后才会被赋值. *compute_output*()方法进行算子的正向计算，在元算子基类中该方法没有被实现，只有继承了元算子基类的具体元算子类才会实现该方法. 同样地，*compute_gradient*(*grad*)方法进行反向的梯度计算，*grad* 参数为当前的梯度，该方法同样需要在继承了元算子基类的具体元算子类中实现.

不同的深度学习框架对于自动梯度计算的实现不同，例如：当计算一个矩阵与一个标量数值相加后，PyTorch 对标量数值求得的梯度同样也是一个标量[32]，而 TensorFlow 给出的梯度结果是一个矩阵. 为了使本文检测方法能够更好地比较不同深度学习框架的梯度计算结果，元算子在采用 TensorFlow 等底层实现时会进一步处理梯度的形状，使得元算子在使用不同底层实现方法时在梯度的形状上可以保持一致.

Add 元算子继承了元算子基类，在本文检测工具中为加法计算的元算子. 该元算子具有多种正向计算和梯度计算的方法，每一种正向计算方法和梯度计算方法通过接口实现对深度学习框架中算子计算方法的调用，以完成具体的计算任务.

4.3　调试器

本文方法在检测过程中，调试器负责执行深度学习模型的计算图，在得到每次算子替换后模型的计算结果后记录结果，以实现对不同深度学习框架统一计算过程得到的结果的对比.

4.3.1　数据流执行器

效仿 TensorFlow 中执行计算图的方式，本文工具同样设计了会话控制类以启动对计算图中数据流的执行. 启动对数据流的执行之前首先需要创建一个会话.

会话类在执行计算图时，用户需要制定一个“目标节点”，会话将以该节点为起点，在计算图中按照广度优先搜索的方式找出该节点所依赖的全部节点. 计算图中节点的计算顺序为广度优先搜索的倒序，这样的顺序可以保证计算时每个节点的依赖节点都已经完成计算.

4.3.2　状态记录器

在按照算法 1 的检测过程检测深度学习框架时，每次替换完一个元算子的底层实现方式之后，由数据流执行器执行计算图，得到模型的计算结果. 状态记录器将每一次执行计算图后得到的正向推断结果和梯度计算结果分别记录下来，并对每次替换算子前后的结果进行比较，最后按照替换算子带来结果差异的大小对元算子排序，并将结果差异与用户定义的阈值比较，给出最终的缺陷定位结果.

4.4　模型静态图

通常，对深度学习任务的模型定义和参数求解方式进行抽象之后，可以确定一个唯一的计算逻辑，将这个逻辑用有向无环图表示，称之为计算图. 计算图定义了数据的流转方式，数据的计算方式，以及各种计算之间的相互依赖关系等. 本文工具使用这样计算图来表示深度学习模型的结构.

TensorFlow 等框架会在运行计算图之前定义好计算图的结构,然后再进行实际的计算,这样的计算图被称为“静态图”.然而,在 PyTorch 等框架中,计算图的构建和计算同时发生,这样的计算图被称为“动态图”[33].用户可以从动态图中实时得到中间结果的值,这使得调试更加容易;但在对深度学习模型的实际使用中很少采用这种机制,且本文方法无需获取中间计算结果.因此,本文采取类似于 TensorFlow 中静态图的形式表示深度学习模型.即使用户在使用本文工具时,采用 PyTorch 等框架作为模型算子的底层实现,本文工具也使用工具自身所定义的静态图来表示模型结构.

计算图中的节点根据功能主要分为计算节点(Operator)、存储节点(Variable)、常数节点(Constants)和数据节点(Placeholder)四类.每一个计算节点对应一个元算子,主要负责算法逻辑表达或者流程控制.存储节点通常用来存储模型权重.常数节点定义了计算图中不可被训练的、常值不变的权重.数据节点用于定义输入数据的类型和形状等属性,是对数据的统一抽象[34].

计算图中的边是有向边,边的方向通常为前向求值的方向,定义不同节点之间的关系.边可以分为两类:一类用来传输数据,称为数据边;另一类用来定义依赖关系,称为控制边.所有的节点都通过数据边或者控制边连接,其中入度为 0 的节点没有前置依赖,可以立即执行;入度大于 0 的节点,要等待其依赖的所有节点执行结束之后,才可以执行.

4.5 用户接口

本文检测工具所提供类似于各个深度学习框架所提供的算子函数式接口,通过类似函数调用的形式完成深度学习模型的搭建.例如,对于一个损失函数为均方误差(mean square error)的一元线性回归模型,用户可以采用如代码块 1 的方式搭建用于检测的模型.

代码块 1. 搭建用于检测的模型.

```
x=sf.Placeholder()
y_=sf.Placeholder()
w=sf.Variable([[1.0]], name='weight')
b=sf.Variable(0.0,name='threshold')
y=x*w+b
loss=sf.mse(y,y_)
```

在对深度学习框架的缺陷检测阶段,用户在指定模型的输入数据和待检测的两种深度学习框架后,只需启动检测会话,本文检测工具即可自动利用用户所搭建的深度学习模型,对用户指定的两种深度学习框架计算结果进行比较.以 TensorFlow 和 PyTorch 两种深度学习框架为例,检测会话的启动方式如代码块 2 所示.

代码块 2. 启动检测会话

```
with sf.DebugSession() as sess:
    sess.run(loss,feed_dict=feed_dict,
             frameworks=['tf', 'pytorch'])
```

5 实验评估与实例分析

本文共实现了TensorFlow[5]、TensorFlow Lite[35]、PyTorch[6]和 MXNet[36]四种深度学习框架的共 20 种不同的元算子,实现所用框架版本分别为 TensorFlow 2.0.0(TensorFlow Lite为TensorFlow 框架的移动端版本)、PyTorch 1.7.1 和 MXNet 1.6.0.本文工具目前已实现的元算子及其底层深度学习框架如表 1 所示.除此以外,本文的方法也可扩展至其他深度学习框架和其他算子.

表 1 检测工具实现的元算子及其底层深度学习框架

元算子	深度学习框架			
	TensorFlow	TensorFlow Lite	PyTorch	MXNet
add	√	√	√	√
multiply	√	√	√	√
matmul	√	√	√	√
sigmoid	√	√	√	√
log	√	√	√	√
exp	√	√	√	√
negative	√	√	√	√
inv	√	√	√	√
div	√	√	√	√
square	√	√	√	√
transpose	√	√	√	√
reshape	√	√	√	√
reduce_sum	√	√		
reduce_mean	√	√		
softmax	√	√	√	√
ReLU	√	√	√	√
BatchNorm	√	√	√	
Conv2D(GPU)	√		√	
einsum	√			√
Dropout	√		√	√

为了验证方法的有效性,对本文方法进行了实验评测.本章将主要分可行性评估、性能评估和实例分析三个部分进行介绍.

5.1 可行性评估

可行性评估是为了验证本文方法所提出的元算子在采用不同深度学习框架作为底层实现时,

计算结果可以准确反映深度学习框架本身的计算情况.

5.1.1　实验设计

在可行性评估实验中，对于本文实现的每一个不包含随机计算过程的元算子，对比元算子采用某种深度学习框架作为底层实现时的计算结果和直接使用该深度学习框架的算子的计算结果，以此验证元算子正向推断计算和反向梯度计算的正确性. 测试每个元算子的输入数据为随机产生.

不同深度学习框架对梯度计算结果的形状处理不同，为保证深度学习模型中不同元算子采用不同底层实现时仍可以正常推算或训练，元算子在采用某些深度学习框架作为底层实现时，在原计算结果的基础上改变了部分算子梯度的形状. 因此在验证这些元算子梯度计算的正确性时，需要比较直接使用深度学习框架计算得到的梯度和元算子计算梯度的中间结果.

5.1.2　实验结果

经比较，除 Dropout 等具有随机性的元算子外，在输入数据相同时，每个已实现的元算子和深度学习框架中相应算子的正向推断计算结果之差均为 0；除元算子对梯度形状的改变操作之外，梯度计算结果之差也均为 0. 因此元算子底层替换的计算结果可以准确反映深度学习框架中算子的计算结果，本文工具的元算子底层实现替换操作具有可行性.

5.2　性能评估

本文检测方法的计算单元“元算子”与深度学习框架的计算单元“算子”相比，增加了接口调用、逻辑判断和部分计算，因此元算子的运行效率低于相应深度学习框架算子计算的效率，也会占用更多的内存. 本节从元算子在进行运算时效率和内存占用情况两方面对元算子的性能进行测试. CRADLE[11] 以及 Srisakaokul 等人[28] 提出的对深度框架的缺陷检测方法均没有对深度学习框架以及编程接口进行修改，应用相应方法时算子的性能与原框架相同，因此我们只需将本文所实现的元算子性能与框架原有算子性能进行对比.

5.2.1　实验设计

为测试元算子对计算效率的影响，在性能评估实验中，在使用同样的数据作为输入时，对比元算子采用某种深度学习框架作为底层实现时的计算时间和直接使用该深度学习框架的算子的计算时间. 根据不同算子对输入数据的不同要求，如果没有特殊说明，本文实验采用的输入数据的精度均为 32 位浮点数，形状为[1,100]、[100,1]、[1,1,1,100]、[1,100,1,1]或[1,1,100]. 其中 einsum 算子的方程输入为“bnij,bnjc—>bnic”，矩阵输入的形状为[1,1,5,3]和[1,1,3,2]. 对于每一个算子，记录该算子 10 次计算时间的平均值，以代表该算子单次计算的时间. 在对元算子内存占用情况的性能评估实验中，在使用同样的数据作为输入时，对比某深度学习框架作为底层实现的元算子和该深度学习框架中的相应算子在进行计算时的内存占用情况. 同样地，每一个算子做 10 次计算，在本文实验中记录每次计算时的内存增加并取平均值，以代表该算子单次计算的内存占用情况. 由于 TensorFlow Lite 中没有算子的接口，使用 TensorFlow Lite 模型进行推断的唯一方式是使用转换器将 TensorFlow 模型转换为 TensorFlow Lite 模型，再使用 TensorFlow Lite 解释器加载模型、优化模型计算图的结构，最后再进行推断操作. 因此，对于每一个使用 TensorFlow Lite 作为底层实现的元算子，都需要运行一次模型转换、加载模型和优化模型的操作. 这导致使用 TensorFlow Lite 作为元算子底层实现时，其计算时间和内存消耗远远超过其他三个框架. 因此，在 5.2.2 节的图示中没有展示 TensorFlow Lite 框架的结果.

5.2.2　实验结果

实验中观察到，元算子不如其在深度学习框架中对应的算子运行效率高，但除 TensorFlow Lite 外，元算子采用其他深度学习框架作为底层实现时的运行效率与深度学习框架中的算子相比差距较小. 图 5 展示了本文工具元算子单次计算的计算时间与 TensorFlow、Pytorch 和 MXNet 三种框架的算子相比的时间增加的累计分布曲线(CDF). 观察到，使用本文检测方法中的元算子计算时，单次正向推断计算的计算时间均比使用原深度学习框架直接

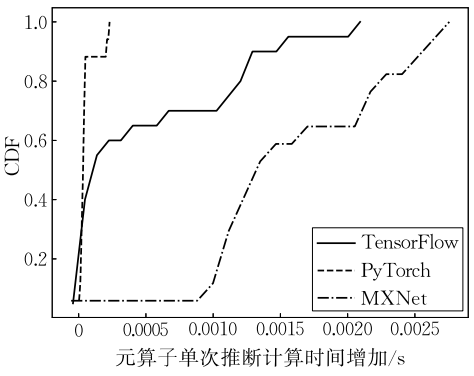

图 5　元算子算子推断计算时间增加

计算高,其中使用 PyTorch 作为底层实现时,元算子的平均单次计算时间为直接使用 PyTorch 的 1.7 倍,且单次计算的时间增加均在 0.0002 s 以内.而使用 MXNet 作为元算子底层实现时,平均单次计算时间为直接使用 MXNet 算子时的 5.81 倍.这主要是因为使用 MXNet 作为元算子底层实现时,需要在原有计算的基础上增加数据类型转换的操作,以保证算子的计算可以正常执行.

除了图 5 中展示的三个框架之外,在本文所实现的使用 TensorFlow Lite 作为底层实现的加法元算子中,单次推断计算平均耗时为 4.70 s,这主要是因为其在计算之前的准备工作消耗了大量的时间,计算效率较低.

图 6 为除不支持梯度计算的 TensorFlow Lite 外,本文所实现的全部元算子与相应深度学习框架算子梯度计算效率对比.在采用 PyTorch 框架作为远算子的底层实现时,本文工具所实现的元算子单次梯度计算时间相比使用原深度学习框架的梯度计算时间,均不增加超过 1.2 ms.而 TensorFlow 框架本身在进行自动梯度计算时对张量形状的处理方式不同于其他框架,因此在使用 TensorFlow 作为底层实现时,本文工具对部分算子增加了张量形状处理操作,这导致部分算子使用 TensorFlow 作为底层实现时的提督计算时间略长.本文检测方法为了增加对深度学习框架的缺陷检测功能,牺牲了元算子的计算效率.

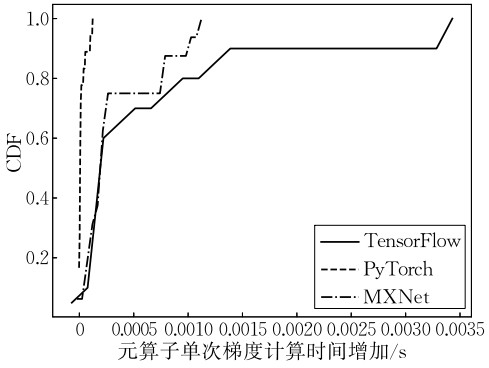

图 6 元算子梯度计算时间增加

图 7 和图 8 分别展示了除 TensorFlow Lite 外 TensorFlow、Pytorch 和 MXNet 三种框架的加法算子单次计算的内存使用情况.观察到,大部分元算子只有少量的额外内存占用.使用 TensorFlow 为元算子底层实现时,单次推断计算的内存占用较大,主要内存占用来自于推断计算时的数据类型转换.

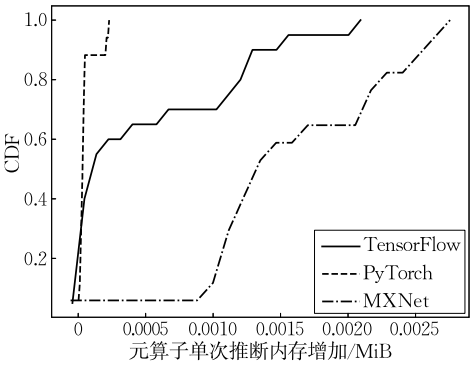

图 7 元算子推断计算内存占用增加

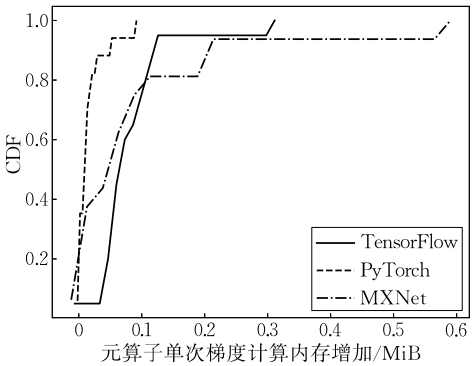

图 8 元算子梯度计算内存占用增加

在 TensorFlow Lite 中,本文所实现的算子单次计算时的平均内存占用均不超过 0.03 MiB,而以 TensorFlow Lite 作为加法元算子底层实现时,单次计算的内存平均增加高达 8.42 MiB.这同样是由于使用 TensorFlow Lite 作为底层实现的元算子,需要运行模型转换、加载模型和优化模型等一系列操作.

由性能评估实验结果得知,除了 TensorFlow Lite 以外,使用其他深度学习框架作为本文工具元算子底层实现时,其性能虽然比直接使用原深度学习框架算子差,但是对整体性能的影响不大.若使用 TensorFlow Lite 作为本文工具元算子底层实现,由于受到 TensorFlow Lite 本身特性的限制,元算子的计算不得不为了满足本文方法的功能而牺牲性能.

5.3 缺陷检测

本文首先使用仅包含一个元算子的“单层”模型,对工具中已实现的所有算子计算结果进行检测.表 2 和表 3 为本文所实现的全部元算子采用不同底层实现时计算结果的平均绝对离差.在实验中,对每个单层模型使用同样的、随机生成的输入数据,且全部采用与 5.2 节相同的参数设置.可以看出,只有

einsum 元算子采用 TensorFlow 和 MXNet 作为底层实现时的平均绝对离差较大，为 0.745. 对于该结果，我们将在 5.4.1 节进一步探讨.

表 2　检测工具实现的元算子正向计算结果 MAD

元算子	深度学习框架		
	TensorFlow v. s. PyTorch	TensorFlow Lite v. s. TensorFlow	TensorFlow v. s. MXNet
add	0	0	0
multiply	0	0	0
matmul	0	7.37e-6	2.54e-6
sigmoid	0	0	0
log	4.32e-9	4.62e-9	4.62e-9
exp	1.22e-6	7.38e-6	7.38e-6
negative	0	0	0
inv	0	—	0
div	1.25e-8	—	1.25e-8
square	0	0	0
transpose	0	0	0
reshape	0	0	0
reduce_sum	—	1.22e-4	—
reduce_mean	—	9.54e-7	—
softmax	1.24e-11	8.96e-10	9.87e-9
ReLU	0	0	0
BatchNorm	1.15e-4	0	
Conv2D(GPU)	8.91e-5	—	
einsum	—	—	9.82e-8

表 3　检测工具实现的元算子梯度计算结果 MAD

元算子	深度学习框架	
	TensorFlow v. s. PyTorch	TensorFlow v. s. MXNet
add	0	0
multiply	0	0
matmul	3.77e-8	0
sigmoid	0	0
log	0	0
exp	1.22e-6	7.38e-6
negative	0	0
inv	0	4.02e-9
div	1.10e-8	9.94e-8
square	0	0
transpose	0	0
reshape	0	0
softmax	7.39e-19	2.22e-8
ReLU	0	0
BatchNorm	1.02e-8	—
Conv2D(GPU)	0	—
einsum	—	**0.745**

为了保证错误算子都被检测到，且要避免产生过多假阳性结果，通过在实验中不断改变阈值 T_C 和 T_G 的取值，本文实验最终将 T_C 和 T_G 均设为 1.5e-4. 通过将已实现的算子逐一与各深度学习框架官方的错误报告进行比对验证，可以确认在该阈值取值下，本文的诊断不存在假阳性和假阴性结果. 本文工具可以拓展至其他目前尚未实验的算子，该阈值的取值在其他算子上可能会引入假阳性和假阴性结果，对不同的元算子集合、不同的数据精度，阈值的合理取值不同. 对于不带有随机性算子的模型，在相应数值精度下，阈值的取值应和模型中算子计算结果的可接受浮动范围(即近似阈值[9]的二倍)接近.

5.4　实例分析

本小节通过搜集已知的深度学习框架中的计算错误或有差异的计算结果，以及现有深度学习框架缺陷检测方法的检测报告和结果，然后应用本文方法对包含已有方法没有检测出的算子的深度学习模型进行缺陷检测，通过应用实例验证本文方法的有效性.

5.4.1　实例一：MXNet 中 einsum 算子的梯度计算错误

“多头注意力”机制是 Transformer[37]、BERT 等自然语言处理模型中的核心组件，可以自动学习和计算输入数据对输出数据的贡献大小. 其核心结构包含了矩阵乘法、softmax 等计算[38]，其中，多维矩阵乘法可以通过爱因斯坦求和约定(einsum)计算. 由于多头注意力模型中的 Dropout 算子带有随机性，不适用于本文提出的方法，因此本文采用去除 Dropout 算子后的多头注意力模型进行实验. 第 6 节有对带有随机性算子的更多讨论. 将本文方法应用于该模型，我们发现 MXNet 与 TensorFlow 的 einsum 算子的梯度计算结果有较大差异. 例如，使用图 9 随机产生的矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 作为模型输入数据时，MXNet 和 TensorFlow 的 einsum 算子计算得到的 A 的梯度计算结果的平均绝对离差高达 3.96，而在该实例的检测过程中，对其他算子的底层实现方式进行替换前后，计算结果的平均绝对离差均未超过 1.5e-4.

使用如图 9 所示的随机输入数据作为模型输入，本文方法检测到，替换 einsum 算子前后模型整体梯度之差高达 0.398，远远超过 T_G；其中，TensorFlow 和 MXNet 的 einsum 算子计算得到的 value 的梯度分别如图 9 所示. 而在该实例的检测过程中，对其他算子的底层实现方式进行替换前后，计算结果的平均绝对离差均未超过 T_G. 因此，本文方法检测出 MXNet 框架的 einsum 算子存在错误. 为了对该结果进行验证，我们按照 3.4 节的方法比较两个模型的梯度计算结果：(1) 所有元算子均用 TensorFlow 作为底层实现的模型；(2) einsum 元算子用 TensorFlow 作为底层实现、其他元算子用 MXNet

作为底层实现的模型. 两个模型的整体梯度之差为 0. 因此,einsum 算子是导致计算结果错误的原因. 比较本文实验所用版本深度学习框架底层代码后,可以确认 MXNet 的 einsum 算子计算有误.

```
query: [[[[1.17109962][0.37565252][0.82949107] [0.29293999][-0.61608501]]
         [[1.75082922][-0.13850867][1.59422947][-0.09395125][1.56230319]]
         [[0.43450781][1.77360748][0.94412398] [1.90139302][1.55708293]]]]
key: [[[0.25439725] [0.96581968] [-0.32869229][2.16545762][2.07900606]]
      [[2.301548]  [1.85931635][ 1.24133237] [0.40015171][-0.060046]]
      [[0.93944128] [0.8107018] [ 0.70168201] [1.36959909][0.34947845]]]]
value: [[[[1.25536117] [0.22812862][0.33883442] [0.88289629][0.91486889]]
         [[1.58969553] [1.33733386][0.14963483] [-1.44493619][2.6568602]]
         [[1.86181002] [0.31700891][1.25405337] [-0.68810828][0.63587646]]]]

TensorFlow_grad: [[[[0.23091541][0.5312316][0.11665206][2.1648228] [1.9563781]]
                   [[2.9955711] [1.381076] [0.46807754][0.10732592][0.04794922]]
                   [[1.035875][0.97952074][0.93420273][1.2487628][0.8016389]]]]
MXNet_grad: [[[[0.68662703][0.79202116][0.69883525][1.4509143][1.3716023]]
               [[2.0841556][1.208312][0.7005228][0.51207346][0.49493614]]
               [[1.0247087][0.8674304][0.7563526][1.8527048][0.49880356]]]]
```

图 9 MXNet 和 TensorFlow 的 eimsum 算子梯度计算差异

目前,已经有用户向 MXNet 官方提出这一错误,MXNet 的开发人员在对这一错误进行了核实和验证后,在 MXNet 的最新发行版本中修复了该错误.

5.4.2 实例二:对 MXNet 中 ReLU 算子计算结果的验证

Keras 是一个可以使用不同深度学习系统作为后端的高级神经网络 API[7]. 有用户发现,使用如图 10 所示的图片作为图像分类任务模型 MobileNetV1 的输入时,若 Keras 采用 TensorFlow 或 CNTK[39] 作为后端,得到的分类结果为“浴帘”,而使用 MXNet 作为后端时,得到的分类结果为“天鹅绒”[40]. 通过采用类似于 CRADLE[11] 的方法比较模型每一层计算的中间结果,该用户发现,无论使用哪一种深度学习系统作为后端,模型前三层的计算结果都几乎相同,而第四层 ReLU 计算得到的结果差异性较大. 因此,该用户猜测可能是 MXNet 的 ReLU 算子存在计算错误.

图 10 导致 Keras 结果差异的图片输入

通过本文诊断工具的对 MobileNetV1 模型中 TensorFlow 和 MXNet 两种深度学习系统算子的对比,发现在使用图 10 作为模型的输入时,替换 ReLU 算子底层实现后的计算结果只差为 0,即两种深度学习框架的 ReLU 算子计算结果完全相同. 因此,Keras 的结果差异很有可能是 Keras 本身导致的,而非 MXNet 的 ReLU 算子计算错误导致的. 该结论得到了提出这一问题的用户的认同.

6 结束语

本文设计并实现了基于元算子的深度学习框架缺陷检测方法,其将不同深度学习框架中算子的共性计算逻辑抽象为“元算子”,支持在不改变模型代码的前提下绑定元算子的具体实现,从而高效地实现算子的细粒度替换,进而可以通过比较同一模型在不同框架下的运行结果来发现框架缺陷. 实验表明,本文方法为满足功能上的需求牺牲了部分性能,方法可以有效地检测和定位不同深度学习框架中算子计算的差异.

本文方法具有一定的局限性:首先,在一些情况下,本文方法无法准确检测到深度学习框架中算子的计算错误. 一方面,深度学习框架中可能包含复杂的、不确定的算子,因此在给定相同输入的情况下,这些算子输出可能会略有不同. 为了避免这种不确定性影响本文方法的检测效果,本文的仅适用于不包含随机性的算子,且这些算子不会在使用过程中进行权重初始化等改变算子属性的操作. 另一方面,有时两种框架中同一算子的实现出现了相同的错误导致输出结果相同;或有错误的算子经过和其他算子组合,导致在不同深度学习框架实现之下的深度学习模型输出差异不大. 这些情况下的缺陷仍是在未来需要探究的课题,本文方法目前无法检测这些错误. 其次,用户使用本文检测方法在运行监测过程之前需使用本文检测工具的编程接口搭建深度学习模型,过程较繁琐. 另外,由本文性能评估实验结果知,本文提出的元算子的计算时间和计算时内存占用开销具有提升空间.

因此,本文未来工作主要集中在对缺陷检测方法的功能性补全和性能优化上. 主要包括以下两个方面:首先,为检测工具增加模型转换功能,即用户直接使用深度学习框架模型文件即可对模型所包含的算子进行检测,而无需使用工具接口重新搭建深度学习模型. 其次,优化元算子类的设计和计算过程,减少不必要的计算和存储,以提升元算子的计算性能. 此外,本文方法还可与深度学习框架的模型生成方法和数据生成方法相结合,从而在保证全面检测深度学习框架行的算子的同时有效触发深度学习框架中的错误.

参 考 文 献

[1] Recurrent Neural Network that Generates Little Stories About Images. https://github.com/ryankiros/neural-storyteller, 2018

[2] Graves A, Fernández S, Gomez F. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks//Proceedings of the 23rd International Conference on Machine Learning. Pittsburgh, USA, 2006: 369-376

[3] Gu J, Neubig G, Cho K, et al. Learning to translate in real-time with neural machine translation//Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics. Valencia, Spain, 2017: 1053-1062

[4] Chen C, Seff A, Kornhauser A, et al. DeepDriving: Learning affordance for direct perception in autonomous driving//Proceedings of the 2015 IEEE International Conference on Computer Vision. Santiago, Chile, 2015: 2722-2730

[5] Abadi M, Barham P, Chen J, et al. TensorFlow: A system for large-scale machine learning//Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation. Savannah, USA, 2016: 265-283

[6] Paszke A, Gross S, Massa F, et al. PyTorch: An imperative style, high-performance deep learning library//Advances in Neural Information Processing Systems 32 (NeurIPS 2019). Vancouver, Canada, 2019: 8024-8035

[7] Keras. https://keras.io, 2015

[8] Uber Finds Deadly Accident Likely Caused by Software Set to Ignore Objects on Road. https://www.theinformation.com/articles/uber-finds-deadly-accident-likely-caused-by-software-set-to-ignore-objects-on-road, 2018

[9] Nejadgholi M, Yang J. A study of oracle approximations in testing deep learning libraries//Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering. San Diego, USA, 2019: 785-796

[10] Jia L, Zhong H, Wang X, Huang L, et al. An empirical study on bugs inside TensorFlow//Proceedings of the Database Systems for Advanced Applications—25th International Conference Part I. Jeju, Korea, 2020: 604-620

[11] Pham H V, Lutellier T, Qi W, et al. CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries//Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering. Montreal, Canada, 2019: 1027-1038

[12] Pearson S, Campos J, Just R, et al. Evaluating and improving fault localization//Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering. Buenos Aires, Argentina, 2017: 609-620

[13] Weiser M. Programmers use slices when debugging. Communications of the ACM, 1982, 25(7): 446-452

[14] Tip F. A survey of program slicing techniques. Journal of Programming Languages, 1995, 3: 121-189

[15] Abreu R, Gemund A. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis//Proceedings of the 8th Symposium on Abstraction, Reformulation, and Approximation. Lake Arrowhead, USA, 2009: 2-9

[16] Wotawa F, Stumptner M, Mayer W. Model-based debugging or how to diagnose programs automatically//Proceedings of the International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems: Developments in Applied Artificial Intelligence. Cairns, Australia, 2002: 746-757

[17] Le T, Oentaryo R, Lo D. Information retrieval and spectrum based bug localization: Better together//Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo, Italy, 2015: 579-590

[18] Hong S, Kwak T, Lee B, et al. MUSEUM: Debugging real-world multilingual programs using mutation analysis. Information and Software Technology, 2017, 82: 80-95

[19] Moon S, Kim Y, Kim M, et al. Ask the mutants: Mutating faulty programs for fault localization//Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. Cleveland, USA, 2014: 153-162

[20] Zhang Y, Chen Y, Cheung S, et al. An empirical study on TensorFlow program bugs//Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. Amsterdam, Netherlands, 2018: 129-140

[21] Islam M, Nguyen G, Pan R, et al. A comprehensive study on deep learning bug characteristics//Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn, Estonia, 2019: 510-520

[22] Humbatova N, Jahangirova G, Bavota G, et al. Taxonomy of real faults in deep learning systems//Proceedings of the 42nd International Conference on Software Engineering. Seoul, Korea, 2020: 1110-1121

[23] Zhang J M, Harman M, Ma L, et al. Machine learning testing: Survey, landscapes and horizons. arXiv preprint arXiv: 1906.10742, 2019

[24] Pei K, Cao Y, Yang J, et al. DeepXplore: Automated whitebox testing of deep learning systems//Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China, 2017: 1-18

[25] Guo J, Jiang Y, Zhao Y, et al. DLFuzz: Differential fuzzing testing of deep learning systems//Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista, USA, 2018: 739-743

[26] Tian Y, Pei K, Jana S, Ray B. DeepTest: Automated testing of deep-neural-network-driven autonomous cars//Proceedings of the 40th International Conference on Software Engineering. Gothenburg, Sweden, 2018: 303-314

[27] Papernot N, Mcdaniel P, Swami A, et al. Crafting adversarial input sequences for recurrent neural networks//Proceedings of the 2016 IEEE Military Communications Conference. Baltimore, USA, 2016: 49-54

[28] Srisakaokul S, Wu Z, Astorga A, et al. Multiple-implementation testing of supervised learning software//Proceedings of the 23rd AAAI Conference on Artificial Intelligence. New Orleans, USA, 2018: 384-391

[29] Ma L, Juefei-Xu F, Zhang F, et al. DeepGauge: Multi-granularity testing criteria for deep learning systems//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Montpellier, France, 2018: 120-131

[30] Du X, Xie X, Li Y, et al. Deepstellar: Model-based quantitative analysis of stateful deep learning systems//Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn, Estonia, 2019: 477-487

[31] Wang Z, Yan M, Chen J, et al. Deep learning library testing via effective model generation//Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Virtual Event, USA, 2020: 788-799

[32] Paszke A, Gross S, Chintala S, et al. Automatic differentiation in PyTorch//Proceedings of the 31st Conference on Neural Information Processing Systems Workshop Autodiff. Long Beach, USA, 2017

[33] Neubig G, Dyer C, Goldberg Y, et al. DyNet: The dynamic neural network toolkit. arXiv preprint arXiv: 1701.03980, 2017

[34] Peng Jing-Tian, Lin Jian, Bai Xiao-Long. Deeply Understand the Architecture Design and Implementation Principle of TensorFlow. Beijing: The People's Posts and Telecommunications Press, 2018(in Chinese)
(彭靖田, 林健, 白小龙. 深入理解 TensorFlow 架构设计与实现原理. 北京: 人民邮电出版社, 2018)

[35] TensorFlow Lite|ML for Mobile and Edge Devices. https://www.tensorflow.org/lite/2017, 2017

[36] Chen T, Li M, Li Y, et al. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274, 2015

[37] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need//Proceedings of the 30th Annual Conference on Neural Information Processing Systems. Long Beach, USA, 2017: 5998-6008

[38] Devlin J, Chang M, Lee K, et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv preprint arXiv: 1810.04805, 2018

[39] Seide F, Agarwal A. CNTK: Microsoft's open-source deep-learning Toolkit//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Francisco, USA, 2016: 2135-2135

[40] The Output of the ReLU Layer in MXNet is Different from that in TensorFlow and CNTK. https://github.com/apache/incubator-mxnet/issues/17684, 2020

GU Dian-Dian, Ph. D. candidate. Her research interests include system software and machine learning systems.

SHI Yi-Ning, M. S. candidate. His research interests include deep learning and high-performance computing.

LIU Xuan-Zhe, Ph. D., associate professor. His research interests include services computing and system software.

WU Ge, B. S. Her research interests include big data and artificial intelligence.

JIANG Hai-Ou, Ph. D., assistant researcher. Her research interests include cloud computing, big data, and machine learning.

ZHAO Yao-Shuai, M. S. His research interests include big data and artificial intelligence.

MA Yun, Ph. D., assistant professor. His research interests include system software and Web computing.

Background

This work is related to the field of detecting and localizing defects in deep learning frameworks. Previous work focuses on performing cross-implementation inconsistency checking to detect bugs in deep learning frameworks and leveraging anomaly propagation tracking and analysis to localize faulty functions in deep learning frameworks. However, on the one hand, existing deep learning framework defect detection methods can only detect large calculation differences of

operators between different frameworks through comparison and speculation. These methods cannot verify the accuracy of the detection results. On the other hand, existing methods can only detect calculation errors of deep learning models in the inference process, and cannot detect calculation errors generated in the training process.

In order to overcome the shortcomings of current defect detection methods and to make the defect detection of deep learning frameworks more accurate and more comprehensive, in this paper, we design and implement a defect detection method for deep learning frameworks based on meta operators. We abstract common computing logic such as forward computation and gradient computation of operators in different deep learning frameworks as "meta operator" and bind the specific implementation of operators without changing the code of deep learning models. In this way, users can make fine-grained replacement of operators in DL models. Through fine-grained operator replacement, not only can the calculation errors of the deep learning frameworks during the inference process be found, but also the defect occurred during the training process can be detected. Also, the localization of these defects can be verified.