

# Deep-SBFL: 基于频谱的深度神经网络缺陷定位方法<sup>\*</sup>

李 铮<sup>1</sup>, 崔展齐<sup>1</sup>, 陈 翔<sup>2</sup>, 王荣存<sup>3</sup>, 刘建宾<sup>1</sup>, 郑丽伟<sup>1</sup>

<sup>1</sup>(北京信息科技大学 计算机学院, 北京 100101)

<sup>2</sup>(南通大学 信息科学技术学院, 江苏 南通 226019)

<sup>3</sup>(中国矿业大学 计算机科学与技术学院, 江苏 徐州 221116)

通信作者: 崔展齐, E-mail: [czq@bistu.edu.cn](mailto:czq@bistu.edu.cn)



**摘 要:** 深度神经网络已经在自动驾驶和智能医疗等领域取得了广泛的应用. 与传统软件一样, 深度神经网络也不可避免地包含缺陷, 如果做出错误决定, 可能会造成严重后果. 因此, 深度神经网络的质量保障受到了广泛关注. 然而, 深度神经网络与传统软件存在较大差异, 传统软件质量保障方法无法直接应用于深度神经网络, 需要设计有针对性的质量保障方法. 软件缺陷定位是保障软件质量的重要方法之一, 基于频谱的缺陷定位方法在传统软件的缺陷定位中取得了很好的效果, 但无法直接应用于深度神经网络. 在传统软件缺陷定位方法的基础上提出了一种基于频谱的深度神经网络缺陷定位方法 Deep-SBFL. 该方法首先通过收集深度神经网络的神经元输出信息和预测结果作为频谱信息; 然后将频谱信息进行处理作为贡献信息, 以用于量化神经元对预测结果所做的贡献; 最后提出了针对深度神经网络缺陷定位的怀疑度公式, 基于贡献信息计算深度神经网络中神经元的怀疑度并进行排序, 以找出最有可能存在缺陷的神经元. 为验证该方法的有效性, 以  $E_{\text{Inspect}}@n$  (结果排序列表前  $n$  个位置内成功定位的缺陷数) 和  $EXAM$  (在找到缺陷元素之前必须检查元素的百分比) 作为评测指标, 在使用 MNIST 数据集训练的深度神经网络上进行了实验. 结果表明, 该方法可有效定位深度神经网络中不同类型的缺陷.

**关键词:** 软件质量保障; 软件缺陷定位; 深度神经网络 (DNN); 频谱; 怀疑度

**中图法分类号:** TP311

中文引用格式: 李铮, 崔展齐, 陈翔, 王荣存, 刘建宾, 郑丽伟. Deep-SBFL: 基于频谱的深度神经网络缺陷定位方法. 软件学报, 2023, 34(5): 2231–2250. <http://www.jos.org.cn/1000-9825/6403.htm>

英文引用格式: Li Z, Cui ZQ, Chen X, Wang RC, Liu JB, Zheng LW. Deep-SBFL: Spectrum-based Fault Localization Approach for Deep Neural Networks. Ruan Jian Xue Bao/Journal of Software, 2023, 34(5): 2231–2250 (in Chinese). <http://www.jos.org.cn/1000-9825/6403.htm>

## Deep-SBFL: Spectrum-based Fault Localization Approach for Deep Neural Networks

LI Zheng<sup>1</sup>, CUI Zhan-Qi<sup>1</sup>, CHEN Xiang<sup>2</sup>, WANG Rong-Cun<sup>3</sup>, LIU Jian-Bin<sup>1</sup>, ZHENG Li-Wei<sup>1</sup>

<sup>1</sup>(School of Computer Science, Beijing Information Science and Technology University, Beijing 100101, China)

<sup>2</sup>(School of Information Science and Technology, Nantong University, Nantong 226019, China)

<sup>3</sup>(School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China)

**Abstract:** Deep neural networks have been widely used in fields such as autonomous driving and smart healthcare. Like traditional software, deep neural networks inevitably contain defects, and it may cause serious consequences if they make wrong decisions. Therefore, the quality assurance of deep neural networks has received extensive attention. However, deep neural networks are quite different from traditional software. Traditional software quality assurance methods cannot be directly applied to deep neural networks, and targeted quality assurance methods need to be designed. Software fault localization is one of the important methods to ensure software quality. The

<sup>\*</sup> 基金项目: 江苏省前沿引领技术基础研究专项 (BK20202001); 国家自然科学基金 (61702041); 江苏省自然科学基金面上项目 (BK20181353); 北京信息科技大学“勤信人才”培育计划 (QXTCP C201906)

收稿时间: 2020-12-21; 修改时间: 2021-02-24, 2021-05-17; 采用时间: 2021-06-24; jos 在线出版时间: 2022-07-07

CNKI 网络首发时间: 2022-11-15

spectrum-based fault localization method has achieved good results in traditional software fault localization methods, but it cannot be directly applied to deep neural networks. In this study, based on the traditional software fault localization methods, a spectrum-based fault localization approach named Deep-SBFL for deep neural network is proposed. The approach firstly collects the neuron output information and the prediction results of deep neural network as the spectrum. The spectrum is then further calculated as the contribution information, which can be used to quantify the contribution of neurons to the predicted results. Finally, a suspicious formula for the defect localization of deep neural network is proposed. Based on the contribution information, the suspiciousness scores of neurons in deep neural network are calculated and ranked to find out the most likely defective neurons. To verify the effectiveness of the method,  $E_{\text{Inspect}@n}$  (the number of defects successfully located by inspecting the first  $n$  positions of the sorted list) and  $EXAM$  (the percentage of elements that must be checked before finding defect elements) are evaluated on a deep neural network trained by the MNIST data set. Experimental results show that this approach can effectively locate different types of defects in deep neural networks.

**Key words:** software quality assurance; software fault localization; deep neural networks (DNN); spectrum; suspiciousness scores

## 1 引言

作为一种重要的机器学习模型, 深度神经网络 (deep neural network, DNN) 已在很多领域中取得了广泛的应用. 在医学图像分析<sup>[1]</sup>、语音识别<sup>[2]</sup>、恶意软件检测<sup>[3]</sup>等特定任务上 DNN 的性能已经达到甚至超过了人类水平, 在很多场景下可以代替人做出重要决策. 但是, 所有软件都不可避免地包含缺陷, DNN 也不例外. 尤其在自动驾驶和智能医疗等安全关键系统中, 如果 DNN 做出错误决定, 将会造成非常严重的后果. 例如: Uber 无人驾驶汽车在夜间测试的过程中发生严重事故致使一名行人死亡<sup>[4]</sup>; 苹果手机的 FaceID 可以被使用 3D 打印的人脸图片破解<sup>[5]</sup>等. 因此, 如何检测 DNN 中的缺陷以保证其质量, 成为了软件工程和人工智能交叉领域的热点研究问题.

目前, 适用于 DNN 的缺陷检测技术受到了广泛关注, 如使用蜕变关系对 DNN 进行测试<sup>[6]</sup>, 通过对比测试数据蜕变前后分类的一致性来检测 DNN 是否存在缺陷. 此外, 还有研究提出了不同的 DNN 测试覆盖准则, 如 DeepGauge<sup>[7]</sup>、DeepHunter<sup>[8]</sup>、DeepXplore<sup>[9]</sup>和 TensorFuzz<sup>[10]</sup>等, 这类工作目的是提高神经元的覆盖率, 并根据覆盖率制导的测试数据生成方法来检测 DNN 中存在的缺陷. 在成功检测出 DNN 中存在缺陷后, 更重要的是定位缺陷所在位置, 以便于修复缺陷. 但是上述工作既无法定位缺陷的具体位置, 也不能检测出 DNN 中特定结构存在缺陷, 如层数过多或者过少、激活函数不正确等.

传统软件通常是预先编写好的程序生成一组规则<sup>[11]</sup>, 使用相同的输入数据执行同一个程序, 执行状态和输出一定是相同的. 而 DNN 是先基于数据集训练后得到模型, 然后将测试数据输入模型输出预测结果, 这个结果与指定的 DNN 结构及训练得出的权重都有关. 现有的程序缺陷定位方法主要针对传统软件, 给予其不同的输入, 程序会执行不同的区域, 可通过对程序代码或者程序运行中的信息进行分析来得出缺陷定位结果. 然而 DNN 与传统软件不同, 在一条输入数据的预测过程中, 每个神经元都存在输出, 因此针对传统软件的缺陷定位方法不能直接应用到 DNN 上.

在使用过程中, DNN 可被视为一个黑盒, 它可以通过大量神经元中的简单函数来模拟任何一个连续函数. 以全连接 DNN 为例, 其中每个神经元都有输出, 这些输出经过权重将影响传递到下一层的神经元, 或者经过激活函数计算出新的输出. DNN 中的组成元素都有可能对最终结果产生影响, 无法准确找出某一个神经元、某种结构或者某种激活函数对 DNN 所产生结果的具体影响. 从这个角度来看, 其存在的缺陷难以被定位. 如果 DNN 的输出结果错误, 则难以准确找出造成错误的具体元素, 也就无法有针对性地构造数据集进行重训练或调试. 因此, 如何定量分析 DNN 的组成元素对预测结果的影响就成了定位 DNN 缺陷的一个切入点.

基于频谱的缺陷定位方法<sup>[12]</sup>作为一种轻量级的缺陷定位方法受到了广泛关注, 并取得了较好的定位效果. 该方法通过获取程序执行的覆盖信息和运行结果评估每个程序实体出错的可能性, 并按照程序实体出错的可能性, 从高到低进行审查, 直至定位到实际缺陷实体为止. 鉴于该方法在传统程序中的成功应用, 本文也将基于频谱的缺陷定位的思想运用在 DNN 中, 进而提出了一种基于频谱的 DNN 缺陷定位方法 Deep-SBFL (deep spectrum-based fault localization), 以在神经元的粒度上定位 DNN 中的缺陷. 该方法首先依据 DNN 中各神经元对测试数据的输出信息和预测结果量化 DNN 中神经元对不同预测结果的影响, 然后利用所提出的怀疑度公式计算各神经元的怀疑

度, 根据怀疑度对神经元进行降序排序, 从而定位最有可能存在缺陷的位置. 使用 Deep-SBFL 对使用 MNIST 数据集训练的 DNN 进行缺陷定位, 该方法可有效定位深度神经网络中不同类型的缺陷. 在 4 种变异体上的 EXAM 中位数都在 0.080 以下, 最好的结果在 0.001 以下, 实验还发现, 存在缺陷的神经元越靠近输出层越容易被准确定位.

本文的主要贡献如下.

- (1) 提出一种基于频谱的 DNN 缺陷定位方法 Deep-SBFL, 该方法能在神经元的粒度上定位 DNN 中隐含的缺陷.
- (2) 基于所提出的方法实现原型工具, 在 MNIST 数据集上开展实验, 验证 Deep-SBFL 的有效性.

本文第 2 节介绍研究背景和相关工作. 第 3 节介绍 Deep-SBFL 的框架和具体流程. 第 4 节介绍实验设计. 第 5 节对实验结果进行分析和讨论. 第 6 节进行总结并对未来工作加以展望.

## 2 相关工作

本节将分别从软件缺陷定位和 DNN 缺陷检测及调试两个方面对已有研究工作进行总结.

### 2.1 软件缺陷定位

软件缺陷定位的目标是识别与软件缺陷相关的程序元素, 它可以帮助程序开发人员更快、更准确地发现程序中需要修复的地方, 以提升程序的可靠性. 早期以人工调试为主的缺陷定位不仅耗时长, 而且难度大. 为此, 研究人员提出了多种不同类型的缺陷定位方法.

#### 2.1.1 基于频谱的缺陷定位方法

基于频谱的缺陷定位方法是目前传统软件缺陷定位研究中的重要方法<sup>[12]</sup>, 它通过运行大量测试用例, 获取程序的测试覆盖信息和测试结果来计算每个程序实体的怀疑度, 根据怀疑度定位有缺陷的实体.

程序频谱<sup>[13]</sup>指程序执行过程中产生的关于程序语句等元素的覆盖情况, 以及执行是否通过的信息. 在使用测试用例集对程序进行测试的过程中, 收集程序频谱信息, 并代入怀疑度公式计算程序实体对应的怀疑度, 怀疑度越高说明程序实体越有可能存在缺陷. 常用的怀疑度公式有 Ochiai<sup>[14]</sup>, D\*<sup>[15]</sup>等. Wong 等人<sup>[16]</sup>针对执行信息提出了假设: 一条语句包含缺陷的概率和它被失败用例执行的次数成正相关. 基于频谱的缺陷定位方法对测试用例的要求比较严格, 测试用例需要足够多以防止语句的覆盖不够全面, 但也不能过多, 否则会使程序语句重复执行, 影响最后的定位精度<sup>[17]</sup>.

#### 2.1.2 基于变异的缺陷定位方法

基于变异的缺陷定位方法将变异分析<sup>[18]</sup>和缺陷定位相结合来定位缺陷所在位置. 变异分析通过对程序的源代码进行变异操作生成变异体, 然后在变异体上执行测试用例获取执行信息和结果, 最后结合程序结构进行分析. 基于变异的缺陷定位方法考虑一条语句的执行是否会影响程序的执行结果<sup>[19]</sup>, 如果一条程序语句的执行影响的失败测试用例较多而成功测试用例较少, 那么这条语句就更有可能存在缺陷, 其中失败测试用例是指测试输出与测试预言 (test oracle) 不一致的用例, 而成功测试用例是指测试输出与测试预言一致的用例. 如果一条测试用例在变异体和原始程序上执行得到了不同的结果, 那么就说这个变异体被“杀死”了<sup>[20]</sup>. 可以“杀死”变异体的测试用例可能带有帮助定位缺陷的信息, 基于变异的缺陷定位方法通过生成变异体执行测试用例来收集这些信息. MUSE<sup>[21]</sup>和 Metallaxis-FL<sup>[22]</sup>是两种经典的基于变异的缺陷定位方法.

#### 2.1.3 基于程序切片的缺陷定位方法

程序切片由 Weiser 等人<sup>[23]</sup>提出, 是基于特定需求所截取的程序片段集合. 程序切片可分为静态切片和动态切片. 静态切片由输出语句和与输出语句有关的所有变量构成, 通常和整个程序规模相近, 因此使用静态切片技术定位缺陷效率不高. 动态切片由某一具体输入和与之相关的输出语句和变量构成, 与静态切片相比, 动态切片规模更小, 效率更高. 基于切片的定位技术优点在于可以简化程序, 降低了程序的复杂度, 提高了缺陷定位的精度. 但是当被测软件规模比较大时, 程序切片的效率会比较低<sup>[24]</sup>.

#### 2.1.4 基于模型的缺陷定位方法

Reiter 等人<sup>[25]</sup>最早提出基于模型的缺陷定位方法, 该方法基于 3 元组 (SD, COMPS, OBS). 其中 SD 表示对源

程序的描述, *COMPS* 表示程序构件的集合, *OBS* 表示根据输入得到的观测结果. 该类方法通过代码结构和上下文依赖关系构造缺陷模型, 因为充分利用了程序的执行信息, 所以定位效果较好<sup>[26]</sup>. 但是建立模型需要大量的数学知识, 而且过程复杂, 耗时较多.

上述缺陷定位方法主要针对传统软件, 传统软件的编程范式与 DNN 的编程范式具有显著差异<sup>[27]</sup>. 因此这类缺陷定位方法不能直接应用于 DNN 中, 需要针对性地设计 DNN 缺陷定位方法.

## 2.2 DNN 缺陷检测及调试

### 2.2.1 DNN 缺陷

Ma 等人<sup>[28]</sup>将神经网络的缺陷分成两种: 结构缺陷和训练缺陷. 结构缺陷是由不恰当的模型结构引起的, 例如由神经网络模型中的隐藏层数、每层中的神经元数目和神经元连通性等不恰当所致. 训练缺陷是由于训练过程或训练数据集中的偏差导致的, 例如 DNN 模型中一些权重没有得到良好的训练. Glorot 等人<sup>[29]</sup>发现, 训练集中的一些输入可能彼此冲突, 即两个输入或两批输入对权重的更新方向可能相反. 神经网络中的缺陷可能是由不恰当的权重或权重集导致的, Zhang 等人<sup>[30]</sup>将这些权重称为训练不良的权重 (ill-trained weights), 它们可能会导致神经网络输出错误. 训练集质量不高或者训练过程中的过拟合或者欠拟合会导致神经网络中的权重因训练不良而存在缺陷. 不恰当的权重会导致神经元的输出存在异常, 并经过所连接的边及对应权重向后传递, 因此不恰当的权重可能会导致最后的输出结果存在错误. 类似的, 不恰当的激活函数也可能使神经元的输出存在异常, 从而导致最终结果存在错误. 本文中讨论的缺陷神经元是指因为权重、激活函数等影响因素导致的存在异常输出的神经元.

### 2.2.2 DNN 缺陷检测

DNN 结构复杂, 且相对使用者来说是一个黑盒, 数据中的微小变化即有可能对其最终结果产生明显影响. 随着 DNN 被更多地应用于安全性要求高的场景中, 甚至有时会代替人做出重要的决定, 如何保障其可靠性就成为一个重要的问题.

目前研究人员针对 DNN 的缺陷检测已经展开了深入研究, 如蜕变测试、变异测试和 DNN 的覆盖准则等. Dwarakanath 等人<sup>[31]</sup>使用蜕变测试来检测图片分类器中无实现上的错误. Murphy 等人<sup>[32]</sup>实现了一个自动化蜕变测试的方法, 使之可以处理大规模的复杂数据集, 该方法对真实的程序缺陷也有效. Cheng 等人<sup>[33]</sup>使用  $\mu$ Java<sup>[34]</sup>对 Weka 平台<sup>[35]</sup>上的决策树和支持向量机等算法进行变异, 然后对它们进行蜕变测试, 也取得了较好的结果. Kim 等人<sup>[36]</sup>提出了一种深度学习系统测试充分性准则, 它定量地衡量深度学习系统相对于其训练数据行为的差异, 可以作为评价深度学习系统对未知输入做出反应的指标. Wang 等人<sup>[37]</sup>也提出了一种针对 DNN 的路径测试准则.

还有一些工作主要关注于 DNN 缺陷的实证研究. Zhang 等人<sup>[38]</sup>对 TensorFlow<sup>[39]</sup>中机器学习项目的错误进行分类, 总结了 7 种导致错误的原因, 并发现由于神经网络内部元素之间的相互依赖关系, 传统的调试技术 (例如切片) 几乎没有帮助, 因此需要新的调试研究技术. Sun 等人<sup>[40]</sup>研究了 3 个开源项目 Scikit\_learn<sup>[41]</sup>、Paddle<sup>[42]</sup>和 Caffe<sup>[43]</sup>, 结合软件工程中程序错误的常见分类和有关机器学习程序的特征, 把其中出现的错误分成了 7 个类别.

### 2.2.3 DNN 缺陷定位及修复

Pham 等人<sup>[44]</sup>通过检查交叉实现是否一致来检测深度学习库中是否存在错误, 然后使用异常传播跟踪和分析的方法来定位导致错误的深度学习库中的函数. 但是这种方法只能定位 DNN 模型在某一学习库上与实现相关的缺陷, 而不能定位到 DNN 本身存在的缺陷.

现有的 DNN 缺陷修复方法主要是通过重训练或直接修改的方式来调整权重, 以达到修复的目的. Ma 等人<sup>[28]</sup>使用差分热图来度量 DNN 中特征对分类结果的重要性, 找出对于错误分类影响较大的神经元, 然后使用类似于回归测试中输入选择的算法选择对这些神经元有较大影响的新样本进行重训练来调整权重. 但是这种方法只适用于模型中存在的欠拟合或者过拟合问题, 即训练过程中出现的缺陷. Zhang 等人<sup>[30]</sup>提出了一种权重自适应方法 Apricot 来修复训练不良的权重, 通过在原始训练集的许多不同子集上进行训练得到简化深度学习模型 (reduced deep learning model, rDLM), 使用其权重来辅助修改原始模型中的权重以达到修复的目的. 他们还发现要隔离一个权重 (或一组权重) 对 DNN 模型的性能 (例如准确性) 的影响比较困难, 暂时没有相关工作能准确定位训练不良的

权重. 因为其不能定位训练不良的权重, 该方法针对所有权重进行修改, 效率较低, 此外该方法在使用中需要训练大量 rDLM, 开销较大, 且权重之间的更新方向可能相反, 会影响到修复的结果. 本文所提出的基于频谱的 DNN 缺陷定位方法 Deep-SBFL, 可用于定位训练不良的权重所在的神经元, 为 DNN 的修复工作提供指导信息, 以有针对性的构造数据集进行重训练, 或隔离出需要修改的权重, 最终可提高修复效率.

3 基于频谱的 DNN 缺陷定位方法

本节将从方法框架、怀疑度公式和方法具体实现 3 个方面详细介绍所提出的基于频谱的 DNN 缺陷定位方法 Deep-SBFL.

3.1 方法框架

DNN 模型中每层包含一组神经元, 两个连续层中的神经元通过边连接起来, 每条边上都有表征其连接强度的权重. 神经元间可以通过带权重的边来传递信号. 两个连续层间的连接关系由一个矩阵表示, 矩阵中每个单元的值对应神经元间权重. 训练神经网络模型的本质就是更新权重, 以便最后一层神经元可以产生正确的预测输出. 反向传播是用于训练的核心方法, 它根据权重参数计算输出损失的梯度. 直观地讲, 较大的梯度表示如果更改了相应的权重参数, 结果将产生大的变化. 在反向传播中, 梯度会向后反馈以更新权重矩阵. 所以如果矩阵的权重没有被恰当的更新, 那么这样的模型给出的结果就会出现错误. 除了权重以外, DNN 的结构和其所使用的激活函数也会对结果产生较大影响. DNN 隐藏层的数目越多, DNN 越能学习到抽象的特征, 但是当层数过多时, DNN 学习到的特征会变得过于抽象而导致过拟合. 激活函数也一样, 选择合适的激活函数可以使神经网络的效果更好. 但是无论是权重还是激活函数, 最后都会作用在神经元的输出上, 然后再作为输入传入下一层神经元或者作为最终结果.

基于上述特征, 本文提出了基于频谱的 DNN 缺陷定位方法 Deep-SBFL, 其框架如图 1 所示, 主要包括 3 个步骤: (1) 将测试用例集输入待测模型, 收集模型预测过程中神经元的输出和预测结果作为频谱信息; (2) 进一步对频谱信息进行分析 and 计算, 得出贡献信息来量化模型中各神经元对预测结果的影响; (3) 将上一步中得出的贡献信息代入到怀疑度公式中进行计算, 然后对怀疑度进行归一化并排序, 得到最终的缺陷定位结果.

3.2 怀疑度公式

与神经元覆盖率不同, 神经元频谱不是简单统计在某一测试数据集下, 神经元是否被激活过, 也不像传统基于频谱的缺陷定位方法那样统计神经元的覆盖数. 在本方法中, 通过提取贡献信息来量化神经元对结果的贡献程度. 在 Deep-SBFL 中, 可以正向计算贡献信息 (以下简称正向计算) 和反向计算贡献信息 (以下简称反向计算). 在获取贡献信息之前, 假定已经获得了各条测试数据的预期输出, 每条测试数据及其预期输出组成为一条测试用例. 在执行测试用例集的过程中, 若是 DNN 的输出与测试用例的预期输出不一致, 则称该测试用例为负效测试用例, 反之则称为正效测试用例.

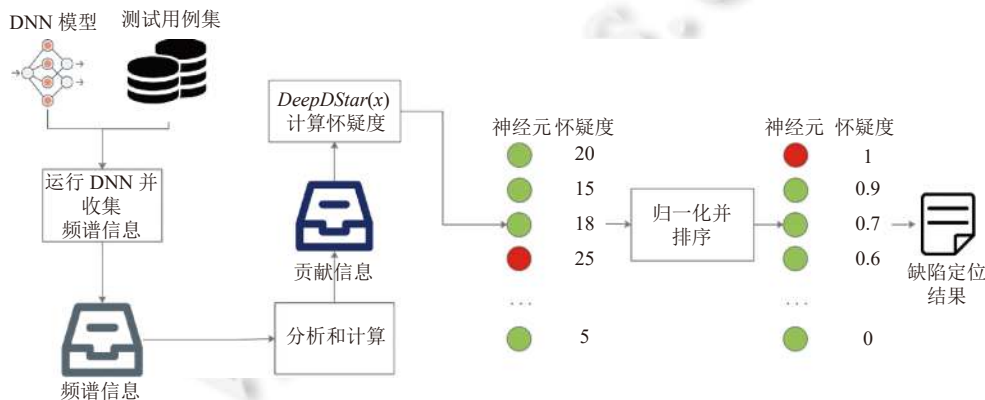


图 1 基于频谱的深度神经网络缺陷定位方法框架图

在正向计算中, 获取每个测试用例执行时, 各个神经元的输出作为用例贡献值, 然后分别累加单个神经元在正效测试用例和负效测试用例下的输出. 正效测试用例下的输出之和代表了神经元对于结果正确所做的贡献, 称之为正贡献; 负效测试用例下的输出之和代表了神经元对于结果错误所做的贡献, 称之为负贡献.

另一种是反向计算, 它从测试集的输出结果出发, 首先获取每一个测试用例的输出结果, 然后计算与该输出相连的权重各自所占的比例, 最后将该输出与各个权重所占的比例相乘, 作为上一层的神经元对下一层的神经元所做的用例贡献值. 之后与正向计算相似, 分别累加单个神经元在正效测试用例和负效测试用例下的贡献值, 正效测试用例下的贡献值之和代表了神经元对于结果正确所做的贡献, 为正贡献, 负效测试用例下的贡献值之和代表了神经元对于结果错误所做的贡献, 为负贡献.

对于神经元来说, 正贡献所对应的数值越大, 代表该神经元对结果正确所做的贡献越大, 即该神经元存在缺陷的概率越小, 反之, 负贡献所对应的数值越大, 代表该神经元对结果错误所做的贡献越大, 即神经元存在缺陷的概率越大. 这种关系与传统的基于频谱的缺陷定位技术中程序频谱与怀疑度的关系类似. 我们将传统频谱测试中的怀疑度公式进行了调整, 使其能够应用于定位 DNN.

DStar<sup>[15]</sup>是基于频谱的软件缺陷定位中效果最好的怀疑度公式之一<sup>[16]</sup>. 在其基础上, 我们设计了面向深度神经网络的缺陷定位公式 *DeepDStar*, 如公式 (1) 所示. 其中,  $x$  代表神经元,  $n_{ef}(x)$  为对应神经元的负贡献,  $n_{ep}(x)$  为对应神经元的正贡献. 因为深度神经网络的权重存在负值, 所以在计算贡献信息的时候也会出现负值, 当  $n_{ef}(x)$  的值为负时, 认为神经元  $x$  在负效用例中对于结果正确起到了正向作用, 应该将其取绝对值加到  $n_{ep}(x)$  上, 同样的, 当  $n_{ep}(x)$  为负时, 应该将其取绝对值加到  $n_{ef}(x)$  上, 以解决公式中存在的负值问题. 当  $n_{ep}(x)$  为正值,  $n_{ef}(x)$  为负值时, 认为  $x$  在正效用例和负效用例中都对于结果正确起到了正向作用, 因此怀疑度为 0.

$$DeepDStar(x) = \begin{cases} n_{ef}(x)^* / (n_{ef}(x) + n_{ep}(x)) & * = 2, 2.5, \dots, n_{ef}(x) \geq 0, n_{ep}(x) \geq 0 \\ 0 & * = 2, 2.5, \dots, n_{ef}(x) < 0, n_{ep}(x) \geq 0 \\ (n_{ef}(x) - n_{ep}(x))^* / (n_{ef}(x) - n_{ep}(x)) & * = 2, 2.5, \dots, n_{ef}(x) \geq 0, n_{ep}(x) < 0 \\ (-n_{ep}(x))^* / (-n_{ef}(x) - n_{ep}(x)) & * = 2, 2.5, \dots, n_{ef}(x) < 0, n_{ep}(x) < 0 \end{cases} \quad (1)$$

### 3.3 方法实现

DNN 模型可定义为一个 5 元组  $D(N, L, E, A, W)$ , 其中  $N = \{n_1, n_2, n_3, \dots\}$  为 DNN 中神经元的集合,  $L = \{l_1, l_2, l_3, \dots\}$  为 DNN 中层的集合 (含输入层、隐藏层和输出层),  $E = \{e_{1,2}, e_{1,3}, e_{1,4}, \dots\}$  为 DNN 中边的集合, 其中  $e_{i,j}$  为连接  $n_i$  和  $n_j$  的一条边,  $A = \{a_1, a_2, a_3, \dots\}$  为 DNN 中激活函数的集合,  $W = \{w_{1,2}, w_{1,3}, w_{1,4}, \dots\}$  为 DNN 中权重的集合,  $w_{i,j}$  即为边  $e_{i,j}$  的权重. 假设有测试用例集  $T = \{t_1, t_2, t_3, \dots\}$  和一个待测 DNN 模型  $D$ , 对于测试用例  $t_i \in T$ ,  $D$  的整体输出为  $f_D(t_i)$ ,  $f_{n_j}(t_i)$  为对应神经元  $n_j$  的输出.

基于频谱的深度神经网络缺陷定位法如算法 1 所示. 算法 1 的输入包括待测 DNN 模型  $D$  和测试用例集  $T$ , 输出为针对待测 DNN 模型的缺陷定位结果  $R$ , 即神经元按可疑度降序排列的序列. 首先输入测试用例集  $T$  运行待测 DNN 模型  $D$ , 获取每条测试用例的运行信息 *excute\_result* (第 2 行), 然后遍历  $N$  中所有神经元, 判断当前测试用例是否为负效测试用例, 若是, 则累加当前测试用例下的输出至  $n_{ef}(n)$  即负贡献 (第 5 行), 否则累加当前测试用例下的输出至  $n_{ep}(n)$  即正贡献 (第 7 行). 最后, 遍历  $N$  中所有神经元, 将  $n_{ef}(n)$  和  $n_{ep}(n)$  代入 *DeepDStar* 中计算怀疑度 (第 13 行), 对怀疑度集合进行归一化和排序后得出最终排序结果  $R$ .

**算法 1.** 深度神经网络的缺陷定位算法.

输入: DNN 模型  $D$  和测试用例集  $T$ ;

输出: 神经元序列  $R$ .

```

1. for each  $t$  in  $T$ :
2.   excute_result ← excute( $D$ ,  $t$ )
3.   for each  $n$  in  $N$ :
```

---

```

4.   if (excute_result=false)
5.        $n_{ej}(n) += f_n(t)$ 
6.   else
7.        $n_{ep}(n) += f_n(t)$ 
8.   end if
9.   end for
10. end for
11.  $R \leftarrow []$ 
12. for each  $n$  in  $N$ :
13.    $R[n] \leftarrow \text{DeepDStar}(n_{ej}(n), n_{ep}(n))$ 
14. end for
15.  $R \leftarrow \text{rank}(\text{normalize}(R))$ 

```

---

深度神经网络的贡献信息是在模型的运行过程中提取的,用来量化分析各个神经元为每条测试数据做出预测结果贡献程度的信息。每个神经元的贡献信息包含两个数据,即正贡献和负贡献。除了权重的变化会影响到神经元的输出外,无论是激活函数还是神经网络结构的缺陷,最终都会表现在神经元上,如修改或删除激活函数会使神经元对下一层的输出发生变化。神经网络结构的缺陷,如多了一层神经元或是少了一层神经元亦会使神经元的输出发生变化。因此本方法主要在神经元粒度进行缺陷定位,即识别包含缺陷的神经元所在位置(如权重异常增大,则应定位到该权重所连接的输出位置)。

在 Deep-SBFL 中,可使用正向计算和反向计算两种方式来获取神经元的贡献信息。正向计算使用输入数据和权重按照神经网络的计算顺序从前向后计算正贡献和负贡献信息,分别为各神经元对应所有成功测试数据的输出之和与所有失败测试数据的输出之和。反向计算使用输出结果和权重从后往前计算获取贡献信息。首先计算神经元中各个权重所占权重之和的比例,然后从输出层开始,将每层中各个神经元的输出分别乘以与之相连的权重所占的比例,然后将对应权重得到的结果求和得到每个权重的贡献信息,最后将神经元中所有权重的贡献信息求和作为神经元的贡献信息。

### 3.3.1 深度神经网络缺陷定位正向计算具体实现

(1) 对于待测 DNN 模型  $D$ ,从  $D$  中提取出神经元集合  $N$ ,然后输入测试用例集  $T$  运行该神经网络,获取运行测试数据  $t_i$  ( $t_i \in T$ ) 时,各个神经元  $n_j$  ( $n_j \in N$ ) 的输出  $f_{n_j}(t_i)$  及该条测试用例的预测结果  $f_D(t_i)$ ,若该结果与  $t_i$  的预期输出相同则  $t_i$  为正效测试用例,反之则为负效测试用例。

(2) 待测试用例集  $T$  全部运行完成后,计算  $N$  中所有神经元对于全部正效测试用例的输出之和与全部负效测试用例的输出之和,作为神经元的贡献信息。

(3) 将第 (2) 步中获得的  $N$  中所有神经元对应的贡献信息代入公式 (1),得出每个神经元对应的怀疑度,然后分别在不同层内对神经元的怀疑度做归一化处理。进行归一化的原因是在某一层内排名靠前的神经元存在缺陷的可能性比较大,但是不同层间的神经元输出值范围大小有差异,若不进行归一化处理,当存在缺陷的神经元所在层的输出整体偏小时可能会被排到后面。最后将归一化后的所有神经元的怀疑度统一进行排序,排名靠前的神经元即为最有可能存在缺陷的神经元。

### 3.3.2 深度神经网络缺陷定位反向计算具体实现

(1) 对于待测神经网络  $D$ ,从  $D$  中提取出神经元集合  $N$ ,获取  $D$  的权重集合  $W$ 。对于  $n_j$  ( $n_j \in N$ ),计算  $n_j$  中各个权重所占的比例。权重的比例为当前权重占与之相连的神经元对应前一层的所有权重之和的百分比。因为想要得到各个权重分别对最终结果影响的占比,因此将指定权重与权重之和相除得到该权重所占比例,而不使用权重的绝对值之和。权重所占比例可能会出现负值,这表示在对应用例的计算过程中该权重造成的效果与最终结果相反。之后输入测试用例集  $T$  运行该神经网络,获取每条测试数据  $t_i$  ( $t_i \in T$ ) 运行时,各个神经元  $n_j$  ( $n_j \in N$ ),的输出  $f_{n_j}(t_i)$

及该条测试用例的预测结果  $f_D(t_i)$ , 若该结果与  $t_i$  的预期输出相同则  $t_i$  为正效测试用例, 反之则为负效测试用例.

(2) 待测试用例集  $T$  全部运行完成后, 遍历测试结果  $f_D(t_i)$ , 分别将其中正效测试用例和负效测试用例对应的输出结果和与之相连的权重所占的比例相乘, 将对应权重得到的结果求和得到每个权重的贡献信息. 因为神经网络中权重有正值和负值, 在反向计算的过程中, 附加到每一个权重上的贡献信息也有正值和负值. 若正贡献或者负贡献出现负值, 则应该认为该权重起了相反的作用. 若将权重的贡献信息简单相加作为神经元的贡献信息则可能会导致该信息无法体现神经元所起的作用. 因此在计算神经元的贡献信息时需要对各权重的贡献信息进行判断, 如果其中存在负值, 则将其绝对值加到相反的贡献信息中. 最后得到  $N$  中每个神经元对应的贡献信息.

(3) 将第 (2) 步中获得的  $N$  中所有神经元对应的贡献信息代入计算公式 (1), 得出每个神经元对应的怀疑度. 与正向计算一样, 对不同层神经元的怀疑度进行归一化处理. 最后将归一化后的所有神经元怀疑度放在一起进行排序, 排名靠前的神经元即为最有可能存在缺陷的神经元.

3.4 示 例

如图 2 所示的简单神经网络, 有 2 个输入神经元 1.1 和 1.2, 3 个隐藏神经元 2.1、2.2 和 2.3 和 3 个输出神经元 3.1、3.2 和 3.3, 经由 Softmax 函数计算后得出最后的分类结果为 0、1 或 2. 神经网络中的权重和偏置的值如图 2 所示. 为方便展示, 图中展示的权重和怀疑度计算过程中的数值均保留 2 位小数. 该神经网络训练时所用的数据集为 3 个类别的二维坐标点和其对应的分类标签 0, 1, 2.

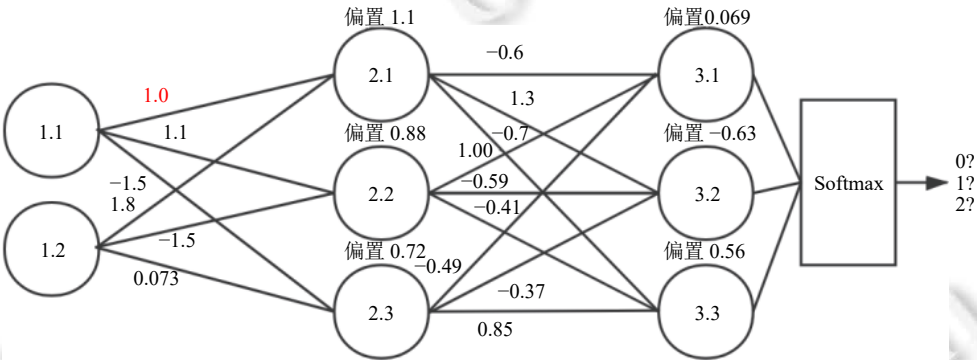


图 2 示例神经网络结构图

在该示例中, 假设植入一个缺陷, 即连接输入 1.1 和 2.1 的权重 1.0 被扩大 5 倍变成 5.0, 将造成神经元 2.1 存在缺陷. 使用差分测试<sup>[45]</sup>的思想来区分正效和负效测试用例. 将测试用例在原神经网络模型中得到的结果作为预期输出, 对于该测试用例, 若修改后的神经网络模型的输出与预期输出相同, 则该测试用例不能检测到植入缺陷, 为正效测试用例; 若修改后的神经网络模型的输出与预期输出不一致, 则该测试用例能检测到植入缺陷, 为负效测试用例. 对于不同类型的测试用例, 可以从运行过程中提取到相应的贡献信息.

我们选择了 4 条数据分别使用正向计算和反向计算来计算神经元的怀疑度. 如表 1 所示, #1 和#2 分别是分类结果为 1 的正效和负效测试用例, #3 和#4 分别是分类结果为 2 的正效和负效测试用例, 基于原始模型的预测结果为各测试用例在原始神经网络模型上得到的结果, 基于修改后模型的预测结果为各测试用例在修改后的神经网络模型上得到的结果. 假设公式 (1) 中参数\*的取值为 33, 即文献 [16] 中提出的效果最好的\*取值, 下面分别使用正向计算和反向计算来得出贡献信息.

表 1 示例测试数据

| 测试用例 | 数据             | 基于原始模型的预测结果 | 基于修改后模型的预测结果 |
|------|----------------|-------------|--------------|
| #1   | (0.31, 0.10)   | 1           | 1            |
| #2   | (0.33, -0.055) | 0           | 1            |
| #3   | (-0.30, 0.26)  | 2           | 2            |
| #4   | (-0.23, 0.33)  | 1           | 2            |

### 3.4.1 正向计算

正向计算首先需要获得每个神经元的输出,在该示例中直接使用神经网络测试时的运行数据,如表2所示,例如神经元 2.1 对测试用例#1 的输出为:  $0.31 \times 5 + 0.10 \times 1.8 + 1.1 \approx 2.8$ ,  $n_{ep}(2.1) = 2.8 + 0.068 \approx 2.9$ ,  $n_{ef}(2.1) = 2.7 + 0.54 \approx 3.2$ , 将其代入公式 (1) 中可得  $DeepDStar(2.1) = n_{ef}(2.1)^* / (n_{ep}(2.1) + n_{ef}(2.1)) = 3.2^{33} / (2.9 + 3.2) \approx 7.7 \times 10^{15}$ . 其余神经元的怀疑度计算同样按照上述过程,结果如表中“怀疑度”列所示,可以看出神经元 2.1 的怀疑度最高,即神经元 2.1 中最有可能存在缺陷,准确定位出缺陷所在位置.

表2 正向计算结果

| 神经元    | 测试用例贡献值 |       |       |       | $n_{ep}$ | $n_{ef}$ | 怀疑度                   | 排名 |
|--------|---------|-------|-------|-------|----------|----------|-----------------------|----|
|        | #1      | #2    | #3    | #4    |          |          |                       |    |
| 2.1    | 2.8     | 2.7   | 0.068 | 0.54  | 2.9      | 3.2      | $7.7 \times 10^{15}$  | 1  |
| 2.2    | 1.1     | 1.3   | 0.16  | 0.13  | 1.3      | 1.4      | $2.5 \times 10^4$     | 3  |
| 2.3    | 0.26    | 0.22  | 1.2   | 1.1   | 1.5      | 1.3      | $2.1 \times 10^3$     | 4  |
| 3.1    | -0.69   | -0.36 | -0.40 | -0.66 | -1.1     | -1.0     | $1.1 \times 10^1$     | 5  |
| 3.2    | 2.3     | 2.0   | -1.1  | -0.41 | 1.2      | 1.6      | $1.9 \times 10^6$     | 2  |
| 3.3    | -1.6    | -1.7  | 1.5   | 1.1   | -0.10    | -0.60    | $1.4 \times 10^{-33}$ | 6  |
| 测试用例类型 | 正效      | 负效    | 正效    | 负效    | —        | —        | —                     | —  |

### 3.4.2 反向计算

反向计算需要获得与对应神经元相连的各权重占总权重的比例,如连接神经元 3.2 与前一层神经元的权重分别为 1.3、-0.59 和 0.37,它们占总权重的比例分别为  $1.3 / (1.3 - 0.59 - 0.37) \approx 3.8$ 、 $-0.59 / (1.3 - 0.59 - 0.37) \approx -1.7$  和  $-0.37 / (1.3 - 0.59 - 0.37) \approx -1.1$ . 使用同样的方法计算出 DNN 中每个权重所占的比例,然后从输出结果开始反向计算每个神经元的频谱信息. 输出结果只考虑经 Softmax 函数后的最终分类结果. 以神经元 2.1 和测试用例#1 为例,测试用例#1 是一个预测结果为 1 的正效测试用例,使用预测结果为 1 的正效测试用例数量 (1) 乘以连接神经元 2.1 和输出 3.2 的权重所占的比例 (3.8),其贡献值为  $1 \times 3.8 = 3.8$ . 按照此方法计算执行其余测试用例时各神经元的贡献值. 最后计算 DNN 中每个神经元的贡献信息,以神经元 2.1 为例,将所有正效测试用例下的贡献值相加得到  $n_{ep}(2.1) = 3.8 + 2.7 = 6.5$ ,将所有负效测试用例下的贡献值相加得到  $n_{ef}(2.1) = 3.8 + 2.7 = 6.5$ . 代入公式 (1) 中可得  $DeepDStar(2.1) = n_{ef}(2.1)^* / (n_{ep}(2.1) + n_{ef}(2.1)) = 6.5^{33} / (6.5 + 6.5) \approx 5.1 \times 10^{25}$ . 对 DNN 中每个神经元计算怀疑度,结果如表3所示,神经元 2.1 的怀疑度最高,由此可知神经元 2.1 中最有可能存在缺陷,准确定位出缺陷所在位置.

表3 反向计算结果

| 神经元    | 测试用例贡献值 |      |      |      | $n_{ep}$ | $n_{ef}$ | 怀疑度                   | 排名 |
|--------|---------|------|------|------|----------|----------|-----------------------|----|
|        | #1      | #2   | #3   | #4   |          |          |                       |    |
| 2.1    | 3.8     | 3.8  | 2.7  | 2.7  | 6.5      | 6.5      | $5.1 \times 10^{25}$  | 1  |
| 2.2    | -1.7    | -1.7 | 1.6  | 1.6  | -0.1     | -0.1     | $5.0 \times 10^{-33}$ | 5  |
| 2.3    | -1.1    | -1.1 | -3.3 | -3.3 | -4.4     | -4.4     | $1.9 \times 10^{20}$  | 2  |
| 3.1    | 0       | 0    | 0    | 0    | 0        | 0        | 0                     | 6  |
| 3.2    | 1       | 1    | 0    | 0    | 1        | 1        | 0.5                   | 3  |
| 3.3    | 0       | 0    | 1    | 1    | 1        | 1        | 0.5                   | 3  |
| 测试用例类型 | 正效      | 负效   | 正效   | 负效   | —        | —        | —                     | —  |

## 4 实验设计

### 4.1 实验问题的设计

本文通过研究以下问题来验证所提出的缺陷定位方法 Deep-SBFL 和怀疑度计算公式  $DeepDStar$  在深度神经

网络上的有效性.

RQ1: 所提两种贡献信息计算方式对 DNN 中缺陷的定位效果如何? 对不同类型的缺陷, 不同计算方式的定位效果是否存在差异?

在 Deep-SBFL 中我们设计了正向计算和反向计算两种方式获取神经元的贡献信息, 为了探究两种方式的缺陷定位效果是否有差异, 我们设计了 RQ1, 在 Deep-SBFL 中分别使用正向计算和反向计算获取待测 DNN 中神经元的贡献信息, 然后使用相同的怀疑度计算公式计算其怀疑度并排序, 观察其定位效果, 另外也将分析两种计算方式对不同类型缺陷的定位效果是否有差异.

RQ2: 缺陷所在层的位置是否会影响 Deep-SBFL 的缺陷定位效果?

DNN 中可能存在很多层, 不同层中的缺陷对预测结果的影响不同, 因此也会影响到 Deep-SBFL 中对贡献信息的获取, 我们设计了 RQ2, 分析缺陷所在层的位置是否会对缺陷定位的效果产生影响.

RQ3: 测试数据是否会影响 Deep-SBFL 的缺陷定位效果?

本文中使用了差分测试的思想来区分正效和负效测试用例, 不同使用场景下, 正效测试用例和负效测试用例的数量会有不同. Deep-SBFL 通过正效测试用例和负效测试用例来计算贡献信息中的正贡献和负贡献, 然后将其代入怀疑度公式计算怀疑度, 测试数据中不同比例的负效测试用例可能会影响该方法缺陷定位的效果, 因此我们设计了 RQ3, 研究测试数据是否会影响缺陷定位的效果.

RQ4: DNN 的准确率是否会影响 Deep-SBFL 的缺陷定位效果?

DNN 的准确率是评价其性能的一个直观的指标, 缺陷会影响 DNN 的输出, 从而造成准确率差异, 我们通过设计 RQ4, 来研究 Deep-SBFL 在准确率不同的 DNN 上的缺陷定位效果.

## 4.2 实验对象和变异算子

我们使用 PyTorch 基于 MNIST 数据集训练了一个全连接神经网络模型, 该模型一共有 4 层, 有 784 个输入神经元, 10 个输出神经元, 中间有 2 个隐藏层, 分别有 500 个神经元, 使用激活函数为 ReLU, 该神经网络模型在 MNIST 的测试集中的准确率为 96.51%.

变异分析是指对程序代码执行变异操作生成变异体, 然后在变异体上执行测试用例, 获得执行结果, 最后结合程序结构分析测试用例的执行路径来优化或调试程序. 变异测试将变异分析应用于软件测试中, 根据程序的特性设计并应用变异算子为程序生成大量模拟程序错误的变异体<sup>[46]</sup>, 以评价测试用例集质量. 测试用例集“杀死”变异体的数量越多, 其质量越高. Andrews 等人<sup>[47]</sup>发现在软件测试研究中很难找到合适的真实缺陷程序来进行实验, 即使存在这样的程序, 其数量通常也较少. Xie 等人<sup>[48]</sup>发现变异测试中的变异与现实世界中的软件缺陷存在相似性. 类似地, Just 等人<sup>[49]</sup>分析了 5 个大型开源程序中的 357 个真实缺陷, 发现变异和真正的缺陷之间有很强的相关性. 因此, 许多研究人员使用由变异算子生成的变异体作为实验对象来评价测试方法<sup>[50-52]</sup>.

与传统软件的变异测试思想类似, 研究人员提出了针对深度学习程序的变异方法. Shen 等人<sup>[53]</sup>提出了一种针对 DNN 特点的变异方法 MUNN, 可以在一定程度上部分模拟 DNN 模型的缺陷, 以评估测试集的质量. Ma 等人<sup>[54]</sup>针对 DNN 上定义了一组变异算子, 将可能出现的缺陷和问题引入 DNN 中, 这些缺陷和问题可能出现在收集训练数据或实施训练计划的过程中. 一些针对深度学习程序的研究工作使用通过变异算子生成的变异体来验证所提测试方法的有效性<sup>[55-57]</sup>.

本文在实验中使用 DeepMutation<sup>[54]</sup>中定义的变异算子来生成变异体, 以模拟存在缺陷的 DNN 模型. 变异算子分为 source-level 和 model-level 两类. 其中, source-level 变异算子主要对训练程序或者数据进行变异, 改变其结构、超参数或者训练所用的数据集, 需要进行训练才能得到变异体, 这类变异算子可以模拟现实生活中程序员的错误导致的问题, 如少输入或者多输入了一行代码、参数小数点位置错误、误删部分数据集等, 最终训练得到的变异体模型的结构或者学习到的知识与理想模型不同, 存在缺陷, 导致性能下降. 而 model-level 变异算子则是对已经完成训练的模型进行变异, 如增加或删除层、改变神经元的激活状态、修改神经元内部的权重等, 这类变异算子虽然没有直接模拟程序员的错误, 但是也会改变模型的结构或者影响到神经元的输出. source-level 模型变异

算子生成变异体的代价较高,且最终都能以 model-level 的模型变异表现出来.因此,我们在实验中使用 model-level 变异算子来批量生成变异体,在待测神经网络的每一层随机生成变异体来模拟真实世界中的缺陷,以验证 Deep-SBFL 的效果.此外,通过对程序员行为的分析,我们发现程序员很容易出现输错小数点的错误,因此对 Gaussian Fuzzing 算子进行了调整,对权重进行一定程度的缩放以模拟这种错误,将该算子命名为 Weight Scaling.表 4 为实验中所用到的变异算子.

表 4 实验中所用变异算子

| 变异算子                      | 描述                                              |
|---------------------------|-------------------------------------------------|
| Weight scaling            | 选择一个神经元的权重对其进行一定程度的缩放                           |
| Weight shuffling          | 选择一个神经元的权重和一个和它有联系(connection)的神经元交换            |
| Neuron effect blocking    | 将一个神经元对下一层的影响阻断,通过将连接下一层的权重设为0,移除了神经元对整个神经网络的影响 |
| Neuron activation inverse | 逆转神经元的激活状态,通过在运用激活函数之前将输出反转(正变负,负变正)            |
| Neuron switch             | 交换同一层的两个神经元,也即是交换它们的角色和对下一层的影响                  |

为了回答所提出的研究问题,首先使用表 4 中所列出的变异算子对待测 DNN 的每一层随机选取权重或神经元进行变异,然后使用本文所提出的方法对其进行缺陷定位,观察其结果.具体的变异体生成情况如表 5 所示,共有 5 类变异体 weight scaling、weight shuffling、neuron effect blocking、neuron activation inverse 和 neuron switch,分别生成了 400、300、310、310 和 345 个变异体.为研究缺陷所在层是否对缺陷定位效果有影响,使用每个变异算子对所使用神经网络模型的每一层都随机生成 100 个变异体来做测试对象,其中由于该神经网络输出层只有 10 个神经元,在此情况下 neuron effect blocking 和 neuron activation inverse 变异算子在该层只能生成 10 个变异体,neuron switch 变异算子只能生成 45 个变异体.此外,weight shuffling 需要对相邻两层神经元中的权重进行交换,因此只能在相邻两层间生成变异体.

表 5 变异体生成情况

| 变异算子                      | 输入层 | 隐藏层1 | 隐藏层2 | 输出层 | 总计  |
|---------------------------|-----|------|------|-----|-----|
| Weight scaling            | 100 | 100  | 100  | 100 | 400 |
| Weight shuffling          | 100 | 100  | 100  | —   | 300 |
| Neuron effect blocking    | 100 | 100  | 100  | 10  | 310 |
| Neuron activation inverse | 100 | 100  | 100  | 10  | 310 |
| Neuron switch             | 100 | 100  | 100  | 45  | 345 |

### 4.3 评测指标

实验使用  $E_{\text{Inspect}}@n$ <sup>[58]</sup>和  $EXAM$ <sup>[59]</sup>来评价缺陷定位的效果.其中, $E_{\text{Inspect}}@n$ 为统计结果中排序列表的前  $n$  个位置内成功定位缺陷的数量.Xia 等人<sup>[60]</sup>发现程序开发者应该只检查排名靠前的位置,而  $E_{\text{Inspect}}@n$ 正好反映了这一特点,因此  $E_{\text{Inspect}}@n$  在实验中主要用来体现缺陷定位的效果.除了 neuron switch 外,其余变异算子都只对一个神经元进行操作,生成的变异体中也只有一个神经元中存在缺陷.对于使用 neuron switch 变异算子生成的变异体,在定位到第一个缺陷的时候, $n$  的取值越小则说明缺陷定位方法的效果越好,实验中  $n$  取不同的值来以观察缺陷定位方法针对不同类型变异体的定位效果. $EXAM$ 为在发现缺陷元素之前必须检查元素的百分比,它是另一种缺陷定位技术中的常用度量标准<sup>[61]</sup>. $EXAM$  的值越小说明缺陷定位方法的效果越好,在本实验中检查的元素为 DNN 中的神经元.

### 4.4 运行平台

本文中的所有实验均是在 CPU (i5-9300H), 16 GB 物理内存的 64 位 Windows 10 操作系统的计算机上进行的.本实验的开发和运行环境为 Python 3.6,训练和测试神经网络模型的机器学习库为 PyTorch.

## 5 实验结果及分析

我们使用差分测试<sup>[45]</sup>的方法来区分正效测试用例和负效测试用例,若在原始 DNN 模型和变异体上预测结果不同,则认为  $t_i$  是该变异体的负效测试用例,反之则为正效测试用例.对于差分测试不能发现输出和原模型存在差异的 DNN,视作不包含缺陷  $t_i$  的等价变异体,存在负效测试用例的变异体为非等价变异体.此外,如果 DNN 模型的准确率低于预期,通常可认为 DNN 模型存在缺陷<sup>[28]</sup>.在实验中,部分变异体虽然是非等价变异体,但其准确率与原始 DNN 模型差别过小,不存在明显缺陷.在本实验中,假设所生成的非等价变异体集合为  $M$ ,我们选择准确率差异超过 0.1% 的非等价变异体集合  $M'$  ( $M' \subseteq M$ ) 作为实验对象.

### 5.1 RQ1 实验结果

在 DNN 模型上生成变异体之后使用 Deep-SBFL 对所有变异体进行缺陷定位,将结果与真实缺陷位置进行对比,其中 *EXAM* 的情况如图 3 所示.横轴为  $M'$  中不同变异算子生成的变异体,纵轴为 *EXAM* 的值,每类变异体都分别使用正向计算和反向计算 2 种方式来进行缺陷定位.

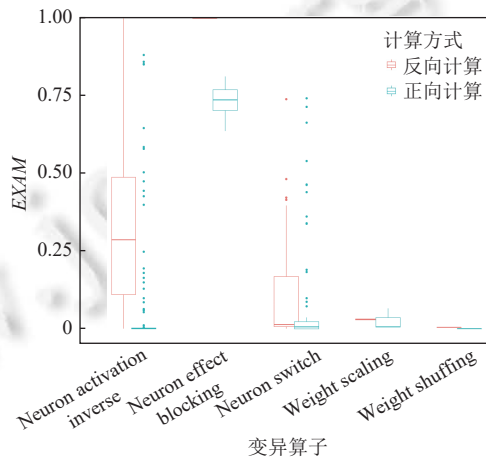


图3 各类变异体的 *EXAM* 情况

从图 3 中可以看出,正向计算对各种变异算子生成的变异体有着更好的缺陷定位效果,*EXAM* 的中位数更低.如图 3 所示,变异算子从左到右,正向计算的 *EXAM* 中位数相比反向计算分别低 0.285、0.263、0.008、0.024 和 0.004.其中,对于变异算子 neuron effect blocking 所生成的变异体,正向计算的 *EXAM* 中位数为 0.735,而反向计算的 *EXAM* 为 0.997,这两种计算方式对于该类变异体定位效果均较差,反向计算甚至要遍历所有神经元才能定位到缺陷所在位置.经分析后发现,因为变异算子 neuron effect blocking 阻断了神经元输出的传导,将其输出变为 0,因此其生成变异体在正向计算过程中,存在缺陷神经元的影响无法表现在神经元的输出数值上,反而使得其影响降低,导致正向计算无法获取该神经元的正确贡献信息,因而定位效果不理想.同样的,在反向计算过程中,因为缺陷神经元的输出为 0,其对于后续神经元的影响也消失了,导致在反向计算的过程中也没有足够的失败测试用例覆盖在缺陷神经元上,从而影响了缺陷定位效果.

为详细分析两种计算方式对不同类型的缺陷定位效果是否有差异,表 6 统计了不同的变异算子在不同  $E_{\text{Inspect}}@n$  和 *EXAM* 下的数据.在表 6 中,第 1 行 3~7 列为各变异算子及生成的变异体数量,如 neuron activation inverse (127) 表示  $M'$  中变异算子 neuron activation inverse 生成的变异体数量为 127;  $E_{\text{Inspect}}@1 \sim 100$  为在对应范围内可以定位到缺陷的变异体数量,如第 2 行第 3 列 86 (67.72%) 表示对  $E_{\text{Inspect}}@1$ ,正向计算定位到变异算子 neuron activation inverse 所生成变异体中的 86 个缺陷,占  $M'$  中该类变异体总数的 67.72%.

对  $E_{\text{Inspect}}@1$ ,正向计算一共定位到了 116 个缺陷,反向计算一共定位到了 10 个缺陷.正向计算时,对变异算

子 neuron activation inverse 生成变异体,能成功定位到缺陷的数量最多,为 86 个,占该类变异体总数的 67.7%;对变异算子 weight shuffling 生成变异体,成功定位 1 个缺陷.没有定位到变异算子 weight scaling 所生成变异体. Weight shuffling 和 weight scaling 变异算子生成变异体被成功定位的数量较少也与  $M'$  中该类变异体的总数较少(分别为 1 个和 3 个)有关.反向计算时,成功定位了 8 个变异算子 neuron switch 生成的变异体,占  $M'$  中该类变异体总数的 9.64%;成功定位了 2 个变异算子 neuron activation inverse 生成的变异体,占  $M'$  中该类变异体总数的 1.57%.

表 6 变异体在不同  $E_{\text{Inspect}@n}$  下的缺陷定位结果(总计 1 510 个神经元)

| $E_{\text{Inspect}@n}$ | 计算方式 | Neuron activation inverse (127) | Neuron effect blocking (18) | Neuron switch (83) | Weight scaling (3) | Weight shuffling (1) | 合计  |
|------------------------|------|---------------------------------|-----------------------------|--------------------|--------------------|----------------------|-----|
| $n=1$                  | 正向计算 | <b>86 (67.72%)</b>              | 0                           | <b>29 (34.94%)</b> | 0                  | <b>1 (100%)</b>      | 116 |
|                        | 反向计算 | 2 (1.57%)                       | 0                           | 8 (9.64%)          | 0                  | 0                    | 10  |
| $n=5$                  | 正向计算 | <b>100 (78.74%)</b>             | 0                           | <b>35 (42.17%)</b> | 0                  | <b>1 (100%)</b>      | 136 |
|                        | 反向计算 | 2 (1.57%)                       | 0                           | 11 (13.25%)        | 0                  | 0                    | 13  |
| $n=10$                 | 正向计算 | <b>102 (80.31%)</b>             | 0                           | <b>46 (55.42%)</b> | <b>2 (66.67%)</b>  | 1 (100%)             | 151 |
|                        | 反向计算 | 4 (3.15%)                       | 0                           | 20 (24.10%)        | 0                  | 1 (100%)             | 25  |
| $n=50$                 | 正向计算 | <b>103 (81.10%)</b>             | 0                           | <b>66 (79.52%)</b> | 2 (66.67%)         | 1 (100%)             | 172 |
|                        | 反向计算 | 10 (7.87%)                      | 0                           | 53 (63.86%)        | <b>3 (100%)</b>    | 1 (100%)             | 67  |
| $n=100$                | 正向计算 | <b>106 (83.46%)</b>             | 0                           | <b>67 (80.72%)</b> | 3 (100%)           | 1 (100%)             | 177 |
|                        | 反向计算 | 22 (17.32%)                     | 0                           | 56 (67.47%)        | 3 (100%)           | 1 (100%)             | 82  |

对  $E_{\text{Inspect}@5}$ , 正向计算一共定位到了 136 个缺陷, 反向计算一共定位到了 13 个缺陷. 正向计算仍然对变异算子 neuron activation inverse 生成变异体的缺陷定位效果最好, 能成功定位 100 个变异体中的缺陷, 占  $M'$  中该类变异体总数的 78.7%; 其次是变异算子 neuron switch, 增加了 6 个变异体被成功定位. 反向计算对变异算子 neuron switch 生成变异体的缺陷定位效果最好, 有 11 个变异体被成功定位, 占  $M'$  中该类变异体总数的 13.3%, 但与正向计算相比仍有较大差距. 其余变异算子所生成变异体被成功定位到的数量不变.

对  $E_{\text{Inspect}@10}$ , 正向计算一共定位到了 151 个缺陷, 反向计算一共定位到了 36 个缺陷. 正向计算仍然对变异算子 neuron activation inverse 生成变异体的缺陷定位效果最好, 与  $E_{\text{Inspect}@5}$  能定位的缺陷数量相比提升了 2 个; 对变异算子 weight scaling 生成变异体, 成功定位 2 个缺陷. 反向计算对变异算子 neuron switch 生成变异体定位到 20 个缺陷, 占  $M'$  中该类变异体总数的 24.1%. 对  $E_{\text{Inspect}@50}$ , 正向计算一共定位到了 172 个缺陷, 反向计算一共定位到了 67 个缺陷. 对  $E_{\text{Inspect}@100}$ , 正向计算一共定位到了 177 个缺陷, 反向计算一共定位到了 82 个缺陷.

Deep-SBFL 中正向计算比反向计算效果更好, 整体 EXAM 更低, 在  $E_{\text{Inspect}@n}$  ( $n=1 \dots 100$ ) 中成功定位的缺陷数量也更多. 反向计算对变异算子 neuron activation inverse、neuron switch、weight scaling 和 weight shuffling 生成的变异体进行缺陷定位的效果远不如正向计算. 在正向计算时, 对变异算子 neuron activation Inverse 生成的变异体,  $E_{\text{Inspect}@1}$  已经取得了较好的效果, 可以成功定位 67.7% 该类变异算子生成的变异体. 对于变异算子 Weight shuffling 生成的变异体, 正向计算在  $E_{\text{Inspect}@1}$  即可成功定位到缺陷, 而反向计算在  $E_{\text{Inspect}@10}$  才定位到缺陷. 总体上看, 正向计算相比反向计算有着更好的缺陷定位效果, 因此在后续问题中主要对正向计算进行分析.

对 RQ1 的回答: Deep-SBFL 中正向计算比反向计算的缺陷定位效果更好, 4 类变异算子所生成变异体定位的 EXAM 平均数均低于 0.08, 中位数均低于 0.01. Deep-SBFL 对不同类型缺陷的定位效果不同, 在正向计算时, 对变异算子 neuron effect blocking 生成变异体的定位效果较差, 对变异算子 neuron activation inverse 生成变异体的定位效果较好, 体现了 DNN 中不同类型缺陷的差异性.

## 5.2 RQ2 实验结果

为了分析缺陷所在的不同位置对定位效果的影响, 统计了不同变异算子在 DNN 模型各层的变异体下的平均 EXAM, 结果如图 4 所示. 其中, 输入层、隐藏层 1、隐藏层 2 和输出层的平均 EXAM 分别为 0.390、0.286、0.024 和 0.170. 相比较而言, 越靠近输出层, EXAM 的值越小, 定位效果也越好. 除了 neuron effect blocking 和 neuron

activation inverse 生成的变异体外, 其余情况下都是缺陷越靠近输出层越容易被定位到, 特别是变异算子 neuron switch 生成的变异体, 对于在后两层的缺陷神经元, 平均 *EXAM* 都非常低. 在 RQ1 中已经分析过变异算子 neuron effect blocking 生成变异体不能被 Deep-SBFL 准确定位的原因, 在这里更能明显看出, 该类变异体的平均 *EXAM* 高达 0.700. 对变异算子 neuron activation inverse 生成的变异体, 当缺陷位于隐藏层 2 时, 定位效果最好, 平均 *EXAM* 只有 0.002; 当缺陷位于输出层时,  $E_{\text{Inspect}}@100$  能定位到 4 个 (40%) 变异体, 但平均 *EXAM* 却高达 0.371. 这是因为在  $M'$  中, 使用该变异算子只能在输出层生成 10 个变异体, 除了被定位到的 4 个变异体以外, 剩余 6 个变异体的定位效果较差, 导致平均 *EXAM* 较高.

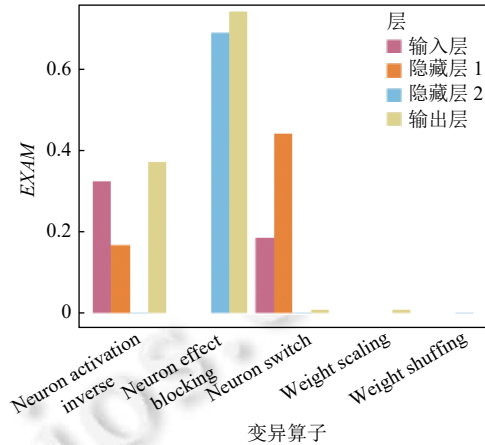


图 4 缺陷分布在不同层中的 *EXAM* 情况

对 RQ2 的回答: 缺陷深度神经网络中所在层的位置会影响缺陷定位效果. 实验数据表明, 在大部分情况下, 存在缺陷的神经元越靠近输出层越容易被准确定位.

### 5.3 RQ3 实验结果

为分析测试数据是否会影响该方法的定位效果, 生成统计  $M'$  中变异体的负效测试用例数占所有测试用例数的百分比及其对应缺陷定位结果. 其中, 图 5(a)–(e) 分别为变异算子 neuron activation inverse、neuron effect blocking、neuron switch、weight scaling 和 weight shuffling 所生成变异体的 *EXAM* 分布情况, 图 5(f) 为所有变异体的总体情况. 在图 5 中, 我们将变异体对应的负效测试用例比例分为 (0, 0.5%]、(0.5%, 1%] 等 7 个不同的区间. 如图 5 所示, 除变异算子 neuron effect blocking 所生成的变异体, 随着负效测试用例比例增加平均 *EXAM* 持续升高外, 其余各类变异体平均 *EXAM* 随着负效测试用例比例增加一直下降, 且当负效测试用例所占比例大于 1% 后, 平均 *EXAM* 大幅降低. 如 RQ1 中分析, 变异算子 neuron effect blocking 所生成的缺陷模型阻断了神经元输出的传导, 使影响不能向后传递, 因此此处的结果无法体现随着负效测试用例比例增大而变好.

变异算子 neuron activation inverse 和 neuron switch 所生成的变异体在不同比例负效测试用例下的 *EXAM* 有相似的走势. 变异算子 neuron activation inverse 所生成的变异体在负效测试用例所占比例为 (0, 0.5%) 和 (0.5%, 1%) 时, *EXAM* 较高, 但在负效测试用例所占比例超过 1% 后, *EXAM* 大幅下降, 中位数为 0.001. 变异算子 neuron switch 所生成的缺陷模型在负效测试用例所占比例为 (0, 0.5%) 和 (0.5%, 1%) 时 *EXAM* 中位数分别为 0.005 和 0.010. 在负效测试用例所占比例超过 1% 时, *EXAM* 中位数下降至 0.004.

由于变异算子 weight scaling 和 weight shuffling 只对神经元中的 1 个或 2 个权重进行变异, 对最终结果生成的影响较小. 在实验中, 它们所生成的变异体只有少量个体准确率差异超过 0.1%, 分别为 3 个和 1 个.  $M'$  中 weight scaling 所生成的 3 个变异体中有 2 个的负效测试用例所占比例在 (0, 0.5%) 间, *EXAM* 平均数为 0.036, 缺陷定位效果较好. 剩余 1 个变异体的负效测试用例所占比例在 (0.5%, 1%) 间, *EXAM* 为 0.007. Weight shuffling 所生成变异体准确率差异大于 0.1% 的只有一个, 负效测试用例比例在 (0, 0.5%) 间, *EXAM* 为 0.001, 缺陷定位效果较好.

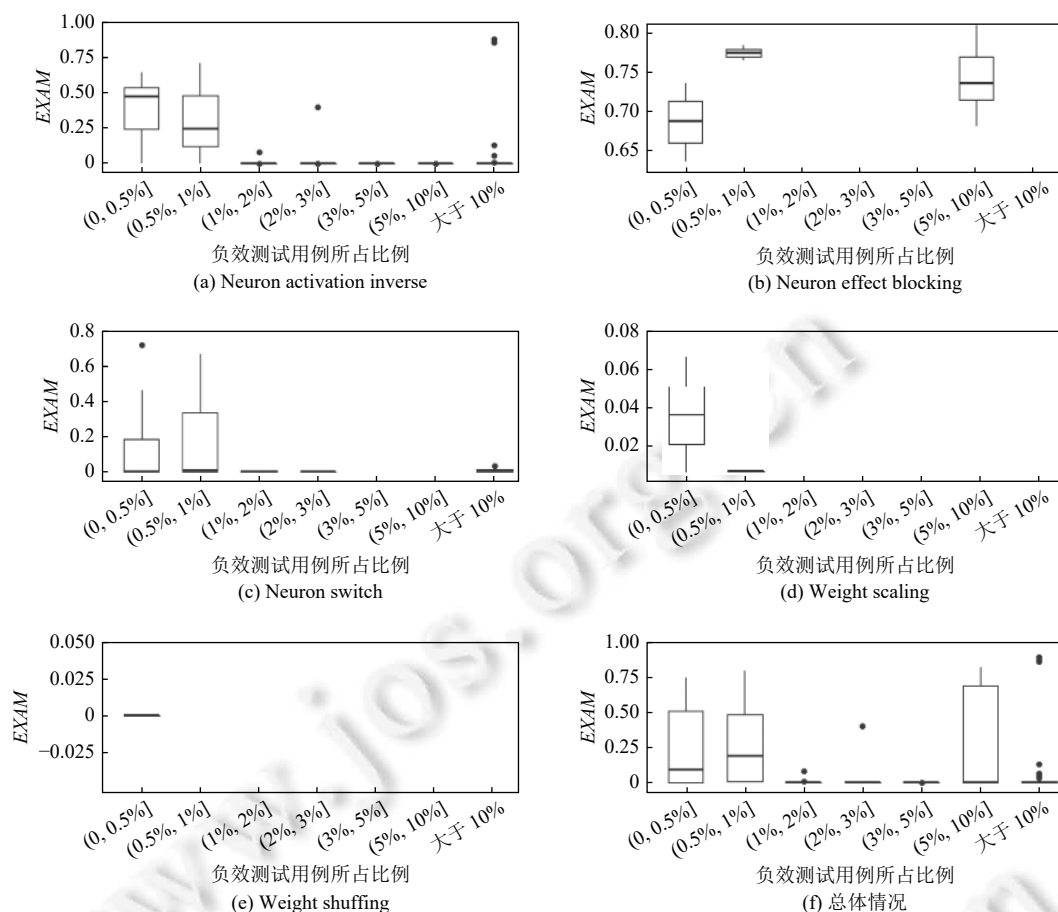


图5 负效测试用例对 DNN 模型缺陷定位 EXAM 的影响

总体上看,当测试集中负效测试用例所占比例越高,缺陷定位的效果越好.图5(f)中当负效测试用例所占比例在(0.5%, 1%]和(5%, 10%]时 EXAM 较高,这主要是因为  $M'$  中变异体在各区间的数量不太均衡, neuron effect blocking 变异算子所生成变异体在上述区间中的 EXAM 均较高,导致总体情况下对应区间的 EXAM 较高.在 RQ2 的实验中发现,越靠近输出层的缺陷越容易被定位.经过分析,靠近输出层的缺陷通常具有较多的负效测试用例.这是因为缺陷靠近输出层,能对结果产生更大的影响,从而有更多的测试用例能够检测到该缺陷.

对 RQ3 的回答:测试用例集中负效测试用例的比例会影响 Deep-SBFL 的缺陷定位效果.通常情况下,测试用例集中负效测试用例占的比例越大,Deep-SBFL 的缺陷定位效果就越好.

#### 5.4 RQ4 实验结果

图6统计了  $M$  中不同准确率 DNN 模型缺陷定位的 EXAM 分布情况,以分析准确率是否会影响 Deep-SBFL 的定位效果.其中,图6(a)–(e)分别为变异算子 neuron activation inverse、neuron effect blocking、neuron switch、weight scaling 和 weight shuffling 所生成变异体的 EXAM 分布情况,图6(f)为所有变异体的总体情况.在图6中,我们将变异体与原始 DNN 模型准确率的差异分为(0, 0.1%]、(0.1%, 0.2%]等11个不同的区间.可以看出,除对变异算子 neuron effect blocking 所生成变异体进行缺陷定位的 EXAM 随着准确率差异增加而升高外,其余各类变异体的 EXAM 随着准确率差异增加持续下降.

对于变异算子 neuron activation inverse 所生成变异体,当准确率差异在(0, 0.1%]间, EXAM 中位数为0.463,当准确率差异在(0.1, 0.2%]间, EXAM 中位数下降至0.248,当准确率差异超过0.2%后,各区间 EXAM 中位数均

低于 0.214, 当准确率差异超过 0.4% 后, 各区间 *EXAM* 中位数均低于 0.002. 对于变异算子 *neuron switch* 所生成的变异体, 当准确率差异在 (0, 0.1%] 间, *EXAM* 中位数为 0.208, 当准确率差异在 (0.1, 0.2%] 间, *EXAM* 中位数下降至 0.079, 当准确率差异超过 0.2% 后, *EXAM* 大幅下降, 各区间 *EXAM* 中位数均小于 0.005. 变异算子 *weight scaling* 和 *weight shuffling* 所生成的变异体数量较少, 只出现在部分区间中, 但依然可以看出当变异体与原始 DNN 模型准确率差异较小时, *EXAM* 较大, 差异较大时, *EXAM* 较小.

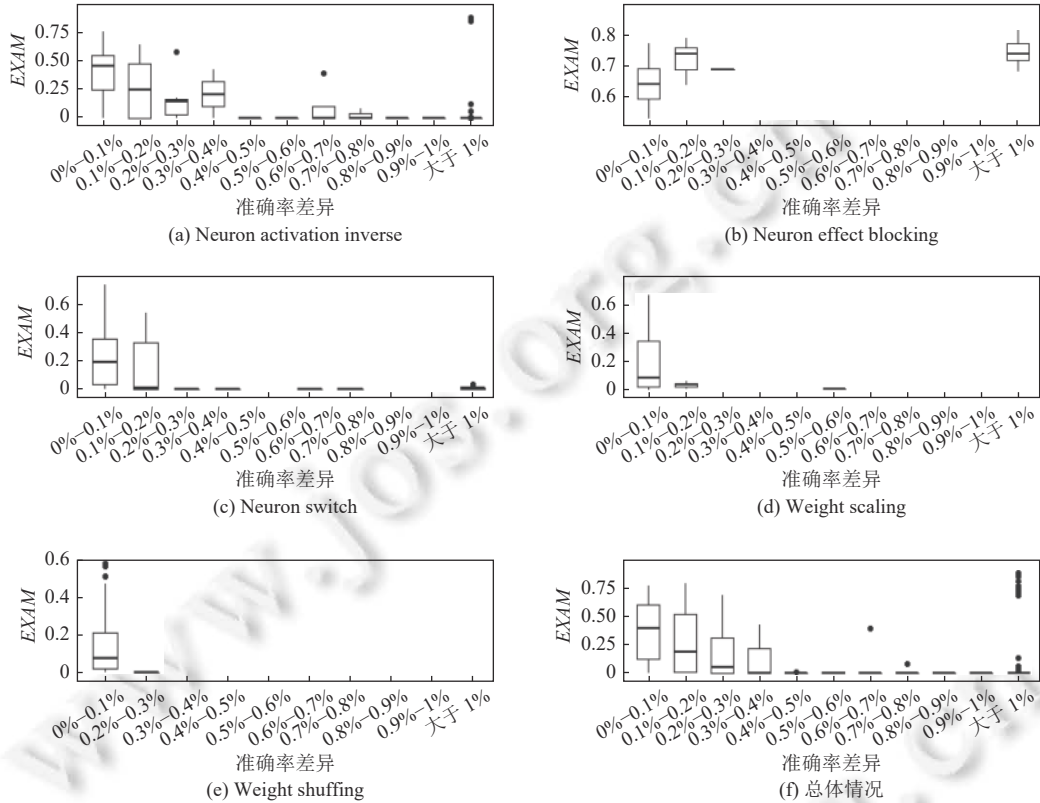


图 6 不同准确率 DNN 模型缺陷定位的 *EXAM* 分布情况

总体上看, 变异体相比原始 DNN 模型的准确率差异越大, 缺陷定位的效果就越好. 如图 6(f) 所示, 随着准确率差异的增大, 变异体的 *EXAM* 持续下降. 当准确率差异超过 0.4% 后, 在各区间中, 除少数异常点 (如 *neuron effect blocking* 所生成变异体), 其余变异体的 *EXAM* 中位数均小于 0.001. 因此, 变异体的准确率会影响 Deep-SBFL 的缺陷定位效果. DNN 模型的准确率与预期相差越大, 就越容易定位到其中的缺陷. 在 RQ1-RQ3 中, 我们选择准确率差异超过 0.1% 的变异体进行实验, 取得了较好的缺陷定位效果, 若选择准确率差异更大的变异体, Deep-SBFL 将取得更好的效果.

对 RQ4 的回答: DNN 模型的准确率会影响 Deep-SBFL 的缺陷定位效果. 通常情况下, DNN 模型的准确率与预期相差越大, Deep-SBFL 的缺陷定位效果就越好.

### 5.5 有效性分析

本节从内部有效性、外部有效性和构造有效性 3 个方面进行有效性分析.

本方法内部有效性的影响主要在于两个方面, 一是实验所使用的变异体是否可以模拟真实的深度神经网络中的缺陷, 本文实现了 DeepMutation 中定义的神经网络变异算子来生成神经网络变异体, 使用这些变异算子生成的变异体可以用来模拟神经网络中的真实缺陷. 二是怀疑度公式的合理性, 基于程序频谱的传统软件缺陷定位方法

中有很多怀疑度公式, Ochiai 和  $D^*$  等公式在不同场景中都取得过最优的性能<sup>[14,16]</sup>, 本文所采用的 *DeepDStar* 公式是受到  $D^*$  的启发设计的. 后续将验证其他公式在 Deep-SBFL 中的效果.

本方法外部有效性的影响主要在于所构建存在缺陷的 DNN 模型是否能代表真实的缺陷. 变异分析在 DNN 测试中是一种常用的方法<sup>[55-57]</sup>, 我们使用 DeepMutation 中的 neuron activation inverse、neuron switch 等多种变异算子来生成大量变异体, 以提高所生成变异体的多样性. 此外, 我们通过随机对 DNN 模型不同层中的组成元素进行变异, 以缓解缺陷所在位置可能会影响 Deep-SBFL 定位效果的威胁. 但是, 通过变异生成的实验对象仍有一定的局限性, 不能完全代表含有真实缺陷的 DNN 模型, 下一步我们将使用蜕变测试或者使用人工检测的方法对含有真实缺陷的 DNN 模型进行实验. 本实验中所使用的 DNN 模型为全连接神经网络 (fully connected neural network), 是一种常见的 DNN 模型, 被广泛用作实验对象<sup>[1,37,46]</sup>, 未来我们会使用更多不同类型的 DNN 模型进行实验.

本方法构造有效性的影响主要在于评价缺陷定位效果的评价指标, 本文采用了缺陷定位中常用的指标  $E_{\text{Inspect}@n}$  和 EXAM 来评价缺陷定位效果, 在各类缺陷定位方法中, 上述指标被广泛采用<sup>[19,46,58,61]</sup>.

## 6 总结和展望

本文分析了现有的软件缺陷定位和 DNN 缺陷检测方法, 并提出了一种基于频谱的深度神经网络缺陷定位方法 Deep-SBFL, 用来定位深度神经网络中存在缺陷的神经元. 具体来说, 该方法首先输入测试用例集来运行模型, 获取模型中神经元的贡献信息, 然后将各神经元对应的贡献信息代入怀疑度公式计算, 并根据怀疑度进行排序, 以定位存在缺陷的神经元. 基于上述方法, 本文在使用 MNIST 数据集训练的深度神经网络上进行了实验, 以验证 Deep-SBFL 的有效性. 实验结果表明, Deep-SBFL 在深度神经网络的缺陷定位上取得了一定的效果, 其中对变异算子 neuron switch 所生成变异体的定位效果最好, 83 个变异体的平均 EXAM 达 0.064. 对  $E_{\text{Inspect}@1}$ , 正向计算可以定位到 116 个缺陷. 此外, 实验还发现当负效测试用例越多、缺陷所在位置越靠近输出层或变异体准确率差异越大时, 越容易被 Deep-SBFL 成功定位.

在下一步工作中将考虑使用蜕变测试等方法来生成大量负效测试用例, 以提高缺陷定位的准确性. 其次, 怀疑度公式在本方法中占据重要的地位, 对其改进也是一个重要的研究方向. 再次, 尝试将本方法应用于 Apricot 等方法中, 以提升其修复 DNN 模型缺陷的效率. 最后, 基于变异的 DNN 缺陷定位方法也是未来我们研究的重点之一.

## References:

- [1] Litjens G, Kooy T, Bejnordi BE, Setio AAA, Ciompi F, Ghafoorian M, van der Laak JAWM, van Ginneken B, Sánchez CI. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 2017, 42: 60–88. [doi: 10.1016/j.media.2017.07.005]
- [2] Schneider S, Baevski A, Collobert R, Auli M. WAV2vec: Unsupervised pre-training for speech recognition. In: *Proc. of the 20th Annual Conf. of the Int'l Speech Communication Association*. Graz: ISCA, 2019. 3465–3469. [doi: 10.21437/Interspeech.2019-1873]
- [3] Vinayakumar R, Alazab M, Soman KP, Poornachandran P, Venkatraman S. Robust intelligent malware detection using deep learning. *IEEE Access*, 2019, 7: 46717–46738. [doi: 10.1109/ACCESS.2019.2906934]
- [4] Uber's self-driving cars were struggling before Arizona crash. 2018. <https://www.nytimes.com/2018/03/23/technology/uber-self-driving-cars-arizona.html>
- [5] Hackers say they've broken face ID a week after iPhone X release. <https://www.wired.com/story/hackers-say-broke-face-id-security/>
- [6] Chen TY, Cheung SC, Yiu SM. Metamorphic testing: A new approach for generating next test cases. *arXiv:2002.12543*, 1998.
- [7] Ma L, Juefei-Xu F, Zhang FY, Sun JY, Xue MH, Li B, Chen CY, Su T, Li L, Liu Y, Zhao JJ, Wang YD. DeepGauge: Multi-granularity testing criteria for deep learning systems. In: *Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering*. Montpellier: ACM, 2018. 120–131. [doi: 10.1145/3238147.3238202]
- [8] Xie XF, Ma L, Juefei-Xu F, Xue MH, Chen HX, Liu Y, Zhao JJ, Li B, Yin JX, See S. DeepHunter: A coverage-guided fuzz testing framework for deep neural networks. In: *Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*. Beijing: ACM, 2019. 146–157. [doi: 10.1145/3293882.3330579]
- [9] Pei KX, Cao YZ, Yang JF, Jana S. DeepXplore: Automated whitebox testing of deep learning systems. In: *Proc. of the 26th Symp. on Operating Systems Principles*. Shanghai: ACM, 2017. 1–18. [doi: 10.1145/3132747.3132785]

- [10] Odena A, Olsson C, Andersen DG, Goodfellow I. TensorFuzz: Debugging neural networks with coverage-guided fuzzing. In: Proc. of the 36th Int'l Conf. on Machine Learning. Long Beach: PMLR, 2019. 4901–4911.
- [11] Marijan D, Gotlieb A, Ahuja MK. Challenges of testing machine learning based systems. In: Proc. of the 2019 IEEE Int'l Conf. on Artificial Intelligence Testing. Newark: IEEE, 2019. 101–102. [doi: [10.1109/AITest.2019.00010](https://doi.org/10.1109/AITest.2019.00010)]
- [12] Collofello JS, Cousins L. Towards automatic software fault location through decision-to-decision path analysis. Int'l Workshop on Managing Requirements Knowledge, AFIPS, 1987. 539–544.
- [13] Harrold MJ, Rothermel G, Sayre K, Wu R, Yi L. An empirical investigation of the relationship between spectra differences and regression faults. Software Testing, Verification and Reliability, 2000, 10(3): 171–194. [doi: [10.1002/1099-1689\(200009\)10:3<171::AID-STVR209>3.0.CO;2-J](https://doi.org/10.1002/1099-1689(200009)10:3<171::AID-STVR209>3.0.CO;2-J)]
- [14] Wen WZ, Li BX, Sun XB, Li JK. Program slicing spectrum-based software fault localization. In: Proc. of the 23rd Int'l Conf. on Software Engineering & Knowledge Engineering. Miami Beach: Knowledge Systems Institute Graduate School, 2011. 213–218.
- [15] Wong WE, Debroy V, Li YH, Gao RZ. Software fault localization using DStar (D\*). In: Proc. of the 6th IEEE Int'l Conf. on Software Security and Reliability. Gaithersburg: IEEE, 2012. 21–30. [doi: [10.1109/SERE.2012.12](https://doi.org/10.1109/SERE.2012.12)]
- [16] Wong WE, Debroy V, Gao RZ, Li YH. The DStar method for effective software fault localization. IEEE Trans. on Reliability, 2014, 63(1): 290–308. [doi: [10.1109/TR.2013.2285319](https://doi.org/10.1109/TR.2013.2285319)]
- [17] Jiang B, Chan WK, Tse TH. On practical adequate test suites for integrated test case prioritization and fault localization. In: Proc. of the 11th Int'l Conf. on Quality Software. Madrid: IEEE, 2011. 21–30. [doi: [10.1109/QSIC.2011.37](https://doi.org/10.1109/QSIC.2011.37)]
- [18] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans. on Software Engineering, 2011, 37(5): 649–678. [doi: [10.1109/TSE.2010.62](https://doi.org/10.1109/TSE.2010.62)]
- [19] Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B. Evaluating and improving fault localization. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering. Buenos Aires: IEEE, 2017. 609–620. [doi: [10.1109/ICSE.2017.62](https://doi.org/10.1109/ICSE.2017.62)]
- [20] Zhang J, Zhang LM, Harman M, Hao D, Jia Y, Zhang L. Predictive mutation testing. IEEE Trans. on Software Engineering, 2019, 45(9): 898–918. [doi: [10.1109/TSE.2018.2809496](https://doi.org/10.1109/TSE.2018.2809496)]
- [21] Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. In: Proc. of the 7th IEEE Int'l Conf. on Software Testing, Verification and Validation. Cleveland: IEEE, 2014. 153–162. [doi: [10.1109/ICST.2014.28](https://doi.org/10.1109/ICST.2014.28)]
- [22] Papadakis M, Le Traon Y, Metallaxis-FL: Mutation-based fault localization. Software Testing, Verification and Reliability, 2015, 25(5–7): 605–628. [doi: [10.1002/stvr.1509](https://doi.org/10.1002/stvr.1509)] [doi: [10.1002/stvr.1509](https://doi.org/10.1002/stvr.1509)]
- [23] Weiser M. Programmers use slices when debugging. Communications of the ACM, 1982, 25(7): 446–452. [doi: [10.1145/358557.358577](https://doi.org/10.1145/358557.358577)]
- [24] Wen WZ, Li BX, Sun XB, Liu CC. Technique of software fault localization based on hierarchical slicing spectrum. Ruan Jian Xue Bao/Journal of Software, 2013, 24(5): 977–992 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/4342.htm> [doi: [10.3724/SP.J.1001.2013.04342](https://doi.org/10.3724/SP.J.1001.2013.04342)]
- [25] Reiter R. A theory of diagnosis from first principles. Artificial Intelligence, 1987, 32(1): 57–95. [doi: [10.1016/0004-3702\(87\)90062-2](https://doi.org/10.1016/0004-3702(87)90062-2)]
- [26] Chen X, Ju XL, Wen WZ, Gu Q. Review of dynamic fault localization approaches based on program spectrum. Ruan Jian Xue Bao/Journal of Software, 2015, 26(2): 390–412 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/4708.htm> [doi: [10.13328/j.cnki.jos.004708](https://doi.org/10.13328/j.cnki.jos.004708)]
- [27] Wang Z, Yan M, Liu S, Chen JJ, Zhang DD, Wu Z, Chen X. Survey on testing of deep neural networks. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1255–1275 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/5951.htm> [doi: [10.13328/j.cnki.jos.005951](https://doi.org/10.13328/j.cnki.jos.005951)]
- [28] Ma SQ, Liu YQ, Lee WC, Zhang XY, Grama A. MODE: Automated neural network model debugging via state differential analysis and input selection. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 175–186. [doi: [10.1145/3236024.3236082](https://doi.org/10.1145/3236024.3236082)]
- [29] Glorot X, Bengio Y. Understanding the difficulty of training deep feedforward neural networks. In: Proc. of the 13th Int'l Conf. on Artificial Intelligence and Statistics. Sardinia: JMLR.org, 2010. 249–256.
- [30] Zhang H, Chan WK. Apricot: A weight-adaptation approach to fixing deep learning models. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering. San Diego: IEEE, 2019. 376–387. [doi: [10.1109/ASE.2019.00043](https://doi.org/10.1109/ASE.2019.00043)]
- [31] Dwarakanath A, Ahuja M, Sikand S, Rao RM, Bose RPJC, Dubash N, Podder S. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 118–128. [doi: [10.1145/3213846.3213858](https://doi.org/10.1145/3213846.3213858)]
- [32] Murphy C, Shen K, Kaiser G. Automatic system testing of programs without test oracles. In: Proc. of the 18th Int'l Symp. on Software Testing and Analysis. Chicago: ACM, 2009. 189–199. [doi: [10.1145/1572272.1572295](https://doi.org/10.1145/1572272.1572295)]

- [33] Cheng DW, Cao C, Xu C, Ma XX. Manifesting bugs in machine learning code: An explorative study with mutation testing. In: Proc. of the 2018 IEEE Int'l Conf. on Software Quality, Reliability and Security. Lisbon: IEEE, 2018. 313–324. [doi: 10.1109/QRS.2018.00044]
- [34]  $\mu$ Java. <https://cs.gmu.edu/offutt/mujava/>
- [35] Weka. <https://www.cs.waikato.ac.nz/ml/weka/>
- [36] Kim J, Feldt R, Yoo S. Guiding deep learning system testing using surprise adequacy. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 1039–1049. [doi: 10.1109/ICSE.2019.00108]
- [37] Wang D, Wang ZY, Fang CR, Chen YS, Chen ZY. DeepPath: Path-driven testing criteria for deep neural networks. In: Proc. of the 2019 IEEE Int'l Conf. on Artificial Intelligence Testing. Newark: IEEE, 2019. 119–120. [doi: 10.1109/AITest.2019.00013]
- [38] Zhang YH, Chen YF, Cheung SC, Xiong YF, Zhang L. An empirical study on TensorFlow program bugs. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 129–140. [doi: 10.1145/3213846.3213866]
- [39] TensorFlow. <https://github.com/tensorflow>
- [40] Sun XB, Zhou TC, Li GJ, Hu JJ, Yang H, Li B. An empirical study on real bugs for machine learning programs. In: Proc. of the 24th Asia-Pacific Software Engineering Conf. Nanjing: IEEE, 2017. 348–357. [doi: 10.1109/APSEC.2017.41]
- [41] Scikit\_learn. <https://scikit-learn.org/stable/>
- [42] Paddle. <https://github.com/PaddlePaddle/Paddle>
- [43] Caffe. <http://caffe.berkeleyvision.org/>
- [44] Pham HV, Lutellier T, Qi WZ, Tan L. CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 1027–1038. [doi: 10.1109/ICSE.2019.00107]
- [45] McKeeman WM. Differential testing for software. Digital Technical Journal, 1998, 10(1): 100–107.
- [46] Naish L, Lee HJ, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Trans. on Software Engineering and Methodology, 2011, 20(3): 11. [doi: 10.1145/2000791.2000795]
- [47] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? In: Proc. of the 27th Int'l Conf. on Software Engineering. St Louis: ACM, 2005. 402–411. [doi: 10.1145/1062455.1062530]
- [48] Xie XY, Chen TY, Kuo FC, Xu BW. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. on Software Engineering and Methodology, 2013, 22(4): 31. [doi: 10.1145/2522920.2522924]
- [49] Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing? In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 654–665. [doi: 10.1145/2635868.2635929]
- [50] Granda MF, Condori-Fernández N, Vos TEJ, Pastor O. Effectiveness assessment of an early testing technique using model-level mutants. In: Proc. of the 21st Int'l Conf. on Evaluation and Assessment in Software Engineering. Karlskrona: ACM, 2017. 98–107. [doi: 10.1145/3084226.3084257]
- [51] Jahangirova G, Clark D, Harman M, Tonella P. Test oracle assessment and improvement. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 247–258. [doi: 10.1145/2931037.2931062]
- [52] Ahmed I, Gopinath R, Brindescu C, Groce A, Jensen C. Can testedness be effectively measured? In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 547–558. [doi: 10.1145/2950290.2950324]
- [53] Shen WJ, Wan J, Chen ZY. MuNN: Mutation analysis of neural networks. In: Proc. of the 2018 IEEE Int'l Conf. on Software Quality, Reliability and Security Companion. Lisbon: IEEE, 2018. 108–115. [doi: 10.1109/QRS-C.2018.00032]
- [54] Ma L, Zhang FY, Sun JY, Xue MH, Li B, Juefei-Xu F, Xie C, Li L, Liu Y, Zhao JJ, Wang YD. DeepMutation: Mutation testing of deep learning systems. In: Proc. of the 29th IEEE Int'l Symp. on Software Reliability Engineering. Memphis: IEEE, 2018. 100–111. [doi: 10.1109/ISSRE.2018.00021]
- [55] Chetouane N, Klampfl L, Wotawa F. Investigating the effectiveness of mutation testing tools in the context of deep neural networks. In: Proc. of the 15th Int'l Work-Confer. on Artificial Neural Networks. Gran Canaria: Springer, 2019. 766–777. [doi: 10.1007/978-3-030-20521-8\_63]
- [56] Klampfl L, Chetouane N, Wotawa F. Mutation testing for artificial neural networks: An empirical evaluation. In: Proc. of the 20th IEEE Int'l Conf. on Software Quality, Reliability and Security. Macao: IEEE, 2020. 356–365. [doi: 10.1109/QRS51102.2020.00054]
- [57] Wang JY, Dong GL, Sun J, Wang XY, Zhang PX. Adversarial sample detection for deep neural network through model mutation testing. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 1245–1256. [doi: 10.1109/ICSE.2019.00126]
- [58] Zou DM, Liang JJ, Xiong YF, Ernst MD, Zhang L. An empirical study of fault localization families and their combinations. IEEE Trans. on Software Engineering, 2021, 47(2): 332–347. [doi: 10.1109/TSE.2019.2892102]

- [59] Wong WE, Wei TT, Qi Y, Zhao L. A crosstab-based statistical method for effective fault localization. In: Proc. of the 1st Int'l Conf. on Software Testing, Verification, and Validation. Lillehammer: IEEE, 2008. 42–51. [doi: 10.1109/ICST.2008.65]
- [60] Xia X, Bao LF, Lo D, Li SP. “Automated debugging considered harmful” considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: Proc. of the 2016 IEEE Int'l Conf. on Software Maintenance and Evolution. Raleigh: IEEE, 2017. 267–278. [doi: 10.1109/ICSME.2016.67]
- [61] Mahapatra RP, Negi A. Effective software fault localization using GA-RBF neural network. Journal of Theoretical and Applied Information Technology, 2016, 90(1): 168–174.

#### 附中文参考文献:

- [24] 文万志, 李必信, 孙小兵, 刘翠翠. 一种基于层次切片谱的软件错误定位技术. 软件学报, 2013, 24(5): 977–992. <http://www.jos.org.cn/1000-9825/4342.htm> [doi: 10.3724/SP.J.1001.2013.04342]
- [26] 陈翔, 鞠小林, 文万志, 顾庆. 基于程序频谱的动态缺陷定位方法研究. 软件学报, 2015, 26(2): 390–412. <http://www.jos.org.cn/1000-9825/4708.htm> [doi: 10.13328/j.cnki.jos.004708]
- [27] 王赞, 闫明, 刘爽, 陈俊洁, 张栋迪, 吴卓, 陈翔. 深度神经网络测试研究综述. 软件学报, 2020, 31(5): 1255–1275. <http://www.jos.org.cn/1000-9825/5951.htm> [doi: 10.13328/j.cnki.jos.005951]



李铮(1998—), 男, 硕士生, CCF 学生会员, 主要研究领域为智能软件工程.



王荣存(1979—), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件测试, 缺陷定位, 软件维护.



崔展齐(1984—), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件测试及分析, 智能软件工程.



刘建宾(1963—), 男, 博士, 教授, CCF 专业会员, 主要研究领域为程序建模理论与方法, 智能化软件开发技术, 模型驱动软件工程.



陈翔(1980—), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件缺陷预测, 软件缺陷定位, 回归测试, 组合测试.



郑丽伟(1979—), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为需求工程, 群体协同, 大数据挖掘.