УДК 004.9 doi: 10.15827/0236-235X.142.551-560 2023. Т. 36. № 4. С. 551–560

Автоматическая локализация ошибок времени выполнения в программном обеспечении с помощью нейронных сетей

А.М. Достовалова
А.А. Шерминская

Ссылка для цитирования
Достовалова А.М., Шерминская А.А. Автоматическая локализация ошибок времени выполнения в программном обеспечении с помощью нейронных сетей // Программные продукты и системы. 2023. Т. 36. № 4. С. 551–560. doi: 10.15827/0236-235X.142.551-560
Информация о статье
Поступила в редакцию: 02.05.2023 После доработки: 22.07.2023 Принята к публикации: 13.09.2023

Аннотация. Разработан метод автоматической локализации ошибок времени выполнения с помощью нейронной сети по данным трассировки осуществления функций программы. Метод сопоставляет каждой функции вероятность содержания ошибки, которая считается пропорциональной степени влияния значений параметров функции на результат выполнения программы. Влияние параметров определяется численной характеристикой (весом), вычисляемой по алгоритму Хашема. Метод применялся для отладки нескольких программ, различных по типам и причинам возникновения ошибок времени выполнения. Ошибки были расположены во вложенных функциях и проявлялись при определенных значениях входных данных. В каждой программе проведено сопоставление наиболее вероятных мест возникновения ошибок, которые определил метод, с их реальными местоположениями. Особенностями разработанного метода являются возможность работы с вложенными функциями, локализация множественных ошибок, а также ошибок, у которых место возникновения и место проявления в программе не совпадают. Во всех случаях параметры, содержащие ошибку, имели больший вес в сравнении с остальными, даже если ошибок в программе было несколько. При этом метод выделяет в программе полный путь ошибки, включающий в себя все параметры, связанные с ее возникновением. Благодаря этому с помощью предложенного метода можно определять положение логических ошибок в программах. Метод может применяться для отладки как программного, так и аппаратного обеспечения технических систем, поскольку логика его работы не зависит от источника исходных данных.

Ключевые слова: автоматическая локализация ошибок, ошибки времени выполнения, отладка ПО, нейронные сети, трассировка

Требования к функциональности современного программного и аппаратного обеспечения сложных технических систем с каждым годом ужесточаются. Это приводит к увеличению числа входящих в них устройств и программ. Усложнение системы ведет к увеличению возможного количества появляющихся в ней неисправностей, комплексная отладка такой системы становится более трудоемкой. Возникает необходимость автоматизации процесса поиска неисправностей, особенно ошибок времени выполнения, определение положения которых наиболее сложно.

Для ПО существуют методы автоматической локализации ошибок времени выполнения [1]. Эти методы можно разделить на две группы: параметрические и непараметрические. Первые (например, методы срезки и их модификации [1]) для локализации ошибок используют значения программных переменных. Непараметрические методы, к которым относятся спектральные, статистические и некоторые методы машинного обучения, определяют положение ошибки в коде по данным трассировки (трассам) программы – численным данным о результатах работы отслеживаемых участков кода (функций, инструкций), полученным во время выполнения программы, и по результатам выполнения программы (выполнилась она до конца или завершилась с ошибкой) [2]. При этом в непараметрических методах используются только данные об успешном или неудачном завершении выполнения участков программы.

Непараметрические методы обычно более производительные по сравнению с параметрическими, поскольку программист может сам назначить подозрительные на наличие ошибки участки кода, в том числе проводя поиск ошибки в несколько этапов, каждый раз измельчая исследуемые блоки. Но эти методы пригодны для ограниченного круга задач – для программ, в которых нет участков вложенных функций и местоположение ошибки времени выполнения совпадает с местом ее проявления (характерно для спектральных и статистических методов). Непараметрическими методами невозможно локализовать ошибки, проявляющиеся лишь при определенных значениях входных параметров, особенно если программа завершает свое выполнение в момент ее возникновения. Кроме того, при трассировке алгоритм судит

о наличии ошибки в функции по тому, соответствуют ли ее выходные параметры заданному диапазону. Такая информация сильно подвержена искажениям, поскольку при определении границ диапазона уже может быть допущена ошибка.

Основным затруднением в процессе поиска неисправностей как в ПО, так и в технической системе в целом, является сопоставление большого числа разнородных параметров, неявно связанных между собой. Для выявления подобных закономерностей перспективным представляется использование нейронных сетей. Преимуществом этого подхода является универсальность: нейронная сеть способна искать закономерности в данных независимо от их природы, то есть в общем случае в состав исследуемой системы могут входить как программные, так и аппаратные средства.

Была поставлена задача разработать метод автоматической локализации ошибок времени выполнения с помощью нейросети в программах по данным трассировки выполнения функций. Каждой функции метод должен сопоставлять вероятность содержания ошибки, которая считается пропорциональной степени влияния значений параметров функции на результат работы программы. Численная характеристика влияния параметра вычисляется с помощью алгоритма Хашема [3]. Новизна разработанного метода заключается в возможности работы с вложенными функциями и в локализации множественных ошибок, а также ошибок, у которых место возникновения и место проявления в программе не совпадают. Предложенный метод тестировался на наборе программ с заложенными в коде ошибками времени выполнения, находящимися внутри последовательности вложенных функций. Предполагаемые места возникновения ошибок сравнивались с их реальным местоположением.

Подходы к автоматизации процесса поиска неисправностей в ПО

Задача автоматизации поиска неисправностей в ПО имеет долгую историю. Одной из первых работ по данной тематике является [4], в которой описан метод статической срезки. В нем положение некорректных инструкций в программе определяется среди набора выделенных срезов – строк кода, содержащих переменные, связанные с появлением ошибки. В последние годы методы срезки использовались в гибридных подходах в связке с альтернативными методами автоматической локализации ошибок в программном коде [5, 6].

Кроме методов срезки, существует множество других подходов к локализации ошибок [1, 7]. Среди них можно выделить два активно развивающихся направления – спектральные методы и методы машинного обучения.

В спектральных методах программа разбивается на участки, подозрительные на наличие в них ошибок. При отладке для каждого участка отслеживаются правильность его работы и степень покрытия тестами и вычисляется его подозрительность – численное значение, характеризующее вероятность наличия ошибки [1]. Формула для вычисления подозрительности с течением времени претерпела изменения [1], в нее были добавлены вероятностные элементы [8, 9]. Созданы модификации спектральных методов, повышающие для больших программ точность определения участков с ошибкой [10, 11].

Методы машинного обучения для автоматической локализации ошибок разделяются на методы, направленные только на определение положения ошибки, и методы, и локализующие, и исправляющие найденную ошибку. Методы первой группы близки по идее к спектральным. С помощью полносвязной нейронной сети определяются взаимосвязи между покрытием тестами отслеживаемых участков кода и соответствующими им значениями подозрительности [12].

Методы второй группы рассматривают задачу поиска и исправления ошибки как задачу машинного перевода [13]. Такая постановка позволяет использовать архитектуры LLM, например, инструмент ChatGPT [14]. В работе [15] также был продемонстрирован потенциал языковых моделей к исправлению ошибок времени выполнения при дополнении набора для обучения сети данными трассировки программы.

Перечисленные подходы различны, но обладают общей чертой – каждый метод накладывает ограничения на обрабатываемые данные. Методы срезки требуют, чтобы в исследуемом участке кода присутствовала только одна ошибка. Спектральные методы не используют данные о значениях программных переменных, из-за чего их способность к локализации ошибок, проявляющихся только при определенных значениях входных данных, ограничена. Кроме того, оценка правильности прохождения участков трасс требует создания критериев оценива-

ния, при разработке которых также могут возникнуть ошибки. Соответственно, исходные данные для спектрального метода могут быть недостоверными.

Использование большинства языковых моделей для исправления ошибок времени выполнения ограничено случаями, когда ошибки не являются логическими и не проявляются только при определенных значениях входных данных. Добавление к обучающему набору данных трассировки программы расширяет возможности языковых моделей. Однако в работе [15] задача разметки подозрительных участков трасс перед обучением решена не была – разметка проводилась вручную.

Кроме того, при отладке программно-аппаратных комплексов часто возникает ситуация, когда исходного кода у программиста нет. Часть функционала может быть доступна только в формате библиотек с известным интерфейсом, но неизвестной реализацией. В таких библиотеках могут существовать ошибки, проявляющиеся только при их функционировании как части системы, но не отдельного блока.

Предложенный в данной работе метод не сталкивается с вышеперечисленными ограничениями. Для локализации ошибок он использует только данные трассировки программы, к которым легко получить доступ при отладке программно-аппаратных комплексов и программ, даже если исходный код последних не известен. Метод использует непосредственно значения переменных, отслеживаемых при отладке программы, поскольку эти данные не искажены дополнительными проверками правильности. Кроме того, применение метода не ограничено случаями, когда в программе присутствует только одна ошибка.

Общее описание метода локализации ошибок времени выполнения

Представляемый метод локализует ошибки времени выполнения в программе, состоящей из набора функций $f_1(\bar{\theta}_1)...f_n(\bar{\theta}_n)$, $n \geq 1$, где $\bar{\theta}_1...\bar{\theta}_n$ – векторы выходных параметров функций. При трассировке для этих функций отслеживаются значения входных и выходных параметров. Считается, что логика работы программы (блок-схема) не изменяется в зависимости от входных данных: число вызовов данных функций и их последовательность остаются неизменными.

Если в программе среди f_i нет вложенных функций, ее можно однозначно представить в виде одной функции $F(\bar{\theta})$, где $\bar{\theta} = (\bar{\theta}_1...\bar{\theta}_n)$ – вектор всех выходных параметров функций программы. Функция $F(\bar{\theta})$ принимает два значения: $\omega_1 = 1$ в случае успешного выполнения программы, когда все входящие в нее функции отработали без ошибок, и $\omega_2 = 0$ в противном случае. В пространстве параметров, где $F(\bar{\theta})$ является разделяющей поверхностью, возможно определить направления, по которым эта функция изменяется наиболее сильно – больше всего значения градиента $F(\bar{\theta})$. Эти направления позволяют определить набор параметров из вектора $(\bar{\theta})$, изменение которых сильнее всего влияет на результат выполнения программы – значение ω_i. Определив эти параметры и предположив, что вероятность содержания ошибки пропорциональна степени влияния значений параметров функций на результат выполнения программы, можно выделить функции программы, которые содержат ошибку.

Однако связи между значениями $\bar{\theta}$ и ω_i, а также вид зависимости (линейная, квадратичная и пр.) заранее не известны. Функция $F(\bar{\theta})$ задана набором точек ($\bar{\theta}$, ω_i), $i = 1,2$, каждая из которых описывает результат выполнения программы ω_i для заданных параметров функций $\bar{\theta} = (\bar{\theta}_1...\bar{\theta}_n)$. Чтобы вычислить градиент функции $F(\bar{\theta})$, вначале необходимо построить ее аппроксимацию. Для этого в представляемом методе используется полносвязная нейронная сеть. Выбор такого способа аппроксимации $F(\bar{\theta})$ обусловлен тем, что сеть является универсальным аппроксиматором. Согласно теореме Хехт-Нильсена [16], с помощью двухслойной нейронной сети можно аппроксимировать любую непрерывную функцию. Сеть обучается на наборе данных трассировки программы. Она учится определять для вектора, составленного из значений параметров $(\bar{\theta})$, метку ω_i – соответствующий этим значениям результат работы программы.

После того как сеть обучена, необходимо определить для каждого входного параметра нейросети степень его влияния на выходное значение метки ω_i – результат выполнения трассы. Для этого каждому параметру ставится в соответствие численная характеристика – вес параметра, вычисляемый по алгоритму Хаше-

ма [3] усреднением по обучающему набору значений градиента $F(\bar{\theta})$ в точках данных.

Обработка участков вложенных функций

Описание метода было сделано только для работы с программами, которые не содержат вложенных функций. Если в программе встречаются вложенные участки (что характерно для многих реальных задач), метод не сможет локализовать в ней ошибку из-за неспособности сформировать полный вектор параметров $(\bar{\theta})$. Если программа завершается с ошибкой на нижнем уровне вложенности, то функции верхнего уровня не отрабатывают до конца и значения их выходных параметров оказываются недоступными. Ошибку в таком случае можно локализовать только в функции нижнего уровня вложенности. Если фактически ошибка расположена в одной из функций верхнего уровня, определить ее местоположение становится невозможным: нет ни одного параметра, связанного с нужной функцией.

Чтобы преодолеть ограничение на наличие вложенности в обрабатываемых программах, представляемый метод дополняют еще одним шагом, предшествующим шагу составления вектора параметров $\bar{\theta}$. Данные трассировки программы обрабатываются алгоритмом разворачивания вложенности. Назначение этого алгоритма в том, чтобы заменить вызовы вложенных функций на последовательность вызовов обычных функций. Например, рассмотрим последовательность вложенных функций

$$f_1 \supset (f_2 \supset (f_3 \rightarrow f_4)), \quad (1)$$

где $\supset$ – вызов из левой функции вложенного правого участка; $\rightarrow$ – последовательные вызовы функций.

Функции f_3, f_4 принадлежат нижнему уровню вложенности и выполняются на нем последовательно друг за другом. Развернутая трасса этой последовательности имеет вид

$$f_{1\_}\, f_2 \rightarrow f_{2\_}\, f_3 \rightarrow f_3 \rightarrow f_4 \rightarrow f_{4\_}\, f_2 \rightarrow f_{2\_}\, f_1, \quad (2)$$

где функции $f_{1\_}\, f_2$ и $f_{2\_}\, f_3$ – части функций f_1, f_2 до вхождения во вложенный участок; $f_{4\_}\, f_2$ и $f_{2\_}\, f_1$ – части f_1, f_2 после выхода из вложенности (участки между несколькими последовательными уровнями вложенности внутри f_i обозначаются $f_{i\_}\, f_{j\_}\, f_k$). Различия между схемами вызовов исходной и развернутой трасс представлены на рисунке 1.

В развернутой трассе выходные параметры функций f_3, f_4, $f_{4\_}\, f_2$, $f_{2\_}\, f_1$ – это выходные параметры функций f_3, f_4, f_2, f_1 из исходной трассы. Для функций $f_{1\_}\, f_2$, $f_{2\_}\, f_3$ в исходной трассе аналогов нет. Для них выходными параметрами считаются входные параметры следующих после них функций из вложенных участков (соответственно, функций) $f_{2\_}\, f_3$. Аналогично предлагается делать для функций, в которых программа завершается с ошибкой. Их выходные параметры неинформативны, поскольку содержат значение, приводящее к ошибке. Для развернутой трассы уже можно составить полный вектор параметров $(\bar{\theta})$ и определить среди них обладающие наибольшим влиянием на результат выполнения программы.

Определение параметров с наибольшим влиянием на результат выполнения программы

Для установления зависимости между значениями параметров $\bar{\theta} = (\bar{\theta}_1 ... \bar{\theta}_n)$ и результатом выполнения программы ω_i с помощью нейронной сети строилась аппроксимация функции $F(\bar{\theta})$. Влияние параметров на результат неравнозначно. Оценить степень этого влияния по построенной аппроксимации можно с помощью алгоритма Хашема [15]. В нем влияние

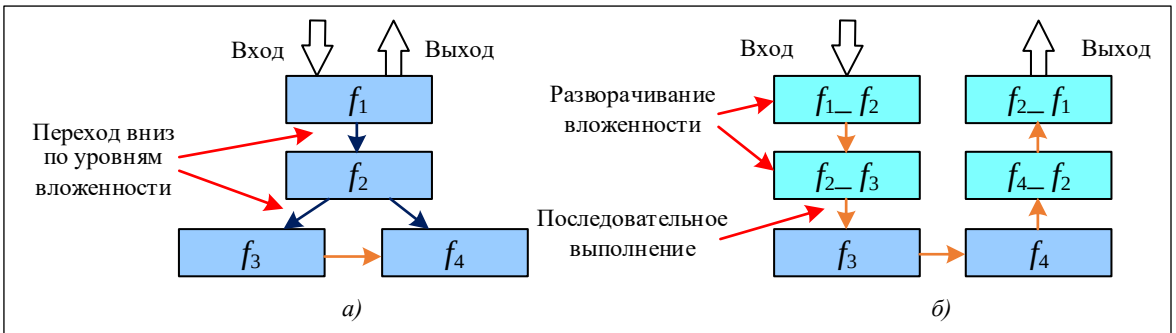

Рис. 1. Участок трассы следования программы: а) исходный; б) с разворачиванием вложенности

Fig. 1. Part of program trace: a) original view; б) view after unfolding nesting

входных параметров $\bar{\theta} = (\bar{\theta}_1 ... \bar{\theta}_n)$ на результат работы нейросети оценивается в ходе вычислений значений градиента $F(\bar{\theta})$ в точках данных. Обозначим O_k выходное значение k-го узла выходного N+1-го слоя, h_i^j – выходное значение i-го узла внутреннего слоя ($j = 1 \ldots N$) сети, I_i – значение на i-м входе нейросети, net_l^j – взвешенную сумму в l-м узле на j-м слое, w_{il}^j – весовой коэффициент, связывающий i-й узел с j-го слоя с l-м узлом на j+1-м, $j = 0 \ldots N$. Градиент в точке вычисляется с помощью алгоритма обратного распространения ошибки. Формулы в приведенных обозначениях имеют вид:

$$\frac{\partial O_k}{\partial h_i^N} = \frac{\partial O_k}{\partial net_k^{N+1}} \cdot \frac{\partial net_k^{N+1}}{\partial h_i^N} = O_k(1 - O_k) w_{ik}^N, \forall i,k, \quad (3)$$

$$\frac{\partial O_k}{\partial h_i^j} = \sum_l \frac{\partial O_k}{\partial h_l^{j+1}} \cdot \frac{\partial h_l^{j+1}}{\partial net_l^{j+1}} \cdot \frac{\partial net_l^{j+1}}{\partial h_i^j} = \\ = \sum_l \frac{\partial O_k}{\partial h_l^{j+1}} \cdot h_l^{j+1}(1 - h_l^{j+1}) \cdot w_{il}^j \quad \forall i,k, \quad (4)$$

$$\frac{\partial O_k}{\partial I_i} = \sum_l \frac{\partial O_k}{\partial h_l^1} \cdot \frac{\partial h_l^1}{\partial net_l^1} \cdot \frac{\partial net_l^1}{\partial I_i} = \\ = \sum_l \frac{\partial O_k}{\partial h_l^1} \cdot h_l^1(1 - h_l^1) \cdot w_{il}^0 \quad \forall i,k. \quad (5)$$

Для оценки приоритета входных параметров нейросети полученные значения градиента в точках данных усредняются по всему обучающему набору. Для каждого входа I_i нейронной сети вычисляется вес зависимости D_{ik} между ним и каждым выходным узлом O_k (N_{data} – число элементов в обучающем наборе) по формуле

$$D_{ik} = \frac{\sum_{s=1}^{N_{data}} \left| \frac{\partial O_k^s}{\partial I_i^s} \right|}{N_{data}}. \quad (6)$$

Эти значения нормируются и объединяются в матрицу весов DM (N_I – число узлов входного слоя, N_o – число узлов выходного слоя), для вычисления которой используется формула

$$DM = \begin{pmatrix} \frac{D_{11}}{\max_j(D_{j1})} & \ldots & \frac{D_{1k}}{\max_j(D_{jk})} \\ \ldots & \ldots & \ldots \\ \frac{D_{i1}}{\max_j(D_{j1})} & \ldots & \frac{D_{ik}}{\max_j(D_{jk})} \end{pmatrix} \forall j,k, \quad (7)$$

$j = 1 \ldots N_I, k = 1 \ldots N_O$.

Достоинство алгоритма Хашема в сравнении с альтернативными методами определения приоритета входных данных [3] в том, что он может работать с глубокими нейросетями, которые обычно приближают одну и ту же функцию более эффективно, чем неглубокие, даже если общее число нейронов у них совпадает [17]. Вычисленные с помощью алгоритма Хашема значения весов используют в предложенном методе локализации ошибок времени выполнения, чтобы ранжировать функции, составляющие программу, по вероятности содержания ошибки. Она тем выше, чем больше вес соответствующих функций параметров.

Экспериментальная отработка метода

Были исследованы пять тестовых программ. При трассировке отслеживались значения входных и выходных параметров функций. Завершение программы с ошибкой происходило, если значение хотя бы одного из параметров выполняемой функции выходило за пределы установленного для него диапазона.

Каждая исследуемая программа представляет собой последовательность вызовов вложенных функций, организованных в одну из двух структур (рис. 2), содержащих три уровня вложенности. На нижнем уровне функции выполняются последовательно. В каждой программе заложено условие, при выполнении которого она завершается с ошибкой в *func*5 или *func3*. Первопричина ошибки во всех случаях существует в предшествующих функциях: места возникновения и проявления ошибки не совпадают. Пять тестовых программ отличаются друг от друга характером ошибок, их количеством и причиной возникновения.

В **Программе 1** на вход *mod*1 подавался вектор длиной 10, состоящий из случайным образом сгенерированных чисел заданного диапазона. В этой функции создавался второй вектор, полученный из элементов первого умножением их значений на 2. В *mod*3 значения двух векторов еще раз умножались на 2, после чего генерировались два числа – *val*1 и *val*2. Первое принимало значения от 0 до 10, второе – 0 или 1. Оба вектора и два числа передавались в *mod*5. Далее в *func*4 вычислялся максимум из элементов векторов. В *func*5 происходила проверка значений *val*1 и *val*2: если второе число равнялось 1, а первое принимало любое значение, кроме 5 или 9, считалось, что программа завершилась с ошибкой. Программа 1 моделировала случай распределенной ошибки, проявляющейся при определенной комбинации значений параметров функций.

Программа 2 повторяла Программу 1, за исключением того, что в векторе входных па-

раметров не было значений *val*1 и *val*2. Программа 2 моделировала случай, когда ошибка не зависела от параметров, составляющих вектор параметров $(\bar{\theta})$.

Программа 3 тоже повторяла Программу 1, но в векторе параметров *val*1 и *val*2 повторялись три раза (названия переменных при этом были разными). Программа 3 моделировала случай, когда в векторе содержащие ошибку параметры повторялись несколько раз.

В **Программе 4** на вход *mod*2 подавался вектор длины 10, состоящий из заданных чисел (от 10 до 20). В этой функции создавался второй вектор, полученный из элементов первого умножением их значений на 2. В *mod*4 для элементов векторов вычислялись сумма значений *summVals* и максимум *maxVals*. Полученные значения передавались на вход *func*1, в которой вычислялся *res*1 = *summVals* + *maxVals* *c*1 и *res*2 = *res*1*c*2, *c*2 = 0; 1, *res*1 и *res*2 подавались на вход *func*2, в которой эти числа умножались на 1. В *func*3 программа завершалась ошибкой, если параметр *res*2 функции *func*2, передающийся ей на вход как параметр *inp*1, был отрицательным: ошибка была заложена в функции *func*1.

Программа 5 повторяла Программу 4 до момента выполнения функции *func*2. В Программе 5 *func*2 повторяла *func*1, за исключением условия, по которому *c*2 принимала отрицательное значение; *func*2 принимала на вход *maxVals* и *summVals* и возвращала значения *res*1 и *res*2; *func*3 принимала на вход значения и от *func*1, и от *func*2. Программа завершалась в этой функции с ошибкой при условии, что или *res*2 функции *func*1, или *res*2 функции *func*3 отрицательно.

Программа 5 моделировала случай, когда в ней присутствовало несколько ошибок. Программы 2 и 3 позволяли оценить влияние ошибок при составлении вектора параметров на результат обработки данных трассировки. В Программах 4 и 5, в отличие от Программ 1–3, ошибки проявлялись неявно: в вектор параметров входили только значения, на которые влияли параметры с ошибкой. Сами эти параметры в векторе $(\bar{\theta})$ не содержались.

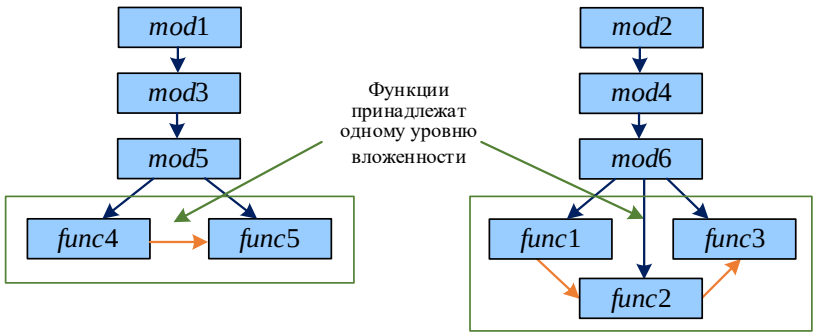

Рис. 2. Структуры исследуемых программ

Fig. 2. Structures of programs under study

Результаты локализации ошибок в исследуемых функциях

Для вычисления весов параметров функций программ 1–5 использовалась пятислойная полносвязная нейронная сеть с одним остаточным блоком [17]. Схема нейронной сети представлена на рисунке 3. Число нейронов в слоях сети изменялось в соответствии с длиной вектора данных *data_size*. В таблице для пяти тестовых программ приведены значения весов для параметров с ошибкой. Для параметров без ошибки были определены максимальное и среднее значения весов, а также их среднее квадратичное отклонение. Функции в программе ранжируются по вероятности содержания ошибки в соответствии с максимальными значениями весов их параметров. Соответствующий график для пяти тестовых программ представлен на рисунке 4.

На рисунке 4 каждой функции присвоен номер, определяющийся в соответствии с высотой столбца функции на графике. Столбцы наибольшей высоты соответствуют функциям, параметры которых обладают наибольшим весом.

В четырех из пяти случаев, для Программ 1, 3, 4 и 5, удалось выделить функции, содержащие ошибки. Параметры с ошибкой выделялись в общем множестве значений как пиковые значения: их веса были больше средних значений весов параметров без ошибки в 3–10 раз. У параметров без ошибки разброс значений весов относительно среднего, характеризующийся величиной среднеквадратичной ошибки, невелик.

Среди рассмотренных программ выделяется случай Программы 2, когда в векторе $(\bar{\theta})$ нет ни одного параметра, связанного с возник-

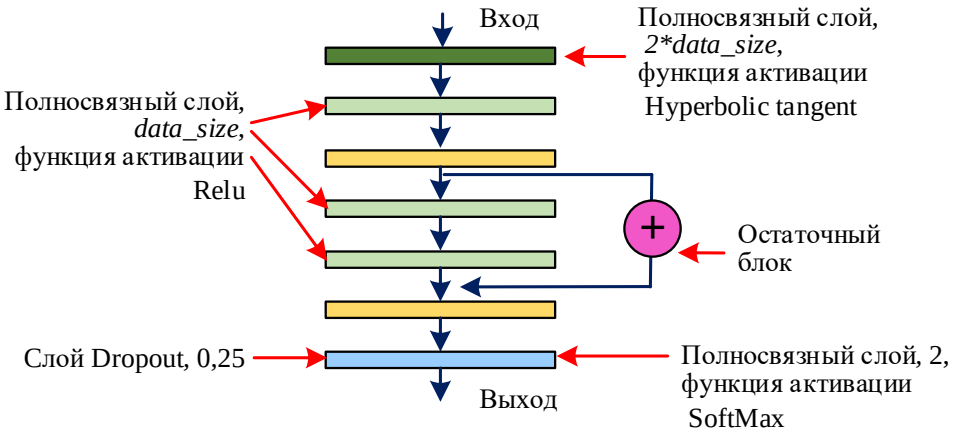

Рис. 3. Схема используемой нейронной сети

Fig. 3. Neural network scheme

новением ошибки. Функции этой программы возможно ранжировать по вероятности содержания ошибки. Но значения весов параметров всех функций близки друг к другу. Среди них нет ни одного яркого пикового значения, которое явно указывало бы на связь с появлением ошибки. Подобные случаи требуют дополнительного тестирования, в котором будут изменены отслеживаемые параметры или функции.

Интерес представляет случай Программы 4. В ней, кроме содержащей ошибку функции *func*1, выделяются еще функции *func*2 и *func*3, потому что у параметров *res*1 и вычисленных с его помощью *res*1 из *func*2 и у *inp*1 из *func*3 близкие значения весов. Метод может выделять лишние параметры, кроме тех, что действительно содержат ошибку. Однако все выделенные параметры оказываются связанными с возникновением ошибки: это параметры, значения которых являются условием появления ошибки, параметры, которым присваивается значение с ошибкой, и т.д. Для каждой ошибки можно проследить ее полный путь в программе – в данном случае от *func*1 до *func*3. В случае Программы 4 полный путь ошибки избыточен для определения ее местонахождения. Однако если допущенная ошибка является логической (например, условие, по которому она возникает, не должно выполняться вообще), исследование полного пути является единственным способом определения ее реального положения.

С помощью предложенного метода удалось локализовать ошибки в случае нескольких независимых ошибок в программном коде (Программа 5), а также когда значения, содержащие

Характеристики значений весов параметров функций тестовых программ

Weight values characteristics for test program function parameters

Номер программы	Параметры с ошибкой			Максимальный вес параметров без ошибки	Средний вес параметров без ошибки	Среднеквадратичная ошибка весов параметров без ошибки
	Функция	Имя	Вес			
1	*mod*5	*val*1	0.1883	0.026	0.0145	0.007
	*mod*5	*val*2	0.1408			
2	-	-	-	0.047	0.019	0.012
3	*mod*5	*val*1	0.0927	0.019	0.0084	0.005
	*mod*5	*val*1_0	0.0977			
	*mod*5	*val*1_1	0.0916			
	*mod*5	*val*2	0.0781			
	*mod*5	*val*2_0	0.0598			
	*mod*5	*val*2_1	0.0847			
4	*func*1	*res*1	0.1998	0.1853	0.015	0.009
5	*func*1	*res*1	0.08	0.12	0.02	0.089
	*func*2	*res*1	0.18			

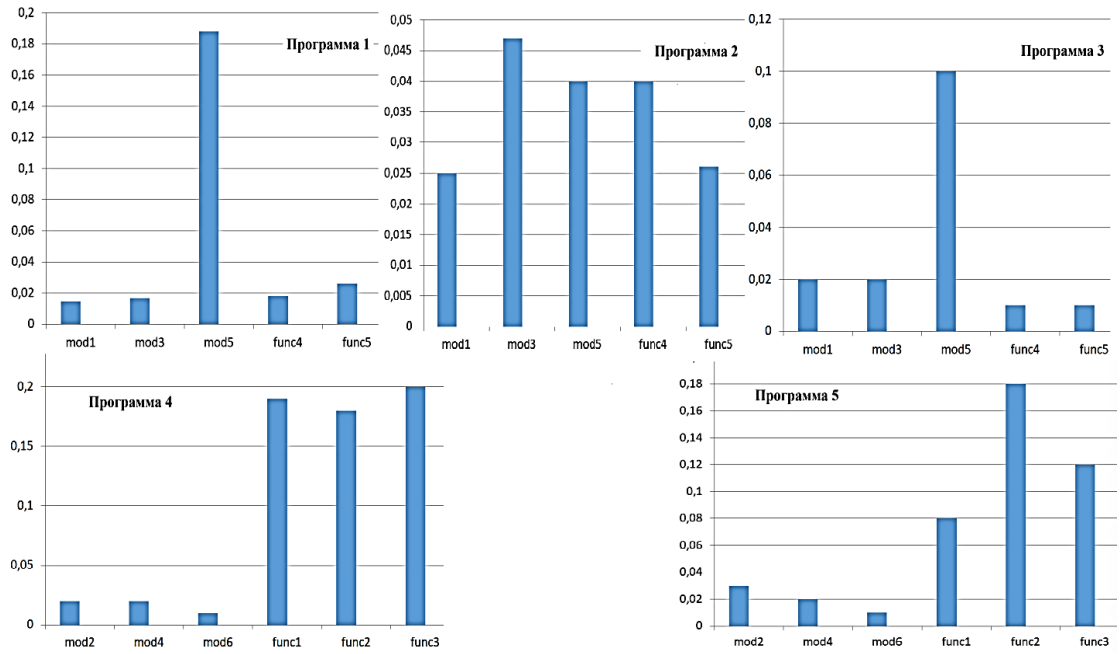

Рис. 4. Ранжирование функций в программе по весу (максимальной вероятности содержания ошибки) их параметров

Fig. 4. Ranking of program functions by parameter weights (maximum probability of containing errors)

ошибку, в векторе параметров повторялись (Программа 3). В обоих случаях значения весов параметров с ошибкой ярко выделялись среди остальных. При этом у повторяющихся параметров веса оказались близкими, так как значения градиентов по этим направлениям меняются схожим образом. Это является преимуществом предложенного метода, поскольку нет необходимости накладывать на исследуемые данные дополнительные ограничения на число независимых ошибок в трассе или то, чтобы каждый параметр встречался в векторе ровно один раз. При обработке реальных данных это позволит увеличить число случаев, в которых метод даст возможность определить местоположение ошибки.

Заключение

В результате исследования разработан метод автоматической локализации ошибки времени выполнения с использованием нейронных сетей в программах по данным трассировки выполнения функций. Предложенный метод производителен, как и непараметрические методы автоматической локализации ошибки времени выполнения, но при этом лишен их главного недостатка: с каждой функцией в трассе связаны неискаженные данные, по значениям которых производится локализация ошибки.

Метод применен для обработки данных трассировки пяти тестовых программ, различных по числу и природе возникающих в них ошибок времени выполнения. С его помощью удалось определить параметры, связанные с возникновением ошибки, когда они присутствовали в векторе параметров, подающемся на вход нейросети. Метод способен выделять ошибки в программах с вложенными функциями, в том числе и когда их несколько, а также определять положение ошибок, возникающих только при определенных значениях параметров функций. Он имеет потенциал в качестве средства определения положения логических ошибок в коде. Метод можно применить для отладки как аппаратного обеспечения, так и ПО сложных технических систем, поскольку для него не важна природа исследуемых данных.

Список литературы

1. Wong W.E., Gao R., Li Y., Abreu R., Wotawa F. A survey on software fault localization. IEEE Transactions on Software Eng., 2016, vol. 42, no. 8, pp. 707–740. doi: 10.1109/TSE.2016.2521368.

2. Kraft J., Wall A., Kienle H. Trace recording for embedded systems: Lessons learned from five industrial projects. Proc. RV. LNPSE, 2010, vol. 6418, pp. 315–329. doi: 10.1007/978-3-642-16612-9_24.

3. Комаров Я.Б., Шерминская А.А., Николаев А.А. Методы определения приоритета входных параметров функционально-математических библиотек с закрытым исходным кодом // Тр. КГНЦ. 2021. Спец. № 2. С. 91–96.

4. Weiser M. Program slicing. IEEE Trans. Soft. Eng., 1984, vol. SE-10, no. 4, pp. 352–357. doi: 10.1109/TSE.1984.5010248.

5. Reis S., Abreu R., d'Amorim M. Demystifying the combination of dynamic slicing and spectrum-based fault localization. Proc. IJCAI, 2019, pp. 4760–4766.

6. Soremekun E., Kirschner L., Böhme M., Zeller A. Locating faults with program slicing: An empirical analysis. Empirical Software Engineering, 2021, vol. 26, no. 3, art. 51. doi: 10.1007/s10664-020-09931-7.

7. Zakari A., Lee S.P., Alam K.A., Ahmad R. Software fault localisation: A systematic mapping study. IET Software, 2019, vol. 13, no. 1, pp. 60–74. doi: 10.1049/iet-sen.2018.5137.

8. Yang Y., Deng F., Yan Y., Gao F. A fault localization method based on conditional probability. Proc. IEEE 19th Int. Conf. on QRS-C, 2019, pp. 213–218. doi: 10.1109/QRS-C.2019.00050.

9. Landsberg D., Barr Earl T. Doric: Foundations for statistical fault localization. ArXiv, 2018, art. 1810.00798. URL: https://arxiv.org/abs/1810.00798 (дата обращения: 13.07.2023).

10. Keller F., Grunske L., Simon H., Filieri A., van Hoorn A., Lo D. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. Proc. IEEE Int. Conf. QRS, 2017, pp. 114–125. doi: 10.1109/QRS.2017.22.

11. Kong L., Wang J., Zhou S., Wang M. A multiple-fault localization method for embedded software with applications in engineering. Math. Problems in Engineering, 2021, vol. 2021, pp. 1–17, art. 7038979. doi: 10.1155/2021/7038979.

12. Heris S.R., Keyvanpour M. Effectiveness of weighted neural network on accuracy of software fault localization. Proc. ICWR, 2019, pp. 100–104. doi: 10.1109/ICWR.2019.8765262.

13. Chalambous Y., Tihanyi N., Jain R., Sun Y. et.al. A new era in software security: Towards self-healing software via large language models and formal verification. ArXiv, 2023, art. 2305.14752. URL: https://arxiv.org/abs/2305.14752 (дата обращения: 13.07.2023).

14. Haque M.A., Li S. The potential use of ChatGPT for debugging and bug fixing. EAI Endorsed Trans AI Robotics, 2023, vol. 2. doi: 10.4108/airo.v2i1.3276.

15. Bouzenia I., Ding Y., Pei K., Ray B., Pradel M. TraceFixer: Execution trace-guided program repair. ArXiv, 2023, art. 2304.12743. URL: https://arxiv.org/abs/2304.12743 (дата обращения: 13.07.2023).

16. Hecht-Nielsen R. Kolmogorov's mapping neural network existence theorem. Proc. IEEE Int. Conf. on Neural Networks, 1987, vol. 3, pp. 11–14.

17. Николенко С., Кадурин А., Архангельская Е. Глубокое обучение. СПб: Питер, 2018. 480 с.

Software & Systems doi: 10.15827/0236-235X.142.551-560 2023, vol. 36, no. 4, pp. 551–560

Automatic runtime error localization for software debugging using neural networks

Anastasiya M. Dostovalova
Anna A. Sherminskaya

For citation
Dostovalova, A.M., Sherminskaya, A.A. (2023) 'Automatic runtime error localization for software debugging using neural networks', *Software & Systems*, 36(4), pp. 551–560 (in Russ.). doi: 10.15827/0236-235X.142.551-560
Article info
Received: 02.05.2023 After revision: 22.07.2023 Accepted: 13.09.2023

Abstract. A new automatic runtime error localization method using a neural network was developed. The method determines bug position by program functions tracing data. The method matches an error containing probability for each function. This probability is proportional to the effect of each function parameter on a program execution result. A numerical characteristic calculated by Hashem algorithm determines a parameter effect. The method was applied for debugging several programs with nested functions. The most probable bug locations defined by the method were compared with their real positions. The developed method has some features. It is able to localize multiple bugs, nested function bugs, and bugs with unequal manifestation and location places. In all cases, bugged parameters had more weight in comparison with rest parameters. At the same time, method determines a full error trace for each bug. The trace consists of all program parameters related to a bug. Therefore, the method is able to localize logical errors in program. The method can be used for debugging both technical system software and hardware since method logic does not depend on source data.
Keywords: program debugging, neural network, program tracing, automatic bug localization, runtime error

References

1. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F. (2016) 'A survey on software fault localization', *IEEE Transactions on Software Eng*., 42(8), pp. 707–740. doi: 10.1109/TSE.2016.2521368.
2. Kraft, J., Wall, A., Kienle, H. (2010) 'Trace recording for embedded systems: Lessons learned from five industrial projects', *Proc. RV. LNPSE*, 6418, pp. 315–329. doi: 10.1007/978-3-642-16612-9_24.
3. Komarov, Ya.B., Sherminskay, A.A., Nikolaev, A.A. (2021) 'Methods for determining the priority of input parameters of functional-mathematical libraries with a closed source code', *Transactions of KSRC*, (spec. 2), pp. 91–96 (in Russ.).
4. Weiser, M. (1984) 'Program slicing', *IEEE Trans. Soft. Eng*., SE-10(4), pp. 352–357. doi: 10.1109/TSE.1984.5010248.
5. Reis, S., Abreu, R., d'Amorim, M. (2019) 'Demystifying the combination of dynamic slicing and spectrum-based fault localization', *Proc. IJCAI*, pp. 4760–4766.
6. Soremekun, E., Kirschner, L., Böhme, M., Zeller, A. (2021) 'Locating faults with program slicing: An empirical analysis', *Empirical Software Engineering*, 26(3), art. 51. doi: 10.1007/s10664-020-09931-7.
7. Zakari, A., Lee, S.P., Alam, K.A., Ahmad, R. (2019) 'Software fault localisation: A systematic mapping study', *IET Software*, 13(1), pp. 60–74. doi: 10.1049/iet-sen.2018.5137.
8. Yang, Y., Deng, F., Yan, Y., Gao, F. (2019) 'A fault localization method based on conditional probability', *Proc. IEEE 19th Int. Conf. on QRS-C*, pp. 213–218. doi: 10.1109/QRS-C.2019.00050.
9. Landsberg, D., Barr E.T. (2018) 'Doric: Foundations for statistical fault localization', *ArXiv*, art. 1810.00798, available at: https://arxiv.org/abs/1810.00798 (accessed July, 2023).
10. Keller, F., Grunske, L., Simon, H., Filieri, A., van Hoorn, A., Lo, D. (2017) 'A critical evaluation of spectrum-based fault localization techniques on a large-scale software system', *Proc. IEEE Int. Conf. QRS*, pp. 114–125. doi: 10.1109/QRS.2017.22.
11. Kong, L., Wang, J., Zhou, S., Wang, M. (2021) 'A multiple-fault localization method for embedded software with applications in engineering', *Math. Problems in Engineering*, 2021, pp. 1–17, art. 7038979. doi: 10.1155/2021/7038979.
12. Heris, S.R., Keyvanpour, M. (2019) 'Effectiveness of weighted neural network on accuracy of software fault localization', *Proc. ICWR*, pp. 100–104. doi: 10.1109/ICWR.2019.8765262.
13. Chalambous, Y., Tihanyi, N., Jain, R., Sun, Y. et.al (2023) 'A new era in software security: Towards self-healing software via large language models and formal verification', *ArXiv*, art. 2305.14752, available at: https://arxiv.org/abs/2305.14752 (accessed July, 2023).
14. Haque, M.A., Li, S. (2023) 'The potential use of ChatGPT for debugging and bug fixing', *EAI Endorsed Trans AI Robotics*, 2. doi: 10.4108/airo.v2i1.3276.
15. Bouzenia, I., Ding, Y., Pei, K., Ray, B., Pradel, M. (2023) 'TraceFixer: Execution trace-guided program repair', *ArXiv*, art. 2304.12743, available at: https://arxiv.org/abs/2304.12743 (accessed July, 2023).
16. Hecht-Nielsen, R. (1987) 'Kolmogorov's mapping neural network existence theorem', *Proc. IEEE Int. Conf. on Neural Networks*, 3, pp. 11–14.
17. Nikolenko, S., Kadurin, A., Arkhangelskaya, E. (2018) *Deep Learning*. St. Petersburg, 480 p. (in Russ.).

Авторы
Достовалова Анастасия Михайловна [1, 2],
техник-программист,
инженер-исследователь,
dost.bmstu99@gmail.com
Шерминская Анна Алексеевна [1],
начальник сектора, SherminskayaAA@yandex.ru

[1] Концерн «Моринсис-Агат»,
г. Москва, 105275, Россия
[2] Федеральный исследовательский центр
«Информатика и управление» РАН,
г. Москва, 119333, Россия

Authors
Anastasiya M. Dostovalova [1, 2],
Technical Fellow Programmer,
Research-Engineer
dost.bmstu99@gmail.com
Anna A. Sherminskaya [1], Head of Sector,
SherminskayaAA@yandex.ru

[1] Morinsis-Agat Concern JSC,
Moscow, 105275, Russian Federation
[2] Federal Research Center
"Computer Science and Control" RAS,
Moscow, 119333, Russian Federation