Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

НА ТЕМУ:

Разработка ПО для анализа исходного кода на основе векторного представления AST.

Студент ИУ8-34 (Группа) _____ (Подпись, дата) Р.А. Кулагин (И.О. Фамилия)

Руководитель курсового проекта _____ (Подпись, дата) А.А. Бородин (И.О. Фамилия)

Консультант _____ (Подпись, дата) А.А. Алексеев (И.О. Фамилия)

Москва, 2020 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ8
(Индекс)
_____ _____
(И.О.Фамилия)
«18» 09 2020 г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине «Технологии и методы программирования»

Студент группы ИУ8-34

Кулагин Руслан Алексеевич
(Фамилия, имя, отчество)

Тема курсового проекта: Разработка ПО для анализа исходного кода на основе векторного представления AST.
Направленность КП (учебный, исследовательский, практический, производственный, др.) исследовательский
Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения проекта: 25% к 3 нед., 50% к 9 нед., 75% к 12 нед., 100% к 15 нед.

Задание
Разработать программное обеспечение для деобфускации исходного кода на языке программирования JAVA. ПО должно анализировать абстрактное синтактическое (AST) дерево исследуемого исходного кода. Разработать нейронную сеть для генерации векторного представления AST. На основе вышеуказанного вектора, ПО должно анализировать функции, предлагать их имена, вычислять идентификаторы переменных, изменённые при обфускации исходного кода.

Оформление курсового проекта:

Расчетно-пояснительная записка на _____ листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

Дата выдачи задания « 1 » сентября 2020 г.

Руководитель курсового проекта _____ 18.09.20 **А. А. Бородин**
(Подпись, дата) (И.О.Фамилия)

Студент _____ 18.09.20 **Р. А. Кулагин**
(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Adam — метод адаптивной оценки моментов, оптимизатор, использующийся для изменения весов у нейронов.

F1 сумма — метрика, используемая для оценки обученности нейронной сети. Равна среднему гармоническому точности и полноты.

accuracy — метрика, отражающая процент предсказаний, в которых нейронная сеть была права. Вычисляется по формуле $\text{accuracy} = \frac{\text{число корректных предсказаний}}{\text{общее число предсказаний}}$.

keras — высокоуровневая оболочка для tensorflow, предоставляющая готовые классы для быстрого создания моделей нейросетей.

tensorflow — библиотека с открытым исходным кодом для машинного обучения.

абстрактное синтаксическое дерево для фрагмента кода C — это кортеж $\langle N, T, X, s, \delta, \phi \rangle$, где N — множество узлов, не являющихся конечными, T — множество узлов, являющихся конечными, X — множество значений, $s \in N$ — корневой узел, $\delta : N \to (N \cup T)$ — функция, соединяющая неконечный узел со списком его потомков, $\phi : T \to X$ — функция, соединяющая конечный узел с его значением. Каждый узел кроме корневого появляются в списке детей только один раз.

векторное представление объекта (эмбеддинг) — набор (вектор) байт, соответствующий объекту.

геттеры и сеттеры — функции, отвечающие за получение и присваивание значения какого-либо поля класса.

гиперпараметр — конфигурация, внешняя по отношению к модели, значение которой невозможно оценить по данным.

датасет — набор данных для обучения нейронной сети.

деобфускация — преобразование исходного кода обратное обфускации.

кортеж — упорядоченный набор фиксированной длины.

метрика — функция, которая используется для оценки предсказаний нейронной сети.

недокументированная возможность — функциональные возможности программы, не описанные или не соответствующие описанным в документации, при использовании которых возможно нарушение конфиденциальности, доступности или целостности обрабатываемой информации.

нейронная сеть (нейросеть) — логические структуры, составленные из формальных нейронов [1].

обфускация — преобразование исходного кода, которое на всех допустимых для исходной программы входных данных выдаёт тот же самый результат, что и оригинальная программа, но более трудна для анализа, понимания и модификации [2]

перекрёстная энтропия — функция, показывающая, сколько информации было потеряно между предполагаемым результатом и предсказанным.

полносвязный слой — слой, каждый элемент которого связан со всеми нейронами предыдущего слоя. Значение каждого нейрона вычисляется как взвешенная сумма всех нейронов предыдущего слоя.

полнота — метрика, используемая для оценки обученности нейронной сети. Вычисляется по формуле полнота $= \frac{TP}{TP+FN}$, где TP — число истинно положительных предположений, FN — число ложно отрицательных предположений.

репозиторий — место, где хранятся и поддерживаются какие-либо данные. В данной работе имеются в виду хранилища проектов на сайте github.com

слой — набор нейронов, работающих вместе на определённой глубине в нейронной сети.

точность — метрика, используемая для оценки обученности нейронной сети. Вычисляется по формуле точность $= \frac{TP}{TP+FP}$, где TP — число истинно

положительных предположений, FP — число ложно положительных предположений.

уязвимость — ошибка или недостаток, позволяющие злоумышленнику получить несанкционированный доступ к компьютеру.

функция потерь — функция, отражающая степерь ошибки нейронной сети.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

AST — абстрактное синтаксическое дерево

ПО — программное обеспечение

ЦЕЛЬ РАБОТЫ

Целью проекта является создание нейросети, которая должна вычислять идентификаторы функций и переменных для деобфускации исходного кода на языке Java. В ходе работы я должен получить:

— представление о способах векторного представления данных при помощи нейронных сетей,

— представление об абстрактном синтаксическом дереве (AST) кода,

— навыки работы с нейронными сетями и обработки данных.

ВВЕДЕНИЕ

Анализ исходного кода является составной частью информационной безопасности.

Любые программы могут включать в себя недокументированные возможности, уязвимости, а также быть вредоносными. Для программ на безопасность не всегда доступна документация к исходному коду, более того большинство несвободного программного обеспечения (ПО) дополнительно защищено от подобного внешнего вмешательства при помощи обфускации.

При анализе такого ПО применяется деобфускация. Однако данный процесс представляет собой нетривиальную задачу, особенно в случае работы с большими функциями. Мой проект поможет частично автоматизировать одну из самых трудоёмких частей деобфускации — вычисление идентификаторов переменных и методов при помощи нейросетей.

Конечно, участия человека в этом процессе не избежать, так как, если анализируемая функция небольшая или слишком зависит от сторонних библиотек, нейронной сети будет трудно понять, с чем она имеет дело. Однако преимущества от частичной автоматизации неоспоримы: большую часть работы программа сделает гораздо быстрее человека.

ОСНОВНАЯ ЧАСТЬ

1 Анализ существующих решений.

Среди литературы, изученной при подготовке проекта, мне не попались примеры нейросетей, которые работали как деобфускатор для переменных. Однако, есть пример архитектуры нейронной сети [3], которая может генерировать имена функций, основываясь на их содержимом.

1.1 Code2vec

Нейронная сеть code2vec [3] использует векторное представление функций, получаемое из AST этих функций. При обработке каждый лист дерева и каждый путь между листьями получает уникальный набор байт, эмбеддинг, который изменяется во время обучения. Полученные эмбеддинги преобразуются нейронной сетью и на выходе получается векторное представление AST, на котором можно обучить классификатор имён функций. Для наглядности на рисунке 1 можно увидеть графическое изображение AST функции из листинга 1.

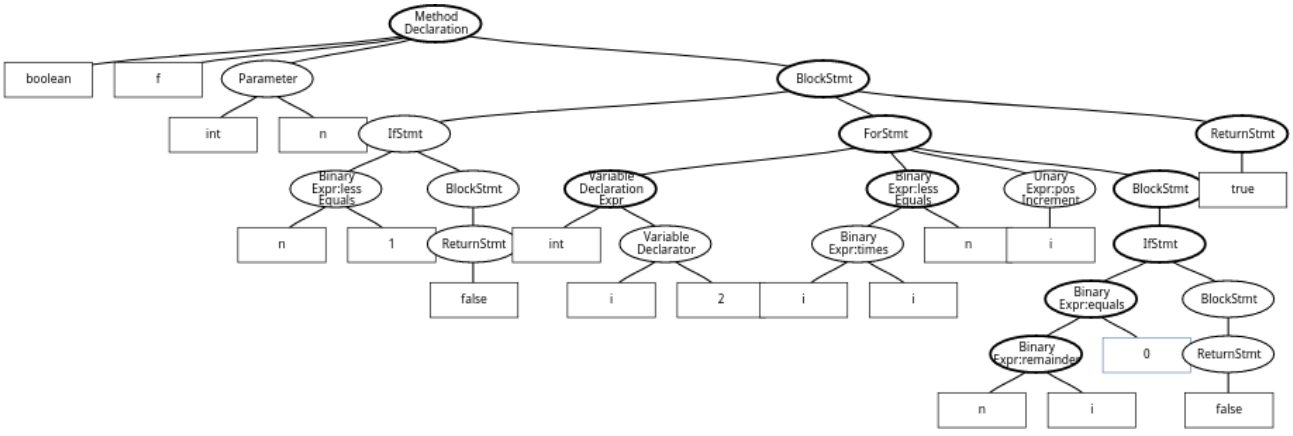

Рисунок 1 — Графическое изображение AST функции из листинга 1.

Листинг 1 — Пример функции, проверяющей число на простоту.

```
1 boolean isPrime(int n) {
2     if (n <= 1) {
3         return false;
4     }
5     for (int i = 2; i * i <= n; i++) {
```

```
6           if (n % i == 0) {
7               return false;
8           }
9       }
10      return true;
11  }
```

У эмбеддингов есть особенность, связанная с количеством данных, для которых они генерируются. Для каждого идентификатора должен быть создан вектор фиксированной длины, из-за этого количество параметров может быть сколь угодно большим. Чтобы улучшить работу программы, приходится фильтровать данные, что ведёт к частичной потере контекста. Авторы code2vec собрали более 12 миллионов функций для обучения, но им пришлось выбрать только 1 миллион, который помещался на их видеокарте — Tesla K80 GPU, имеющей 24 гигабайта памяти. Для обучения в домашних условиях убирать нужно ещё больше функций, что ведёт к ухудшению демонстрируемых результатов.

1.2 Code2vec с обфускацией.

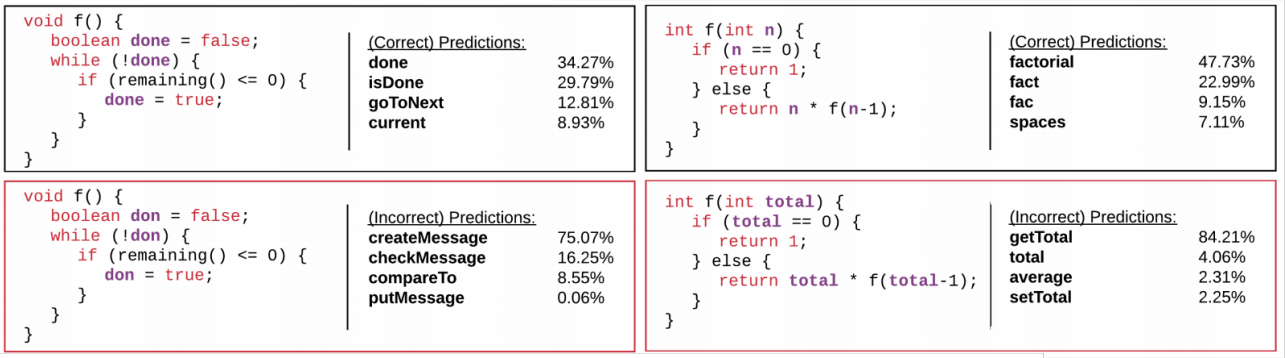

Рисунок 2 — Пример того, как изменение имён переменных порождает ошибки в предсказании code2vec. [4]

Нейронная сеть напрямую связана с данными, на которых она обучалась. Code2vec была обучена на репозиториях с открытым исходным кодом, использующих язык программирования Java. Так как тренировочные данные имели подходящие идентификаторы для переменных (соответствующие смыслу и назначению переменной), вектор функции получился слишком зависимым от имён переменных (см. рисунок 2) [4].

Такой зависимости можно избежать, если предварительно обфусцировать имена переменных. Этот подход делает невозможным работу code2vec для некоторых функций, например, для геттеров и сеттеров путей может просто не хватить. Однако у авторов [4] получилось обучить code2vec на обфусцированных данных, F1 сумма которой была несущественно ниже, чем у оригинала. Это наталкивает на мысль, что code2vec применима для деобфускации при обучении на обфусцированных данных.

2 Адаптация существующих технологий для решения задачи проекта.

Проект состоит из нескольких последовательных частей (см. рисунок 3), которые аналогичны для идентификаторов методов и переменных.

Рисунок 3 — Блок-схема, иллюстрирующая процесс работы программы.

При разработке я опирался на репозиторий code2vec [5], так как статья авторов кода [3] даёт крайне расплывчатое представление о реальной архитектуре сети и её особенностях.

2.1 Данные.

Для генерации AST было принято решение модифицировать программу JavaExtractor, распространяемую вместе с code2vec [5]. Данная программа была переработана, чтобы соответствовать всем требованиям, описанным в пунктах 2.1.1–2.1.4.

2.1.1 Контекст переменной.

Если для функции контекстом можно назвать все пути в её AST, то что считать контекстом переменной? При обработке данных для code2vec для каждой функции генерируются пути-контексты — кортежи, состоящие из 2-х листьев AST и пути[1], связывающего эти листья. Причём все идентификаторы будут содержатся только в листьях. Значит для каждой переменной мы можем взять только те кортежи, которые будут включать в себя идентификатор этой переменной. Этот подход позволит учесть все листья, встретившиеся после инициализации переменной, при этом все пути будут относится только к конкретной переменной, что позволит избежать наложения контекстов.

2.1.2 Необходимость обфускации.

Главная причина обфускации при обучении сети объясняется в разделе 1.2. Так как наша задача деобфусцировать код, мы должны исключить ошибку из-за переменных с обманчивым названием.

В процессе разработки была обнаружена интересная особенность, если в двух независимых блоках функции (см. листинг 2) встречались переменные с одинаковыми идентификаторами, то такие переменные никак не различались для нейронной сети, что могло приводить к ошибкам. Эту проблему также позволяет решить обфускация.

Листинг 2 — Пример кода, в котором две переменные с одинаковым идентификатором.

```
1 public void foo(int n){
```

[1]Под путём здесь и далее понимается результат работы хэш функции hashCode от символьного представления пути между двумя листьями в дереве.

```
2       if (n%2 == 0){
3           int divided = n / 2;
4           System.out.println(divided);
5       }
6       if (n%3 ==0){
7           int divided = n / 3;
8           System.out.println(divided);
9       }
10  }
```

2.1.3 Способы обфускации.

Изначально решил прислушаться к результатам статьи obfuscated-code2vec [4], в которой лучше себя показала идея замены идентификаторов переменных на случайный набор символов. Однако это оказалось не лучшим решением: обфусцированные листья генерировали наборы эмбедингов огромных размеров, которые обучались впустую, так как при тестировании ни один лист не совпадал, а значит аналогий не находилось. В результате нейронная сеть просто не запускалась, так как не хватало памяти. Пришлось накладывать жёсткие ограничения на данные, урезая гибкость результатов (подробнее про ограничения в пункте 2.2.3). В конце у сети было более 70 миллионов параметров, она обучалась долго и была требовательна к памяти: видеокарта NVIDIA GEFORCE GTX 1060 6GB была стабильно загружена на 98%.

Использовать обфускацию основанную на типе данных также не имеет смысла: любой пользовательский тип не позволит корректно сгенерировать имя, а рассчитывать на обфусцированную функцию, использующую только встроенные типы — странно.

Гораздо интереснее выглядит вариант, в котором все листья разделяются на 4 класса: имена переменных, функций, классов и константы. Такие листья заменяются соответственно на VAR, FUNC, CLASS_NAME, CONSTANT. Такой подход делает акцент на типе листа, участвующего в кортеже, при этом мы избавляемся от имён, которые могут искажать результаты. Чтобы оставить стандартные и общеупотребимые листья было принято решение не обфусцировать имена встроенных в Java классов и константы 0, 1, INT_MAX,

NaN, пустую строку, null. Это всё позволило существенно сократить количество используемых листьев, за счёт освободившейся памяти были смягчены ограничения для идентификаторов.

В code2vec имя метода скрывается при помощи листа METHOD_NAME, логично скрывать имя переменной, для которой ищется идентификатор, при помощи VARIABLE_NAME.

2.1.4 Датасет.

В качестве данных для обучения были взяты репозитории с крупными проектами с github: cassandra, elasticsearch, gradle, hibernate-orm, liferay-portal, spring-framework, wildfly. Эти проекты были выбраны, так как они содержат достаточную кодовую базу, при этом в них высокое качество кода, значит переменные имеют идентификаторы, которые могут быть выходами нейронной сети.

В рамках подготовки данных для деобфускации происходит обфускация кода, описанная в пункте 2.1.3, после чего происходит фильтрация имён по встречаемости. Идентификаторы, которые появлялись менее чем в 2х проектах или менее 35 раз суммарно в финальную выборку не включаются, они скорее всего специфичны для какой-то программы и вряд ли помогут при деобфускации либо мы не наберём достаточно данных для обучения для данного идентификатора. При такой обработке получаются файлы с данными размером более 470 МБайт.

Также на данном этапе генерируются словари: отфильтрованным данным (идентификаторам, листьям, путям) присваиваются индексы, уникальные в каждом наборе, потом всё сохраняется в файл. Этот файл понадобится уже при работе нейросети, так как по нему будут составляться эмбеддинги.

2.2 Code2var.

2.2.1 Архитектура.

Архитектура нейронной сети для деобфускации переменных аналогична архитектуре code2vec. В рамках проекта нейросеть была написана самостоятельно. Основное отличие от оригинальной нейросети — использование особенностей новой версии tensorflow 2.0 и keras. Разработчики code2vec использовали самостоятельно написанные слои, однако tensorflow 2.0 позволяет использовать встроенные в keras для выполнения тех же задач. Данное решение упрощает разработку, позволяя быть уверенным в корректности работы компонентов нейронной сети.

В процессе разработки была обнаружена проблема, связанная со слоем эмбеддингов. Из-за того, что нейросеть была написана с использованием режима мгновенного исполнения (eager execution), ставшего режимом по умолчанию в tensorflow 2.0, а keras не полностью был адаптирован для этого, слой эмбеддигов запускался для исполнения на процессоре, а не на видеокарте. Из-за этого скорость работы катастрофически падала, процессор был перегружен. Проблему решило написание слоя-оболочки для слоя эмбеддингов, которая была взята из описания проблемы в репозитории tensorflow на github [6].

В основе векторного представления кода лежит работа с эмбеддингами, которые при помощи полносвязного слоя объединяются в контекстный вектор. Данное преобразование одинаково для всех путей-контекстов и позволяет выделить определённую комбинацию среди похожих [3]. Так как каждому идентификатору соответствует множество контекстных векторов, для объединения используется дополнительный полносвязный слой, который генерирует коэффициенты — вес контекста в итоговом векторе. После этого происходит взвешенное суммирование всех контекстных векторов. На выходе нейросети стоит классификатор, который по контекстному вектору определяет, с какой вероятностью идентификатор соответствует входным данным.

Граф иллюстрирующий архитектуру нейронной сети представлен на рисунке 4.

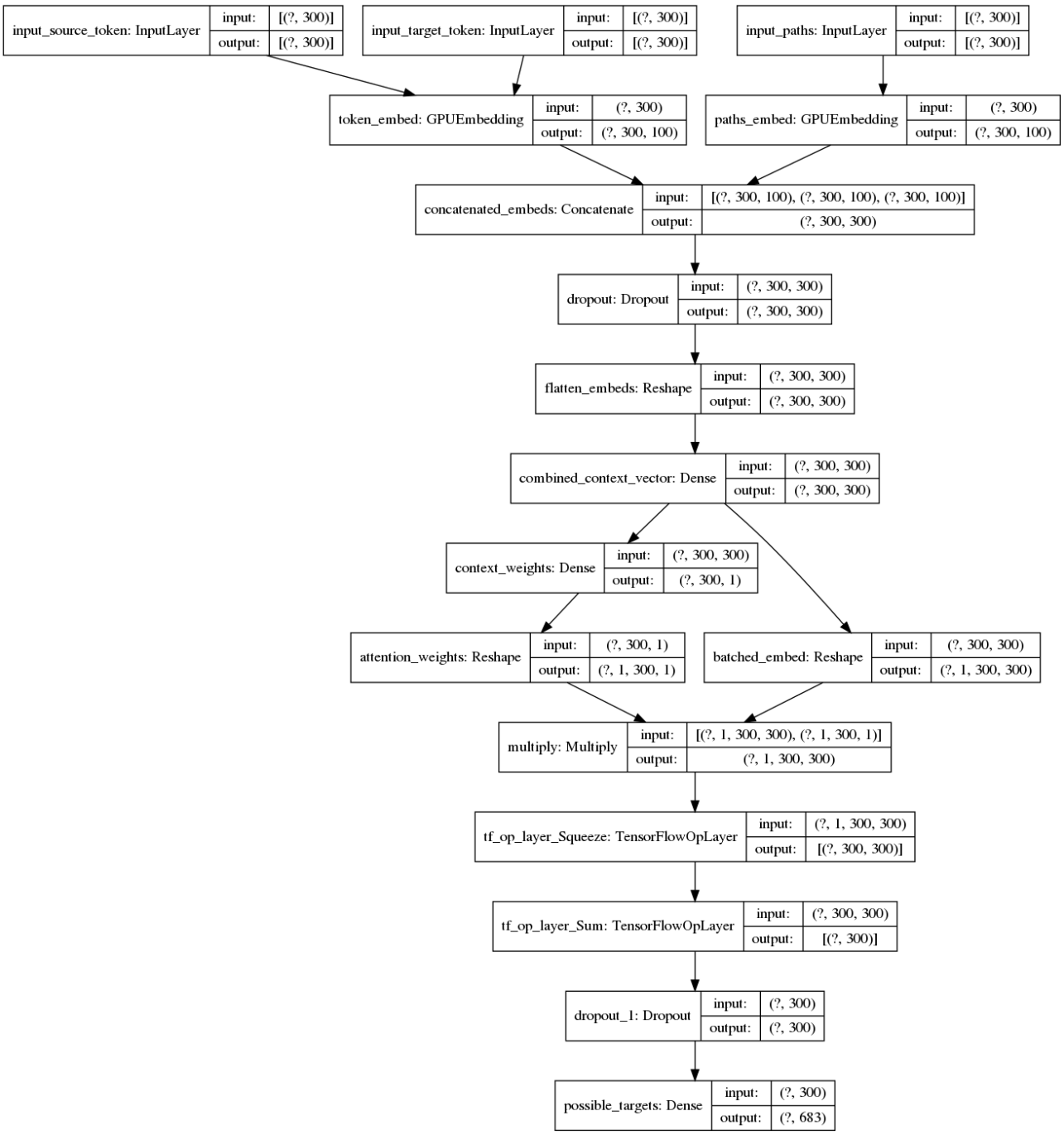

Рисунок 4 — Архитектура нейронной сети.

2.2.2 Параметры и метрики нейросети.

На обучение нейронной сети можно влиять несколькими способами: изменять оптимизатор, функцию потерь, добавлять новые метрики или работать над данными.

Функцию потерь и оптимизатор было принято взять аналогично оригинальной нейросети: перекрёстную энтропию и Adam.

В качестве метрики изначально было принято решение использовать accuracy, однако в процессе работы выяснилось, что это оказалось не самой удачной идеей. Во-первых, accuracy учитывает истинно ложные значения (не подходящим идентификаторам соответствует близкое к нулю значение выходного слоя). Во-вторых, в качестве предполагаемого значения accuracy считает то, которому в выходном слое нейросети соответствует наибольшее значение, при этом никак не учитывается ситуация, когда искомое значение находится в первой пятёрке выходного слоя. Так как проект предполагает выбор анализирующего из представленных пяти вариантов, то описанную ситуацию необходимо включать во внимание, поэтому будем использовать точность, в которой положительным результатом будет считаться попадание в топ 5, а отрицательным — все остальные ситуации.

Работа с параметрами оптимизатора требует большого опыта и уверенности в данных, на которых обучается нейросеть, в моём случае попытки как-либо изменить эти параметры не улучшили результаты работы.

2.2.3 Анализ данных.

Часть работы над данными уже была описана в пункте 2.1. Здесь описание будет дополнено и будут приведены некоторые графики обучения для обучения на разных данных.

Чем больше данных, тем больше различных листьев, путей, идентификаторов, которые порождают увеличение числа обучаемых параметров нейронной сети. Я был ограничен ресурсами моего компьютера, а также временем, которое уходило на обучение.

Прежде всего нужно оценить осмысленность идентификаторов. Найдём первые 50 слов по встречаемости (гистограммы представлены на рисунках 5 и 6). Среди переменных находится много однобуквенных идентификаторов, которые, за исключением i, j, k, o, e (первые три — счётчики, четвёртый обозначает объект, а пятый — для исключения), s (описание сессии или сканера) мало что скажут об объекте. При визуальном анализе полного списка идентификаторов бросаются в глаза переменные, имеющие специфичное название, подходящее

только для проекта, в котором использовалось, например, zzUnpackAttribute. Чтобы обобщить идентификаторы для переменных, было принято решение не рассматривать те из них, которые встречаются только в одном проекте. Также, чтобы избавиться от малоинформативных идентификаторов, был поставлен фильтр, не пропускающий идентификаторы короче двух символов за исключением i, j, k, e, s, o, db, fs, it, is, in, to, которые несут определённую смысловую нагрузку. Это решение позволило значительно уменьшить число идентификаторов, при этом они стали универсальными. Конечно, такое решение имеет свои недостатки: большинство идентификаторов стало выглядеть слишком обобщёнными, часть полезных идентификаторов было утеряно, но в собранном датасете более 120 тысяч уникальных имён переменных и более 210 тысяч уникальных имён функций и отфильтровать их вручную на данном этапе не представляется возможным.

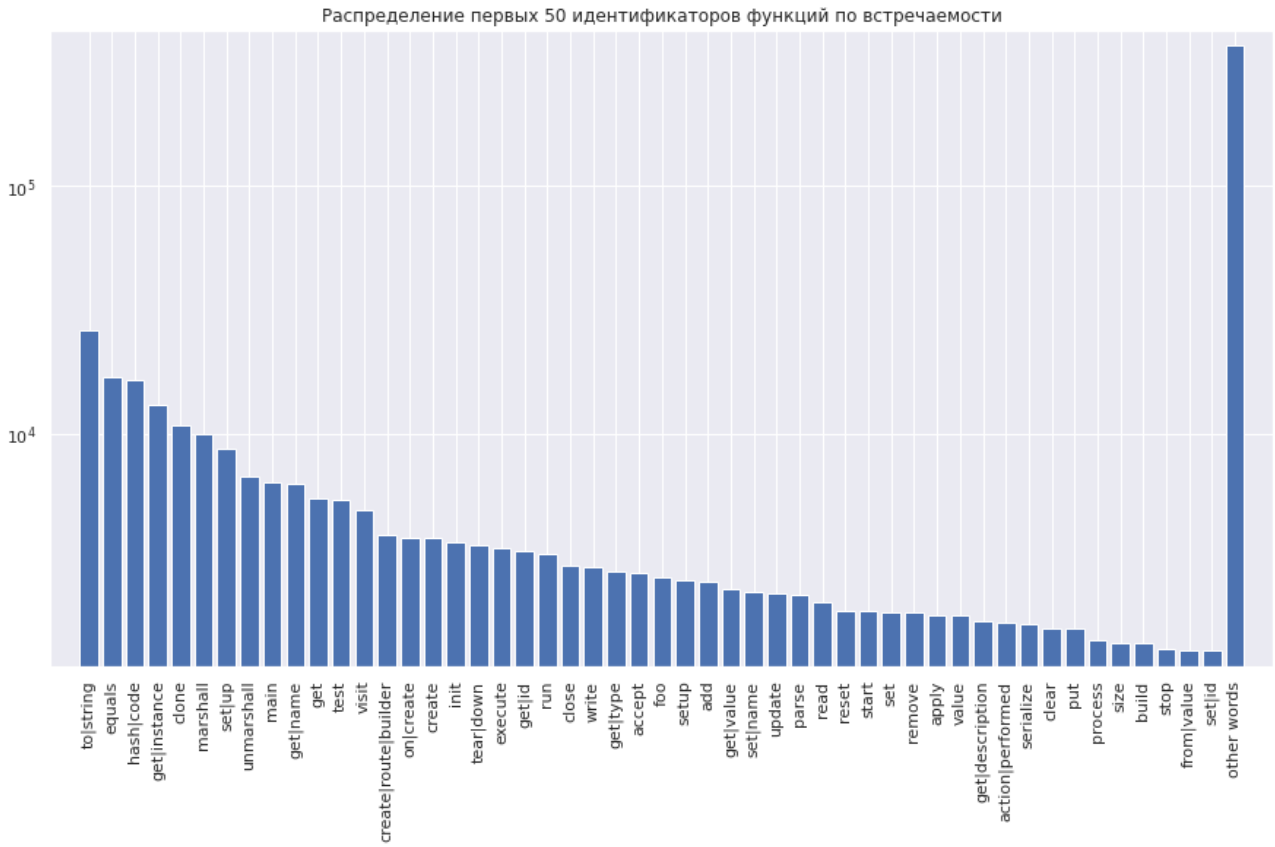

Рисунок 5 — Первые 50 идентификаторов функций по встречаемости.

Для функций такое решение лишает набор идентификаторов какой-либо смысловой нагрузки, так как становится слишком узким. Поэтому для них было принято решение ограничить число идентификаторов по их встре-

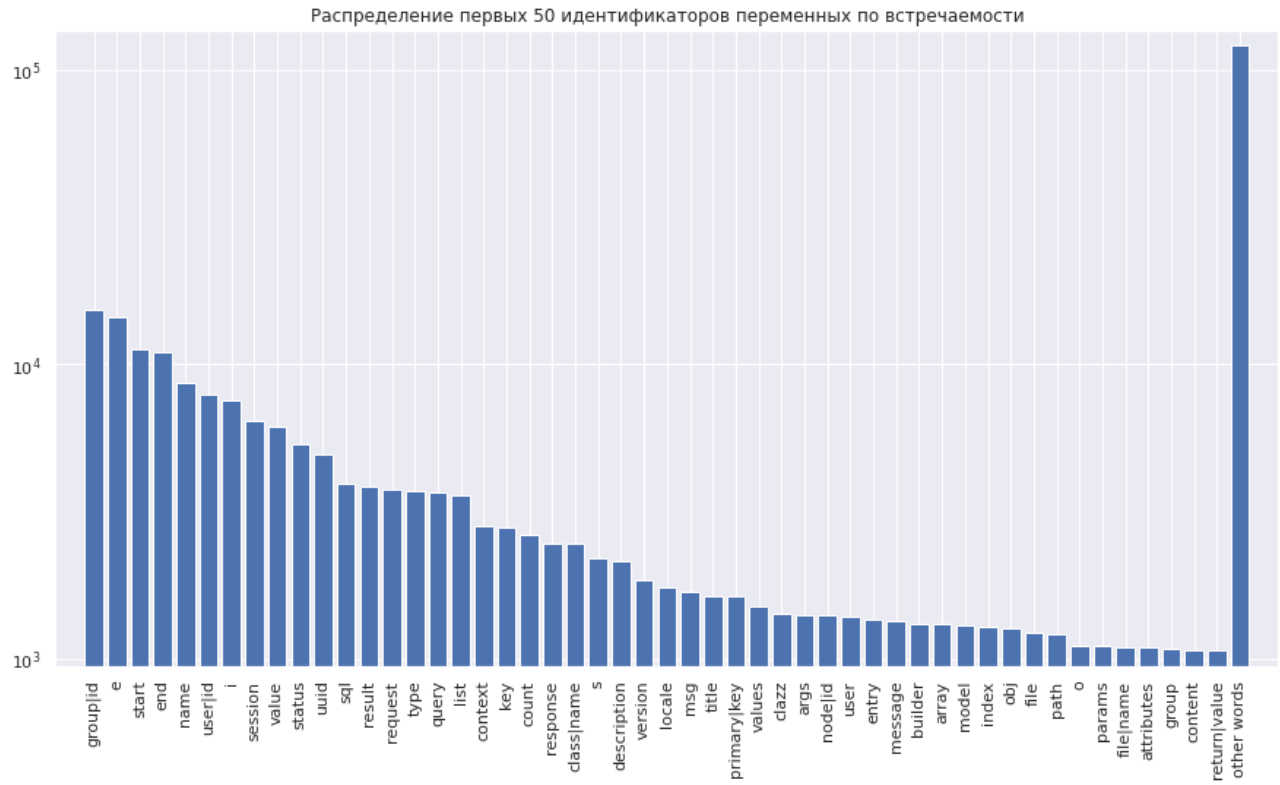

Рисунок 6 — Первые 50 идентификаторов переменных по встречаемости.

чаемости. Если для каких-то из них слишком мало данных для обучения, то такие идентификаторы будут мешать корректной работе, так как нейросеть не научится их распознавать в должной мере. Понять, какой лимит стоит поставить, чтобы не потерять слишком много данных, можно при помощи графиков, отражающих количество идентификаторов, встречающихся в датасете, в зависимости от частоты встречаемости этих идентификаторов. Для имён функций эта гистограмма представлена на рисунке 7, для переменных — на рисунке 8. По этим гистограммам видно, что отфильтровав функции, встречающиеся реже 35 раз, и переменные, встречающиеся реже 35 раз, мы не существенно уменьшим количество данных, при этом уберём редко используемые идентификаторы.

При обучении требуется отдельный набор данных, на котором нейронная сеть будет тестироваться. Изначально предполагалось использовать данные с совпадающими идентификаторами из проекта, который не был включён в тренировочный датасет, однако решение было неудачным, график метрики accuracy, полученной при таком обучении для идентификаторов функций, можно увидеть на рисунке 9. Проанализировав идентификаторы,

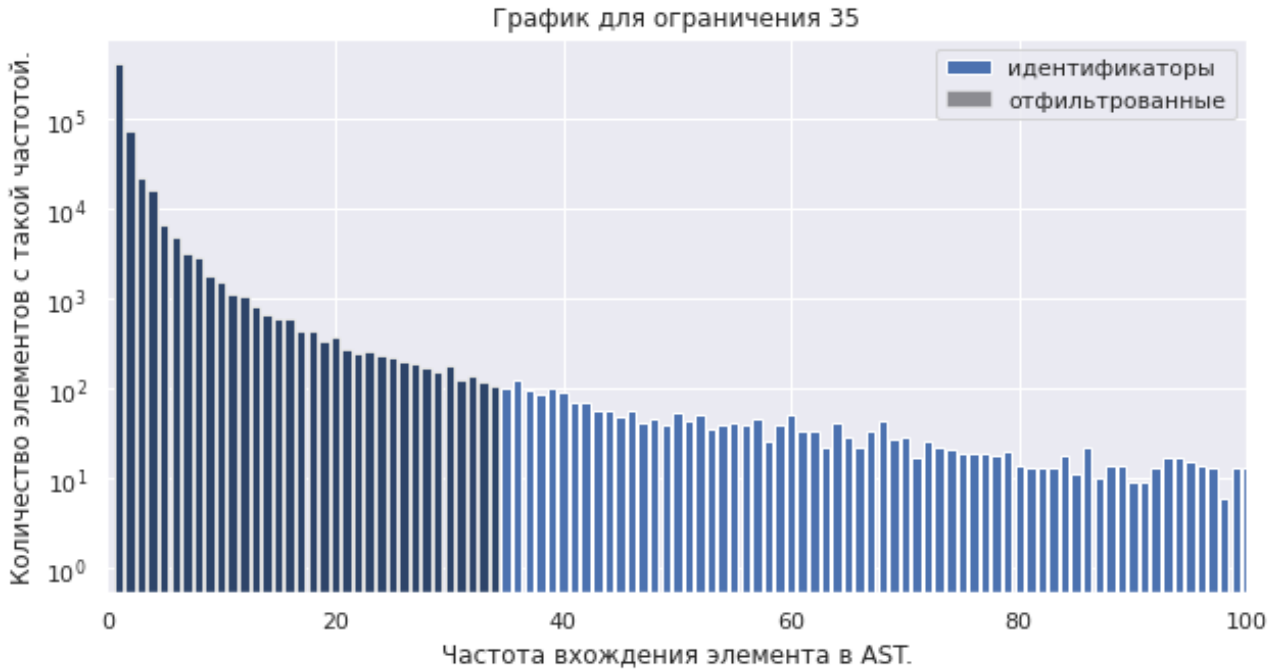

Рисунок 7 — График зависимости количества идентификаторов функций от частоты встречаемости идентификаторов в датасете.

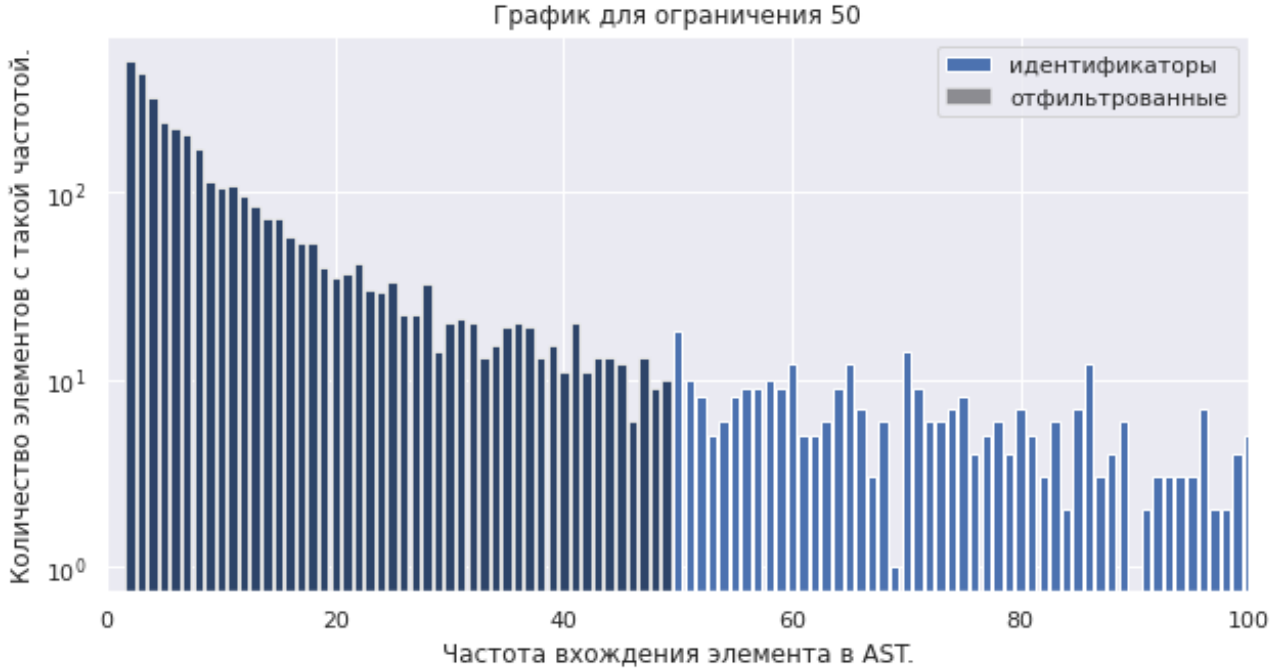

Рисунок 8 — График зависимости количества идентификаторов переменных от частоты встречаемости идентификаторов в датасете.

стала понятна причина расхождений: совпадали play, run, execute и другие имена, которые для каждого класса работали совершенно по-разному, из-за чего стало понятно нулевое значение accuracy.

После этого было принято решение случайным образом отбирать из тренировочного датасета определённое количество строк (примерно 10%) и исключать эти данные из процесса обучения, на них проводилось тестирование во время обучения. Такое решение позволило сделать корректным оценку качества обучения нейронной сети (график accuracy для нейронной сети, обученной на идентификаторах представлен на рисунке 10).

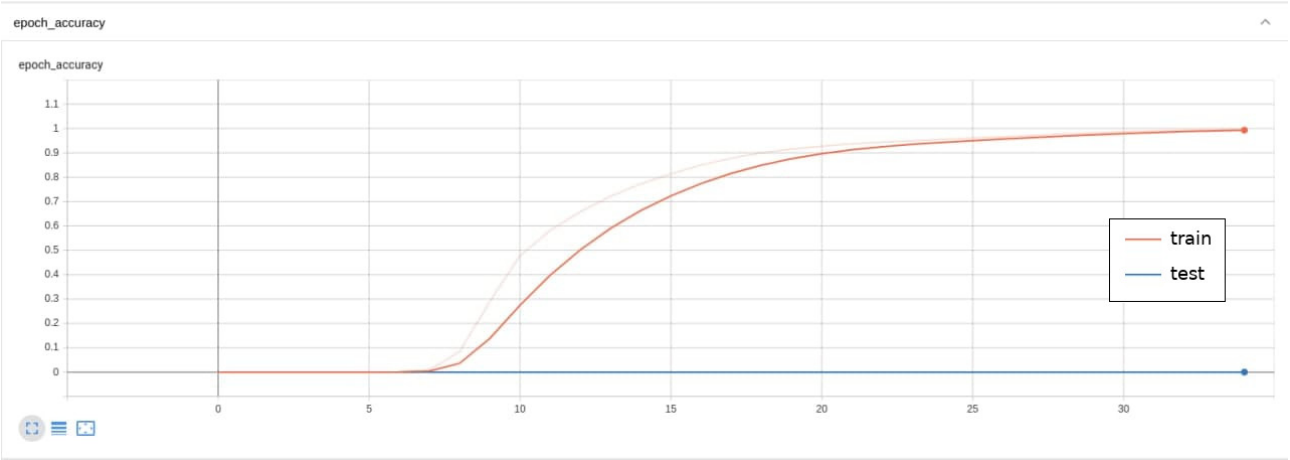

Рисунок 9 — График метрики accuracy для нейросети, обучавшейся для идентификаторов функций, тестировавшейся на данных из стороннего проекта.

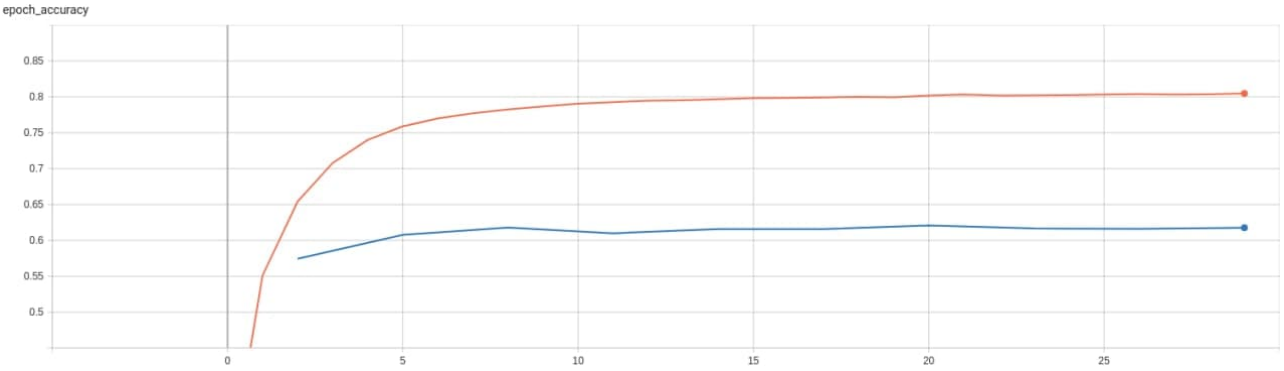

Рисунок 10 — График метрики accuracy для нейросети, обучавшейся для идентификаторов функций, тестировавшейся на данных из проектов, использовавшихся в обучении.

Количество уникальных листьев AST было уменьшено благодаря обфускации, однако число уникальных путей остаётся большим, что сказывается на размерах нейронной сети и скорости обучения. Попытка отфильтровать пути по количеству упоминаний стала плохой идеей: при минимуме в 25, обре-

завшем 91% от общего числа уникальных путей для функций (см. рисунок 11), нейронная сеть не смогла обучаться, данные оказались слишком обобщёнными, об этом свидетельствует график функции потерь (см. рисунок 12), которая сначала резко снижалась, но на поздних эпохах начала возрастать. Учитывая то, что из-за обфускации листьев, основная смысловая нагрузка ложится на пути, от их фильтрации было принято решение отказаться.

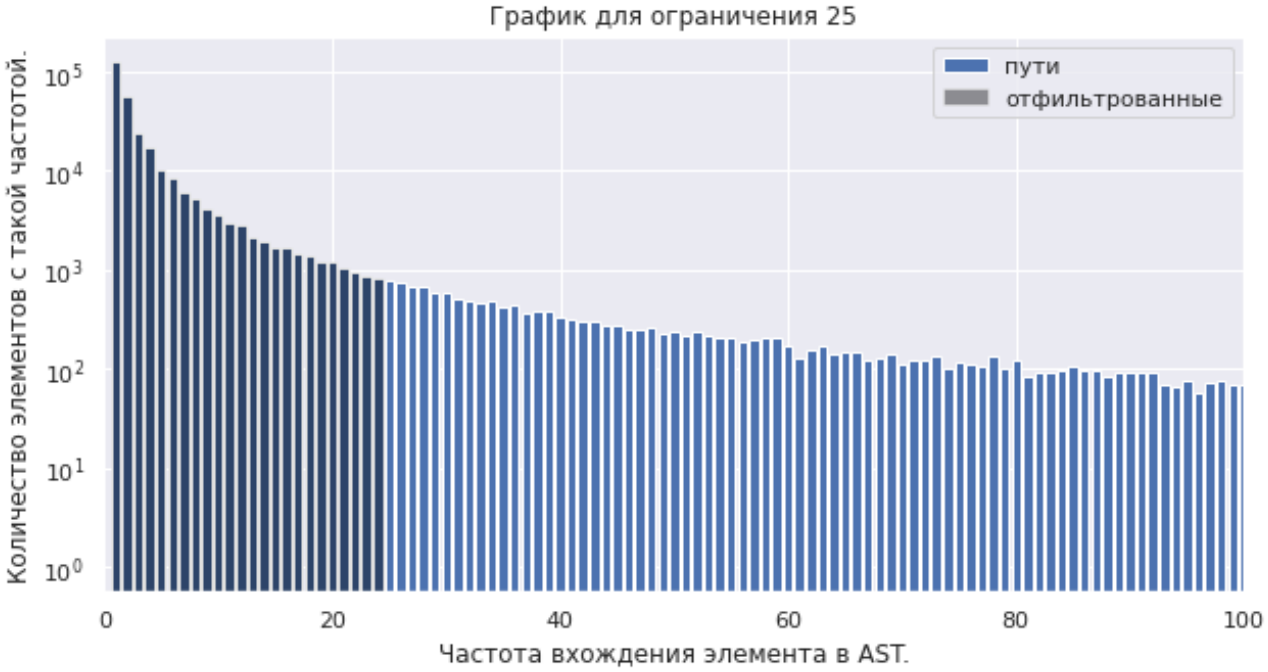

Рисунок 11 — Распределение количества уникальных путей в зависимости от частоты встречаемости этих путей в данных для переменных.

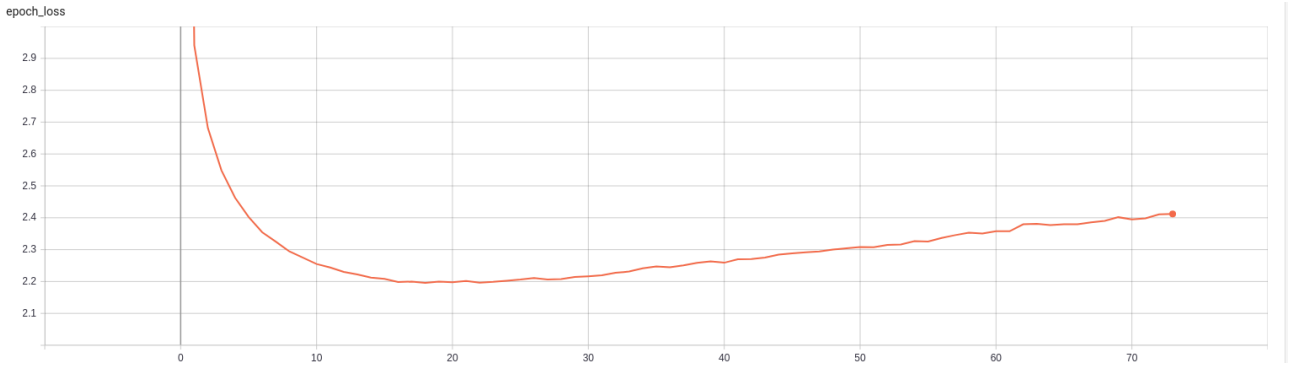

Рисунок 12 — График функции потерь для нейросети, обучавшейся для идентификаторов переменных, с ограничением минимального количества упоминаний для путей равной 25.

С функцией потерь для тестовых данных возникла проблема, которую решить так и не получилось: несмотря на рост accuracy для тестовых данных, функция потерь также растёт (хотя должна наоборот стремиться к минимуму). Скорее всего, это связано с тем, что функция потерь учитывает не только отклонение от 1 для вероятности правильного идентификатора, но и отклонение от 0 для остальных вероятностей. Найти замену перекрёстной энтропии не удалось.

2.2.4 Итоги обучения

Обучение происходило до тех пор, пока функция потерь не прекращала уменьшаться.

Нейросеть для определения идентификаторов функций обучилась за 2 эпохи до точности 46% для обучающего датасета и до точности 30% для тестового датасета, график функции потерь представлен на рисунке 13. Нейросеть для идентификаторов переменных обучилась за 5 эпох до точности 46.5% для тренировочного датасета и 38% для тестового, график функции потерь представлен 14.

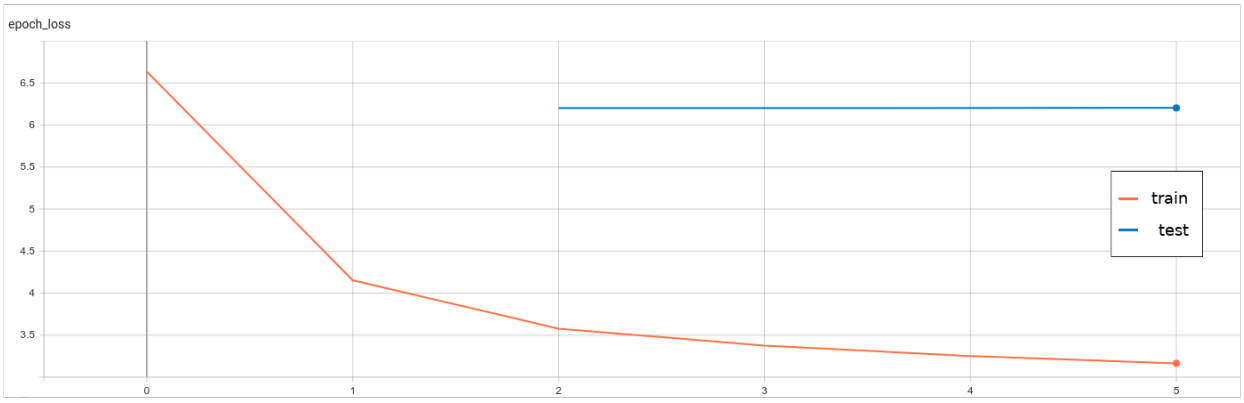

Рисунок 13 — График функции потерь для нейросети, обучавшейся для идентификаторов функций.

Такое быстрое обучение, которое потом практически не улучшается скорее всего связано с тем, что слои эмбеддингов учатся гораздо быстрее, чем полносвязные слои, которым обычно требуется не один десяток эпох. В результате эмбеддинги подстраиваются под случайные веса полносвязных слоев.

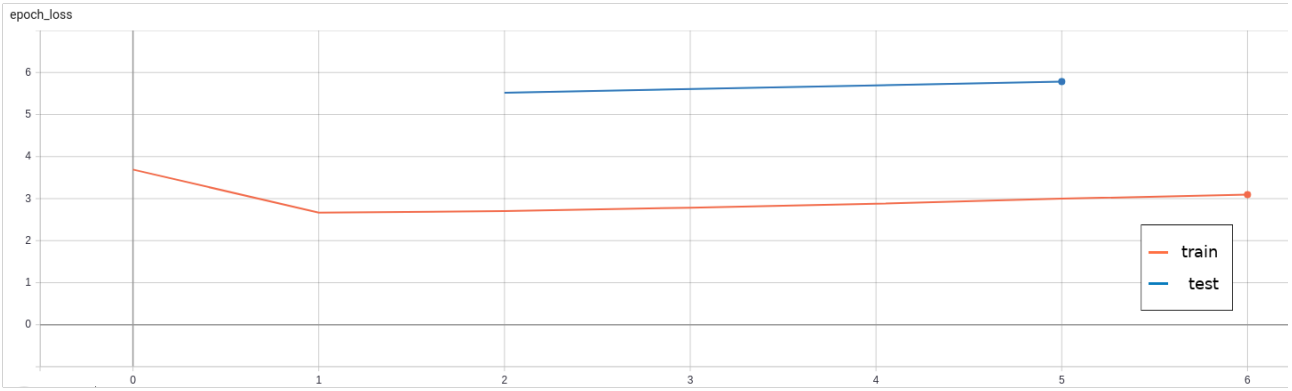

Рисунок 14 — График функции потерь для нейросети, обучавшейся для идентификаторов функций.

3 Демонстрация результатов работы.

Программа получает на вход файл с расширением java, который содержит код, написанный на языке программирования Java. Происходит обработка файла, которая включает в себя обфускацию файла, после чего для каждой функции из AST генерируются пути-контексты, которые сохраняются в csv файл. На вход нейронной сети поступают данные, полученные из этого csv файла, в которых все данные заменены на индексы из словаря, созданного при обучении. Результатом работы нейросети является новый csv файл, в котором в каждой строке записаны индентификатор, который переменная имела в оригинальном файле, а также 5 идентификаторов, которые, по мнению нейронной сети, наиболее подходят для подстановки. Этот файл читается программой, которая предлагает пользователю выбрать, какая из альтернатив подходит больше всего и заменяет в деобфусцируемом коде имена переменных и функций на выбранные альтернативы. Идентификаторы отсортированы по убыванию степени уверенности нейросети в этой альтернативе.

В Приложении Б приведён пример кода, поступающего на вход программе (до деобфускации), в Приложении В приведён результат работы программы (после деобфускации).

ЗАКЛЮЧЕНИЕ

В рамках проделанной работы была опробована идея использовать архитектуру нейронной сети code2vec для деобфускации исходного кода на языке программирования Java. Анализируя результаты из пункта 3, можно сказать, что эта идея вполне реальна.

Нейронная сеть работает неидеально на данном этапе, но её потенциал виден. Она корректно определяет переменные-счётчики, переменные, отвечающие за массив или его длину. Однако большинство трудностей возникает с переменными, инициализированными в конце, так как контекст для них крайне мал. Также в процессе разработки возникали и продолжают возникать проблемы с данными: малое количество идентификаторов не покрывает все рассматриваемые случаи, но для увеличения набора требуется много времени на ручной отбор, а также вычислительные мощности большие, чем есть у меня на данный момент.

Работа была для меня сложной, так как это первый опыт разработки крупного проекта, при подготовке к которому отсутствовала достаточная теоретическая подготовка и необходимый опыт. В большинстве вопросов приходилось обращаться к источникам на английском языке или разбираться в исходном коде библиотек.

В дальнейшем я планирую, после наработки дополнительных теоретических и практических знаний, вернуться к этому проекту с целью доработки до состояния, подходящего для полноценного применения в деобфускации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Галушкин А. И. НЕЙРОННЫЕ СЕТИ. — 2017. — Access mode: https://bigenc.ru/technology_and_technique/text/v/2256451 (online; accessed: 01.12.2020).

2. Чернов А. В. Анализ запутывающих преобразований программ. // Труды ИСП РАН. — 2002. — Access mode: https://cyberleninka.ru/article/n/analiz-zaputyvayuschih-preobrazovaniy-programm (online; accessed: 01.12.2020).

3. Code2vec: Learning Distributed Representations of Code / Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav // Proc. ACM Program. Lang. — 2019. — Jan. — Vol. 3, no. POPL. — Access mode: https://doi.org/10.1145/3290353 (online; accessed: 19.09.2020).

4. Embedding Java Classes with Code2vec: Improvements from Variable Obfuscation / Rhys Compton, Eibe Frank, Panos Patros, Abigail Koay // Proceedings of the 17th International Conference on Mining Software Repositories. — MSR '20. — New York, NY, USA : Association for Computing Machinery, 2020. — P. 243–253. — Access mode: https://doi.org/10.1145/3379597.3387445 (online; accessed: 17.10.2020).

5. Репозиторий с кодом нейронной сети code2vec. — Access mode: htpps://github.com/tech-srl/code2vec/ (online; accessed: 30.09.2020).

6. Описание проблемы с слоем эмбедингов в keras. — Access mode: https://github.com/tensorflow/tensorflow/issues/44194/ (online; accessed: 30.10.2020).

ПРИЛОЖЕНИЕ А

Ссылка на репозиторий с проектом

https://github.com/bmstu-iu8-g4-2020-project/code2var

ПРИЛОЖЕНИЕ Б

Файл data.java

```
import java.util.ArrayList;

public class data {
    public String featuresToString(
                        ArrayList<ProgramFeatures> features) {
        if (features == null || features.isEmpty()) {
            return Common.EmptyString;
        }
        List<String> methodsOutputs = new ArrayList<>();

        for (ProgramFeatures singleMethodfeatures : features) {
            StringBuilder builder = new StringBuilder();

            String toPrint = singleMethodfeatures.toString();
            if (m_CommandLineValues.PrettyPrint) {
                toPrint = toPrint.replace(" ", "\n\t");
            }
            builder.append(toPrint);
            methodsOutputs.add(builder.toString());
        }
        return StringUtils.join(methodsOutputs, "\n");
    }

    public static void bubbleSort(int[] arr) {
        Integer n = arr.length;
        for (int I = 0; I < n - 1; I++)
        for (int J = 0; J < n - I - 1; J++)
        if (arr[J] > arr[J + 1]) {
            int temp = arr[J];
```

```
            arr[J] = arr[J + 1];
            arr[J + 1] = temp;
        }
    }

    public static void insertionSort(int[] sort_arr) {
        for (int i = 0; i < sort_arr.length; ++i) {
            int j = i;
            while (j > 0 && sort_arr[j - 1] > sort_arr[j]) {
                int key = sort_arr[j];
                sort_arr[j] = sort_arr[j - 1];
                sort_arr[j - 1] = key;
                j = j - 1;
            }
        }
    }

    public static int average(int[] arr){
        int sum = 0;
        for (int i = 0; i<arr.length; i++){
            sum+=arr[i];
        }
        int avg = sum / arr.length;
        return avg;
    }

}
```

ПРИЛОЖЕНИЕ В

Файл data-deobfuscated.java

```
import java.util.ArrayList;

public class data {
    public String encrypt(ArrayList<ProgramFeatures> nodes) {
        if ((nodes == null) || nodes.isEmpty()) {
            return Common.EmptyString;
        }
        List<String> paths = new ArrayList<>();
        for (ProgramFeatures node : nodes) {
            StringBuilder builder = new StringBuilder();
            String name = node.toString();
            if (m_CommandLineValues.PrettyPrint) {
                name = name.replace(" ", "\n\t");
            }
            builder.append(name);
            paths.add(builder.toString());
        }
        return StringUtils.join(paths, "\n");
    }

    public static void sort(int[] array) {
        Integer length = array.length;
        for (int i = 0; i < (length - 1); i++)
            for (int j = 0; j < ((length - i) - 1); j++)
                if (array[j] > array[j + 1]) {
                    int value = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = value;
        }
```

```
    }

    public static void sort(int[] array) {
        for (int i = 0; i < array.length; ++i) {
            int i1 = i;
            while ((i1 > 0) && (array[i1 - 1] > array[i1])) {
                int value = array[i1];
                array[i1] = array[i1 - 1];
                array[i1 - 1] = value;
                i1 = i1 - 1;
            }
        }
    }

    public static int sum(int[] value) {
        int counter = 0;
        for (int i = 0; i < value.length; i++) {
            i += value[i];
        }
        int index = counter / value.length;
        return index;
    }
}
```