Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.М07-мм

Разработка набора инструментов для обучения искусственных нейронных сетей выбору оптимального пути для символьного исполнения

Нигматулин Максим Владиславович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н. С. В. Григорьев

Санкт-Петербург
2024

Оглавление

Введение

Тестирование — распространенный способ проверки корректности программного кода. Это дополнительная программная (или программно-аппаратная) инфраструктура, позволяющая проверить приложение на ожидаемое поведение. Однако, как утверждал Дейкстра, тестирование программы может эффективно продемонстрировать наличие ошибок, но неадекватно для демонстрации их отсутствия [5]. Поэтому для проверки кода в критически важных программных компонентах необходим другой, более надежный способ.

Одним из наиболее надежных методов проверки корректности программ является символьное исполнение. Это техника анализа ПО, позволяющая выполнять программу не с конкретными входами, а с их символьным представлением [12]. С помощью эффективных решателей логических уравнений этот подход позволяет перебрать все возможные состояния исполнения программы и проверить каждое из них на соответствие условию корректности [17]. Преимуществом этого метода также является возможность воспроизвести путь символьной машины к состоянию программы, исполнив её с конкретными данными, таким образом создавая тест на поведение, соответствующее пути символьного исполнения к этому состоянию. Успех этого подхода подтверждается множеством инструментов, использующих символьное исполнение для генерации тестов [18, 7, 1, 2, 11]. Целью работы этих инструментов является увеличение тестового покрытия, метрики, сигнализирующей о качестве тестирования программы.

Символьное исполнение сталкивается с проблемой экспоненциального увеличения количества путей, которые нужно исследовать для достижения целевых состояний [3]. Одним из способов решения этой проблемы является оптимизация выбора путей исполнения. Оптимизируя выбор путей, можно сократить количество исследуемых состояний исполнения.

Определить заранее, увеличит ли выбор определенного пути исполнения тестовое покрытие в разумное время, невозможно. Поэтому ин-

струменты, использующие символьное исполнение, полагаются на вручную разработанные стратегии, которые аппроксимируют идеальный алгоритм (или эвристику). Эти эвристики могут направлять исполнение в те части программы, где максимизируется целевая метрика, а не туда, где присутствует уязвимость. В таких случаях, где отношения между данными и целевым признаком слишком сложны для выявления закономерностей вручную, эффективно использовать машинное обучение.

В задаче выбора оптимального пути существующие решения на основе машинного обучения опираются на анализ данных об особенностях состояний исполнения [13, 6]. Так как множество состояний программы и путей между ними представляют собой граф, называемый графом исполнения [4], новые данные для анализа могут быть предоставлены графовыми нейронными сетями [8] (англ. *Graph Neural Networks, GNN*). Графовые нейронные сети могут улавливать зависимости между вершинами в графах, что можно использовать для разработки стратегии выбора оптимальных путей в графе исполнения, учитывая не только особенности состояний, но и взаимосвязи между ними.

Хотя существующие решения позволяют использовать машинное обучение для создания стратегий выбора оптимального пути, нельзя сказать, что эти решения подходят для внедрения графовых нейронных сетей и их обучения. Таким образом, возникает идея создания собственного набора инструментов для осуществления обучения.

При разработке инфраструктуры для создания стратегии выбора пути с использованием машинного обучения можно выделить несколько основных пунктов. К ним относятся проектирование и реализация алгоритма взаимодействия с символьной машиной во время обучения, рациональное использование вычислительных ресурсов, инструменты внедрения результатов обучения, визуализация и сравнение результатов обучения. В данной работе будут рассмотрены разработанные решения, адресующие вышеописанные пункты.

1. Постановка задачи

Цель работы: реализация набора инструментов, предназначенного для обучения графовых нейронных сетей выбору оптимального пути для символьных машин.

Для достижения этой цели были сформулированы следующие задачи.

1. Спроектировать инфраструктуру для обучения графовых нейронных сетей выбору оптимального пути.

2. Реализовать инструмент, использующий машинное обучение нейронных сетей во время взаимодействия с сервером обучения для реализации алгоритма выбора оптимального пути.

3. Реализовать протокол взаимодействия с сервером обучения для конкретной символьной машины V♯ с целью получения сигнала о начале и окончании взаимодействия и состоянии символьного исполнения во время обучения.

4. Реализовать возможность сравнения результатов работы различных нейронных сетей.

5. Реализовать компонент, способный обеспечить возможность интеграции графовых нейронных сетей в качестве механизма выбора пути.

2. Термины

В данном разделе приведено описание основных терминов и понятий, используемых в работе.

2.1. Графы

Граф — это математическая абстракция реальной системы любой природы, объекты которой обладают парными связями. Граф как математический объект есть совокупность двух множеств — множества объектов, называемого множеством вершин, и множества их парных связей, называемого множеством рёбер. Элемент множества рёбер есть пара элементов множества вершин.

Графы находят широкое применение в современной науке и технике. Они используются в естественных науках (физике и химии) и в социальных науках (например, социологии), но наибольшего масштаба применение графов достигло в информатике и сетевых технологиях.

Отношение доминирования на графе определяется следующим образом: вершина v доминирует над вершиной $w \neq v$ в графе G, если путь из любой вершины r в w содержит v [14].

Поток управления программы. В программировании предполагается, что процесс выполнения программы заключается в выполнении её инструкций вычислителем. Переход к выполнению следующей по порядку инструкции называется передачей управления. Последовательность передач управления в процессе выполнения программы формирует её поток управления.

Базовый блок. Последовательность инструкций образует базовый блок, если:

1. Инструкция в каждой позиции доминирует (всегда выполняется раньше) всех инструкций в более поздних позициях.

2. Никакая другая инструкция не выполняется между двумя инструкциями в последовательности.

В таких условиях, когда выполняется первая инструкция в базовом блоке, остальные инструкции обязательно выполняются ровно один раз и по порядку [22].

Граф потока управления (Граф исполнения, Control Flow Graph, CFG). CFG является направленным графом, имеющим вершину для каждого базового блока и ребро (ветвь) для каждой возможной передачи управления [4]. В качестве примера на рисунках 1 и 2 приведены программа и её граф исполнения.

```
1 def not_abs(x: int) -> int:
2     x = x + 10
3     print(f"value of x is {x}")
4     if x >= 0:
5         print("do nothing")
6     elif x < 0:
7         x = x * -1
8         print("invert x's sign")
9     if x < 0:
10        assert False
11    return x
```

Рис. 1: Код программы

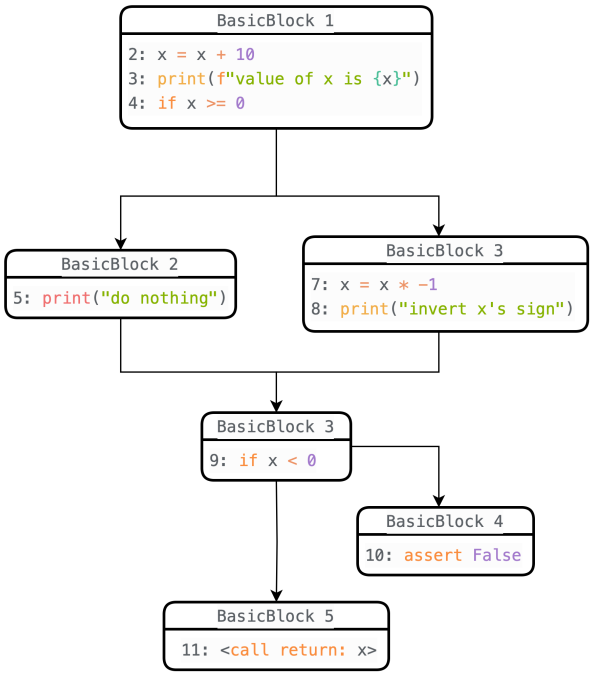

Рис. 2: Граф исполнения программы

Статический анализ кода — анализ кода, производимый без исполнения программы (в отличие от динамического анализа).

Тестовое покрытие — отношение покрытых тестами рёбер графа исполнения или инструкций программы к общему количеству рёбер или инструкций (покрытие ветвей и покрытие строк кода соответственно), в процентах.

2.2. Символьное исполнение и анализ программ

Символьное исполнение — техника анализа программы, позволяющая исследовать все возможные пути (ветви) исполнения программы путём использования символьных данных вместо реальных. Во время символьного исполнения программы конкретные переменные заменяются на символьные: такую замену можно сравнить с конкретными числами и переменными в формуле.

Процесс символьного исполнения можно представить как постепенное продвижение символьных состояний исполнения по графу исполнения программы. Во время продвижения состояние собирает условия пути для каждого пройденного оператора ветвления, накладывающие ограничения на конкретные данные, которые могли бы привести к попаданию программы в текущее состояние при конкретном исполнении. Решив условия пути для произвольного состояния, можно определить набор таких конкретных данных. Эти данные позволяют сгенерировать тест, воспроизводящий символьное состояние.

На рисунке 3 показано содержание условий пути для части графа после символьного исполнения. Так как программа на изображении 1 подразумевает взятие модуля от числа на строках $4 - 8$, то в блоке, содержащем строку 10, в условии пути содержится формула $x >= 0$. В таком случае состояние программы, приходящее в *BasicBlock 4* недостижимо, так как результирующие условия пути для этого блока не могут быть выполнены одновременно.

2.3. Машинное обучение

Эвристический алгоритм (эвристика) — это алгоритм решения задачи, который не является гарантированно точным или оптимальным, но достаточный для решения поставленной задачи. Он позволяет ускорить решение задачи в тех случаях, когда точное решение не может быть найдено.

Q-learning — это алгоритм обучения с подкреплением, позволяющий определять предпочтительность действия из набора возможных

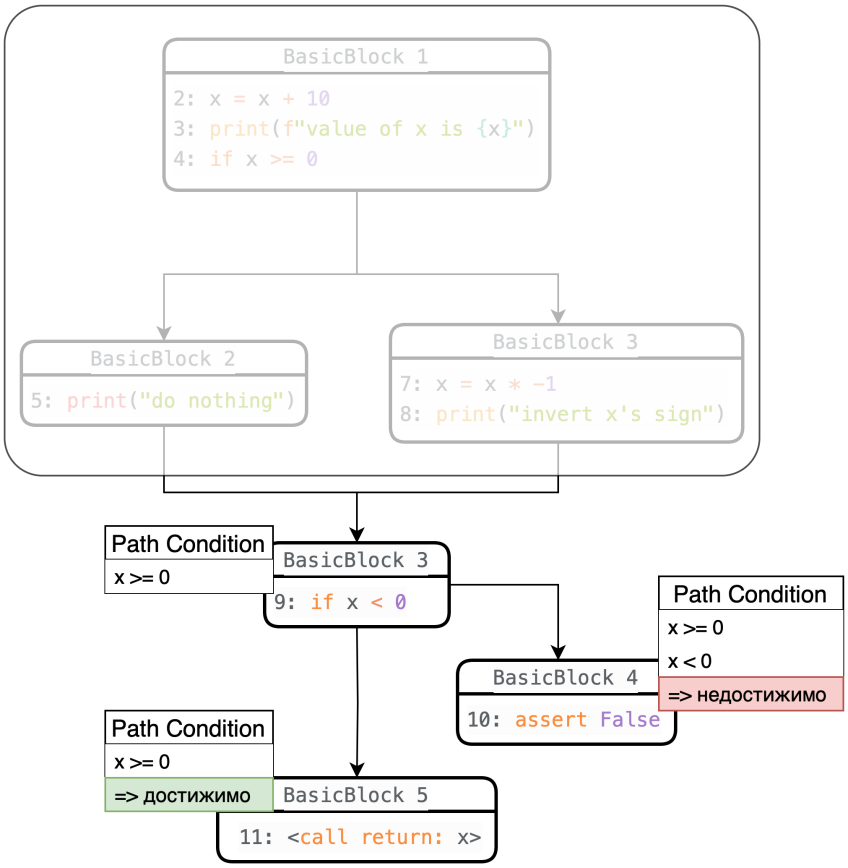

Рис. 3: Пример символьного исполнения программы

действий для каждого состояния системы. Информация о предпочитаемых действиях в определённых состояниях хранится в **Q-table**, которая обновляется по ходу обучения: каждый шаг алгоритма должен быть отмечен наградой, которая изменяет значение предпочтительности для совершаемого шага [20].

3. Обзор инфраструктур для машинного обучения

Существует широкий спектр инструментов для решения различных задач, связанных с применением машинного обучения. Рассмотрим некоторые из них.

В качестве базового решения для реализации инфраструктуры можно рассматривать инструменты для обучения с подкреплением. Обучение с подкреплением — метод машинного обучения, в ходе которого агент (модель) обучается во время взаимодействия с какой-либо средой. В качестве среды может выступать символьная машина, в качестве агента — графовая нейронная сеть.

Один из таких инструментов — Gymnasium. Он позволяет задавать пользовательское окружение и агентов для взаимодействия с ним. Gymnasium поддерживает создание нескольких агентов параллельно, установку временных ограничений для сеансов взаимодействия, а также использование и визуализацию результатов взаимодействия с окружением [9]. Несмотря на возможности по реализации пользовательских окружений и агентов, а также визуализации результатов взаимодействия, преимущества Gymnasium по сравнению с собственными решениями не очевидны. Использование Gymnasium требует согласования инфраструктуры кода с инфраструктурой фреймворка для получения визуализаций, механизм настройки которых может оказаться недостаточно гибким для целей исследования. Несмотря на непригодность инструмента в целом, идея обучения с подкреплением о взаимодействии агента со средой применима для отслеживания процесса работы модели машинного обучения во время фазы валидации.

Другим примером инфраструктуры для обучения является платформа MLflow[1]. MLflow представляет собой многокомпонентное решение для коллаборации (MLflow Projects[2]), разработки

[1]MLflow: A Machine Learning Lifecycle Platform https://github.com/mlflow/mlflow (дата обращения: 30.05.2024)

[2]MLflow Projects: инструмент для воспроизведения запусков https://mlflow.org/docs/latest/projects.html (дата обращения: 30.05.2024)

(MLflow Tracking[3]) и развертывания (MLflow Models[4], MLflow Model Registry[5]) моделей машинного обучения. Несмотря на преимущества в удобстве разработки моделей при использовании MLflow, на начальном этапе разработки и прототипирования данное решение не является жизненно важным и может быть внедрено на более позднем этапе создания продукта.

[3]MLflow Tracking: API для экспериментов в области машинного обучения https://mlflow.org/docs/latest/tracking.html (дата обращения: 30.05.2024)

[4]MLflow Models: формат и инструменты для развертывания моделей https://mlflow.org/docs/latest/models.html (дата обращения: 30.05.2024)

[5]MLflow Model Registry: хранение и управление жизненным циклом моделей https://mlflow.org/docs/latest/model-registry.html (дата обращения: 30.05.2024)

4. Обзор инфраструктур, использующих символьные машины

Инфраструктурные решения для машинного обучения уже существуют и успешно применяются в области создания стратегий выбора оптимального пути.

Рассмотрим их с точки зрения используемых символьных машин и алгоритмов машинного обучения.

LEARCH. Самый известный подход с использованием машинного обучения — LEARCH. Основной компонент LEARCH — модель машинного обучения, оценивающая награду за выбор состояния. Для каждого состояния вычисляется вектор признаков, который затем подаётся в модель машинного обучения для оценки [13]. Для создания набора данных для обучения и оценки моделей LEARCH использует символьную машину KLEE. Несмотря на то, что подход предоставляет возможность использовать нейронные сети в качестве модели предсказания пути, LEARCH сильно связан с KLEE через использование PYTHON-кода из C++-кода, что значительно усложняет потенциальному пользователю интеграцию своих символьных машин.

Q-KLEE. В данном подходе авторы используют Q-learning [20] — метод машинного обучения с подкреплением для обучения выбору оптимального пути, статический анализ программы для определения инструкций, которые Q-learning будет оценивать посредством награды, и KLEE в качестве символьной машины [21]. Принципиально Q-learning может быть заменён графовой нейронной сетью, но отсутствие исходного кода инструмента не позволяет переиспользовать инфраструктурные решения.

ParaDySE. В данном подходе авторы представляют решение для автоматической генерации эвристических стратегий выбора путей для символьного исполнения. В качестве символьных машин ParaDySE использует KLEE [2] и CREST [1]. В процессе символьного исполнения программы ParaDySE оптимизирует функцию выбора состояния путём подбора весов вектора признаков состояний исполнения (например,

находится ли ветвь в цикле), составляющих оптимальный эвристический алгоритм для конкретной программы [6]. В процессе изучения исходного кода инструмента выяснилось, что кода интеграции с KLEE в открытом доступе нет и существующая инфраструктура жёстко связана с символьной машиной CREST.

Выводы

Хотя существующие подходы используют машинное обучение для создания оптимального алгоритма выбора пути для символьного исполнения, рассмотренные подходы не предполагают использование графовых нейронных сетей и/или реализованы на базе конкретных символьных машин. Таким образом, возникает необходимость в разработке нового набора инструментов.

5. Архитектура

Назначение работы. Данный набор инструментов разработан для обучения графовых нейронных сетей выбору оптимальных путей исполнения в графе программы, обеспечения возможности внедрения результатов обучения и сравнения результатов работы с другими стратегиями выбора пути.

Требования. Механизм обучения является стандартным для данной области и состоит из чередующихся фаз обучения и валидации. Во время фазы обучения нейронная сеть использует существующие данные для подбора оптимальных весов нейронов, во время фазы валидации инфраструктура действует как внешний сервер для инференса (англ. *inference — вычисление, вывод*) нейронной сети при взаимодействии с сервером-обёрткой. Во время валидации происходит оценка работы модели на валидационном наборе данных, результаты которой записываются в логи.

Для реализации механизма обучения с использованием символьных машин были выделены основные компоненты для реализации: модуль для взаимодействия с серверами, модуль для конвертации результатов обучения в общепринятый формат для интеграции с символьными машинами, модуль для визуализации результатов. Отдельной задачей является ускорение процесса обучения.

UML-диаграмма компонентов. Разработанное решение состоит из нескольких компонентов, UML-диаграмма которых отображена на рисунке 4. Подробно рассмотрим каждый из них.

PySymGym. Данный компонент содержит составляющие машинного обучения: алгоритм обучения и валидации. Алгоритм валидации использует Connector для взаимодействия с сервером-обёрткой над символьной машиной, используя веб-сокеты в качестве протокола взаимодействия. Во время фазы валидации модель управляет символьным исполнением подключённой символьной машины, каждый шаг логгируется для анализа возможных ошибок в протоколе взаимодействия. Задачу управления серверами с символьными машинами решает Broker. Этот

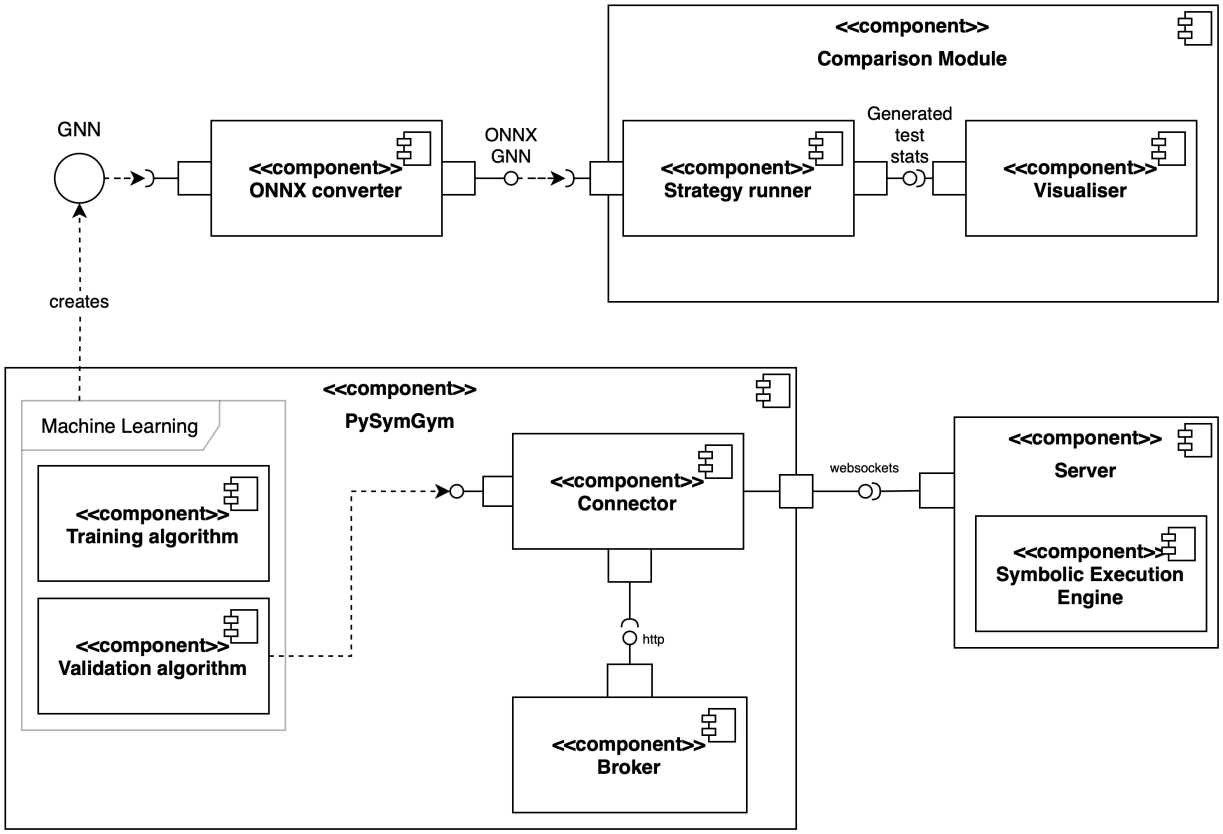

Рис. 4: Схема компонентов

компонент содержит информацию о максимальном количестве запускаемых серверов, запускает их и выдаёт сетевые адреса их веб-сокетов по запросу Connector по протоколу HTTP.

Server. Данный компонент является обёрткой над символьной машиной и позволяет управлять ей с помощью заданного протокола общения. Разработанный протокол позволяет использовать различные символьные машины в качестве среды для валидации при условии реализации обёртки в виде сервера с реализованным протоколом.

ONNX converter. В результате обучения компонент PySymGym сохраняет обученные модели. Для использования модели в качестве механизма выбора пути необходимо преобразовать модель в формат ONNX.

Comparison Module. После конвертации обученной модели можно сравнить её результативность с существующими стратегиями, в том числе и с другими версиями модели, с помощью компонента Strategy

RUNNER. Этот компонент принимает на вход название стратегии и программы, после чего запускает символьную машину, генерирует тесты для заданных программ, анализирует и сохраняет статистику по всем сгенерированным тестам. Имея два набора результатов запуска стратегий, можно визуализировать их сравнение, используя VISUALISER. Этот компонент создаёт параметризованное сравнение двух стратегий в виде графика.

Пользовательский сценарий. IDEF0-диаграмма пользовательского сценария представлена на рисунке 5. Для начала работы с инструментом пользователю необходимо предоставить описание скомпилированных программ в *json*-формате в качестве тестовой и валидационной выборок, конфигурацию серверов-обёрток в формате *yaml*. Когда все необходимые зависимости будут переданы, пользователь может запустить обучение. В процессе работы этапа 1, занимающего самую значительную часть времени, графовая нейронная сеть будет обучаться. После завершения обучения пользователь передаёт обученную нейронную сеть в компонент конвертации, после чего запускает компонент оценки стратегии. В результате работы компонента оценки пользователь получает данные о результатах работы стратегии, которые затем передаёт в компонент визуализации вместе с оппонирующей стратегией. После завершения генерации визуализации визуализатор сохраняет результаты сравнения на диск.

Использованные технологии. Основным языком, на котором велась разработка, является PYTHON3. Его использование обосновано большим количеством существующих библиотек для машинного обучения. Для анализа и визуализации данных выбрана связка библиотек PANDAS [16] и MATPLOTLIB [10], такой выбор обусловлен большим размером сообщества разработчиков, использующих данные инструменты. В качестве символьной машины был выбран V♯, так как команда разработки имеет большой опыт в использовании данного инструмента.

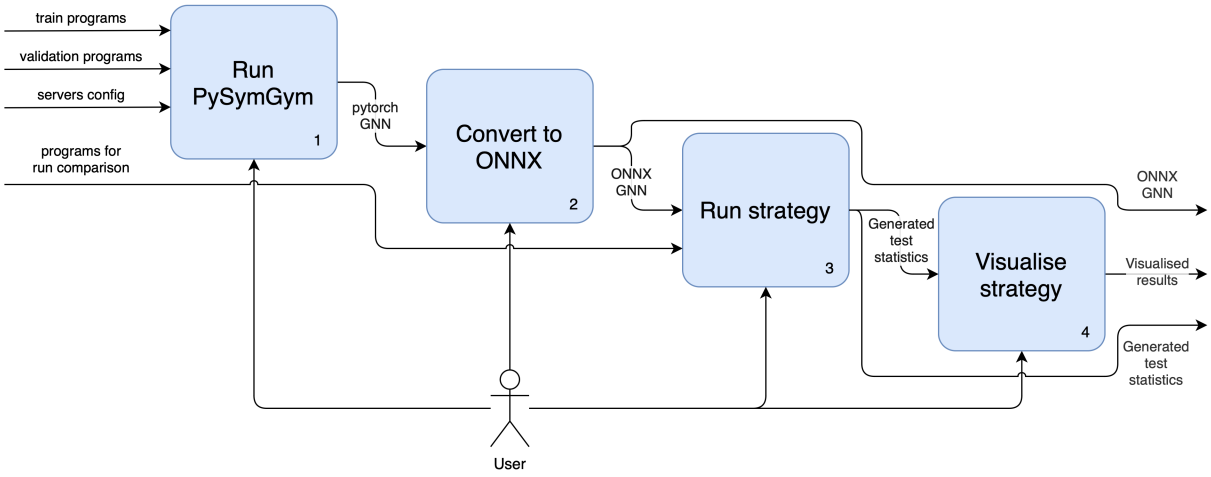

Рис. 5: Пользовательский сценарий в нотации IDEF0

6. Реализация

Данный раздел посвещен описанию особенности реализации основных задач работы.

6.1. Взаимодействие с символьной машиной

Одним из требований к протоколу взаимодействия с символьной машиной является возможность при обучении использовать различные символьные машины: таким образом пользователь инфраструктуры будет иметь возможность использовать решение с учётом специфики своих нужд.

С учётом такого требования был разработан подход, позволяющий клиенту написать обёртку на выбранную символьную машину для реализации интерфейса компонента PySymGym. Эта обёртка должна представлять собой вебсокет-сервер, реагирующий на команды управления компонентом обучения. Диаграмма последовательности процесса взаимодействия представлена на рисунке 6. За реализацию интерфейса взаимодействия со стороны компонента обучения отвечает компонент Connector. Этот компонент может получать и обрабатывать сообщения, содержащие актуальное состояние графа исполнения и информацию о завершении работы символьной машины. Connector отправ-

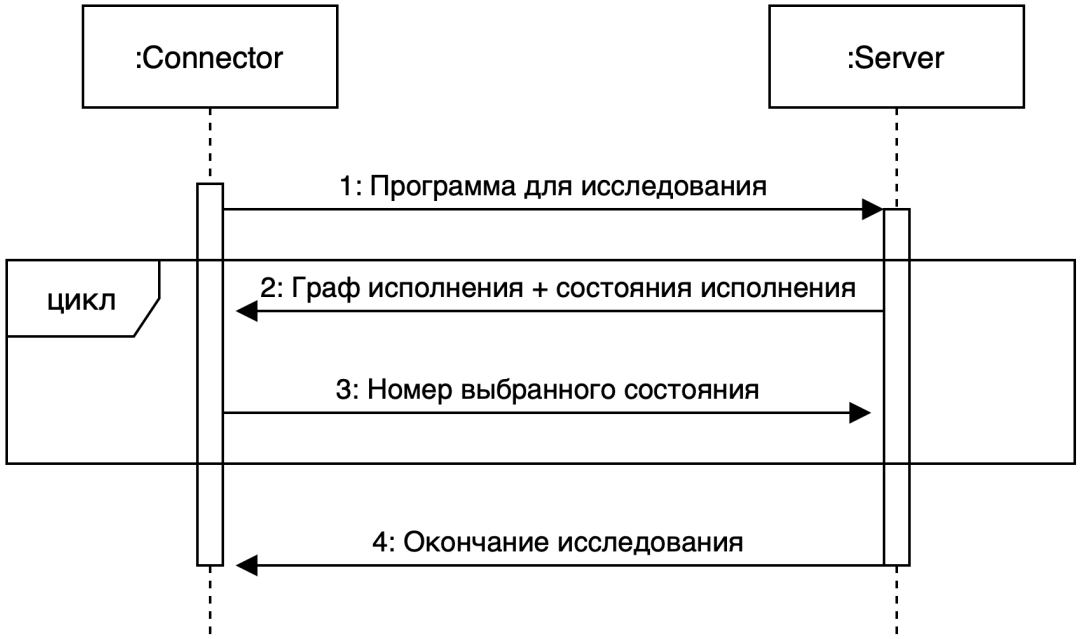

Рис. 6: Диаграмма последовательности протокола взаимодействия

ляет серверу сообщения о начале работы символьной машины и выбранном для исследования состоянии исполнения. На данный момент реализована обёртка над символьными машинами V♯ и USVM [19].

6.2. Ускорение обучения

Процесс обучения состоит из фазы обучения и валидации, во время последней происходит взаимодействие с символьной машиной, таким образом можно напрямую оценить результаты обученной модели. При таком подходе возникает проблема: из-за количества валидационных программ и сравнительно медленной работы символьной машины на большом количестве шагов, общее время на валидацию может составлять несколько часов. Для ускорения валидационной фазы было принято решение запускать несколько серверов-обёрток и осуществлять валидацию нескольких программ параллельно.

Для реализации этой идеи необходимо было решить проблему создания и управления несколькими серверами и проблему распараллеливания алгоритма валидации. В качестве решения первой проблемы был реализован компонент Broker. В его обязанности входит запуск и уни-

чтожение серверов, передача адресов успешно запущенных серверов в компонент Connector для обеспечения взаимодействия алгоритма валидации и сервера. Внутри Broker хранит очередь из конфигураций серверов, используя которую при запросе от Connector Broker запускает сервер. Для решения второй проблемы был использован пул процессов: задачи валидации распределяются по нескольким процессам, после завершения работы результаты задач сохраняются в компонент статистики.

6.3. Интеграция модели с символьной машиной

Во время разработки инструмента было рассмотрено три принципиальных подхода к использованию нейронной сети вместе с символьной машиной.

Первым и самым простым вариантом является реализация сервера на python, принимающего данные о графе исполнения и возвращающего предсказание о наиболее предпочтительном состоянии для продвижения. Этот подход позволяет использовать модуль машинного обучения «нативно», переиспользуя код из алгоритма обучения модели, что в том числе позволяет не решать проблему загрузки весов нейронной сети на GPU для ускорения работы модели. Минусом данного решения являются расходы на коммуникацию с сервером: состояние графа исполнения при длительном исследовании может быстро взрываться, его передача на каждый шаг символьной машины становится узким местом.

Вторым, более интересным решением является использование *биндингов* (англ. bindings, *привязки*), такой подход используют авторы LEARCH. Биндинги — это библиотеки-обёртки, которые служат мостом между двумя языками программирования. Таким образом, библиотека, написанная для одного языка, может быть неявно использована и в другом языке. Этот подход позволяет преодолеть проблему расходов на коммуникацию: данные для инференса могут быть считаны из общей области памяти процесса-клиента, использующего модель. Однако биндинги придется поддерживать для каждой программной платфор-

мы символьных машин. Также выполнение кода все еще будет выполняться виртуальной машиной PYTHON, что может сказаться на производительности.

Третьим решением, выбранным для реализации в данной работе, является преобразование нейронной сети в формат ONNX. Этот формат предназначен для представления моделей машинного обучения: ONNX определяет общий формат файлов, чтобы разработчики могли использовать модели в различных инструментах, средах выполнения и компиляторах [15]. При этом библиотека для использования моделей такого формата поддерживается *Microsoft* и реализована на множестве программных платформ [23].

6.4. Анализ результатов

Для получения данных и визуализации результатов экспериментов был разработан компонент сравнения результатов работы символьной машины с использованием различных стратегий выбора пути исполнения, на схеме компонентов он обозначен как COMPARISON MODULE. Зависимый компонент STRATEGY RUNNER запускает процесс символьной машины с запросом на генерацию тестов для выбранных программ и считывает результаты. Затем STRATEGY RUNNER сохраняет количество сгенерированных тестов и ошибок, измеряет суммарное покрытие сгенерированных тестов и время работы.

После получения результатов производится сравнение двух стратегий с помощью VISUALISER. Этот компонент может быть сконфигурирован с помощью различных опций. Пользователь может выбрать:

- Исходные метрики для сравнения: тесты, ошибки, покрытие, время;
- Способ объединения таблиц с данными по именам методов по аналогии с операциями над таблицами в SQL: $OuterJoin$, $InnerJoin$, $InnerJoinWithEqualCoverage$ (последний нужен для сравнения только тех методов, где покрытие двух стратегий одинаково);

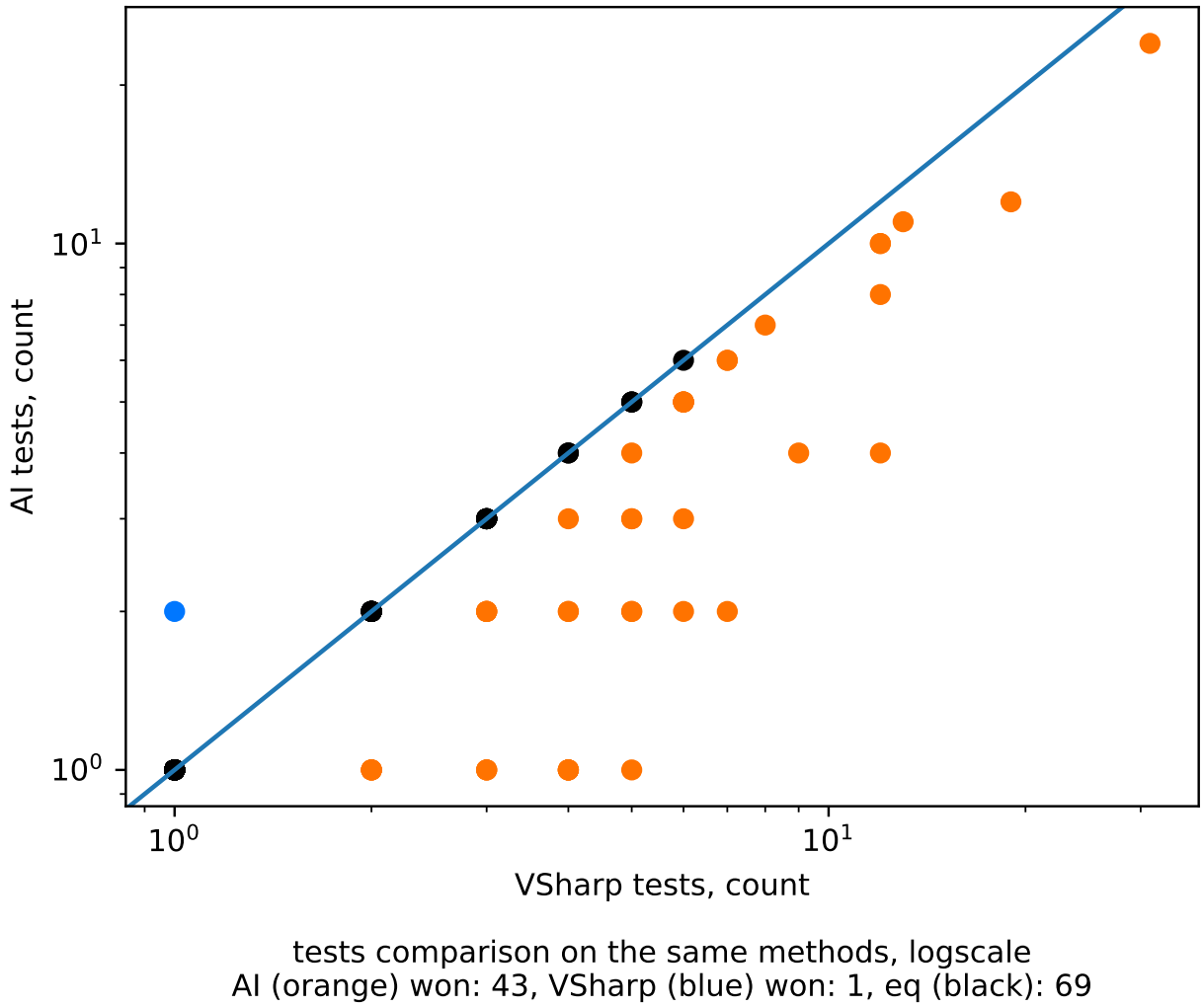

Рис. 7: Сравнение по количеству сгенерированных тестов с существующей стратегией на методах с одинаковым покрытием для двух стратегий. Оранжевым отмечены точки, где предложенный подход выигрывает сравнение.

- Различные настройки графика: название эксперимента, имя метрики, линию разделения, использование логарифмической шкалы, выигрывает ли меньшее или большее значение метрики.

На рисунке 7 изображён график, созданный визуализатором. На нём представлено сравнение результатов работы символьной машины с использованием модели, обученной с помощью предложенной инфраструктуры, с существующими стратегиями.

7. Сравнение с аналогами

Для оценки разработанного инфраструктурного решения приведём сравнение с существующими инфраструктурами для обучения моделей выбора оптимального пути в графе исполнения по параметрам, актуальным для реализуемой идеи.

7.1. Параметры сравнения

Введём параметры, по которым будет осуществляться сравнение инструментов:

1. **Внедрение**: Существует ли в открытом доступе механизм для использования результатов обучения в алгоритме выбора пути с какой-либо символьной машиной.

2. **Ускорение обучения**: Есть ли возможность ускорить обучение за счёт параллельного исполнения кода.

3. **Воспроизведение и визуализация**: Существуют ли в открытом доступе инструменты для воспроизведения экспериментов и визуализации результатов обучения.

4. **Несколько символьных машин**: Возможна ли интеграция с несколькими символьными машинами одновременно, если да, то какие поддержаны.

5. **Логгирование шагов при взаимодействии с символьной машиной**: Реализован ли механизм логгирования шагов при взаимодействии с символьной машиной.

7.2. Результаты

Для оценки инфраструктурных решений использовались репозитории проектов, находящиеся в открытом доступе.

LEARCH поддерживает внедрение и использование обученной модели с символьной машиной KLEE. Репозиторий проекта содержит скрипты для запуска стратегий выбора пути с обученными моделями и отображения результатов в виде таблицы.

ParaDySE поддерживает внедрение полученных векторов весов признаков в символьную машину CREST.

Репозиторий Q-KLEE в открытом доступе не найден.

Инструмент	Внедрение	Ускорение	Воспроизведение и визуализация	Несколько символьных машин
PySymGym	+	+	+	+ (V♯, USVM)
LEARCH	+	—	+	—
ParaDySE	+	—	—	—
Q-KLEE	—	—	—	—

Таблица 1: Сравнение инфраструктур, использующих машинное обучение

7.3. Обсуждение результатов

Результаты сравнения показывают различие в фокусе исследований. Для LEARCH и ParaDySE использование нескольких символьных машин не являлось основной целью. В PySymGym акцент сделан на переносимость подхода, поэтому была реализована возможность интеграции сторонних символьных машин. То же самое относится к ускорению и визуализации: идеи, реализованные в PySymGym, имеют экспериментальную природу, поэтому инфраструктура включает инструменты для быстрой и интерпретируемой обратной связи.

Заключение

В ходе данного исследования был разработан и реализован набор инструментов, предназначенный для обучения нейронных сетей выбору оптимального пути для символьного исполнения программ.

1. Спроектирована инфраструктура инструмента для обучения графовых нейронных сетей выбору оптимального пути.

2. Реализован инструмент, позволяющий осуществлять обучение нейронных сетей и валидировать результаты обучения с помощью взаимодействия с символьной машиной.

3. Реализован протокол взаимодействия с символьной машиной V♯ через сервер-обертку, использующий веб-сокеты для передачи сообщений.

4. Создан компонент для сравнения результатов работы различных нейронных сетей как алгоритма выбора оптимального пути исполнения на различных программах.

5. Создан компонент для конвертации нейронных сетей в формат ONNX, с его помощью выполнено внедрение нейронной сети в качестве механизма выбора пути для символьного исполнения в символьную машину V♯.

Проведённое сравнение показало, что существующие инструменты уступают разработанному решению по параметрам, актуальным для исследуемого подхода. Разработанный набор инструментов позволяет использовать различные символьные машины как в качестве окружения для обучения, ускорить процесс обучения посредством параллельной валидации, экспортировать результаты обучения в общепринятый формат, сравнивать стратегии символьных машин.

Исходный код набора инструментов: https://github.com/gsvgit/PySymGym/tree/dev.

Аккаунт автора работы на GitHub: https://github.com/emnigma.

Список литературы

[1] Burnim J., Sen K. Heuristics for Scalable Dynamic Test Generation // Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering. — ASE '08. — USA : IEEE Computer Society, 2008. — P. 443–446. — URL: https://doi.org/10.1109/ASE.2008.69.

[2] Cadar Cristian, Dunbar Daniel, Engler Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — OSDI'08. — USA : USENIX Association, 2008. — P. 209–224.

[3] Cadar Cristian, Sen Koushik. Symbolic execution for software testing: three decades later // Commun. ACM. — 2013. — feb. — Vol. 56, no. 2. — P. 82–90. — URL: https://doi.org/10.1145/2408776.2408795.

[4] Cooper Keith D., Torczon Linda. Chapter 4 - Intermediate Representations // Engineering a Compiler (Third Edition) / Ed. by Keith D. Cooper, Linda Torczon. — Philadelphia : Morgan Kaufmann, 2023. — P. 159–207. — URL: https://www.sciencedirect.com/science/article/pii/B9780128154120000103.

[5] Dijkstra testing quote. — https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html. — Accessed: 2023-09-23.

[6] Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics / Sooyoung Cha, Seongjoon Hong, Jiseong Bak et al. // IEEE Transactions on Software Engineering. — 2022. — Vol. 48, no. 9. — P. 3640–3663.

[7] Godefroid Patrice, Levin Michael Y., Molnar David. SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact

at Microsoft. // Queue. — 2012. — jan. — Vol. 10, no. 1. — P. 20–27. — URL: https://doi.org/10.1145/2090147.2094081.

[8] The Graph Neural Network Model / Franco Scarselli, Marco Gori, Ah Chung Tsoi et al. // IEEE Transactions on Neural Networks. — 2009. — Vol. 20, no. 1. — P. 61–80.

[9] Towers Mark, Terry Jordan K., Kwiatkowski Ariel et al. Gymnasium. — 2023. — . — URL: https://zenodo.org/record/8127025 (дата обращения: 2023-07-08).

[10] Hunter J. D. Matplotlib: A 2D graphics environment // Computing in Science & Engineering. — 2007. — Vol. 9, no. 3. — P. 90–95.

[11] IntelliTest. — https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/?view=vs-2022. — Accessed: 2023-09-23.

[12] King James C. Symbolic Execution and Program Testing // Commun. ACM. — 1976. — jul. — Vol. 19, no. 7. — P. 385–394. — URL: https://doi.org/10.1145/360248.360252.

[13] Learning to Explore Paths for Symbolic Execution / Jingxuan He, Gishor Sivanrupan, Petar Tsankov, Martin Vechev // Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. — CCS '21. — New York, NY, USA : Association for Computing Machinery, 2021. — P. 2526–2540. — URL: https://doi.org/10.1145/3460120.3484813.

[14] Lengauer Thomas, Tarjan Robert Endre. A Fast Algorithm for Finding Dominators in a Flowgraph // ACM Trans. Program. Lang. Syst. — 1979. — jan. — Vol. 1, no. 1. — P. 121–141. — URL: https://doi.org/10.1145/357062.357071.

[15] Bai Junjie, Lu Fang, Zhang Ke et al. ONNX: Open Neural Network Exchange. — https://github.com/onnx/onnx. — 2019. — Accessed: 2024-05-13.

[16] Pandas. — https://pandas.pydata.org. — 2024. — Accessed: 2024-05-13.

[17] Program Analysis via Satisfiability modulo Path Programs / William R. Harris, Sriram Sankaranarayanan, Franjo Ivančić, Aarti Gupta // SIGPLAN Not. — 2010. — jan. — Vol. 45, no. 1. — P. 71–82. — URL: https://doi.org/10.1145/1707801.1706309.

[18] Symflower. Symflower - Automated Unit Testing for Software Developers. — https://symflower.com/en/. — Accessed: 2024-05-20.

[19] UnitTestBot: Automated Unit Test Generation for C Code in Integrated Development Environments / Dmitry Ivanov, Alexey Babushkin, Saveliy Grigoryev et al. // 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). — 2023. — P. 380–384.

[20] Watkins Christopher J. C. H., Dayan Peter. Q-learning // Machine Learning. — 1992. — . — Vol. 8, no. 3. — P. 279–292. — URL: https://doi.org/10.1007/BF00992698.

[21] Wu Jie, Zhang Chengyu, Pu Geguang. Reinforcement Learning Guided Symbolic Execution // 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). — 2020. — P. 662–663.

[22] Yousefi Javad, Sedaghat Yasser, Rezaee Mohammadreza. Masking wrong-successor Control Flow Errors employing data redundancy // 2015 5th International Conference on Computer and Knowledge Engineering (ICCKE). — 2015. — P. 201–205.

[23] developers ONNX Runtime. ONNX Runtime. — https://onnxruntime.ai/. — 2024. — Accessed: 2024-05-13.