НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування

і спеціалізованих комп'ютерних систем

«На правах рукопису»
УДК _____

До захисту допущено:
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
«___»_____2023 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Системне програмування

та спеціалізовані комп'ютерні системи

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Способи автоматизованого тестування програм з використанням глибоких нейронних мереж**»**

Виконав:
студент ІІ курсу, групи КВ-11мн
Юрченко Валентин Володимирович _____
Науковий керівник:
Професор, д.т.н.
Терейковський Ігор Анатолійович _____
Консультант з нормоконтролю:
доцент, к.т.н., с.н.с.
Боярінова Юлія Євгенівна _____
Рецензент:

_____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2023 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-наукова програма **«Системне програмування та спеціалізовані комп'ютерні системи»**

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___»_____202_ р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Юрченку Валентину Володимировичу

1. Тема дисертації «Способи автоматизованого тестування програм з використанням глибоких нейронних мереж», науковий керівник дисертації Терейковський Ігор Анатолійович, професор, д.т.н., затверджені наказом по університету від «30»березня 2023 р. № 1359-с

2. Термін подання студентом дисертації: 18 травня 2023 року.

3. Об'єкт дослідження: автоматизація тестування програм з використанням глибоких нейронних мереж.

4. Предмет дослідження: процеси автоматизації тестування програм з використанням глибоких нейронних мереж.

5. Перелік завдань, які потрібно розробити: ознайомитись з теоретичним матеріалом, розглянути методології тестування глибоких нейронних мереж, провести експерименти з різними методологіями.

6. Перелік графічного (ілюстративного) матеріалу: презентація.

7. Перелік публікацій: Доповідь на тему «Система автоматизованого аналізу безпеки API» на XV науково-практична конференція магістрантів та аспірантів ПМК-2022, тези на тему «Автоматизоване тестування програм з використанням глибоких нейронних мереж» на Міжнародна науково-практична конференція

«Наука, освіта, технології і суспільство в XXI столітті: наукові ідеї та механіхми реалізації».

8. Дата видачі завдання 20.10.2021р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вивчення матеріалів за темою дисертації. Визначення структури магістерської дисертації	20.02.2022	
2	Підготовка першого розділу дисертації	07.09.2022	
3	Підготовка другого розділу дисертації	11.12.2022	
4	Підготовка третього розділу дисертації	28.02.2023	
5	Проведення експериментів тестування нейронних мереж	17.03.2023	
6	Підготовка четвертого розділу дисертації	30.03.2023	
7	Оформлення пояснювальної записки дисертації	30.04.2023	
8	Попередній розгляд МД на кафедрі	4.05.2023	

Студент Юрченко Валентин Володимирович

Науковий керівник Терейковський Ігор Анатолійович

РЕФЕРАТ

Актуальність теми. Моделі машинного навчання (Machine Learning, ML) відіграють важливу роль в різних застосуваннях. Зокрема, в останні роки глибокі нейронні мережі (Deep neural networks, DNN) використовуються в різних галузях науки та техніки. З огляду на таке зростання використання, можливі помилки у моделях DNN можуть викликати серйозні стурбовання з приводу їхньої надійності та спричинити значні втрати. Тому виявлення помилкової поведінки в будь-якій системі машинного навчання, особливо в DNN, є критичним. Тестування програмного забезпечення - широко використовуваний механізм для виявлення помилок. Однак, оскільки точний вихід більшості моделей DNN не відомий для заданих вхідних даних, традиційні техніки тестування програмного забезпечення не можуть бути безпосередньо застосовані. Останніми роками було запропоновано кілька методів тестування та критеріїв адекватності для тестування DNN. У даній дисертації досліджується три типи методів тестування DNN з використанням текстових та зображень вхідних даних.

Об’єктом дослідження є автоматизація тестування програм з використанням глибоких нейронних мереж..

Предметом дослідження є способи автоматизованого тестування програм з використанням глибоких нейронних мереж.

Мета роботи: розробка ефективних методів тестування програм з використанням глибоких нейронних мереж

Наукова новизна: Запропоновано методи тестування та критерії оцінювання їх ефективності для програм з використанням глибоких нейронних мереж, що за рахунок часткового об’єднання різних методологій в запропоновану одну, дозволяє підвищити точність тестування та оптимізувати сам процес тестування.

Практична цінність: Автоматизоване тестування програм з використанням глибоких нейронних- мереж значно полегшує тестування таких складних програм та значно покращує якість тестування, оскільки при такому підході виключається людський фактор і можливість попадання будь-яких вразливостей системи до кінцевого користувача.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на XV науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2022 та на міжнародній науково-практичній конференції «Наука, освіта, технології і суспільство в XXI столітті: наукові ідеї та механіхми реалізації».

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи, наведено відомості результатів і їхнє впровадження.

У першому розділі було викладено загальну теорію по глибоким нейронним мережам. Детально розглянуто архітектуру нейронних мереж та їхні ключові складові. Розглянуто способи тестування таких нейронних мереж.

У другому розділі було розглянуто декілька методів тестування глибоких нейронних мереж, визначено їхні переваги та недоліки.

У третьому розділі було проведено три експерименти з різними методами тестування глибоких нейронних мереж.

У четвертому розділі наведено інформацію про нову запропоновану методологію, яка в собі поєднує кращі якості з попередньо розглянутих інсуючих методологій.

У висновках представлені результати проведеної роботи.

Робота представлена на 90 аркушах, містить посилання на список використаних літературних джерел.

Ключові слова: Нейронні мережі, тестування, автоматизація, Deep neural networks.

ABSTRACT

Actuality of theme.

Machine Learning (ML) models play an important role in various applications. In particular, in recent years, Deep Neural Networks (DNNs) have been used in a variety of applications. Given this increase in use, possible errors in DNN models can raise serious concerns about their reliability and cause significant losses. Therefore, detecting erroneous behavior in any machine learning system, especially in DNNs, is critical. Software testing is a widely used mechanism for detecting errors. However, since the exact output of most DNN models is not known for a given input, traditional software testing techniques cannot be directly applied. In recent years, several testing methods and adequacy criteria have been proposed for testing DNNs. In this thesis, we investigate three types of testing methods for DNNs using text and image inputs.

The object of research is automation of program testing using deep neural networks..

The subject of research is methods for creating automated testing of programs using neural networks.

Purpose: development of effective methods for testing programs using deep neural networks.

Scientific novelty: Testing methods and criteria for evaluating their effectiveness for programs using deep neural networks are proposed.

Practical value: Automated testing of programs using deep neural networks greatly facilitates the testing of such complex programs and significantly improves the quality of testing, since this approach eliminates the human factor and the possibility of any system vulnerabilities reaching the end user.

Testing of the work. The main provisions and results of the work were presented and discussed at the XIV Scientific Conference of Undergraduate and Postgraduate Students "Applied Mathematics and Computer Science" PMK-2022.

Structure and scope of the work. The master's thesis consists of an introduction, four chapters and conclusions.

The introduction gives a general description of the work, assesses the current state of the problem, substantiates the relevance of the research area, formulates the purpose and objectives of the research, shows the scientific novelty of the results and the practical value of the work, and provides information on the results and their implementation.

In the first chapter, the general theory of deep neural networks is presented. The architecture of neural networks and their key components are discussed in detail. The ways of testing such neural networks are considered. Several types of deep neural network testing are discussed in detail, and their advantages and disadvantages are identified.

In the second section, several methods of testing deep neural networks were reviewed, their advantages and disadvantages were identified.

In the third section, three experiments with different methods of testing deep neural networks were conducted.

The fourth section analyzes the results of the experiments on the use of different methodologies for testing deep neural networks.

The conclusion presents the results of the work.

The work is presented on 90 pages and contains references to the list of used literature sources.

Keywords: Neural networks, testing, automation, Deep neural networks.

ЗМІСТ

ВСТУП

Алгоритми машинного навчання (Machine Learning, ML) [1] зараз використовуються в широкому спектрі областей програмування, таких як прогнозування і оцінка, генерація та синтез, трансформація та багато інших застосувань. Оскільки ці програмні системи на основі МН стають все більш популярними, помилки у цих програмних системах можуть викликати значні проблеми. Тестування програмного забезпечення - це широко використовуваний механізм для виявлення помилок у програмних системах. Тому, для запобігання критичних втрат через несправні програми, програмні застосунки, що базуються на МН, також повинні проходити перевірку за допомогою тестування програмного забезпечення.

Загалом, у тестуванні програмного забезпечення є дві основні проблеми: 1) пошук ефективних тестових вхідних даних, які дозволять виявляти помилки, та 2) надання способу визначення, чи працює система під час тестування, як очікувалося (успішний тест) або ні (невдалий тест), що називається "проблемою тестового оракула" [2]. Чи то ми проводимо ручне тестування, чи автоматизоване, ці проблеми повинні бути вирішені, проте зазвичай вони важче вирішувати, коли генерація тестів та їх оцінка повністю автоматизовані, що є контекстом даної дисертації.

Ці проблеми потрібно вирішувати незалежно від типу застосування, технології, і застосованих алгоритмів. Проте, програмні системи, що базуються на машинному навчанні, особливо ті, що використовують глибокі нейронні мережі, є більш складними і вимагають більше зусиль. Системи, реалізовані на основі глибоких нейронних мереж (DNN), відрізняються від традиційного програмного забезпечення. У традиційних програмних системах розробники явно реалізовують логіку вимог (у вигляді послідовності програмних блоків: присвоєння, цикли, умови тощо) в методі або функції. Однак в системах

глибокого навчання логіка не кодується явно, а навчається з тренувального набору даних з використанням глибокої нейронної мережі, яка представляється як набір ваг, що подаються на не-лінійні функції активації [3].

Загалом, завжди потрібно вирішувати ці складнощі щодо тестування програмного забезпечення незалежно від типу застосування, використаної технології та алгоритмів. Однак, програмні системи на основі машинного навчання, зокрема ті, які використовують глибокі нейронні мережі, є складнішими та більш викликають складності. Системи програмного забезпечення на основі глибоких нейронних мереж (DNNs) відрізняються від традиційного програмного забезпечення. У традиційних системах програмного забезпечення розробники експліцитно реалізовують логіку вимог (як послідовність програмних блоків, призначень, циклів та умов тощо) у вигляді методу або функції. Однак, в системах глибокого навчання, логіка не реалізована явно, а навчається з навчального набору даних за допомогою глибокої нейронної мережі, яка представляє собою набір вагів, що подаються на неелементарні функції активації [3].

Отже, щодо автоматизованого тестування стратегії генерації вхідних тестів та оракулів можуть відрізнятися від традиційних підходів до автоматизованого тестування. Наприклад, у більшості традиційних систем програмного забезпечення є явний очікуваний результат, який відрізняє тестові випадки з відмінним результатом (відмінний від того, що очікується). Однак, системи програмного забезпечення на основі глибокого навчання відомі як проблеми без оракулу [4]. Тому виявлення неправильної поведінки програмного забезпечення на базі DNN є іншим у порівнянні з традиційним програмним забезпеченням. Наприклад, існуючі критерії відповідності тестів для традиційного програмного забезпечення, такі як покриття операцій, гілок та потоку даних, повністю не ефективні для тестування DNN.

Згідно з недавнім дослідженням тестування програмного забезпечення на основі машинного навчання, Ліу та ін. [5], класифікували завдання тестування та відлагодження систем машинного навчання на шість груп: 1) Генерація

тестового оракулу, 2) Оцінка достатності тестів, 3) Генерація тестових вхідних даних, 4) Приоритезація та зменшення кількості тестів, 5) Аналіз звітів про помилки, та 6) Відлагодження та виправлення.

В даній дисертації я розглядаю три основні категорії тестування: 1) генерацію тестових оракулів, 2) оцінку адекватності тестів та 3) генерацію тестових вхідних даних. Я проводжу тестування моделей DNN за допомогою наведених вище методів тестування, організовуючи три експерименти.

По-перше, я пропоную метод для створення тестових оракулів для виявлення помилок в програмному забезпеченні на основі машинного навчання. По-друге, я емпірично оцінюю існуючі критерії адекватності тестування для DNN. Нарешті, я пропоную новий метод Guided Mutation (GM) для генерації тестових вхідних даних для оцінки моделей DNN та їх подальшого перетренування за допомогою нових вхідних даних.

В першому експерименті я застосував техніку перевірки оракулу для перевірки моделі глибокого навчання - Варіаційний автоенкодер (VAE) [6], яка є генеративною моделлю для зменшення розміру вхідних даних. Багатоімплементаційне тестування (MIT) [7] є одним з методів, який вирішує проблему "Відсутність оракула". MIT порівнює різні реалізації однієї і тієї ж моделі, використовуючи той самий тестовий ввід. Отримавши вихід кожної моделі, оракул вважається найбільш повторюваним вихідом серед усіх реалізацій. Якщо будь-який вихід моделі відрізняється від тестового оракула (з допустимим порогом), то вважається, що програма містить помилку.

У другому експерименті я зосередився на оцінці адекватності тестів. Серед усіх критеріїв адекватності тестів у традиційному тестуванні програмного забезпечення, "покриття коду" та "тестування мутацій" є двома популярними техніками, які також застосовуються у тестуванні машинного навчання і, зокрема, у тестуванні глибоких нейронних мереж. Тому у цьому експерименті фокус був зосереджений на критеріях, пов'язаних з покриттям та мутаціями коду.

У традиційному тестуванні програмного забезпечення, покриття коду визначає відсоток елементів вихідного коду (наприклад, операторів, гілок, умов), які були виконані тестовим набором [8]. Вище покриття є приблизним значенням вищої ймовірності виявлення помилок. У відміну від традиційного програмного забезпечення, програма на основі DNN не реалізує логіку програми через явні оператори, гілки та умови. Замість цього логіка вивчається через набір нейронів у нейронних мережах [9]. Тому останні роботи з тестування DNN вводять багато нових критеріїв покриття на основі "покриття нейронів", щоб оцінити, наскільки добре вхідні дані охопили DNN-модель [10].

Хоча Pei et al. [9] були першими, хто ввів метрику покриття нейронів (Neuron Coverage, NC) як метрику тестування DNN. NC визначається як співвідношення активованих нейронів для заданих тестових вхідних даних до загальної кількості нейронів у DNN-моделі. Після пропонування NC як нової метрики, багато дослідників проявили зацікавленість у дослідженні NC та ввели ще кращі метрики покриття. Наприклад, Kim et al. [11] ввели нове поняття для тестування систем машинного навчання, яке називається метрикою "Likelihood Surprise Coverage" (LSC). Вона визначає, наскільки несподіваним є результат випробування в порівнянні з вхідними даними навчання моделі.

Мутаційне тестування є однією з форм тестування білого ящика, яка використовується в традиційному тестуванні програмного забезпечення та передбачає модифікацію програми шляхом внесення невеликих змін [12]. Тестові випадки виявляють та відхиляють кожну модифіковану версію (мутант), спричиняючи різницю у поведінці мутанта від оригіналу. У мутаційному тестуванні потрібна метрика, щоб з'ясувати, наскільки добре тестові набори виявляють дефекти. Тому бал мутацій описується як співвідношення виявлених мутантів до всіх засіяних мутантів.

У машинному навчанні, Ma et al [13] запропонували підхід, який називається DeepMutation. Він полягає у мутації моделей DNN на рівні джерела коду або моделі, щоб внести незначні зміни у границю прийняття рішень в DNN. На основі цієї роботи було визначено метрику покриття мутацій як

відношення кількості тестових випадків, в яких результати змінилися на мутанті порівняно з оригінальною програмою, до загальної кількості тестових випадків.

Критерії адекватності тестування, такі як покриття коду або коефіцієнт мутацій, зазвичай є метриками, які люди використовують для оцінки своєї набору тестів або стратегії тестування. Однак у цьому дослідженні метою є оцінка самого критерію адекватності. У традиційному тестуванні програмного забезпечення це було б зроблено, оцінивши здатність метрик виявляти реальні помилки. У тестуванні на основі нейронних мереж, введення помилок та їх виявлення пов'язані з характеристиками моделі та її вимогами. Тому класичні метрики оцінки, такі як точність та повнота, можуть не бути підходящими для оцінки самого критерію адекватності на основі завдання моделі DNN.

При оцінці критеріїв адекватності на основі DNN-моделей можуть бути використані інші підходи для оцінки їх ефективності. Наприклад, можна використовувати метрики, які базуються на порівнянні результатів виконання оригінальної та мутованої моделей на тестових вибірках з реальними помилками. Також можна використовувати метрики, які враховують специфіку завдання моделі, наприклад, якість класифікації, визначення аномалій, передбачення часових рядів тощо. Крім того, можна проводити порівняльні аналізи різних критеріїв адекватності для визначення їх ефективності на конкретній моделі та в конкретному контексті.

В останній літературі як LSC [11], так і DeepMutation [15] використовуються для виявлення атак на DNN. Тому мій другий експеримент порівнює ці два критерії щодо їх здатності виявляти атаки на DNN.

У третьому експерименті я застосував три різні методи генерації тестів (включаючи новий метод) до моделей DNN і порівняв їх результативність, якщо згенеровані тестові дані використовуються для повторної навчання моделей.

У експериментах 1 та 2 використовуються зображення як вхідні дані. Але в третьому експерименті я використовував текстові дані. Однією з нових

застосувань DNN є застосування їх на вихідному коді в межах області текстових даних. Останні роки були відзначені величезним прогресом у глибокому навчанні для завдань, пов'язаних з вихідним кодом, таких як пошук коду та генерація коментарів. Методи для вивчення розподілених представлень створюють векторні представлення низького розміру для об'єктів, які називаються вкладеннями [16]. Вкладення коду визначається як векторне представлення фрагментів коду. Навчання вкладень коду дозволяє застосовувати нейромережеві техніки в широкому спектрі задач програмування. У цьому дослідженні я оцінюю мотивуючі завдання (пошук коду, підписання коду та передбачення назв методу), експериментуючи з трьома найбільш відомими та новітніми інструментами, такими як Code2Vec [13], Code2Seq [12] та CodeBert [14], в області інженерії програмного забезпечення.

Таблиця 1.1 - Ця таблиця підсумовує три різні техніки тестування, які використовувалися в цій дисертації, а також вхідні дані та DNN-модель.

Категорія тестування	DNN-модель	Вхідні дані	Тип вхідних даних
Генерація тестового оракулу	Варіаційний автоенкодер (VAE)	Набір даних MNIST	Зображення
Оцінка адекватності тесту	LeNet	Набір даних MNIST	Зображення
	Conv5	Набір даних MNIST	Зображення
	GoogleLeNet	Набір даних CIFAR10	Зображення
	Conv12	Набір даних CIFAR10	Зображення
Генерація тестових сценаріїв	Code2seq	Фрагменти коду на мові Java	Текст (фрагмент коду)
	Code2vec	Фрагменти коду на мові Java	Текст (фрагмент коду)
	CodeBERT	Фрагменти коду на мові Java	Текст (фрагмент коду)

Опираючись на обговорення, виявляється, що виявлення адверсальних прикладів є однією з основних тем у тестуванні DNN. Тому для тестування технік вбудовування коду потрібно визначити адверсальні приклади для коду. Хоча існують стратегії генерації текстових адверсальних прикладів, вони не є найкращим варіантом для джерела коду. Аліпур [9] та Томас [10] працювали над генерацією адверсальних прикладів для тестування DNN на основі джерела коду. Їхня основна ідея полягає у використанні рефакторингу джерела коду як способу зміни коду. Їхня мета полягала в тому, щоб показати, що вони можуть обдурити оригінальну DNN, використовуючи ці оператори рефакторингу. У третьому експерименті я запропонував новий метод генерації адверсальних прикладів для джерела коду та порівняв його з підходами Аліпура та Томаса щодо їх здатності обдурити моделі DNN. Крім того, я перетренував моделі, використовуючи адверсальні приклади, і показав, наскільки мої підходи є кращими у порівнянні з альтернативними.

Мій запропонований генератор адверсарних зразків базується на алгоритмі Guided Mutation (GM). Він ітеративно змінює код, використовуючи оператори рефакторингу, щоб максимізувати функцію пристосованості (яка базується на оцінці коефіцієнта мутацій DeepMutation++).

Ця дисертація ставить за мету відповісти на шість дослідницьких питань, експериментуючи з трьома різними аспектами тестування моделей глибокого навчання. Таблиця 1.1 узагальнює всі три експерименти щодо аспекту тестування, моделі DNN та специфікації набору даних. Основний внесок цієї дисертації можна узагальнити наступним чином:

• Експеримент 1: У першому експерименті я застосовую тестування багатьма реалізаціями на моделі глибокої нейронної мережі, називаємої варіаційним автоенкодером, щоб побачити, чи можна її використовувати як тестовий оракул і виявляти помилки в результаті. Мої дослідження мають на меті відповісти на наступне дослідницьке питання:

Питання 1: Як багаторазове тестування може адекватно наблизити тестовий оракул для моделей глибоких нейронних мереж?

Експеримент 2: У другому експерименті я емпірично оцінив дві різні метрики адекватності тестування для тестування глибоких нейронних мереж, що називаються покриттям сюрпризів (Surprise Coverage - SC) та мутаційним балом (Mutation Score - MS) для виявлення адверсарних прикладів. У цьому експерименті я прагну відповісти на наступні дослідницькі питання:

Питання 2: Яка з двох метрик (MS або SC) більш чутлива до виявлення адверсарних прикладів?

Питання 3: Як зміна параметрів метрик може вплинути на їх чутливість?

Експеримент 3: У останньому експерименті я тестую використання моделей DNN у вихідному коді, генеруючи адверсарні приклади. Я використовував два існуючі методи, а також запровадив новий метод для генерації адверсарних прикладів для введень у вихідному коді. Пізніше я перетренував модель, щоб побачити, чи відбувається якась покращення, враховуючи адверсарні приклади для моделі навчання. У цьому експерименті я досліджую наступні дослідницькі питання:

Питання 4: Як змінюється оцінка моделі при наявності адверсарних тестових вхідних даних?

Питання 5: Як впливає повторне навчання моделі на F1-оцінку?

Питання 6: Чи має вибір завдання оцінки якийсь вплив на покращення продуктивності моделі?

1. ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО ГЛИБОКІ НЕЙРОННІ МЕРЕЖІ

1.1. Машинне навчання

Новітня межа штучного інтелекту - Машинне навчання (МН): воно дає можливість машинам вчитися на основі минулих даних або досвіду, не будучи явно програмованими, з метою здійснення, за допомогою статистичного аналізу та відповідності шаблону, класифікації або передбачень щодо майбутнього. Але що ми маємо на увазі, говорячи про навчання? За словами Тома Мітчелла, професора комп'ютерних наук та машинного навчання у Карнегі-Меллон, комп'ютерна програма вважається навченою з досвідом E щодо певної задачі T і певного показника продуктивності P, якщо її продуктивність на T, вимірювану за допомогою P, покращується з досвідом E. Завдання МН зазвичай описуються у термінах того, як МН система має обробляти приклад, тобто набір функцій, таких як пікселі на зображенні, що є частиною набору даних. Деякі з найбільш поширених завдань, що можуть бути вирішені з використанням МН, включають: класифікацію, регресію, транскрипцію, машинний переклад, синтез та вибірковий аналіз. Щоб оцінити можливості алгоритму МН, ми повинні розробити кількісний показник його продуктивності: для завдань, таких як класифікація та транскрипція, ми часто вимірюємо точність моделі, тобто співвідношення прикладів, для яких модель дає правильний результат, або еквівалентно, рівень помилок [17], тобто співвідношення прикладів, для яких модель дає неправильний результат. Залежно від того, якого виду досвід їм дозволено отримати в процесі навчання, алгоритми ML можна умовно розділити на три класи:

- Навчання з вчителем: це найбільш популярний парадигма для виконання операцій машинного навчання. Це широко використовується для даних, де є точний відповідний зв'язок між вхідними та вихідними даними: класифікація зображень,

розпізнавання обличчя, виявлення спаму в електронній пошті та інше. Набір даних має мітки, що означає, що алгоритм визначає функції явно та робить передбачення або класифікацію відповідно. Термін "навчання з вчителем" [18] походить зі зрозумілого погляду на цю парадигму, де цільові значення надає інструктор або вчитель, який показує системі машинного навчання, що робити. Деякі з алгоритмів, які належать до навчання з вчителем, такі як: лінійна та логістична регресія, випадковий ліс, метод опорних векторів, штучні нейронні мережі;

- Некероване навчання: у цьому підході дані не мають явної позначки на різні класи, тобто в них немає позначок інструктора або вчителя, тому алгоритм повинен навчитися зрозуміти дані без цього керівництва. Модель може навчитися з даних, знаходячи неявні шаблони: щільності, структури, схожі сегменти та інші подібні ознаки. Деякі з важливих алгоритмів, що використовуються у некерованому навчанні, це: кластеризація, аналіз головних компонентів та виявлення аномалій;
- Навчання з підсиленням: воно охоплює більше галузей ШІ, що дозволяє машинам взаємодіяти з їх динамічним оточенням, щоб оцінити ідеальну поведінку в конкретному контексті та досягти своїх цілей. За допомогою зворотного зв'язку на основі нагород, агенти можуть навчитись поведінці та покращувати її у майбутньому, тож між системою навчання та її досвідом є зворотний зв'язок. На відміну від навчання з вчителем, агенту не надається ключова відповідь, коли він повинен виконати певне завдання, але він навчається зі свого власного досвіду.

Штучні Нейронні Мережі (ШНМ) - найпопулярніші алгоритми навчання з учителем в наш час, які складають дуже захоплюючий та потужний підвид Машинного Навчання, відомий як Глибинне Навчання (ГН). Винаходження ШНМ відбулося в 1970-х роках, коли Воррен Маккалок [19] та Волтер Пітс [20]

придумали цей термін, бувши натхненними метою моделювання біологічних нейрональних систем. Розвиток ГН частково мотивований невдачею традиційних алгоритмів, таких як лінійна регресія або базові дерева рішень, у загальному використанні на завданнях ШІ, таких як розпізнавання мови або розпізнавання об'єктів, особливо при роботі з високовимірними даними та складними функціями. Наслідуючи структуру людського мозку, ці алгоритми стали невід'ємним інструментом для широкого спектру застосувань, таких як класифікація зображень, розпізнавання мови та обробка природної мови, і вони досягли надзвичайної точності прогнозування, у багатьох випадках на рівні людської продуктивності [21]. ШНМ є ключовою технологією за спиною безпілотних автомобілів, дозволяючи їм розпізнавати знаки зупинки або відрізняти пішохода від ліхтарного стовпа, або вони є ключовим елементом у голосовому керуванні споживчих пристроїв, таких як телефони, планшети, телевізори та динаміки без рук. Моделі ГН можна використовувати для різноманітних складних завдань:

- Глибокі нейронні мережі (DNN) для задач класифікації;
- Згорткові нейронні мережі (CNN) для комп'ютерного зору;
- Рекурентні нейронні мережі (RNN) для аналізу часових рядів.

Штучні нейронні мережі (ШНМ) складаються з великої кількості сильно взаємопов'язаних обчислювальних елементів, нейронів, які працюють разом для вирішення конкретної проблеми. Функціонування ШНМ схоже на роботу нейронів у нашій нервовій системі. Основною обчислювальною одиницею мозку є нейрон: кожен нейрон отримує вхідні сигнали з дендритів і обробляє їх у клітинному тілі, де вони всі сумуються [22]. Якщо остаточна сума перевищує певний поріг, то нейрон може згенерувати викид, відправивши його по своїй єдиній аксоні та виробивши вихідні сигнали (рис. 1.1). Аксон врешті-решт розгалужується і підключається через синаптичні з'єднання [23] до дендритів інших нейронів. Синаптичні сили можна навчитися і вони контролюють силу впливу та його напрям на один нейрон на інший, як збудливий чи гальмівний.

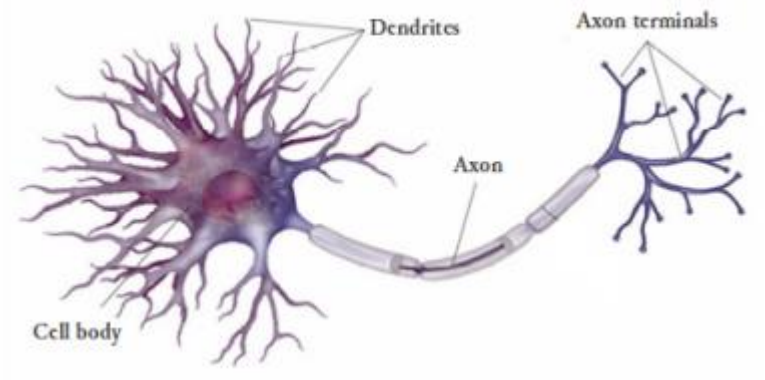

Рисунок 1.1 – Представлення нейрона

Для того, щоб зрозуміти, як працює ШНМ, корисно зрозуміти, як він побудований. ШНМ називають мережами, оскільки вони зазвичай представлені як композиція багатьох різних функцій. Модель пов'язана з орієнтованим ациклічним графом нейронів, що описує, як ці функції компонуються між собою. У мережі немає петель, тому виходи деяких нейронів можуть стати входами для інших нейронів [25]. З цієї причини такі мережі називають нейронною мережею прямого поширення: інформація проходить через функцію, що оцінюється від входу до виходу, тому немає зворотних зв'язків, у яких виходи моделі повертаються до неї самої. Коли нейронні мережі прямого поширення розширюються для включення зворотних зв'язків, їх називають рекурентними нейронними мережами. ШНМ організовані в кілька різних шарів нейронів [26], оскільки ця структура дуже проста та ефективна для оцінки функцій за допомогою операцій матриць та векторів. У ШНМ є три основні шари:

- Вхідний шар: це перший шар ШНМ, який отримує вхідну інформацію у формі текстів, чисел, аудіофайлів, пікселів зображення тощо. Нейрони у цьому шарі називаються вхідними нейронами;
- Прихований шар: посередині ШНМ розташовані приховані шари, які виконують різні типи математичних обчислень над вхідними даними та розпізнають входящі в них патерни. Може бути один прихований

шар, як у випадку з персептроном, або кілька - у випадку з глибокими нейронними мережами (ГНМ), кількість яких визначає глибину ШНМ;

- Вихідний шар: він містить вихідні нейрони та надає нам результат, отриманий за допомогою виконання складних обчислень середніми шарами.

1.2. Персептрон та глибокі нейронні мережі

Раніше версію ШШН представлено Персептроном. Це простий алгоритм, складений з одного вхідного та одного вихідного шару, а також не більше одного прихованого шару між ними, призначений для бінарної класифікації, тому він передбачає, чи належить вхід до певної категорії інтересу чи ні. Очевидно, персептрон не є повною моделлю прийняття рішень людини. З цієї причини останнє десятиліття спостерігається зростання більш складних і глибоких ШШН [27], таких як глибокі нейронні мережі (Deep Neural Networks, DNNs). DNNs отримують один вектор входу та перетворюють його через велику кількість прихованих шарів: отже, їх називають глибокими. Кожен нейрон кожного прихованого шару повністю з'єднаний з усіма нейронами попереднього шару, але працює повністю незалежно і не ділить жодних з'єднань з нейронами свого власного шару. Нарешті, вихідний шар є повністю з'єднаним шаром, який в налаштуваннях класифікації представляє класові бали (рис. 1.2).

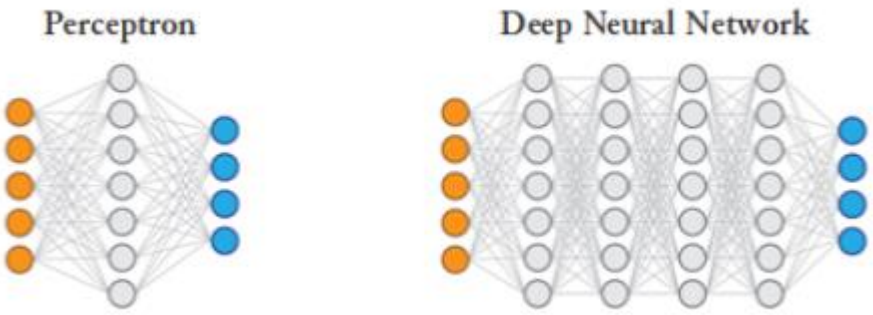

Рисунок 1.2 - Схематичне зображення персептрона та глибокої нейронної мережі.

Метою DNN є знаходження після фази навчання доброї апроксимації f відображення F між вхідними та вихідними даними: при заданому вхідному навчальному наборі X маркованих прикладів x, де кожен елемент x_i є ознакою, та векторі міток y, де y_i надає мітку для ознаки x_i, DNN [29] визначає відображення y = f(x; p) та навчає значення параметрів p, які найкраще відповідають вхідним даним, приводячи f(x) до відповідності з F(x) під час навчання. Кожен прихований шар мережі зазвичай є вектор-значним, і кожен елемент вектора можна розглядати як граючий роль аналогічну до нейрона, у тому сенсі, що він отримує вхід від багатьох інших одиниць та обчислює своє власне значення активації. Замість того, щоб думати про шар, як представлення однієї вектор-векторної функції, ми також можемо думати про шар як складається з багатьох одиниць, які діють паралельно, кожна з яких представляє вектор-скалярну функцію [30]. У кожному шарі кожен нейрон видає одне дійсне число, яке передається до кожного нейрона у наступному шарі, де кожен нейрон утворює власну зважену комбінацію цих значень, додає свій власний зсув та застосовує нелінійну функцію σ, яку називають функцією активації.

Система глибокого навчання (Deep Learning System) визначається як будь-яка програмна система, яка включає компонент Deep Neural Network (DNN). Ця програмна система може мати компоненти DNN, які взаємодіють з традиційними програмними компонентами. Розробники компонентів DNN

можуть опосередковано впливати на правила, які навчається DNN, змінюючи навчальні дані, особливості та архітектурні деталі моделі [3].

Штучний інтелект та машинне навчання, які є підмножиною ШІ, відіграють важливу роль у повсякденному житті в останні роки. Глибокі нейронні мережі (DNN), також є однією з цікавих тем, які були застосовані в різних галузях, таких як автомобілі з автопілотом та діагностика хвороб. DNN працює не тільки згідно з алгоритмом, але й ідентифікує та витягує відповідні високорівневі ознаки з сировинних вхідних даних без будь-якого людського керівництва [9].

Глибокі нейронні мережі складаються з багатьох параметричних функцій, які будують представлення вхідного сигналу. На практиці, DNN складається з послідовних шарів нейронів, які утворюють вихідний шар. Нейрон є окремим обчислювальним блоком всередині DNN, який застосовує функцію активації до своїх вхідних даних та передає результат іншим підключеним нейронам [31]. Вхідний, вихідний та прихований шари є три головні шари, які має кожна DNN.

Кожне з'єднання між нейронами у DNN прив'язане до параметра ваги. Навчання моделі полягає у вивченні ваг з'єднань шляхом мінімізації функції витрат. Кожен шар мережі перетворює інформацію у своєму вхідному сигналі на більш високорівневе представлення даних.

ВИСНОВКИ ДО РОЗДІЛУ 1

У цьому розділі наведена інформація про базову теорії по машинному навчанню та інформація по глибоким нейронним мережам.

2. МЕТОДОЛОГІЇ ТЕСТУВАННЯ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ

2.1. Тестування DNN категорій

У цьому розділі я коротко описую наступні три аспекти (із шести), класифікованих за [5] для тестування систем машинного навчання:

- Створення вхідних тестових даних
- Створення тестового оракулу
- Оцінка відповідності тесту

Створення вхідних тестових даних

У галузі тестування дослідники розробили багато методів та інструментів для генерації тестових вхідних даних [21]. На сьогоднішній день у літературі існує кілька таких методів, які відрізняються способом генерації вхідних даних, стратегією, яку вони використовують для дослідження поведінки тестової моделі, та конкретними евристиками, які вони використовують. Генерація тестових вхідних даних визначається як процес створення набору тестових даних для тестування адекватності програмного забезпечення або моделі.

Створення тестового оракулу

Генерація тестового оракулу описує пошук правила, яке дозволяє здійснити оцінку наявності помилки [32]. У дослідницькому літературному дослідженні було знайдено кілька потенційних категоризацій для тестових оракулів [33]. Метаморфні відношення та множинна реалізація є двома прикладами генерації тестового оракулу.

Оцінка відповідності тесту

Оцінка адекватності тестування спрямована на оцінку можливості існуючого тесту виявити помилки. Охоплення коду та мутаційне тестування - це дві найпопулярніші техніки оцінки адекватності тестування.

2.2. Покриття неочікуваності

Pei та співавтори [9] опублікували перший критерій покриття для моделей DNN, введенням Neuron Coverage (NC) як відношення активованих нейронів для заданих тестових вхідних даних до загальної кількості нейронів в DNN. Активований нейрон - це нейрон, сума його ваг та зсувів (bias) змушує його вогонь, що означає, що ця сума більша за певний поріг. Таким чином, значення активації визначаються порогом, який потрібен для активації кожного нейрона. Пізніше Kim та співавтори [11] запропонували нову концепцію для тестування систем машинного навчання. Вони пропонують Surprise Adequacy (SA), який обчислює, наскільки дивується модель на тестовий ввід порівняно з навчальним ввідом. SA має на меті визначити критерій відповідності, який вимірює поведінкові різниці, спостережувані в заданому наборі входів в порівнянні з навчальними даними.

SA пропонує дві міри для розрахунку несподіванки: міра, заснована на ймовірності (Likelihood-based Surprise Adequacy, LSA) [34] та міра, заснована на відстані (Distance-based Surprise Adequacy, DSA).

LSA розраховує ймовірність щільності функції з використанням оцінювання щільності ядра (KDE) [2]. Оскільки SA має на меті вимірювати сюрприз індивідуального введення x, LSA визначається як метрика, що збільшується зі зменшенням щільності ймовірності.

$$LSA(x) = -log(f(x))$$

де f (x) - щільність функції, визначена як:

$$\hat{f}(x) = \frac{1}{|A_{N_L}(T)|} \sum_{x_i \in T} K_H(\alpha_{N_L}(x) - \alpha_{N_L}(x_i))$$

N_L - це нейрон у вибраному шарі, то $A_{N_L}(x)$ видає набір слідів активації, H - це матриця ширини смуги, а K - гаусівська ядерна функція. $\alpha_{N_L}(x)$ позначає вектор значень активації.

LSA(x) обчислює 1) ймовірність того, що змінна знаходиться в конкретній області, та 2) наскільки ймовірно, що нейрон буде активований для навчальних і тестових вхідних даних, щоб порівняти їх різницю в активації.

DSA використовує Евклідову відстань [36] від тестового вводу до межі двох класів як міру несподіванки. Якщо відстань між першим введенням тесту та межею більше, ніж другий ввід тесту та межа, тоді перший ввід більш здивований, оскільки він не зміг чітко визначити, до якого класу він належить. Отже, DSA визначається таким чином:

$$DSA(x) = \frac{dist_a}{dist_b}$$

де $dist_a$ та $dist_b$ визначаються як:

$$dist_a = //\alpha_N(x) - \alpha_N(x_i)//$$

$$dist_b = //\alpha_N(x_a) - \alpha_N(x_b)//$$

та,

$$x_a = argmin//\alpha_N(x) - \alpha_N(x_i)//$$

$$x_b = argmin//\alpha_N(x_a) - \alpha_N(x_i)//$$

Kim et al [11] провели оцінку свого методу, щоб дослідити, чи можливо виявляти атаки на прикладах на основі значень SA, навчавши класифікатор атак на основі логістичної регресії на значеннях SA. Вони застосовували лише DSA для задач класифікації, для яких це може бути ефективніше, ніж LSA [37]. Нарешті, вони показали, що значення LSA правильно відображають поведінку систем DNN, оскільки класифікатор, побудований на основі DSA, успішно

виявляє атаки на прикладах. Вони також показали, що після повторного навчання моделі з використанням атак на прикладах точність збільшується.

Пізніше вони розрахували різноманітність вхідних даних і назвали її Surprise Coverage (SC). Оскільки LSA та DSA визначені в неперервних просторах, вони використовують розбиття на корзини, щоб дискретизувати простір здивування та визначити Likelihood-based Surprise Coverage (LSC). LSC для набору вхідних даних X визначається наступним чином:

$$LSC(X) = \frac{|\{b_i | \exists x \in X : SA(x) \in (U.\frac{i-1}{n}, U.\frac{i}{n}]\}|}{n}$$

Таблиця 2.1 - Оператори мутації на рівні вихідного коду (Глобальні)

Тип несправності	Цільовий об'єкт	Опис
Повторення даних	Дані	Подвоїти навчальні дані
Помилка мітки	Дані	Фальшиві результати даних
Відсутність даних	Дані	Видалити вибрані дані
Перемішування даних	Дані	Перемішати вибрані дані
Збурення даних	Дані	Додати шум до навчальних даних
Видалення шару	Модель DNN	Видалити шар
Додавання шару	Модель DNN	Додати шар
Видалення функції активації	Модель DNN	Видалити функцію активації

Для заданої верхньої межі U та сегментів B = b_1, b_2, ..., b_n, які ділять проміжок (0,U] на n сегментів SA.

У другому експерименті використовується метрика покриття несподіванності на основі ймовірності (LSC - Likelihood-based Surprise Coverage). LSC використовується замість DSA, оскільки, як зазначено у базовій

статті, DSA використовується тільки для класифікаторів і не використовується для розрахунку покриття.

2.3. DeepMutation Score

Ma et al. [13] запропонували техніку тестування під назвою DeepMutation, яка використовує техніку мутаційного тестування для DNN систем з метою вимірювання якості тестових даних. Спочатку вони використовували техніку мутаційного тестування на рівні початкового коду, яка мутирує навчальні дані та цільову DNN модель. Таблиця 2.1 показує оператори мутаційного тестування на рівні початкового коду для DL систем.

Таблиця 2.2 - Оператори мутації на рівні моделі.

Оператор мутації	Рівень	Опис
Гаусівський розподіл помилок	Вага	Застосовує помилки гаусівського розподілу до ваг моделі
Перестановка ваг	Нейрон	Переставляє ваги нейрона
Блок ефекту нейрона	Нейрон	Блокує ефект нейрона на наступні шари
Інверсія активації нейрона	Нейрон	Інвертує статус активації нейрона
Перемикач нейрона	Нейрон	Перемикає два нейрони в одному шарі
Деактивація шару	Шар	Деактивує ефекти шару
Додавання шару	Шар	Додає шар в нейронну мережу
Видалення функції активації	Шар	Видаляє функцію активації

Друге, вони створили набір операторів мутації моделі, щоб внести дефекти та створити мутанти без навчання моделі. Таблиця 2.2 демонструє операторів мутації моделі. Нарешті, після створення мутантів моделей та даних, кожен

тестовий вхід оцінюється на наборі мутантних моделей.

Припустимо, що маємо задачу класифікації з k класами вхідних даних і нехай $C = \{c_1, ..., c_k\}$ є усі к класів вхідних даних [40]. Для кожної тестової точки $t^j \in T^j$, t^j вбиває $c_i \in C$ мутанта $m^j \in M^j$, якщо виконуються наступні умови:

(1) t^i правильно класифікується як c_i за допомогою початкової моделі глибокого навчання M; (2) t^i не класифікується як c_i m^i. Оцінка мутацій для систем глибокого навчання визначається наступним чином, де KilledClasses(T^i, m^i) - це множина класів mi, вбитих тестовими даними в T^i[13]:

$$MutationScore(T', M') = \frac{\Sigma_{m' \in M'} |KilledClasses(T', m')|}{|M'| \times |C'|}$$

Wang et al. [15] пропонують підхід для виявлення атак на моделі шляхом мутацій. Їхній підхід є інтеграцією тестування DeepMutation [13] та статистичного тестування гіпотез [5]. Вони формулюють проблему як ефективність прийняття моделлю рішення про те, чи є f(x) нормальним зразком чи атакуючим зразком, при заданому вхідному зразку x до моделі глибокого навчання f.

Їхній підхід базується на "чутливості", яку вимірюють за допомогою показника зміни мітки (LCR) [38]. Змутовані моделі DNN на основі заданої моделі DNN, ймовірніше визначать атакований зразок інакше, ніж мітка, створена оригінальною моделлю DNN. Заданий вхідний зразок x (регулярний або атакований) та модель DNN f, спочатку мутують за допомогою набору операторів мутації моделі, що використовуються в DeepMutation, для отримання набору змутованих моделей.

Після отримання такого набору змінених моделей F, вони отримують мітку $f_i(x)$ вхідного зразка x на кожній зміненій моделі $f_i \in F$. LCR визначається для зразка x наступним чином:

$$LCR(x) = \frac{|\{f_i | f_i \in F, f_i(x) \neq f(x)\}|}{|F|}$$

Інтуїтивно LCR(x) вимірює чутливість вхідного зразка x на мутації моделі глибокої нейронної мережі. Вони помітили, що значення LCR для адверсних прикладів значно вищі, ніж для звичайних прикладів [39] на основі результатів. Практична імплікація спостереження полягає в тому, що, маючи вхідний зразок x, LCR може виявити, чи x ймовірно є нормальним чи адверсним.

Оскільки Wang et al [41]. у своїй статті [15] інтегрували моделі і оператори мутацій, а їхня метрика досить схожа на DeepMutation. Однак у нашому експерименті я використовував показник зміни мітки (LCR) для перевірки адекватності критеріїв мутації, зосереджуючись на атаках зловмисників.

2.4. Multiple Implementation тестування

Multiple-implementation testing (MIT) є технікою, яка використовується для вирішення проблем "no oracle" [4]. Ця техніка базується на уявленні, що для функціональності може бути доступно декілька реалізацій, які можна використовувати. Деякі з цих реалізацій можуть містити різні помилки, що призводять до непередбачуваних поведінок для окремих вхідних даних. Проте, якщо більшість виконаних реалізацій дають однаковий результат для певного вводу, то цей результат вважається правильним [42]. Така більшість може бути визначена за допомогою попередньо заданого порогового значення відсотка, позначеного як α, для заданого вводу. Коли відсоток реалізацій, що дають однаковий результат, більший за α, то результат вважається більшістю і використовується як очікуваний вихід.

2.5. Пошуково-орієнтоване тестування

Пошуково-орієнтоване програмне інженерія (SBSE) застосовує метаевристичні методи пошуку, такі як генетичні алгоритми (GA) та Hill Climbing (HC), до програмної інженерії [6]. Це передбачає визначення простору пошуку або набору можливих рішень. Зазвичай пошуковий простір є занадто великим, тому пропонується метаевристичний підхід для дослідження можливих рішень. Потім функція придатності використовується як метрика для вимірювання якості потенційних рішень [8].

Search-Based Software Testing (SBST) - це метод застосування оптимізуючих методів пошуку (наприклад, генетичних алгоритмів) для вирішення проблем тестування програмного забезпечення. SBST використовується для пріоритизації тестових випадків, генерації тестових даних, оптимізації оракулів програмного тестування та верифікації програмних моделей.

Багато задач у програмному інженерії можуть бути сформульовані як задачі оптимізації. У генетичному алгоритмі заданий набір можливих рішень, який еволюціонує до кращих рішень у задачі оптимізації. Еволюція починається зі згенерованого випадково популяції, яку називають індивідуальними рішеннями. Найкращі з них передаються з поточного покоління до наступного, розраховуючи при цьому фітнес кожного індивіда. Решту популяції можна мутувати, щоб створити нові індивідуальні рішення [9]. Алгоритм завершується, коли досягнуто максимальну кількість поколінь.

2.6. Зразки для порівняння

Глибокі нейронні мережі (DNN) проявляють вразливу поведінку до стратегічно модифікованих зразків, які називають адверсарними прикладами.

Ці зразки є спеціалізованими вхідними даними, створеними для зміни результату роботи нейронної мережі шляхом неправильної класифікації вхідних даних. Вони генеруються з невидимими відхиленнями, які можуть змінювати результат роботи DNN та призводити до неправильних передбачень.

Існує кілька способів генерації атакуючих прикладів як для текстових, так і для зображеннєвих вхідних даних. Тому виявлення таких атак залишається важливим завданням для забезпечення стійкості, безпеки та надійності глибоких нейронних мереж.

Метод швидкого знаку градієнту (Fast Gradient Sign Method, FGSM) [10], атака на основі салієнтної карти Якобіана (Jacobian-based Saliency Map Attack, JSMA) [43] та C&W [44] - це деякі відомі методи для генерації адверсарних прикладів як для текстових, так і для зображень [13].

2.6.1. Генератор моделі

Нижче я надам короткий опис ідеї за кожною моделлю генератора атак.

FGSM

Метод швидкого градієнтного знаку (FGSM) [45] розроблений шляхом додавання градієнту функції втрат відносно вхідних даних. Зверніть увагу, що FGSM не гарантує, що адверсна пертурбація є мінімальною.

JSMA

Jacobian-based Saliency Map Attack (JSMA) - це жадібний алгоритм, що змінює один піксель на кожній ітерації. Ця зміна пікселів збільшує ймовірність того, що модель надасть певну мітку цільової класифікації для атакування з мінімальним збуренням. Він спрямовує цільову модель на класифікацію його як певної мітки за допомогою адверсних зразків.

C&W

Карліні та співавтори запропонували групу атак на основі трьох метрик відстані. Основна ідея полягає у вирішенні задачі оптимізації, яка мінімізує спотворення (пертурбацію), одночасно максимізуючи ймовірність отримати цільовий клас.

Deepfool

Ідея DeepFool полягає в тому, щоб зробити оригінальні зразки перетнути границю прийняття рішень з мінімальними змінами.

Blackbox

Ідея полягає в тому, щоб навчити модель-заміну наслідувати поведінку цільової моделі з допомогою аугментації даних. Після цього застосовується один з існуючих алгоритмів атак, наприклад, FGSM або JSMA, для генерації адверсарних прикладів для моделі-заміни.

Всі ці методи широко використовуються для моделей з текстовим та зображеннєвим введенням [13]. У цій роботі я також використовував згадані методи для генерації атакуючих прикладів для зображень. Однак для третього експерименту, оскільки всі ці методи змінюють структуру введення, використання їх для генерації атакуючих фрагментів коду є непрактичним. Невелика безнаглядна зміна може призвести до некорисного фрагменту коду без конкретного значення [48]. Тому я визначаю рефакторинг як методологію генерації атакуючих прикладів коду.

2.6.2. Рефакторинг

У комп’ютерній інженерії, рефакторинг коду є способом змінити кодовий фрагмент, не змінюючи його функціональності. Це використовується для покращення існуючого коду, зробивши його більш зрозумілим, легкочитаємим та чистим. Рефакторинг також допомагає зробити додавання нових

функціональностей зручнішим, створення великих додатків легшим та виявлення помилок швидшим.

У цьому дослідженні для генерації адверсарних прикладів для вихідного коду я використовував оператори рефакторингу. Це гарний вибір, оскільки рефакторингований фрагмент коду семантично збігається та повинен виробляти той самий вихідний результат для більшості DNNs. Однак синтаксичні зміни часто легко призводять до різних результатів DNN. Нижче наведений список з десятьма операторами рефакторингу для генерації семантично еквівалентних фрагментів коду, написаних на мові Java.

- Перейменування локальної змінної: Перейменовує назву змінної за допомогою синоніму.
- Перейменування аргументу: Перейменовує назву аргументу за допомогою синоніму.
- Перейменування назви методу: Перейменовує назву методу за допомогою синоніму.
- Перейменування API: Перейменовує назву API за допомогою синоніму для локальної змінної.
- Додавання локальної змінної: Додає локальну змінну в код.
- Додавання аргументу: Додає аргумент в код.
- Додавання виводу: Додає вивід на екран на випадковому рядку коду.
- Покращення циклу for: Замінює цикл for на цикл while або навпаки.
- Покращення умови IF: Замінює умову IF на еквівалентну логіку.
- Оптимізація return: Змінює повернення змінної, де це можливо.

Незважаючи на те, що функціональність початкового коду не була змінена за допомогою згаданих технік рефакторингу [47], якщо результати DNN змінюються, рефакторингований код тепер вважається новим вхідним даним, що називається атакуючим вхідним даним у цій дисертації.

ВИСНОВКИ ДО РОЗДІЛУ 2

У цьому розділі представлені різні аспекти тестування DNN-моделей. Далі дається визначення багаторазового тестування як метрики тестування в цілому, а потім представлено Surprise Coverage та Mutation Score як дві метрики для оцінювання якості тестування для моделей DNN. Надано визначення змагальних прикладів та різних методів їх створення та їх генерування.

3. ОПИС ТА АНАЛІЗ МЕТОДОЛОГІЙ ВИКОРИСТАНИХ ДЛЯ ТЕСТУВАННЯ DNN

3.1. Тестування DNN: Генерація тестових оракулів

3.1.1. Огляд проблематики

У цьому експерименті я досліджую, чи можна вважати тестування з багатьох реалізацій як один з методів тестування DNN-моделей.

Мета

У цьому експерименті я запускаю десять різних реалізацій алгоритмів автоенкодера з варіаційним засобом (VAE) з однаковими тестовими вхідними даними та конфігураціями. Враховуючи всі вихідні дані моделі, я визначаю більшість голосів відповідей цих реалізацій як тестового оракула. Нарешті, я оцінюю, чи може тестовий оракул бути корисним у виявленні несправних реалізацій. Несправності у цих реалізаціях визначаються будь-якою причиною, що спричинює генерацію моделлю вихідних даних низької якості.

Зокрема, цей підхід включає три основні кроки: перш за все, я збираю багато реалізацій алгоритму VAE з Github. Друге, я запускаю тестові вхідні дані на всіх реалізаціях, щоб визначити тестовий оракул на основі більшості голосів. Нарешті, я оцінюю, чи є тестовий оракул хорошим засобом для виявлення неправильних реалізацій.

Досліджувальні питання

Питання 1: Як можна використовувати багатоімплементаційне тестування для адекватного наближення тестового оракула для моделей глибокого навчання?

3.1.2. Постановка задачі

В цьому розділі я обговорюю модель DNN, конфігурації та набори даних, використані в експерименті.

Дані

Я використовував набір даних MNIST [12], безкоштовний набір даних з понад 30 000 зображень, який включає рукописні англійські цифри з існуючих онлайн-репозиторіїв.

Для тестового вводу я створив два зображення, що містять колекції різних написаних цифр від 0 до 9. Ці тестові вводи допомагають нам побачити, наскільки добре різні реалізації можуть генерувати зображення. Я використовую ці тестові вводи для вимірювання якості тестових вводів моделі. Рисунки 3.3 та 3.4 - це два зразки тестових входів, використаних для відповіді на питання 1 у цьому експерименті.

DNN Модель, що вивчалася

У цьому експерименті я використовував алгоритм варіаційного автоенкодера (VAE) як представника DNN моделі для тестування [50]. Я зібрав кілька реалізацій алгоритму VAE, шукаючи ключові слова VAE та Variational Auto Encoder в репозиторії GitHub. Усі реалізації були написані мовою програмування Python.

Перед тим як перейти до визначення VAE, розглянемо визначення автоенкодера. З наведених джерел, автоенкодери є методом навчання без учителя, який використовує нейронні мережі для завдання навчання представлення даних. Архітектура нейронної мережі спроектована таким чином, щоб в мережі був вузький місток для стиснення вихідного представлення знань. Якщо властивості вхідних даних незалежні один від одного, це стискання та подальша реконструкція буде складним завданням. Однак, якщо дані мають деякі структури, наприклад, кореляції між вхідними

ознаками, ця структура може бути вивчена та використана при примусовому проходженні даних через місток.

Як зображено на Рисунку 3.1, немаркований набір даних можна розглядати як задачу навчання з вчителем, що має на меті виведення xˆ, відтворення оригінального вводу x. Bottleneck - це важлива характеристика мережевого дизайну. Без інформаційного bottleneck [46] мережа може швидко вивчити запам'ятовувати значення вводу, передаючи їх через мережу. Bottleneck обмежує кількість інформації, яка може пройти повну мережу, змушуючи навчитися стисненню вхідних даних.

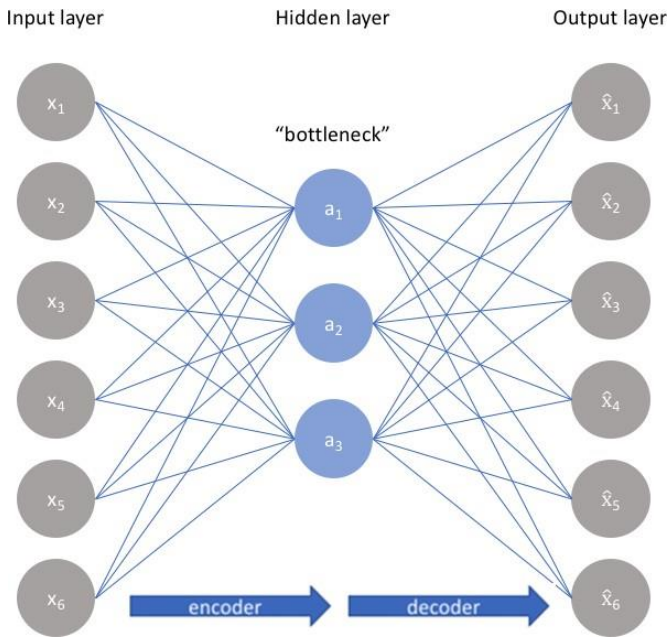

Рисунок 3.1 - Загальна структура моделі автокодера

Найпростіша архітектура для побудови автоенкодера полягає в обмеженні кількості вузлів у прихованому(их) шарі мережі, обмежуючи кількість інформації, що може пройти через нього. Шляхом покарання мережі за реконструкційну похибку, модель може навчитись основних атрибутів вхідних даних та найкращого способу відтворення початкового введення зі "стану кодування". Ідеально, це кодування навчається і описує латентні атрибути вхідних даних.

У автоенкодера в кодері є кілька повністю зв’язаних шарів, які беруть вхід і стискають його до меншого представлення з меншою кількістю вимірів, ніж вхід. Це представлення називається затором або прихованим шаром. Потім з затору він намагається відновити вхід за допомогою генеративно-адверсарної мережі. Використання варіаційного автоенкодера бере протилежний підхід. Розподіл, який слідується латентними векторами, не є питанням, і цільовий розподіл - те, до чого модель намагається дійти.

За лише три роки Варіаційні Автоенкодери (VAE) [51] стали одним з найпопулярніших підходів до навчання без допомоги учителя складних розподілів. VAE мають привабливість тому, що вони ґрунтуються на стандартних апроксимаціях функцій (нейронних мережах) і можуть навчатися за допомогою стохастичного градієнтного спуску [6].

У відміну від автоенкодерів, які перетворюють вхідні дані на фіксовані вектори, Варіаційні автоенкодери (VAE) перетворюють їх на розподіл. Цю ідею можна пояснити так: у стандартному автоенкодері зображення перетворюється в вектор, а в VAE вектори, що отримуються з зображення, перетворюються в розподіл. Це зображено на Рисунку 3.2, де замість звичайного bottleneck у VAE використовують два окремі вектори, один з яких представляє середнє значення, а інший - стандартне відхилення розподілу. Таким чином, коли потрібен вектор для подачі в мережу декодеру, з розподілу береться випадковий зразок і підсилюється через декодер.

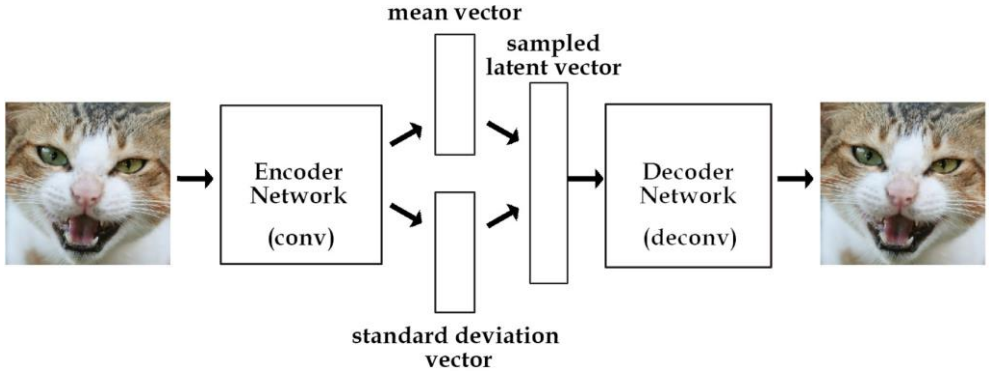

Рисунок 3.2 - Модель варіаційного автокодера з точки зору його функціональності

Конфігурація моделі

У всіх реалізаціях алгоритму VAE я встановив наступні параметри:

• Кількість епох = 50

• Кількість шарів латентних змінних = 20

• Розмір пакету = 128

• Кількість нейронів: 100

• Розмір вхідного зображення: 488 x 488 пікселей.

Де кількість епох вказує кількість ітерацій для кожного алгоритму, Кількість шарів латентних змінних вказує шари звуження, що представляють вектор, Розмір пакету - кількість прикладів в одній партії, для яких виконуються передбачення одночасно. Кількість нейронів вказує кількість вузлів у кожному шарі, крім звуження, а Розмір вхідного зображення вказує розмір кожного зображення для навчання. Я використовував всі конфігурації, базовані на найчастіших числах для кожної функції в усіх реалізаціях.

Оцінка вимірювання

Для оцінки згенерованих зображень я використовував метрику, яка називається Fre´chet Inception Distance [16]. Fre´chet Inception Distance (FID) - це одна з метрик для автоматичної оцінки якості генеративних моделей зображень [7] [8] [6]. FID - це вимірювання, яке порівнює два зображення піксель за пікселем. Розмір зображення та його розмитість можуть впливати на оцінку, яка є числом з плаваючою точкою та показує схожість зображень. Зазначається, що порівняння одного зображення з самим собою буде мати оцінку FID 0,00. У цьому експерименті я використовую FID для порівняння вхідних зображень зі згенерованими зображеннями всіма реалізаціями. Нижче наведено повне визначення оцінки FID.

Inception Score (IS) [16] використовує якість згенерованих зображень та їх різноманітність як два критерії. Я хочу, щоб умовна ймовірність P(y/x) була дуже передбачуваною в згенерованих зображеннях. Тому я використовую мережу Inception для класифікації згенерованих зображень та передбачення P(y/x) - мітки x. Це відображає якість зображень.

Якщо згенеровані зображення є різноманітними, розподіл даних для y повинен бути рівномірним. Щоб поєднати ці два критерії, я обчислюю їх KL-дивергенцію та використовую рівняння нижче для обчислення IS.

$$IS(G) = exp(E_{x \sim p_a} D_{KL}(p(y|x)||p(y)))$$

FID використовує мережу Inception для вилучення ознак з проміжного шару. Потім, для цих ознак я моделюю розподіл даних за допомогою багатовимірного гаусівського розподілу з математичним очікуванням μ та коваріаційною матрицею Σ. FID між реальними зображеннями x та згенерованими зображеннями g обчислюється за допомогою формули, яку я наводжу нижче.

$$FID(x, g) = ||\mu_x = \mu_g||_2^2 + Tr(\Sigma_x + \Sigma_g - 2(\Sigma_x \Sigma_g)^{\frac{1}{2}})$$

Тут Tr позначає суму всіх діагональних елементів матриці. Менші значення FID означають кращу якість та різноманітність зображень. Відстань збільшується при симуляції пропущених мод. FID більш стійкий до шумів, ніж IS. Якщо модель генерує лише одне зображення для кожного класу, то відстань буде високою. Тому FID є кращим показником для різноманітності зображень. У FID є досить висока систематична похибка, але низька дисперсія. Якщо обчислити FID між набором даних для навчання та набором даних для тестування, то я маю очікувати, що FID буде рівним нулю, оскільки обидва набори містять реальні зображення. Однак при тестуванні з різними партіями вибірки для навчання FID показує ненульове значення.

3.1.3. Проблематика експерименту

Для оцінки якості згенерованих зображень я протестував всі 10 реалізацій з двома тестовими вхідними зображеннями. Кожна реалізація зайняла близько

п'яти годин, щоб згенерувати результати у відношенні часу виконання. Для оцінки якості згенерованих зображень я обчислював відстань між згенерованим зображенням та оригінальним зображенням з порогом FID 120. Цей емпіричний поріг оцінювався вручну, оскільки не існує іншої попередньої роботи з тестуванням декількох реалізацій на VAE.

Рисунок 3.3 - Перший зразок вхідного зображення моделі

Рисунок 3.4 - Другий зразок вхідного зображення моделі.

Щоб перевірити, чи залежить оцінка FID згенерованих зображень від конфігурації або моделі DNN, я запустив усі моделі з їх наборами конфігурацій та вхідними даними. Нарешті, маючи тестового оракула, я порівняв різні реалізації вручну, щоб побачити, чи вказує різниця між оцінками FID оригінального зображення та згенерованого зображення на наявність яких-небудь недоліків.

3.1.4. Обговорення проблеми та результати

Далі я розгляну відповіді на дослідницьке запитання експерименту, обговоривши результати. Таблиця 3.1 показує результати кожної реалізації для алгоритму VAE, де I_i - це i^{th} реалізація. На рисунку 3.5 показані згенеровані зображення всіма реалізаціями для введення на рисунку 3.3.

Таблиця 4.2 також показує результати всіх реалізацій на основі другого входу, показаного на Рис. 4.4.

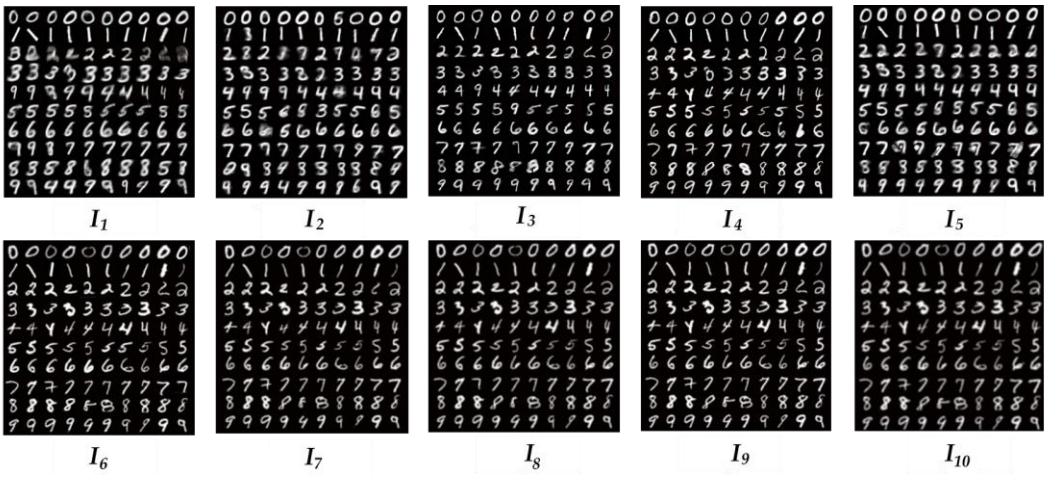

Рисунок 3.5 - Згенеровані зображення всіма 10 реалізаціями від I_1 до I_{10}

Таблиця 3.3 показує значення FID для кожної реалізації з її параметрами за замовчуванням. Незважаючи на те, що інші дві таблиці використовують параметри, зазначені в Розділі 3.1.2, у цій таблиці параметри встановлені так само, як і за замовчуванням для кожної реалізації при клонуванні. Крім того, вхідне зображення також є вхідним зображенням за замовчуванням для реалізації. Чим більше число епох, тим більше часу потрібно для компіляції кожної реалізації.

Таблиця 3.1 - Розрахована оцінка FID для кожної реалізації для першої картинки

Ідентифікатор реалізації	FID Оцінка
I_1	158.89
I_2	229.28
I_3	117.49
I_4	67.5
I_5	205.37
I_6	62.43
I_7	107.74
I_8	95.05
I_9	54.38
I_{10}	135.01

Кінцевим кроком було поєднання FID оцінок наших реалізацій у Таблиці 3.4, відсортувавши їх у порядку зростання FID, який можна знайти в Таблицях 3.1, 3.2 і 3.3, що відповідають першому введенню, другому введенню та різним параметрам та введенням відповідно. Вони були відсортовані, починаючи з найменшого FID.

Нарешті, був встановлений поріг α для знаходження більшості згодних реалізацій. Оскільки не було попередніх робіт з встановленням α, я вибрав α = 120 на основі моїх спостережень та порівняння якості згенерованих зображень. Жирним шрифтом в Таблиці 3.4 показані реалізації, які перевищують поріг.

Питання 1: Як багатореалізаційне тестування може адекватно апроксимувати тестовий оракул для моделей DNN?

Порівняння відсортованих імплементацій в Таблиці 3.4 показує, що у відсортованих імплементаціях з обома першим і другим вхідними даними, I_{10},

I_1, I_5 та I_2 є алгоритмами, які більші за $\alpha = 120$, що означає, що вони мають найбільшу відстань від оригінального входу. Натомість, для останньої колонки, яка є вхідними даними за замовчуванням зі стандартними параметрами кожної реалізації, I_1 та I_2 та I_{10} були тими, які більші за порігове значення.

Результати показують, що при збереженні схожих параметрів не відбувається зміна відсортованих FID-оцінок. Це означає, що, хоча кількість балів FID може відрізнятися залежно від різних вхідних зображень для кожної реалізації, якість згенерованих зображень не змінюється, якщо всі параметри встановлені подібно. З іншого боку, останній стовпець показує, що можуть бути різні результати зі зміною параметрів та вхідного зображення. Тому, щоб відповісти на дослідницьке питання 1, мені потрібно проаналізувати реалізації I_5, I_1, I_2 та I_{10}, щоб переконатися, що не має недоліків. Недоліки у цих реалізаціях визначаються будь-якою причиною, яка зумовлює низьку якість згенерованих вихідних даних.

Таблиця 3.2 - Розрахована оцінка FID для кожної реалізації для першої картинки

Ідентифікатор реалізації	FID Оцінка
I_1	166.23
I_2	257.67
I_3	125.11
I_4	84.55
I_5	193.49
I_6	76.21
I_7	115.89
I_8	102.9
I_9	68.21

I_{10}	143.70

Таблиця 3.3 - Результати реалізацій з параметрами та вхідними даними за замовчуванням

Ідентифікатор реалізації	FID Оцінка	Кількість епох	Кількість латентних шарів	Розмір партії	Кількість нейронів
I_1	200.09	10	20	100	100
I_2	233.42	20	20	128	128
I_3	115.77	75	3	100	100
I_4	79.82	10	20	100	100
I_5	35.37	20	50	128	128
I_6	89.07	50	32	64	64
I_7	40.27	60,000	20	32	32
I_8	84.62	20	20	128	128
I_9	24.1	1000	32	64	64
I_{10}	172.21	301	20	100	100

Це свідчить про те, що зміна конкретного параметру може стати причиною різниці в якості згенерованих зображень. Як показано в таблиці 3.3, за замовчуванням кількість прихованих шарів для цієї реалізації становить 50, тоді як кількість прихованих шарів, встановлених для всіх моделей, дорівнює 20.

Зараз, щоб наблизитися до відповіді на питання 1, мені потрібно порівняти реалізації I_1, I_2 та I_{10}, щоб з'ясувати причини, чому вони не генерують зображення так же як інші реалізації. Після аналізу I_1, я зрозумів, що порівняно з іншими реалізаціями, те, що відрізняє його результати, це шум, який спочатку

додався до вхідного зображення. Після додавання шуму алгоритм намагається згенерувати зображення та намагається ігнорувати ці шуми. Деякі автоенкодери додають шум до зображення, щоб навчити нейронну мережу ігнорувати їх. Тому, даючи шумове зображення і просивши алгоритм згенерувати реальне зображення, він намагається навчитися ігнорувати шуми.

Рисунок 3.6 показує вхідне зображення, шумове вхідне зображення та згенероване зображення на першій та десятій епохах для реалізації I_1. Розглядаючи згенероване зображення за допомогою I_1 на рисунку 3.6, я можу побачити, що додавання шуму може призвести до генерації гірших зображень. Реалізація I_2 також використовує такий же підхід для генерації зображень. Рисунок 3.7 показує процес генерації зображень з шумом для I_2. Варто зазначити, що можуть існувати різні типи додавання шуму до вхідного зображення, які не входять до області дослідження цього дослідження, але вони можуть впливати на якість згенерованих зображень.

Таблиця 3.4 - Сортування реалізацією FID спочатку враховує найнижчий FID, виходячи з наших трьох таблиць: Таблиця 3.1) Вхідне зображення 1 - встановлення параметрів; Таблиця 3.2) Вхідне зображення 2 - встановлення параметрів; Таблиця 3.3) Вхідне зображення за замовчуванням - параметри за замовчуванням

Табл. 3.1	Табл 3.2	Табл 3.3
I_9	I_9	I_9
I_6	I_6	I_5
I_4	I_4	I_7
I_8	I_8	I_4
I_7	I_7	I_8
I_3	I_3	I_6
I_{10}	I_{10}	I_3
I_1	I_1	I_{10}
I_5	I_5	I_1
I_2	I_2	I_2

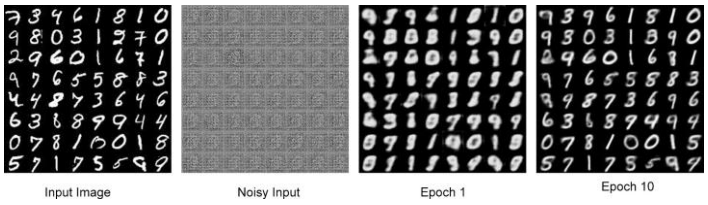

Рисунок 3.6 - Процес додавання шуму для реалізації I_1

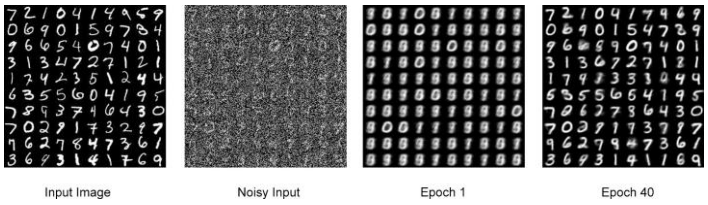

Рисунок 3.7 - Процес додавання шуму для реалізації I_2

Звіт показує, що під час зберігання параметрів в межах приблизно однакових значень немає зміни у відсортованих результатах FID. Це означає, що якщо при кожній реалізації параметри мають приблизно однакові значення, якість згенерованих зображень залишається незмінною, незалежно від вхідного зображення. З іншого боку, останній стовпець показує, що я можу отримати різні результати змінюючи параметри та вхідне зображення. Отже, щоб відповісти на дослідницьке запитання 1, мені потрібно проаналізувати реалізації I_5, I_1, I_2 та I_{10}, щоб дізнатися, чи знайду я якісь проблеми. Проблеми у цих реалізаціях визначаються будь-якою причиною, яка спричиняє генерацію низькоякісних результатів.

У таблиці 3.4 можна зрозуміти, що в перших двох стовпцях, де параметри встановлені подібно, не відбувається зміни порядку FID-оцінок реалізацій. Тому вхідні дані не впливають на результати. Однак порівняння цих двох стовпців з третім показує, що результати можуть відрізнятися в залежності від параметрів.

Підсумок по питанню питанню 1:

Щодо першого дослідження, я можу зробити висновок, що використання MIT може допомогти виявити будь-які помилки, що спричиняють різні виходи на різних імплементаціях для моделей DNN. Отже, відповідь на перше

дослідження полягає в тому, що MIT може допомогти виявити будь-які помилки або відмінності в імплементації моделей DNN. У цьому експерименті я також згадав дві причини, чому імплементація VAE може не давати нам належні результати.

3.2. Тестування DNN: Оцінка адекватності тесту

3.2.1. Огляд проблеми

У цьому експерименті критерії адекватності тестування для DNN-моделей оцінюються за допомогою порівняння технік тестування на основі покриття та мутацій.

Мета:

Крім того, останнім часом було запропоновано кілька метрик адекватності тестування для DNN-програм, проте не проводилось прямого порівняльного дослідження між двома основними групами метрик (тобто критеріїв покриття та мутаційної здатності), щоб виявити адверсарні приклади. У цьому експерименті я запропонував підхід для порівняння метрик Surprise Coverage та Model Mutation здатності для визначення того, який з них може мати більший приріст за допомогою додавання адверсарних прикладів до вихідного набору тестів. У цьому експерименті чутливість визначена як приріст метрики при додаванні частини адверсарних прикладів. Тому, більший приріст в балів метрик показує вищу чутливість до адверсарних атак.

Головна причина вибору цих двох категорій критеріїв полягає в тому, що вони є найбільш вивченими метриками як у традиційному, так і у домені тестування DNN.

Досліджувальні питання:

Питання 2: Який з двох метрик (Surprise Coverage або Label Change Rate) більш чутливий у виявленні адверсальних прикладів?

Питання 3: Як зміна параметрів метрик може впливати на їх чутливість?

3.2.2. Постановка задачі

Набір даних

Я проводив оцінку моделей DNN на наборі даних MNIST, широкому асортименті бази даних рукописних цифр, який зазвичай використовується для тренування різних систем обробки зображень [citelecun-mnisthandwritten digit]. Я також використовував набір даних CIFAR-10 [9], колекцію часто використовуваних зображень для тренування алгоритмів машинного навчання та комп'ютерного зору. Ці два набори даних також використовувалися в обох початкових статтях, які запропонували SA і DeepMutation. Тому наш вибір не буде спрямований на одну метрику.

Ворожі атаки

Оскільки різні атаки можуть мати різні характеристики (для деяких тестових критеріїв можуть бути важчими для виявлення, а для деяких - легшими), необхідно оцінювати два критерії адекватності тестування для кількох типів атак. Я вибрав атаки FGSM, JSMA, C&W, Deepfool та Blackbox для питання 1 і питання 2, які використовувалися в обох початкових роботах Surprise Adequacy та Model Mutation.

Досліджувана модель DNN

У папері Surprise Adequacy [11] цільовими глибокими моделями навчання для наборів даних MNIST та CIFAR-10 є мережі з п'ятьма шарами свертки та дванадцятьма шарами свертки відповідно, тоді як у папері моделі мутації [13] використовується LeNet та GoogleNet відповідно. Щоб бути неупередженим, я

вибрав одну модель з кожного паперу для кожного набору даних наступним чином:

• MNIST: LeNet та 5-шарові згорткові мережі (з точністю відповідно 98,6% та 95,3%)

• CIFAR-10: GoogleNet та 12-шарові згорткові мережі (з точністю відповідно 90,5% та 80,7%)

Конфігурація моделі

У налаштування для кожного дослідження потрібна окрема конфігурація, як описано нижче. Для питання 2 налаштування наступні:

• Конфігурація Surprise Coverage: Був встановлений типовий поріг активації для метрики LSC на рівні 105, так само, як у вихідній статті [11], а також встановив найглибший прихований шар в кожній моделі при обчисленні LSC. Це тому, що Kim et al. [11] не змогли знайти жодної кореляції з глибиною шару [15], і їх експеримент також був проведений на найглибшому прихованому шарі.

• Конфігурація Mutation Model: Для методу мутації моделей потрібно встановити дві конфігурації. По-перше, оператор мутації, який генерує змінені моделі, а по-друге, швидкість мутації. Для цього експерименту я вибрав Gaussian Fuzzing (GF), оскільки він був ефективнішим за інші методи в оригінальній статті. Генеруючи десять мутованих моделей для кожної моделі, я враховую її внутрішню випадковість. Для набору даних MNIST я вибрав найнижчу швидкість мутації 0,01, оскільки в оригінальній статті не було ніяких рекомендацій.

Для питання 3 мета полягає в вимірюванні чутливості метрик щодо змін параметрів. Тому необхідно встановити кілька конфігурацій. Кожна конфігурація змінює один аспект експерименту порівняно з питанням 2 і залишає все інше без змін. Ось елементи, які я контролюю в питанні 3:

• Обрання набору даних та моделі.

Як описано вище, я використовую набір даних MNIST та моделі LeNet і 5-шарові згорткові мережі для експерименту питання 2. Для відповіді на

запитання про те, наскільки чутливими є метрики в питанні 3, я змінюю набір даних MNIST на CIFAR10 і змінюю моделі на GoogleNet і 12-шарові згорткові мережі. Я також використовую різні швидкості мутації для моделей CIFAR-10 (0.007), як рекомендується в оригінальній статті LCR.

• Вибір шару для покриття в Surprise Coverage: одним з важливих гіперпараметрів LSC є вибір шару для вимірювання покриття. У питанні 2 я вирішив вибрати найглибший прихований шар, як запропоновано і використовувалося в оригінальній статті. У питанні 3 я використовую перший прихований шар. Зверніть увагу, що я не повторюю експеримент з усіма атаками та моделями, оскільки я знайшов цікаві спостереження у першому випадковому випадку (тобто наборі даних MNIST, моделі LeNet та атаки FGSM).

• Вибір рівня мутації: У питанні 2 я використовую найнижчий рівень мутації для MNIST і найвищий для CIFAR10, згідно з варіантами, дослідженими в оригінальній статті LCR [15]. Це через те, що в статті немає явної рекомендації, як обирати рівень. У питанні 3 я змінюю рівень на MNIST з найнижчого (0,01) на найвищий (0,05). Я очікую, що ця зміна збільшить приріст, як це було помічено в оригінальній статті. Я використовував набір даних MNIST та модель LeNet, та випадковий атакувальний метод JSMA.

• Вибір оператора мутації. У початковій статті LCR було реалізовано три з шести операторів мутації, згаданих у статті DeepMutation, оскільки інші оператори не підходять для даного дослідження. Знову ж таки, оскільки серед цих трьох операторів немає явного переможця (тому в статті немає рекомендацій), у питанні 2 я випадковим чином вибрав один оператор (GF) і дослідив два інших оператори (NS і WS) в питанні 3. Я використав ту ж саму настройку, що й у попередньому дослідженні: набір даних MNIST, модель LeNet і атака JSMA, оскільки обидва оцінюють гіперпараметри LCR.

Оцінка Вимірювання

Для критерію Surprise Adequacy я використовував метрику покриття Likelihood-based Surprise Coverage (LSC), а не DSA. DSA використовується

лише для класифікації атакованих та оригінальних прикладів, а не для розрахунку покриття нейронів. Для DeepMutation є лише один вибір - це Label Change Rate (LCR), як пояснено в Розділі 2.

Проблематика експерименту

Щоб краще побачити вплив метрики адекватності на виявлення атак, я не дотримуюся дизайну попередніх робіт, які додають всі атаковані приклади одним разом та спостерігають різницю в значеннях метрик [15] [11]. Замість цього, я поступово додаю атаковані приклади (на 1% за раз) до оригінального тестового датасету та спостерігаю зміну значень LCR та LSC порівняно з оригінальними даними.

Таблиця 3.5 показує кількість атакованих тестових вхідних даних для кожної атаки з використанням набору даних MNIST та CIFAR-10. Я починаю з набору оригінальних (невикористаних для атак) тестових вхідних даних для кожного експерименту, такого ж розміру, як і згенерованих адверсарних атак. Цей набір я вибираю випадково з усіх оригінальних тестових даних будь-якого з наборів даних.

Таблиця 3.5 - Кількість прикладів спорб зловмисників на одну атаку, на один набір даних

Ворожі атаки	MNIST	CIFAR-10
Black-Box	2000	1000
Deepfool	1000	1000
FGSM	2000	2000
JSMA	1000	2000
C&W	700	1000

Для оригінальних тестових вхідних даних O, я додаю 1% атакованих вхідних даних A до оригінальних вхідних даних в кожній ітерації, після чого обчислюю нові критерії придатності тестування. Це означає, що я оновлюю набір тестових вхідних даних в кожній ітерації та вимірюю нові показники LCR

та LSC. Таким чином, я можу побачити ефективність показників LCR та LSC у виявленні атакованих прикладів з високою деталізацією. Набір тестових вхідних даних Ti оновлюється наступним чином, для i = {1, 2, ..., 100}:

$$T_i = T_{i-1} + A_i$$

Де Ai є 1% від залишкових атак A у кожній ітерації, а Ti - загальна кількість тестових вхідних даних у i-ій ітерації для i = {1, 2, ..., 100}. Зверніть увагу, що оскільки я використовую 1% атак кожного разу, я округлюю кількість атак до найближчого сотні. Наприклад, загальна кількість згенерованих атак з використанням C&W для MNIST дорівнює 730, але я використовую 700, щоб вибрати сім атак у кожній ітерації.

Щоб вирішити проблему випадковості в дизайні експерименту, я повторив процес десять разів для кожної атаки на кожному датасеті. Медіанні значення обчислюються, щоб повідомити приріст LSC та LCR.

3.2.4. Обговорення проблеми та результати

Питання 1: Яка з двох метрик (MS чи SC) є більш чутливою у виявленні конкурентних іспитів?

Для оцінки LCR та LSC я обчислив їх значення на двох моделях (LeNet та 5-шарових зворотних зв'язків згорткових мережах) з використанням набору даних MNIST та п'яти різних атак.

Спочатку я обчислив кореляцію між «Відсотком атакованих прикладів» та «Інкрементами оцінки адекватності (LCR та LSC) порівняно з оригінальним набором тестів», використовуючи двовимірну кореляцію Пірсона. Як показує таблиця 4.6 (зверніть увагу, що таблиця 4.6 містить значення кореляції для усіх 40 експериментів в обох досліджуваних питаннях), всі кореляції дуже високі (більші 90%), що свідчить про те, що як LCR, так і LSC потенційно чутливі у

відрізненні атакованих прикладів. Знову ж таки, чутливість визначається як будь-який інкремент або декремент оцінки тестування. Однак, значення кореляції самі по собі не розкривають розмір різниці, коли атакований зразок вимірюється через LCR та LSC. Іншими словами, на практиці метрики можуть бути корисними, якщо вони повертають значно різні (набагато вищі або нижчі) результати, коли застосовуються на атакованих зразках порівняно з нормальними тестами. Тому метрика може мати високу кореляцію (збільшуватися з тією самою закономірністю, що і при додаванні атакованих тестів), але не бути дуже практичною. Що буде, якщо інкременти дуже низькі?

На рисунках 3.14 та 3.9 зображено збільшення показника (LCR та LSC), коли до тестового набору додаються деякі випадкові атаки. На кожному кроці додається 1% додаткових атак. При детальному розгляді графіків можна побачити, що для всіх атак на обох моделях глибокого навчання метрика, заснована на мутації (LCR), має значно більші збільшення, ніж метрика, заснована на покритті (LSC).

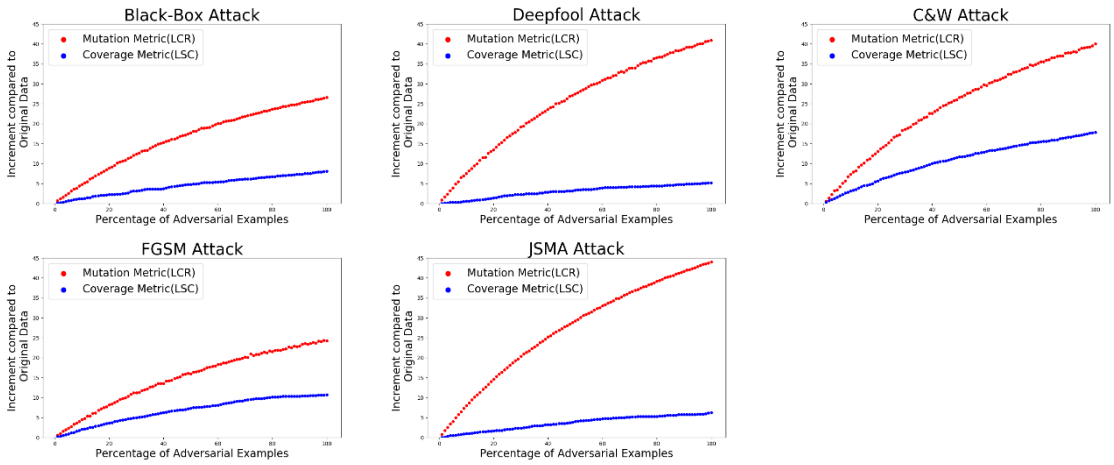

Рисунок 3.8 - Набір даних MNIST був націлений на 5-шарову згорткову модель з п'ятьма різними ворожими атаками.

Висновок для питнання 2 полягає в тому, що за вказаними параметрами LCR є більш чутливим, ніж LSC, виявленні атак на моделі, оскільки експерименти показали більшу зміну метрики для LCR.

Питання 3: Як зміна параметрів метрики може змінити її чутливість?

Вибір набору даних та моделі

Як показано на рисунках 3.10 та 3.11, за винятком чорнуватого атакування GoogleNet, де LCR так само низький, як LSC, інші дев'ять моделей атак показують той самий патерн, де LCR домінує над результатами LSC. Отже, з урахуванням наших даних, я роблю висновок, що результати досліду питання 2 є стійкими щодо зміни датасету та моделей.

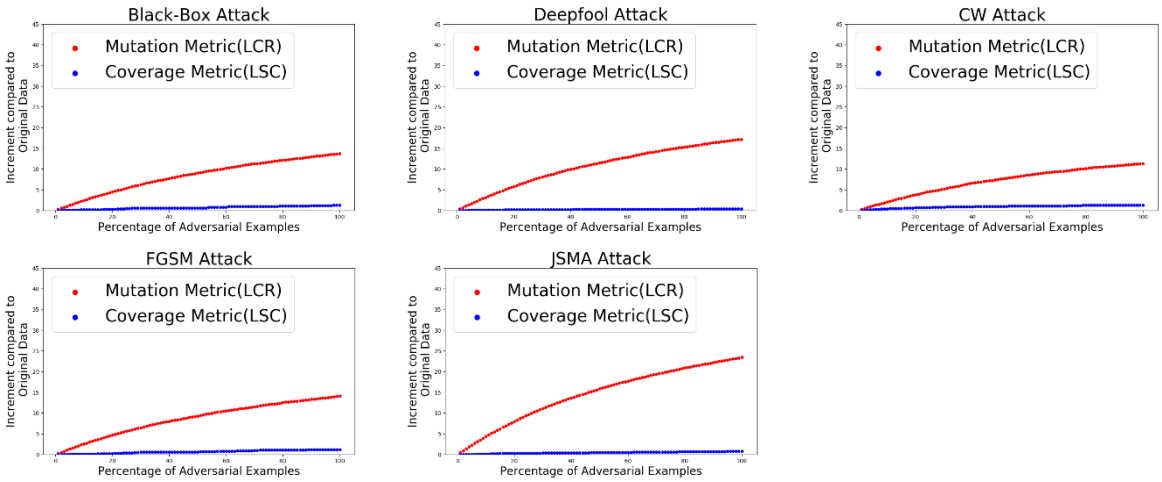

Рисунок 3.9 - Набір даних MNIST націлений на модель LeNet з п'ятьма різними ворожими атаками

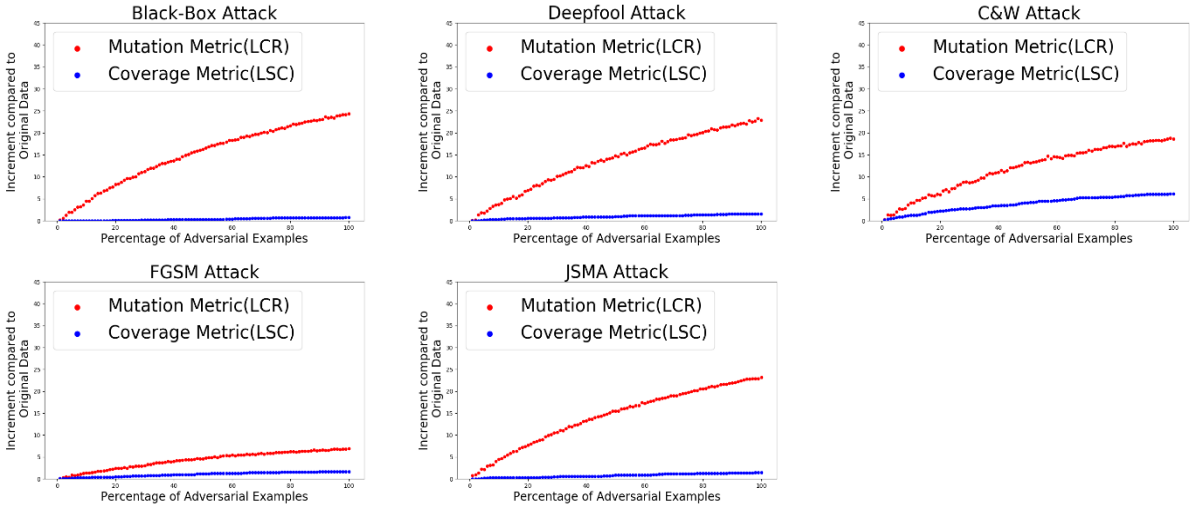

Рисунок 3.10 - Набір даних CIFAR-10 націлений на 12-шарову згорткову модель з п'ятьма різними рекламними прикладами.

Вибір Surprise шару покриття

Як показано на Рисунку 3.13, зміна шару призвела до більшого збільшення покриття несподіванки (LSC), а також зробила LSC краще, ніж LCR. Цей один приклад був достатнім, щоб продемонструвати, що вибір параметра гіперпараметра адекватності метрики (тут шар в LSC) важливий для виявлення атак. Іншими словами, з правильною настройкою можна використовувати LSC порівняно з LCR або навпаки.

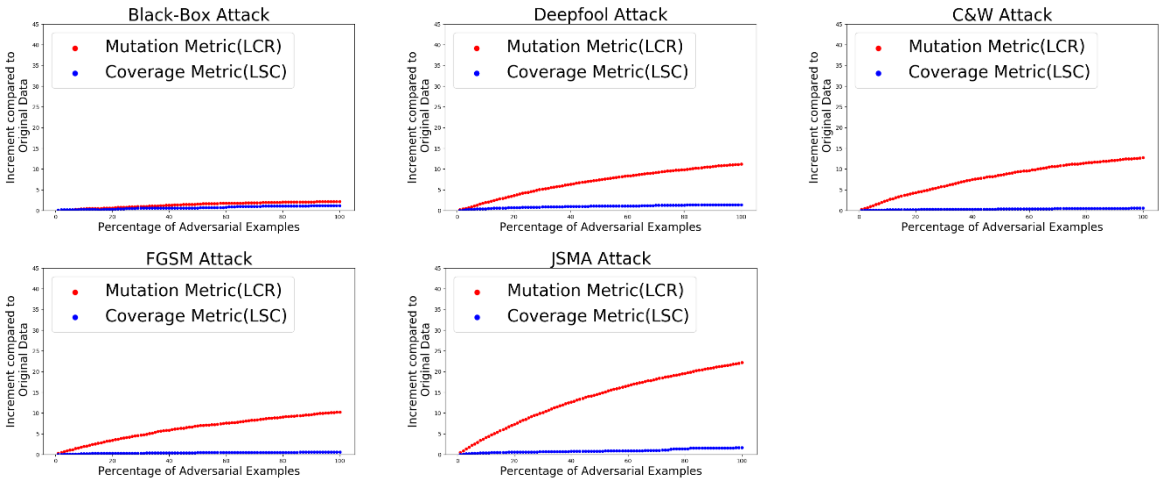

Рисунок 3.11 - Набір даних CIFAR-10 націлений на модель GoogleNet з п'ятьма різними змагальними іспитами.

Таблиця 3.6 - Коефіцієнт кореляції Пірсона між "Відсотком атакованих прикладів" та "Збільшенням оцінки адекватності (LCR та LSC) порівняно з вихідними тестовими даними".

	Модель	Black-Box	Deepfool	FGSM	JSMA	C&W
SC	LeNet	0.987	0.929	0.99	0.978	0.947
	Conv 5	0.994	0.982	0.98	0.985	0.986
	GooglLeNet	0.988	0.956	0.963	0.969	0.972
	Conv 12	0.98	0.989	0.98	0.99	0.986
Mutation	LeNet	0.985	0.984	0.984	0.982	0.984
	Conv 5	0.984	0.984	0.985	0.984	0.984
	Googl LeNet	0.978	0.984	0.985	0.984	0.982

	Conv 12	0.983	0.989	0.98	0.9985	0.98

Вибір швидкості мутації

Рисунок 3.14 показує, що поліпшення швидкості мутації може призвести до більшого зростання показника зміни мітки (LCR). LCR збільшується, але не змінюється патерн з більшими приростами LCR порівняно з LSC. Однак зміна швидкості мутації може змінити значення приростів, що підтримує попередню твердження експерименту, що за належної настройки можна використовувати LCR чи LSC.

Вибір оператора мутації

Рисунок 3.12 показує результати. Як показано, оператор Neuron Switch (NS) навіть не видно на діаграмі, оскільки його приріст є від'ємним. Для оператора Weight Shuffling (WS) приріст є позитивним, але він не настільки великий, як у випадку застосування стандартного оператора GF. Ще раз, ефективність LCR здається чутливою до вибору оператора.

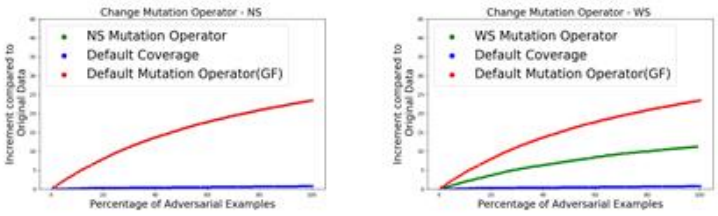

Рисунок 3.12 - Ефект від зміни оператора мутації в LCR

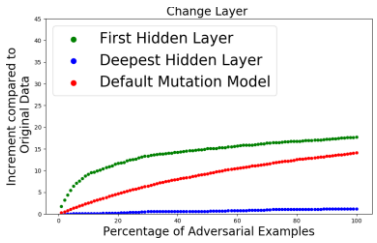

Рисунок 3.13 - Ефект зміни шарів у LSC

Висновок по витанню 3:

Дослід питання 3 показує, що обидва метрики є чутливими до своїх параметрів. Їх продуктивність може бути порушена через неправильне налаштування параметрів до того рівня, коли вони вже не є прогностичними. Тому налаштування гіперпараметрів є суттєвим для інструментів і метрик тестування в цій галузі.

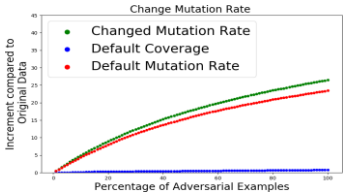

Рисунок 3.14 - Вплив зміни швидкості мутацій в LCR

3.3. Тестування DNN: Генерація тестових вхідних даних

3.3.1. Огляд проблеми

У третьому експерименті досліджується третій аспект тестування нейромереж (DNN) - генерація вхідних тестів. Мета полягає в тому, щоб створювати вхідні тести (адверсальні зразки) для моделей DNN, які знаходять проблеми з моделлю (наприклад, помилкову класифікацію) і потім виправляють модель, перетренуючи її зі згенерованими вхідними тестами.

Мета

У цьому експерименті спочатку я генерую адверсарні приклади за допомогою трьох різних методологій. Далі я досліджую продуктивність моделі на трьох завданнях в галузі програмної інженерії, і нарешті перетреную моделі з використанням згенерованих прикладів тестування, і перевірю, чи є якісь покращення в F1-показнику.

Дослідницьке питання

В данній частині розглядаються такі питання як:

• Питання 4: Як змінюється показник оцінки моделі при використанні адверсарних тестових вхідних даних?

• Питання 5: Як впливає повторне навчання моделі на показник F1-оцінки?

• Питання 6: Чи має вибір задачі оцінювання вплив на покращення продуктивності моделі?

3.3.2. Постановка задачі

У цьому експерименті увага зосереджена на генерації атакуючих прикладів для тестування DNN-моделей, які використовуються у задачах інженерії програмного забезпечення, таких як генерація зведень коду та коментарів. Більшість завдань інженерії програмного забезпечення потребують кодових фрагментів як вхідних даних, що вимагає створення кодових вкладень для живлення DNN. У цій дисертації тестируються три відомі DNN-моделі для генерації кодових вкладень, і їх навчені моделі оцінюються на трьох різних вторинних завданнях, які обговорювалися в розділі 3.3.2.

Досліджувальна DNN модель

У сфері програмної інженерії існує три відомих інструменти для вбудовування коду під назвами Code2vec [16], Code2seq [17] та CodeBERT [18]. Усі три моделі пропонують метод вбудовування фрагментів коду, оскільки широке коло застосувань у програмній інженерії, таких як підсумовування коду, документація та пошук, вимагають їх використання. Code2vec пропонує нейронну модель для представлення фрагментів коду у вигляді неперервно розподілених векторів, тоді як Code2seq запропонувала трансформацію для генерації послідовностей природньої мови з фрагментів вихідного коду. Нарешті, обидві моделі оцінюють свій метод, передбачаючи назву методу на основі вихідного коду.

CodeBERT навчається універсальним представленням, що підтримує додаткові завдання у сфері програмної інженерії, такі як пошук коду за природною мовою і генерація документації для коду. Це двохмодельна

переднавчена модель для природної мови (Natural Language, NL) та мов програмування (Programming Language, PL), таких як Python та Java.

Дані

У цьому дослідженні я використовую оригінальні необроблені java-файли як вхідні дані та генерую адверсарні приклади за допомогою оригінальних. Далі я спочатку пояснюю деталі оригінального набору даних, а потім говорю про генерацію адверсарних вхідних прикладів.

Оригінальні файли Java: Як Code2vec, так і Code2seq підтримують мови програмування Java і C# як вхідні кодові фрагменти. Code2seq також підтримує мову Python. CodeBERT підтримує згадані мови програмування, а також JavaScript, PHP, Ruby та Go.

Датасет, опублікований Code2vec, має передопрацьований формат, але в цьому дослідженні мені потрібні були сирі файли, щоб створити атакуючі приклади. Він також розділяє датасет на тренувальні, валідаційні та тестові набори за одним файлом Java, тоді як Code2seq розділяє їх за проектними папками. На щастя, датасети, які супроводжуються Code2seq, містять сирі файли Java. CodeBERT також забезпечив передопрацьовані набори даних для тренування та валідації і надав код для передопрацювання тестового датасету. Отже, я використовував датасет Java-Large на сторінці Github Code2seq для всіх трьох моделей, включаючи 9000 проектів Java для тренування, 200 для валідації та 300 для тестування. Цей датасет містить близько 16 мільйонів прикладів.

Оскільки Code2vec розбиває набір даних на окремі файли Java, спочатку я перемістив усі файли Java в один каталог і використовував його для навчання. В цілому у мене було близько 1,9 млн файлів Java. Стислий розмір становить близько 4 ГБ, а розпакований розмір - близько 20 ГБ.

Таблиця 3.7 - Оригінальна оцінка F1 моделі наведена у відповідній статті з конфігураціями за замовчуванням на наборі даних за замовчуванням

Інструмент	F1-score	Подальше завдання

Code2vec	58.4	Прогнозування назви методу
Code2seq	59.19	Прогнозування назви методу
CodeBERT	74.84	Пошук коду

Цільові файли Java:

Як обговорювалося у розділі 2.7, є кілька методів для генерації цільових прикладів для текстових і зображень входу. Однак вони не використовні, коли мова йде про фрагменти коду. Альтернативним методом генерації атакуючих прикладів для фрагментів коду без втрати важливої інформації або будь-яких змін в структурі коду є рефакторинг коду. Для цього дослідження я використовував десять методів рефакторингу для генерації атакуючих прикладів, які перелічені у розділі 2.7.

Застосувавши відповідні методи рефакторингу до вихідних java-файлів у тренувальному та тестовому наборах даних, загальна кількість початкових програм становить 1 798 419 файлів для тренування, 44 140 файлів для валідації та 59 404 файлів для тестування. Я створив таку ж кількість ворожих java-файлів для кожного методу, використовуючи три різні методи, описані нижче:

1. Одноразова мутація: застосувати оператор рефакторингу до методу Java один раз.

2. K-кратна мутація: застосувати п'ять різних операторів рефакторингу до методу Java.

3. Керована мутація: застосовувати оператори рефакторингу до методу Java, керуючи тестовим вводом за допомогою бали мутації протягом 100 ітерацій.

Зверніть увагу, що мутація в цих трьох методиках означає застосування методів рефакторингу до кожного java-файлу для створення адверсального

прикладу відповідного java-файлу. Розділ 3.3.3 детально описує кожну генеративну модель.

Конфігурація моделі

Інструменти Code2vec, Code2seq та CodeBERT є доступними для повторення досліджень. У таблиці 4.7 показано значення F1-оцінки, які були звітовані в кожному зі статей для відповідного інструменту. Хоча навчена модель для Code2seq та CodeBERT відповідає продемонстрованій продуктивності у відповідних статтях, Code2vec не міг досягнути F1-оцінки, звітованої в оригінальній статті, через відсутність публічно доступних наборів даних. Крім того, через перенесення всіх java-файлів у єдину теку, значення F1-оцінки для Code2vec також відрізняється від того, що звітується в статті про code2seq, яка використовувала набір даних Java-Large.

У експерименті я встановив кількість епох 20 і залишив інші налаштування такі, як рекомендовано в відповідних статтях. Для експерименту GM я встановив частоту мутації 0,05, оскільки дослідження показують, що це хороша частота мутації для проблеми генетичного алгоритму [3].

Оцінювання Вимірювання

У цьому дослідженні я оцінюю навчену модель на трьох різних вторинних завданнях: прогнозування назв методу, генерування опису коду та пошук коду.

• Прогнозування назв методу: передбачення назви методу на основі тіла методу. Метрикою оцінки є F1-оцінка для підтокенів.

• Генерування опису коду: передбачення повного речення на прикладі короткого фрагменту коду на мові Java. У цьому завданні довжина цільової послідовності становить близько десяти слів у середньому. Модель оцінюється за допомогою ROUGE-N та ROUGE-L F1-оцінок.

• Пошук коду: за вхідний параметр береться природну мову, а завдання полягає в пошуку найбільш семантично пов'язаного коду з колекції кодів. Метрикою оцінки є F1-оцінка.

ROUGE, або Recall-Oriented Understudy for Gisting Evaluation [4], це набір метрик та програмного забезпечення, що використовуються для оцінювання програмного забезпечення автоматичної резюмування та машинного перекладу в обробці природної мови. Метрики порівнюють автоматично створену резюме або переклад з людською резюме або перекладом. Доступні наступні п'ять метрик оцінки:

- ROUGE-N: Належність N-грамів [6] між системою та довідковими резюме.
- ROUGE-1 відноситься до належності одиночних слів (уніграм) між системою та довідковими резюме.
- ROUGE-2 відноситься до належності біграм між системою та довідковими резюме.
- ROUGE-L: статистика на основі найдовшої спільної послідовності (LCS) [3]. Задача знайдення найдовшої спільної підпослідовності враховує подібність структури на рівні речень та автоматично ідентифікує найдовші спільно входящі в послідовність n-грами.

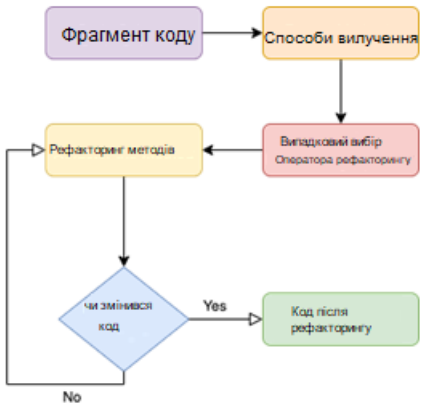

Рисунок 3.15 - Процес 1-кратної мутації, який було використано у третьому експерименті

3.3.3. Проблематика експерименту

1-кратна мутація: як показано на Рисунку 3.15, у методі 1-кратної мутації рефакторяться всі початкові файли всього один раз. Точніше кажучи, спочатку модель виділяє кожен метод Java, а потім випадковим чином вибирає оператор рефакторингу з пулу операторів, який застосовується до методу. Варто зазначити, що деякі випадкові техніки рефакторингу можуть бути непридатні для методу Java. Наприклад, якщо певний метод не містить жодного циклу, то випадково обраний метод покращення циклу не може бути застосований тут. Тому я повторюю процес, поки не переконаюся, що метод рефакторований.

Після того, як всі методи виділені та рефакторовані, генеруються відверсійні файли Java. На Рисунку 3.19 наведено зразок 1-разового рефакторингу, за допомогою оператора додавання аргументу.

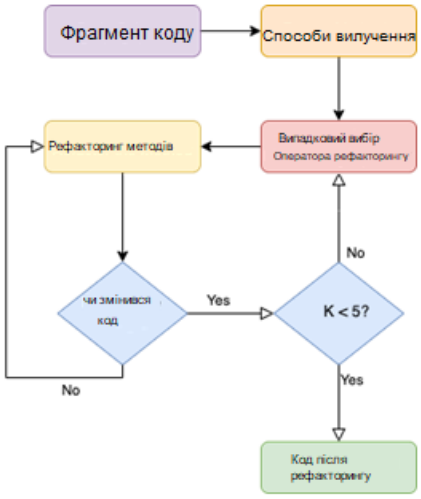

Рисунок 3.16 - процес 5-кратної мутації, що використовувався в третьому експерименті.

К-кратна мутація: Підхід к-кратної мутації схожий на підхід 1-разової мутації, з рефакторингом одного методу к-разів. Після видобування кожного методу Java застосовується випадково вибраний оператор рефакторингу, і цей процес повторюється к-разів для кожного методу. Іншими словами, кожен метод Java рефакторіться к-разів з к-випадково вибраними операторами, як показано на рисунку 3.16. Знову ж таки, деякі випадкові техніки рефакторингу

можуть бути непридатними. Тому я ітерую процес з різними операторами, щоб переконатися, що метод був рефакторингований. У цій дисертації я розглядаю K = 5, оскільки Томас та ін. [15] оцінили різні значення K та запропонували, що K = 5 має найкращий показник F1-оцінки. На рисунку 3.20 показано приклад згенерованого зразка коду за допомогою техніки адверсарної мутації 5-разів. Як зазначено, у цьому зразку коду були використані оператори рефакторингу перейменування локальної змінної, додавання аргументу, покращення циклу for, додавання друку та перейменування імені методу.

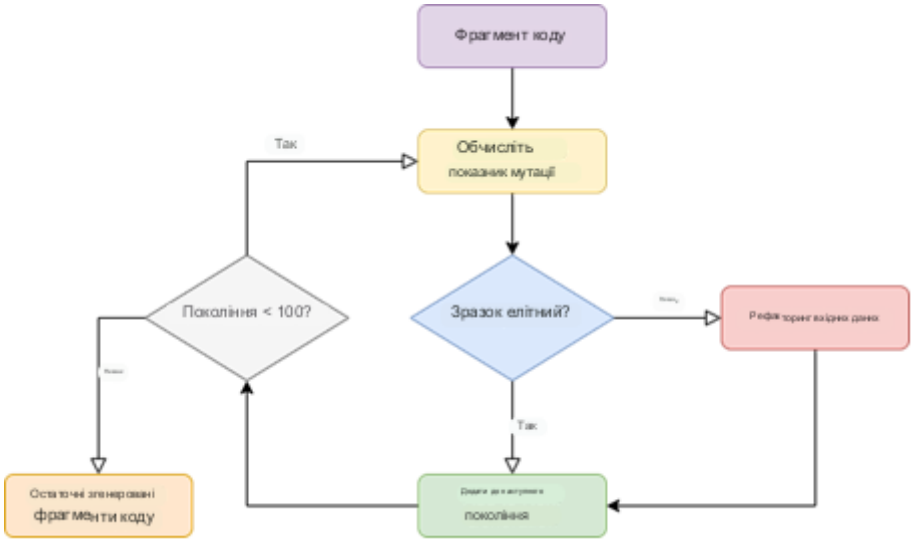

Рисунок 3.17 - Процес керованої мутації, використаний у третьому експерменті

Цей експеримент запропонував методологію Guided Mutation (GM) для генерації адверсарних вхідних даних для вбудовування коду. Метод Guided mutation є еволюційною стратегією, натхненною генетичним алгоритмом (GA) за структурою, але без застосування кросовера на генерацію вхідних даних. Після того, як визначено генетичну репрезентацію та функцію пристосованості, GA продовжує ініціалізувати популяцію рішень та поліпшувати її за допомогою повторних застосувань мутації та кросовера. Як обговорювалось, у методі GM я застосовую лише мутацію, а не кросовер на вхідних даних. Причина полягає в тому, що зміна фрагментів коду за допомогою кросовера

може призвести до багатьох помилкових (навіть некомпільованих) фрагментів коду, не кажучи вже про збереження функціональності коду.

Елітизм у генетичному алгоритмі передбачає копіювання невеликої частини найбільш пристосованих кандидатів без змін до наступного покоління. Це може мати драматичний вплив на продуктивність, забезпечуючи, що еволюційний алгоритм не витрачає час на повторне відкриття раніше відкинутих часткових рішень. Кандидати, які залишаються незмінними завдяки елітизму, можуть бути вибрані батьками для розвитку наступного покоління.

Як показано на малюнку 3.17, для генерації адверсних зразків за допомогою методології GM потрібно виконати наступні кроки:

1. Обчислити бал мутації для поточного покоління.

2. Вибрати елітні кандидати на основі найвищого балу мутації та скопіювати їх у наступне покоління.

3. Мутувати решту кандидатів з вказаними швидкостями мутації.

4. Повторювати крок перший до досягнення критерію зупинки (100 ітерацій).

DeepMutation++ [14] - це фреймворк для тестування мутацій з метою аналізу якості тестових даних. Таким чином, у цій дисертації я використовую метрику балів мутації в DeepMutation++ для оцінки якості тестових даних, які я генерую за допомогою методології GM.

Таблиця 3.8 - Оператори мутації

Рівень		**Оператор**	**Пояснення**
Статичний	**Вага**	Фаззінг ваги за допомогою гаусівського розподілу (WGF)	Фаззінг ваги
		Зменшення точності ваги (WPR)	Зменшення точності ваг
Динамічний	**Статус**	Очистка стану до 0 (SC)	Очистити стан до 0
		Скидання стану до попереднього стану (SR)	Скинути стан на попередній стан

		Фаззінг значення стану за допомогою гаусівського розподілу (SGF)	Фаззінг значення стану
		Зменшення точності значення стану (SPR)	Зменшення точності значення стану
	Гейт	Очистка значення гейту до 0 (GC)	Очистити значення гейту до 0
		Фаззінг значення гейту за допомогою гаусівського розподілу (GGF)	Фаззінг значення гейту
		Зменшення точності значення гейту (GPR)	Зменшення точності значення гейту

Спочатку DeepMutation [13] вперше запропонував вісім операторів на рівні моделі для глибинних нейронних мереж (FNN), щоб створювати модельні мутанти. Пізніше DeepMutation++ запропонував дев'ять нових операторів, спеціалізованих на рекурентних нейронних мережах (RNN). Він підтримує статичне генерування мутантів для аналізу тестових даних в цілому та динамічне генерування мутантів для виявлення вразливих сегментів тестового вводу. У таблиці 3.8 показано 9 операторів, визначених DeepMutation++.

DeepMutation++ спочатку використовує надані оператори мутації для генерації набору високоякісних мутантів DNN з певним заданим порогом якості. Після створення певної кількості мутантів DeepMutation++ аналізує відмінності у поведінці початкової DNN та згенерованих мутантів DNN на наданих вхідних даних. Нарешті, вона виводить якість тестових даних, надаючи мутаційний бал, враховуючи конкретну модель.

Задано вхід t, DNN m та його мутанта mi. Вони визначають, що t вбивається mi, якщо результати відповідних вихідних даних не узгоджуються при t, тобто $m(t) \neq m^i(t)$. Задано набір мутантів DNN, M, і визначено мутаційний бал як:

$$MS(t,m,M) = \frac{|\{m' | m' \in M \wedge m(t) \neq m'(t)\}|}{|M|}$$

Таблиця 3.9 - Code2vec F1-оцінки на оригінальних та перенавчених моделях, що оцінюють завдання передбачення назви методу

Code2vec – Прогнозування назви методу					
		Оригінальне	1-раз	5-разів	GM
Тип	Оригінальне	35.9251	34.7597	33.8735	35.4537
	1-раз	36.4816	35.9213	35.7913	38.7394
	5-разів	36.4008	35.4896	36.1684	37.0826
	GM	36.1465	47.3447	46.5198	53.1309

У цьому експерименті я використовував оцінку якості тестів, що виражається у mutation score, як функцію пристосуваності для оцінки тестових вхідних даних для моделі GM. Я використовував усі дев'ять операторів для моделей RNN та створив по десять мутантів з кожним з них. Таким чином, загалом у мене є 90 мутантів для моделі. На рисунку 3.21 показані адверсарні приклади, згенеровані GM за допомогою операторів локальної зміни назв змінних та покращення циклів IF.

Тренування

Тренування (Retraining) передбачає повторний запуск процесу, який генерував вибрану раніше модель на новому тренувальному наборі даних. Функції, алгоритм моделі та простір пошуку гіперпараметрів повинні залишитися незмінними.

У цьому експерименті я перетренував модель, використовуючи згенеровані зразки ворожих прикладів, щоб дослідити, чи навчилася модель краще чи ні. Очікується, що модель, навчена з новими згенерованими даними, буде більш стійкою під час тестування на ворожих наборах даних.

3.3.4. Розгляд проблеми та результати

У цьому розділі я спочатку повідомляю результати, а потім обговорюю їх для кожного дослідницького запитання окремо. Таблиці 3.9, 3.10 та 3.11 показують результати для експериментів з code2vec та code2seq відповідно. Стовпці представляють оригінальні тестові дані, а також адверсальні дані, а рядки показують оригінальні та перетреновані моделі з використанням адверсальних тренувальних даних.

У наступному розділі розглянуто результати щодо дослідницьких запитань:

Питання 4: Як змінюється оцінка моделі при тестуванні з використанням адверсарних вхідних даних?

Як показано в таблицях 3.9, 3.10 і 3.11, стовпці показують результат тестування на завданні використання навченої моделі. Рядки, однак, вказують на навчену модель, використовуючи оригінальний, 1-разовий, K-разовий та GM набори даних.

Таблиця 3.10 - Code2vec F1-оцінки на оригінальних та перенавчених моделях, що оцінюють завдання передбачення назви методу

Code2vec – Прогнозування назви методу					
		Оригінальне	1-раз	5-разів	GM
Тип	Оригінальне	42.7167	40.7856	40.1655	38.6673
	1-раз	39.5952	42.3478	42.3582	43.8929
	5-разів	39.9891	43.2936	45.9164	44.4910
	GM	41.1948	49.1119	52.2238	56.2494

Для відповіді на це дослідження я спочатку обговорюю результати тестування для навченої моделі з використанням початкового набору даних. Як показано в таблицях 3.9, тестування початкової навченої моделі на початкових тестових випадках має F1-бал 35,92. Цей бал зменшується з використанням адверсарних тестових входів. Цей результат показує, що при наданні адверсарних тестових вхідних даних, початкова модель втрачає бали через непередбачуваний набір даних.

Те ж саме стосується таблиць 3.10 і 3.11. Оцінка тестування адверсарних прикладів має менший бал, ніж початкові тестові дані.

Підсумок по дослідженню питання 4:

Отже, відповідь на це питання полягає в тому, що тестування оригінальної моделі на прикладах знижує точність і F1-Score моделі

Питання 5: Як перетренування моделі впливає на F1-оцінку?

У цьому розділі обговорюється вплив перетренування моделі на F1-оцінку з трьома різними прикладами атак на тестові та тренувальні дані.

У всіх таблицях перетренування моделі з використанням атак на тестові дані призводить до зниження оцінки для оригінального набору тестів. Це очікувано, оскільки модель може перенавчитися під час процесу навчання, і результати можуть бути не такі хороші, як для моделі, навченої за допомогою оригінального набору даних. Як показано в усіх трьох таблицях, порівняння результатів для 1-time, 5-time та GM після перетренування дає кращі результати, ніж оригінальна модель при тестуванні на атакованих прикладах. Це свідчить про те, що з атакованими прикладами перетренування моделі може покращити її результати в цілому. Як бачимо, перетренування моделі з використанням GM тестових даних має найвищу F1-оцінку серед усіх моделей та тестів.

Таблиця 3.11 - Code2vec F1-оцінки на оригінальних та перенавчених моделях, що оцінюють завдання передбачення назви методу

CodeBERT - тест на пошук коду					
		Оригінальне	1-раз	5-разів	GM
Тип	Оригінальне	81.36	80.19	79.81	80.65
	1-раз	80.16	81.45	80.36	82.33
	5-разів	80.53	81.76	82.84	82.49
	GM	81.68	82.37	82.52	82.97

Таблиця 3.12 - Code2seq Rouge-1 F1-рахунок на оригінальних та перенавчених моделях, що оцінюють завдання субтитрування коду

Code2seq - Кодові субтитри					
		Оригінальне	1-раз	5-разів	GM
Тип	Оригінальне	52.0947	43.1603	47.5590	49.3676
	1-раз	40.9169	43.3944	45.8432	46.9653
	5-разів	34.6725	46.9821	49.0806	54.2867
	GM	36.1937	49.8223	55.1286	61.7328

Висновок по дослідженню питання 5:

За результатами досліджень маємо змогу зробити висновок, що перенавчання моделі покращує продуктивність моделі при виконанні тієї ж самої задачі.

Питання 6: Чи має вибір завдання оцінки якийсь вплив на поліпшення продуктивності моделі?

У останньому експерименті я використовував три різних завдання для оцінки наших моделей. Code2vec та Code2seq були оцінені на передбачення імені методу, Code2seq також було оцінено на підписування коду, і нарешті, CodeBERT оцінювався за пошуком коду.

Як показано в таблиці 4.9, F1-бал Code2vec для передбачення імені методу поліпшився на 17,27%. Аналогічно, ми можемо побачити поліпшення на 13,52% для Code2seq на передбаченні імені методу. Code2seq для завдання підписування коду також мав помітне поліпшення, як показано в таблиці 4.12.

Хоча в таблиці 4.11 є невеликий приріст після повторного навчання моделі з трьома адверсарними прикладами, поліпшення свідчить про те, що окрім завдання оцінки, повторне навчання моделі з використанням адверсарних зразків покращує продуктивність моделі.

Висновок по дослідженню питання 6:

Перетренування моделі вбудування коду за допомогою адверсарних зразків покращує продуктивність моделі, незалежно від завдання оцінювання моделі.

```
private static JoinPoint currentJoinPoint() {
  MethodInvocation mi = ExposeInvocationInterceptor.currentInvocation();
  if (!(mi instanceof ProxyMethodInvocation)) {
    throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ mi);
  }
  ProxyMethodInvocation pmi = (ProxyMethodInvocation) mi;
  JoinPoint jp = (JoinPoint) pmi.getUserAttribute(JOIN_POINT_KEY);
  if (jp == null) {
    jp = new MethodInvocationProceedingJoinPoint(pmi);
    pmi.setUserAttribute(JOIN_POINT_KEY , jp);
  }
  return jp;
}
```

Рисунок 3.18 - Оригінальний фрагмент коду Java з набору даних Java-Large.

```
private static JoinPoint currentJoinPoint(String currentJoinPoint) {
  MethodInvocation mi = ExposeInvocationInterceptor.currentInvocation();
  if (!(mi instanceof ProxyMethodInvocation)) {
    throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ mi);
  }
  ProxyMethodInvocation pmi = (ProxyMethodInvocation) mi;
  JoinPoint jp = (JoinPoint) pmi.getUserAttribute(JOIN_POINT_KEY);
  if (jp == null) {
    jp = new MethodInvocationProceedingJoinPoint(pmi);
    pmi.setUserAttribute(JOIN_POINT_KEY , jp);
  }
  return jp;
}
```

Рисунок 3.19 - Згенерований код змагальний за допомогою методу 1-разового рефакторингу. Підкреслені рядки вказують на зміни у порівнянні з оригінальним фрагментом коду

```
private static JoinPoint currentjoinpoint(String new_JoinPoint) {
  MethodInvocation im = ExposeInvocationInterceptor.currentInvocation();
  if (!(im instanceof ProxyMethodInvocation)) {
    throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ im);
    System.out.print("MethodInvocation is not a Spring ProxyMethodInvocation: "+ im);
  }
  ProxyMethodInvocation pmi = (ProxyMethodInvocation) im;
  JoinPoint jp = (JoinPoint) pmi.getUserAttribute(JOIN_POINT_KEY);
  for (i = 0; i < 1; i++){
    if (jp == null) {
      jp = new MethodInvocationProceedingJoinPoint(pmi);
      pmi.setUserAttribute(JOIN_POINT_KEY , jp);
    }
  }
  return jp;
}
```

Рисунок 3.20 - Згенерований код змагальний за допомогою методу 5-разового рефакторингу. Підкреслені рядки вказують на зміни у порівнянні з оригінальним фрагментом коду

```
private static JoinPoint currentJoinPoint() {
  ProcessInvocation returningName = ExposeInvocationInterceptor.currentInvocation();
  if (!(returningName instanceof ProxyProcessInvocation)) {
    throw new IllegalStateException("ProcessInvocation is not a Spring ProxyProcessInvocation: "+ returningName);
  }
  ProxyProcessInvocation declaringVariable = (ProxyProcessInvocation) returningName;
  JoinPoint aspectInstancefactory = (JoinPoint) declaringVariable.getUserAttribute(JOIN_POINT_KEY);
  if (aspectInstancefactory == null && returningName == returningName) {
    aspectInstancefactory = new ProcessInvocationProceedingJoinPoint(declaringVariable);
    declaringVariable.setUserAttribute(JOIN_POINT_KEY , aspectInstancefactory);
  }
  return aspectInstancefactory;
}
```

Рисунок 3.21 Згенерований код рефакторингу методом керованої мутації. Підкреслені рядки вказують на зміни у порівнянні з оригінальним фрагментом коду.

ВИСНОВКИ ДО РОЗДІЛУ 3

У цьому розділі описуються методології, використані для тестування глибоких нейронних мереж через три експерименти: 1) Тестування DNN: Генерація тестового оракула, 2) Тестування DNN: Оцінка достатності тестів та 3) Тестування DNN: Генерація вхідних даних для тестування. У кожному розділі розглядаються такі пункти як: огляд проблеми, проектування, експеримент, результати та обговорення.

4. ТЕСТУВАННЯ DNN СИНЕРГЕТИЧНОЮ МЕТОДОЛОГІЄЮ

4.1. Опис синергетичної методології

Синергетична методологія глибокого тестування програм використовує синергетичний підхід, який базується на аналізі трьох розглянутих методологій: тестування DNN з генерацією тестових оракулів, оцінкою адекватності тесту та генерацією тестових вхідних даних.

Перша методологія, тестування DNN з генерацією тестових оракулів, використовує глибокі нейронні мережі для створення правильних результатів відповідно до вхідних даних. Цей підхід вимагає тренування мережі на вхідних даних з правильними результатами, щоб отримати генератор тестових оракулів.

Друга методологія, оцінка адекватності тесту, використовує глибокі нейронні мережі для аналізу вихідних даних та оцінки ефективності тестових наборів. Це дозволяє виявляти, наскільки ефективними є тести виявлення дефектів програм.

Третя методологія, генерація тестових вхідних даних, використовує глибокі нейронні мережі для створення реалістичних та різноманітних тестових вхідних даних. Це дозволяє покращити покриття коду та виявляти нові помилки програми.

Синергетична методологія об'єднує ці три методології, використовуючи взаємодію між ними для досягнення більшої ефективності тестування програм з використанням глибоких нейронних мереж. Вона забезпечує автоматизований підхід до генерації тестових оракулів, оцінки адекватності тесту та генерації тестових вхідних даних, що в результаті забезпечує покращене покриття коду, виявлення більшої кількості дефектів та підвищення якості тестування програм.

Синергетична методологія починається зі збору і аналізу існуючих методологій тестування DNN з метою виявлення їхніх сильних сторін. В

результаті цього аналізу виокремлюються ключові елементи, які є найбільш ефективними в кожній методології.

Далі проводиться інтеграція цих ключових елементів, де взаємодія між ними створює синергію, що покращує результати тестування. Наприклад, генератор тестових оракулів може використовувати дані, зібрані з оцінки адекватності тесту, для створення більш реалістичних тестових оракулів. Водночас, генератор тестових вхідних даних може використовувати інформацію про покриття коду, отриману з генерації тестових оракулів, для створення більш різноманітних тестових вхідних даних.

Окрім того, в рамках синергетичної методології можуть бути введені нові елементи, що посилюють її ефективність. Наприклад, можна використовувати механізми самоадаптації нейронних мереж, що дозволяють пристосовувати їх до змінних умов тестування та покращувати їх продуктивність.

Загальна мета синергетичної методології полягає в створенні автоматизованого та ефективного підходу до тестування програм з використанням глибоких нейронних мереж. Вона сприяє зниженню залежності від ручного створення тестових сценаріїв та покращує здатність виявляти складні дефекти програм, зокрема ті, які важко виявити за допомогою традиційних методів тестування.

Синергетична методологія дозволяє забезпечити більшу покриття програмного коду шляхом генерації різноманітних тестових вхідних даних, які сприяють виявленню різних шляхів виконання програми та потенційних проблемних областей. Крім того, вона спрощує процес валідації тестових результатів шляхом використання тестових оракулів, забезпечуючи автоматичне порівняння отриманих результатів з очікуваними.

Синергетична методологія також може забезпечити можливість автоматизованої побудови моделей глибоких нейронних мереж для тестування програм. Це дозволяє знизити витрати на тренування та підготовку моделей, а також спростити процес їхнього використання у тестуванні.

Крім того, синергетична методологія може сприяти покращенню швидкості тестування та зменшенню часу, потрібного для виявлення дефектів. Це досягається шляхом оптимізації генерації тестових даних та ефективного використання обчислювальних ресурсів для оцінки адекватності тесту та генерації тестових оракулів.

Узагалі, синергетична методологія глибокого тестування програм надає потужний та інноваційний підхід до автоматизованого тестування програм з використанням глибоких нейронних мереж.

Синергетична методологія базується на підході глибоких нейронних мереж, але вона вирішує деякі проблеми, пов'язані з тестуванням програм, які можуть бути недостатньо вирішені традиційними методами. Методологія складається з двох головних компонентів: генерації тестових оракулів та генерації тестових вхідних даних.

Перша компонента полягає у використанні глибоких нейронних мереж для генерації тестових оракулів, які дозволяють автоматично створювати правильні результати відповідно до вхідних даних. Для цього нейронна мережа навчається на вхідних даних, що містять правильні результати. Після тренування мережа може бути використана для генерації тестових оракулів для нових вхідних даних.

Друга компонента полягає у використанні глибоких нейронних мереж для генерації тестових вхідних даних. Ця компонента дозволяє генерувати нові вхідні дані, які покращують покриття коду і можуть виявити нові помилки програми. Для цього нейронна мережа навчається на вхідних даних та їх відповідних результатів, а потім може генерувати нові вхідні дані, які збільшують покриття коду та можуть виявити нові помилки.

Синергетична методологія може вирішувати проблему нестачі ефективного тестування програм шляхом автоматизації процесу створення тестових оракулів та тестових вхідних даних. Вона дозволяє покращити якість тестування, забезпечувати більш повне покриття коду та виявляти більше помилок програми.

4.2. Опис алгоритму та процесу застосування методології

Алгоритм включає в себе дві основні компоненти: генерацію тестових оракулів та генерацію тестових вхідних даних.

Генерація тестових оракулів передбачає використання глибоких нейронних мереж для автоматичної генерації правильних результатів відповідно до вхідних даних. Нейронна мережа навчається на вхідних даних, які містять правильні результати, і потім може бути використана для генерації тестових оракулів для нових вхідних даних. Ця компонента допомагає покращити ефективність тестування шляхом автоматичного створення правильних результатів.

Генерація тестових вхідних даних використовує глибокі нейронні мережі для генерації нових вхідних даних, які покращують покриття коду та можуть виявити нові помилки програми. Нейронна мережа навчається на вхідних даних та їх відповідних результатів, і після навчання може генерувати нові вхідні дані, що збільшують покриття коду та можуть виявити нові помилки. Ця компонента дозволяє розширити множину вхідних даних для більш повного тестування програми.

Алгоритм синергетичної методології поєднує ці дві компоненти з метою досягнення високої ефективності та покриття в процесі тестування програм. На основі результатів генерації тестових оракулів і тестових вхідних даних проводиться оцінка адекватності тесту. Застосовуються метрики оцінки ефективності тестування, такі як точність, чутливість, специфічність тощо, для порівняння прогнозованих результатів зі зазначеними правильними результатами.

Процес застосування синергетичної методології включає наступні кроки:

1. Ініціалізація: Встановлення початкових значень параметрів та завантаження глибоких нейронних мереж для генерації тестових оракулів та генерації тестових вхідних даних.

2. Генерація тестових оракулів: Обрання вхідних даних з тестового набору даних та використання моделі генерації тестових оракулів для отримання прогнозованих результатів. Порівняння прогнозованих результатів зі зазначеними правильними результатами.

3. Оцінка адекватності тесту: Використання метрик оцінки ефективності тестування для оцінки адекватності прогнозованих результатів. Порівняння прогнозованих результатів зі зазначеними правильними результатами та вимірювання показників, таких як точність, чутливість, специфічність тощо.

4. Генерація тестових вхідних даних: Вибір невикористаних або критичних вхідних даних для покращення покриття коду. Використання моделі генерації тестових вхідних даних для генерації нових вхідних даних, які покращують покриття коду та можуть виявити нові помилки.

5. Перевірка адекватності тесту: Застосування нових вхідних даних на тестову модель та отримання прогнозованих результатів.

Після оцінки адекватності нових тестових даних і отримання прогнозованих результатів, проводиться аналіз та інтерпретація результатів тестування. Цей аналіз допомагає виявити потенційні проблеми, помилки та недоліки програмного забезпечення, які можуть бути виявлені під час виконання синергетичної методології.

За результатами аналізу можуть бути прийняті наступні рішення:

1. Виправлення помилок: Якщо виявлені помилки або недоліки програми, команда розробників може приступити до їх виправлення. Це може включати внесення змін до вихідного коду, встановлення патчів або усунення інших проблем, що були виявлені.

2. Покращення тестового покриття: Якщо під час тестування виявлено недостатнє покриття коду або областей програми, можуть бути розроблені

додаткові тестові сценарії або вихідні дані для покращення тестового покриття. Це може допомогти виявити додаткові помилки або проблеми, які раніше не були виявлені.

3. Оптимізація методології: На основі результатів тестування і аналізу можуть бути проведені покращення самої синергетичної методології. Це може включати оптимізацію параметрів глибоких нейронних мереж, вдосконалення алгоритмів генерації тестових оракулів та вхідних даних або внесення змін до процесу оцінки адекватності тесту.

4. Перевірка вимог та специфікацій: Результати тестування можуть бути порівняні з вихідними вимогами та специфікаціями програми для переконання у відповідності до них та виявлення відхилень. Якщо виявлені відхилення від вимог або специфікацій, можуть бути зроблені відповідні корективи, такі як виправлення документації, зміна функціональності програми або внесення інших змін для забезпечення відповідності вимогам та специфікаціям.

5. Звітність та документація: Усі кроки та результати, отримані під час застосування синергетичної методології, повинні бути добре задокументовані. Це включає збереження вхідних даних, результатів тестування, аналізу та прийнятих рішень. Ця документація може бути використана як засіб звітування, а також як посібник для майбутніх проектів та вдосконалення методології.

В результаті впровадження синергетичної методології можна досягти покращення процесу тестування програмного забезпечення, виявлення більшої кількості помилок та недоліків, покращення покриття коду та забезпечення більшої відповідності вимогам та специфікаціям. Використання глибоких нейронних мереж для генерації тестових оракулів та вхідних даних вносить нові можливості у процес тестування, допомагає автоматизувати його та зменшити залежність від ручного написання тестових сценаріїв.

Всі ці деталі та вказівки спрямовані на те, щоб забезпечити успішне впровадження синергетичної методології у процесі розробки програмного забезпечення. Розуміння алгоритму та процесу застосування цієї методології

дозволить командам розробників та тестувальників ефективно використовувати її переваги для покращення якості та надійності програмного продукту.

4.3. Інтеграція найкращих практик з розглянутих методологій

Синергетична методологія прагне злити усі переваги та ефективність методологій, що були досліджені, створюючи цілісний та потужний підхід до автоматизованого тестування програм з використанням глибоких нейронних мереж.

Одним з ключових етапів інтеграції є аналіз розглянутих методологій з метою ідентифікації їх найсильніших сторін. Ми зосереджуємося на трьох основних методологіях, а саме: тестування DNN з генерацією тестових оракулів, оцінка адекватності тестування DNN та генерація тестових вхідних даних для DNN. Ці методології були обстежені та досліджені для з'ясування їх переваг і обмежень.

Далі, на основі отриманих результатів, ми розробили нову методологію, яка поєднує найкращі аспекти з цих розглянутих методологій. При створенні синергетичної методології ми виявили, що генерація тестових оракулів та генерація тестових вхідних даних взаємодіють між собою, утворюючи потужний симбіоз.

Спочатку, ми використовуємо глибокі нейронні мережі для навчання на вхідних даних, що містять правильні результати, з метою генерації тестових оракулів. Ці тестові оракули використовуються для перевірки правильності робот функціонування навчених глибоких нейронних мереж. Після тренування мережа може автоматично генерувати правильні результати відповідно до нових вхідних даних, що дозволяє ефективно виконувати тестування програм.

Крім того, використовуючи глибокі нейронні мережі, ми також забезпечуємо генерацію тестових вхідних даних. Навчання нейронної мережі на

вхідних даних та їх відповідних результатів дозволяє створювати нові вхідні дані, які розширюють покриття коду та можуть виявити нові помилки програми. Цей підхід підвищує різноманітність тестових вхідних даних і дозволяє виявляти складні тестові сценарії, які можуть бути пропущені іншими методологіями.

Процес застосування синергетичної методології передбачає ітеративний підхід. Спочатку, навчання глибоких нейронних мереж проводиться на базі існуючих вхідних даних та їх відповідних результатів для генерації тестових оракулів. Потім, використовуючи навчену мережу, генеруються нові вхідні дані, що покращують покриття коду та виявляють нові помилки програми. Знову проводиться тренування мережі на розширеному наборі даних, що сприяє поліпшенню її здатностей до генерації тестових оракулів та вхідних даних.

Описаний процес взаємодії компонентів синергетичної методології дозволяє досягти сильної взаємодії між генерацією тестових оракулів та генерацією тестових вхідних даних, створюючи синергію та підвищуючи ефективність процесу тестування.

4.4. Переваги синергетичної методології

У цьому розділі ми розглянемо аргументи, які підтверджують ефективність та переваги синергетичної методології в контексті тестування програмного забезпечення. Злиття найкращих практик з різних методологій створює потужний синергетичний підхід, який дозволяє досягати високої якості та ефективності у процесі тестування.

1. Покращена якість тестування: Синергетична методологія використовує глибокі нейронні мережі для генерації тестових оракулів та вхідних даних. Це дозволяє отримувати більш точні результати тестування, забезпечуючи

виявлення потенційних проблем, які можуть бути пропущені застосуванням окремих методологій. Відтак, покращується якість та надійність програмного забезпечення.

2. Збільшення швидкості тестування: Синергетична методологія сприяє автоматизації процесу генерації тестових оракулів та вхідних даних. Це дозволяє знизити залежність від ручного тестування та збільшити швидкість тестування програм. Команда розробників може ефективно використовувати свій час та ресурси, забезпечуючи швидку доставку програмного продукту на ринок.

3. Розширення покриття коду: Інтеграція найкращих практик з різних методологій дозволяє розширити покриття коду програми. Глибокі нейронні мережі можуть генерувати нові тестові вхідні дані, що виявляють раніше непомічені вразливості та помилки. Це сприяє покращенню якості та стабільності програмного забезпечення, зменшує ризик виникнення критичних помилок у продукті.

4. Мінімізація людських помилок: Синергетична методологія допомагає зменшити ручну працездатність та залежність від індивідуального досвіду тестувальників. Використання глибоких нейронних мереж для автоматизації процесу генерації тестових оракулів та вхідних даних робить тестування більш об'єктивним та знижує ризик людських помилок, що можуть впливати на результати тестування.

5. Гнучкість та адаптивність: Синергетична методологія є гнучкою та адаптивною до різних проектів та вимог. Вона може легко включати елементи інших методологій, таких як водопадна модель, Agile чи DevOps, в залежності від потреб та характеристик проекту. Це дозволяє командам розробників та тестувальників використовувати синергетичну методологію в різних контекстах, забезпечуючи оптимальні результати та відповідність проектним вимогам.

6. Зниження витрат та ефективне використання ресурсів: Синергетична методологія дозволяє ефективно використовувати ресурси, що призводить до

зниження витрат на тестування. Автоматизація процесу генерації тестових оракулів та вхідних даних за допомогою глибоких нейронних мереж забезпечує ефективне використання часу та зусиль команди. Це дозволяє зберігати ресурси, знижуючи фінансові витрати і водночас підвищуючи якість тестування.

7. Інноваційний підхід до тестування: Синергетична методологія пропонує інноваційний підхід до тестування, що базується на використанні сучасних технологій штучного інтелекту та глибокого навчання. Цей підхід дозволяє виявляти нові типи помилок, оптимізувати процес тестування та забезпечувати створення високоякісного програмного забезпечення.

8. Забезпечення постійного вдосконалення: Синергетична методологія сприяє постійному вдосконаленню як процесу тестування, так і самого продукту. Шляхом інтеграції найкращих практик з розглянутих методологій, вона створює основу для постійного розвитку та оптимізації. Команда розробників та тестувальників отримує можливість вдосконалювати свої процеси, використовуючи найкращі методики та інструменти з різних джерел. Це дозволяє забезпечити постійне підвищення якості продукту та впровадження нововведень.

9. Швидкість та ефективність: Синергетична методологія відзначається високою швидкістю та ефективністю процесу тестування. Завдяки автоматизації генерації тестових оракулів та вхідних даних, а також використанню інструментів штучного інтелекту, здійснення тестування стає більш автоматизованим та продуктивним. Команда може швидко генерувати тестові дані, проводити тестування та отримувати швидкі результати, що сприяє зниженню часу розробки та випуску продукту на ринок.

10. Покращення співпраці та комунікації: Синергетична методологія сприяє покращенню співпраці та комунікації між членами команди розробників та тестувальників. Інтеграція різних методологій створює спільний мовник та розуміння, що полегшує обмін інформацією та спільну роботу. Крім того, використання автоматизованих інструментів сприяє покращенню співпраці та

синхронізації робочих процесів, зменшуючи можливість помилок та недорозумінь.

Всі ці переваги синергетичної методології сприяють покращенню якості, продуктивності та ефективності процесу тестування. Вона створює основу для забезпечення високої якості продукту, зменшення ризиків та забезпечення успішності проекту. Синергетична методологія є інноваційним інструментом, який дозволяє команді розробників та тестувальників досягати високих результатів у сфері розробки програмного забезпечення.

Синергетична методологія надає численні переваги та важливість у сфері тестування програмного забезпечення. Вона поєднує найкращі аспекти різних методологій, забезпечуючи покращення якості продукту, зменшення ризиків та ефективне використання ресурсів. Її гнучкість та інноваційний підхід дозволяють застосовувати її в різних проектах та контекстах. Ця методологія виявляється надійним інструментом для досягнення високої якості та успіху в сфері розробки програмного забезпечення.

Загалом, синергетична методологія поєднує найкращі аспекти різних підходів до тестування, сприяючи покращенню якості програмного забезпечення, прискоренню процесу тестування та зниженню ризику помилок. Це робить її потужним інструментом для сучасних команд розробників та тестувальників.

ВИСНОВКИ ДО РОЗДІЛУ 4

У цьому розділі була розглянута синергетична методологія тестування програмного забезпечення, яка представляє собою інноваційний підхід до покращення якості та ефективності процесу розробки. Проведений огляд основних принципів та ідей, що лежать в основі методології, показав, що вона поєднує в собі силу автоматизації, глибокого навчання та аналізу даних.

Був наданий детальний опис алгоритму та процесу застосування синергетичної методології. Цей алгоритм передбачає використання генерації тестових оракулів та вхідних даних за допомогою глибоких нейронних мереж, що покращує покриття коду та виявлення помилок. Процес застосування методології є ітеративним та спрямованим на постійне вдосконалення як самого процесу тестування, так і продукту.

Також було пояснено, як синергетична методологія об'єднує найкращі сторони з попередніх методологій. Інтеграція найкращих практик з інших методологій дозволяє створити цілісний підхід, який поєднує синергію різних методів для досягнення високих результатів.

Завершально, була надана аргументація ефективності та переваг синергетичної методології. Її впровадження може значно покращити якість продукту, зменшити час розробки та сприяти злагодженому співробітництву між розробниками та тестувальниками.

У цілому, синергетична методологія є потужним інструментом для покращення процесу тестування програмного забезпечення, забезпечуючи поєднання передових підходів та методів. Її ефективність і переваги демонструють потенціал для покращення продуктивності та якості розробки програмного забезпечення..

5. ВИСНОВКИ

Забезпечення стійкої, безпечної та надійної роботи глибинних нейронних мереж (DNN) є одним з головних викликів сучасної системи машинного навчання. У традиційних програмних системах найбільш практичним інструментом для досягнення цих цілей є тестування програмного забезпечення.

Оскільки тестування DNN є важливою задачею, в декількох статтях були запропоновані методи тестування та критерії адекватності для тестування DNN. У даній роботі розглядається три різні техніки тестування. У першій техніці використовується тестування мульти-реалізації для створення тестового оракула. У другому експерименті порівнюються дві метрики адекватності щодо їх ефективності у виявленні адверсних прикладів. У останньому експерименті застосовуються три різні техніки генерації тестів та порівнюється їх продуктивність, якщо згенеровані тестові дані використовуються для повторного навчання моделей.

Перший експеримент запропонував підхід множинного тестування реалізацій моделі DNN з назвою Variational Auto Encoder. Оцінки на VAE свідчать, що більшість голосів від багатьох реалізацій, що були протестовані, є адекватною заміною для тестового оракулу. Другий експеримент порівнює дві головні категорії технік тестування, які допомагають DNN стати більш стійкими, зокрема до адверсарних прикладів. Я порівнюю дві провідні метрика адекватності (Surprise Adequacy та DeepMutation) з категорій покриття та мутаційного тестування DNN щодо їх ефективності у виявленні адверсарних прикладів. У останньому експерименті я застосував три різні техніки генерації тестів для оцінки DNN-моделей, використовуваних у задачах вбудовування коду.

Перший експеримент показує, що при пороговому значенні вимірювання (FID) 120, з 10 реалізацій було три, що показали відмінний результат. Порівнюючи їхні реалізації вручну, було виявлено, що дві реалізації не могли виробляти хороші результати, оскільки на початку навчання до вхідного зображення додавалась шумова складова. Інша реалізація використовувала маркування для вхідного зображення, але, оскільки наші вхідні дані не були одним числом, маркування не допомагало генераторам виробляти кращі зображення.

Другий експеримент показує, що обидва показники адекватності з категорій вимірювання охоплення та мутаційного тестування DNN можуть бути дуже ефективними. Крім того, вони вказують, що їх продуктивність дуже чутлива до значень гіперпараметрів, і тому налагодження є важливим для інструментів тестування DNN.

Нарешті, результати останнього експерименту свідчать про те, що повторне навчання моделей за допомогою адверсарних прикладів може покращити продуктивність моделі до 18% в порівнянні з вихідним станом мистецтва.

В цілому, результати проведених експериментів з трьома методологіями тестування виявили їхні обмеження та недоліки, що ставить під сумнів їхню повноту та ефективність в контексті сучасних вимог до якості програмного забезпечення.

З метою подолання цих обмежень і покращення процесу тестування, була запропонована синергетична методологія, яка базується на інтеграції найкращих практик з розглянутих методологій. Цей інноваційний підхід дозволяє поєднати сили автоматизації, глибокого навчання та аналізу даних, забезпечуючи високу якість та ефективність процесу розробки програмного забезпечення.

У цій дисертації було детально описано алгоритм та процес застосування синергетичної методології, розкрито його компоненти та зв'язки між ними. Було показано, як генерація тестових оракулів та тестових вхідних даних за

допомогою глибоких нейронних мереж сприяють покращенню покриття коду та виявленню помилок.

Також було обгрунтовано ефективність та переваги синергетичної методології, вказано на позитивний вплив її впровадження на якість продукту, скорочення часу розробки та підвищення співробітництва між розробниками та тестувальниками.

У підсумку, запропонована методологія посприяє у розвитку тестування програмного забезпечення шляхом розробки та впровадження синергетичної методології. Запропоновані покращення та інноваційний підхід виявили свою ефективність у вирішенні проблем, пов'язаних з обмеженнями та недоліками існуючих методологій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Christian Robert. Машинне навчання з ймовірнісною перспективою 2014. — 143с.

2. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. Проблема оракула в тестуванні програмного забезпечення: огляд досліджень. IEEE Transactions on Software Engineering. — 2014. — 507с.

3. Heaton, J. Гудфеллоу І., Бенджіо Й., та Курвіль А. Глибинне навчання. — 2018. — 97с.

4. Murphy, C., Kaiser, G. E., & Hu, L. Властивості застосувань машинного навчання для використання у метаморфному тестуванні. — 2008. — 69с.

5. Zhang, J.M., Harman, M., Ma, L., & Liu, Y. Тестування машинного навчання: огляд, ландшафти та перспективи. IEEE Transactions on Software Engineering. — 2020. — 5с.

6. Doersch, C. Посібник з варіаційних автокодерів. arXiv preprint arXiv:1606.05908. — 2016.

7. Harel, O., & Zhou, X.H. Множинне відтворення: огляд теорії, реалізації та програмного забезпечення. Statistics in medicine. — 2007. — 26(16), 3057-3077с.

8. Xia Li та Lingming Zhang. Transforming programs and tests in tandem for fault localization. Proceedings of the ACM on Programming Languages. — 2017 1(OOPSLA):1–30с.

9. Kexin Pei, Yinzhi Cao, Junfeng Yang та Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. У збірнику праць 26-го Симпозіуму з принципів операційних систем, сторінки. — 2017 — 1–18с.

10. Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu та інші. Deepgauge: Multi-granularity testing criteria for deep learning systems. У збірнику праць 33-ї міжнародної

конференції з автоматичного програмного забезпечення ACM/IEEE. — 2018 — 120–131с.

11. Jinhan Kim, Robert Feldt та Shin Yoo. Guiding deep learning system testing using surprise adequacy. У 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). — IEEE, 2019 — 1039–1049с.

12. Timothy A Budd та Ajei S Gopal. Program testing by specification mutation. Computer languages. — 1985. — 10(1):63–73с.

13. Arnaud Gotlieb and Bernard Botella. Automated metamorphic testing. In Proceedings 27th Annual International Computer Software and Applications Conference. — COMPAC 2003. — 34с.

14. Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In Advances in neural information processing systems. — 2017. — 662с.

15. Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In 2016 IEEE European symposium on security and privacy (EuroS&P). — 2016. — 372с.

16. Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL) — 2019. — 1– 29с.

17. Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. — 2018. — 36с.

18. Saidur Rahman, Emilio River, Foutse Khomh, Yann Gal Guhneuc, and Bernd Lehnert. 2019. Machine learning software engineering in practice: An industrial case study. — 2019 — 1–21с.

19. Mona Rahimi, Jin L.C. Guo, Sahar Kokaly, and Marsha Chechik. 2019. Toward Requirements Specification for Machine-Learned Components. In 2019 IEEE 27th International Requirements Engineering Conference Workshops (REW). IEEE, 241–244с.

20. Mirko Perkusich, Lenardo Chaves e Silva, Alexandre Costa, Felipe Ramos, Renata Saraiva, Arthur Freire, Ednaldo Dilorenzo, Emanuel Dantas, Danilo Santos, Kyller Gorgônio, Hyggo Almeida, and Angelo Perkusich. 2020. Intelligent software engineering in the context of agile software development: A systematic literature review. Information and Software Technology — 2020 — 119c.

21. Shin Nakajima. 2019. Quality Evaluation Assurance Levels for Deep Neural Networks Software. In 2019 International Conference on Technologies and Applications of Artificial Intelligence (TAAI). IEEE

22. Kayur Patel. 2010. Lowering the barrier to applying machine learning. In Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology - UIST '10. ACM Press, 355–358c.

23. Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski. 2018. Sim-ATAV. In Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week). ACM, 283–284c.

24. Marisa Vasconcelos, Heloisa Candello, Claudio Pinhanez, and Thiago dos Santos. 2017. Bottester. In Proceedings of the XVI Brazilian Symposium on Human Factors in Computing Systems. ACM, 1–4c.

25. Andreas Vogelsang and Markus Borg. 2019. Requirements Engineering for Machine Learning: Perspectives from Data Scientists. In 2019 IEEE 27th International Requirements Engineering Conference Workshops (REW). IEEE, 245–251c.

26. Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 158–168c.

27. Luca Pulina and Armando Tacchella. 2010. An abstraction-refinement approach to verification of artificial neural networks. CEUR Workshop Proceedings 616 (2010), 243–257c.

28. Elizamary Nascimento, Anh Nguyen-Duc, Ingrid Sundbø, and Tayana Conte. 2020. Software engineering for artificial intelligence and machine learning software: A systematic literature review. arXiv preprint arXiv:2011.03751 (2020).

29. M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, “Geo-metric deep learning: Going beyond euclidean data,”IEEE Signal ProcessingMagazine, vol. 34, pp. 18–42, 2017.

30. K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,”CoRR, vol. abs/1409.1556, 2014.

31. S. Kwak, S. Hong, and B. Han, “Weakly supervised semantic segmentation usingsuperpixel pooling network,” inAAAI, 2017.

32. Park, J., Spetka, E., Rasheed, H., Ratazzi, P., and Han, K.Near-real-time cloud auditing for rapid response. InAdvanced Infor-mation Networking and Applications Workshops (WAINA), 2012 26thInternational Conference on(march 2012), IEEE, 1252 –1257c.

33. Mather, T., Kumaraswamy, S., and Latif, S.Cloud security andprivacy: an enterprise perspective on risks and compliance. O’ReillyMedia, Inc., 2009

34. Cochran, M., and Witman, P.Governance and service level agree-ment issues in a cloud computing environment.Journal of InformationTechnology Management 22, 2 (2011), 41–55.c.

35. M. Kydyraliev. CVE-2011-3923: Yet another Struts2 Remote CodeExecution.o0o Security Team blog, 2016.

36. J. Dahse. Multiple vulnerabilities in Apache Struts2 and property orientedprogramming with Java, 2016.

37. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtimeverification. InVerification, Model Checking, and Abstract Interpretation, 2004, 44–57c..

38. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detectionsystems. InProceedings of the 9th ACM Conference on Computer and Com-munications Security, CCS ’02, pages 255–264, New York, NY, USA, 2002.ACM

39. W. M. P. van der Aalst and A. K. A. de Medeiros. Process mining andsecurity: Detecting anomalous process executions and checking process conformance.Electron. Notes Theor. Comput. Sci., 121:3–21, 2005.

40. D. Anderson, T. F. Lunt, H. Javitz, A. Tamaru, and A. Valdes. Detect-ing unusual program behavior using the statistical component of the next-generation intrusion detection expert system (nides). Technical report, SRIInternational. Computer Science Laboratory, 1995

41. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

42. M. R. Shaheen and L. d. Bousquet. Is Depth of Inheritance Tree a Good Cost Prediction for Branch Coverage Testing? In 2009 First International Conference on Advances in System Testing and Validation Lifecycle (VALID), 42–47c.. IEEE, 2009.

43. K. Lakhotia, M. Harman, and H. Gross. Austin: An open source tool for search based software testing of c programs. Information and Software Technology, 55(1):112–125, 2013.

44. A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA ’04, pages 108– 118, New York, NY, USA, 2004. ACM.

45. S. Benlarbi, K. E. Emam, N. Goel, and S. Rai. Thresholds for objectoriented measures. In Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000, 24–38c.

46. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. IEEE Trans. Softw. Eng., 20(6):476–493, 1994.

47. S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab., 22(2):67–120, Mar. 2012.

48. B. Hailpern and P. Santhanam. Software debugging, testing, and verification. IBM Syst. J., 41(1):4–12, Jan. 2002.

49. J. Kindervag. Build security into your network's dna: The zero trust network architecture. Forrester Research Inc, 1–26c. 2010.

50. D. Kreutz et al. Software-defined networking: A comprehensive survey. Proc. of the IEEE, 103(1):14–76, 2015.

51. Peter Mell, James Shook, and Richard Harang. Measuring and improving the effectiveness of defense-in-depth postures. In Proceedings of the 2nd Annual Industrial Control System Security Workshop, pages 15–22, 2016.