МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«Інтелектуальна система розпізнавання шкідливого програмного забезпечення»

Завідувач випускаючої кафедри — **Довбиш А. С.**

Керівник роботи — **Москаленко В. В.**

Студента групи ІН.м – 91н — **Коваленка А. О.**

СУМИ 2021

Сумський державний університет

(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*

Спеціальність *«Інформатика»*

Затверджую:
зав.кафедрою _____
_____
"_____" _____
20___р.

ЗАВДАННЯ

НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

Коваленку Антону Олександровичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інтелектуальна система розпізнавання шкідливого програмного забезпечення*

затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі студентом закінченого проекту (роботи) _____

3. Вхідні данні до проекту (роботи) _____

_____

_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Визначення мети, об'єкту та предмету досліджень. 2) Огляд літератури, вибір методу рішення задачі 3) Побудова моделі розпізнавання шкідливого ПЗ 4) Пошук та формування навчальних та тестових даних для моделі 5) Програмна реалізація та оцінювання ефективності створеної системи розпізнавання

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_____
_____
_____
_____
_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____

(підпис)

Завдання прийняв до виконання _____

(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	*Аналіз проблеми. Постановка мети, об'єкту та предмету дослідження*		
2.	*Огляд літератури та аналіз існуючих рішень*		
3.	*Вибір методу класифікації для інтелектуальної системи*		
4.	*Розробка інформаційної технології розпізнавання, проведення навчання моделі та аналіз результатів*		
5.	*Оформлення пояснювальної записки до дипломної роботи*		

Студент – дипломник _____

(підпис)

Керівник проекту _____

(підпис)

РЕФЕРАТ

Записка: 68 стор., 19 рис., 3 додатки, 65 джерел.

Об'єкт дослідження — шкідливе програмне забезпечення.

Мета роботи — дослідження і реалізація інтелектуальної моделі розпізнавання та класифікації шкідливого програмного забезпечення використовуючи згорткові нейронні мережі з кватерніонними компонентами.

Методи дослідження — штучні нейронні мережі, аналіз файлів шкідливого програмного забезпечення, кватерніонні згорткові та повнозв'язні шари, стохастичний градієнтний спуск, регуляризація нейромережі виключенням, максимізаційне агрегування, оптимізатор Adam, категорійна крос-ентропія як функція втрат моделі

Результати — реалізована інтелектуальна система розпізнавання на основі згорткових нейронних мереж з кватерніонними компонентами, проведений вибір та дослідження методу перетворення файлів шкідливого програмного забезпечення у графічні зображення. Була сформована навчальна вибірка даних для класифікації та проведене тренування моделі. Результати роботи системи показали високу точність класифікації шкідливих зразків ПЗ. Програмна реалізація здійснена з використанням мови Python, бібліотек TensorFlow та Keras.

МАШИННЕ НАВЧАННЯ, ЗГОРТКОВА НЕЙРОННА МЕРЕЖА,
ШКІДЛИВЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, PYTHON,
РОЗПІЗНАВАННЯ ОБРАЗІВ, TENSORFLOW, KERAS,
КВАТЕРНІОН.

ЗМІСТ

ВСТУП

З ростом кількості накопиченої корисної інформації у різних галузях людської діяльності та паралельним розвитком інформаційних систем стало можливим перетворення такої інформації на знання у предметній області з використанням інтелектуального аналізу даних. Все більш актуальною областю застосування таких технологій стає виявлення та класифікація шкідливого програмного забезпечення у зв'язку з його все більшим поширенням в сучасних інформаційних системах. Дослідження щодо вибору оптимальних моделей для системи інтелектуального аналізу даних дають поштовх для створення нових або покращення вже існуючих систем, сприяють практичному застосуванню розробок у області захисту даних та мають позитивний ефект щодо впевненості кінцевих користувачів у надійності інформаційних систем, котрі вони використовують.

Важливою частиною досліджень у області інтелектуального аналізу даних є вибір методів котрі забезпечать оптимальну точність побудованої системи у процесі екзамену. На шляху до побудови якісної системи існує декілька проблем, котрі потребують рішення: дослідження характерних даних які можна використовувати для подальшої побудови моделей аналізу, побудову та порівняння самих моделей ознакового опису, вибір оптимальних методів навчання та розпізнавання, програмну реалізацію, оцінювання ефективності та валідацію фінальної системи аналізу даних.

Практичний аспект створення майбутньої системи аналізу даних полягає у можливості подальшої її інтеграції у діяльність підприємств, критичним для яких є надійне функціонування інформаційних систем та центрів обробки даних. Як відомо, шкідлива активність у таких системах може призвести до значних матеріальних та нематеріальних збитків для її оператора. Зменшення або повна ліквідація ризиків, пов'язаних з впливом шкідливого програмного забезпечення дасть можливості для оптимізації діяльності такого

підприємства. Тому, підвищену увагу при реалізації системи розпізнавання слід звертати на оперативність її роботи, високу функціональну ефективність.

Важливим аспектом функціонування сучасних інформаційних систем є довіра користувачів щодо цілісності та безпечності даних, котрі зберігаються в ній. Безумовно, кожен користувач бажає щоб його дані були доступні у довільний момент часу, а персональна інформація залишались конфіденційними за будь-яких умов. Для забезпечення цього інформаційна система повинна мати відповідні експлуатаційні характеристики і надійний захист від шкідливого програмного забезпечення. Тому, інтеграція інтелектуальних систем виявлення з високими функціональними показниками буде мати позитивний вплив на впевненість клієнтів у надійності інформаційних систем, котрими вони користуються.

Таким чином створення системи розпізнавання шкідливого програмного забезпечення передбачає вирішення декількох важливих проблем: проведення досліджень по вибору оптимальних методів та моделей на яких буде базуватись майбутня система, виконання якісної програмної реалізації, фінальної верифікації та тестування створеного програмного забезпечення. Ефективне розв’язання вищезазначених завдань сприятиме внесенню позитивних змін у науковому, практичному та соціальному аспектах, а саме – отриманню нових результатів у області інтелектуального аналізу даних, введенню продукту в експлуатацію, підвищенню безпечності та надійності інформаційних систем для кінцевих користувачів.

1. АНАЛІЗ ПРОБЛЕМИ РОЗПІЗНАВАННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Сучасний стан та тенденції розвитку технологій аналізу шкідливої активності в інформаційній системі

З кожним роком проблема шкідливого програмного забезпечення та загроз які вони вносять стає все більш актуальною. Шкідливе програмне забезпечення зазвичай потрапляє до інформаційної системи без відома користувачів і може порушити нормальний процес її експлуатації, завдаючи при цьому матеріальних і нематеріальних збитків. Згідно зі звітом організації AV-TEST, в першому кварталі 2020 року було зареєстровано 43 мільйони нових зразків шкідливого програмного забезпечення, при збереженні темпів розробки до кінця 2020 року може з'явитися до 160 мільйонів нових зразків. Таким чином загальна кількість зросте до 700 мільйонів. Так як зловмисники використовують методи автоматичної масової розробки шкідливого програмного забезпечення то поточна швидкість появи нових зразків є 4.2 на секунду [1].

Через велику кількість та поширення різних видів шкідливого програмного забезпечення на сьогодні неможливо обробляти інформацію про нього вручну, тому пріоритетним напрямком вирішення проблеми є створення інтелектуальних систем.

Наведемо основні типи шкідливого програмного забезпечення та їх короткий опис:

- Вірус: шкідлива комп'ютерна програма, котра розповсюджується шляхом вбудовування власного коду у код інших програм. Вірус може поширюватись від однієї програми до інших, також від одного комп'ютера до інших [3].
- Хробак (англ. worm): програма, що розповсюджується всередині системи або мережі та руйнує її файли та дані, можливе знаходження шкідливого коду у власному контейнері. Хробаки

можуть проводити шифрування файлів та розсилати шкідливі листи e-mail [2].

- Троянська програма: різновид шкідливого програмного забезпечення, котра виконує небажані функції, маскуючи себе під корисну програму [3]. Троянські програми зазвичай не розповсюджуються шляхом вбудовування у код інших програм, а поширюються з використанням соціальної інженерії. Зловмисники можуть отримати контроль за інфікованим комп'ютером та заволодіти персональними даними.
- Rootkit: спеціальне програмне забезпечення, яке використовується для несанкціонованого доступу до операційної системи і сприяє розповсюдженню та потраплянню шкідливого програмного забезпечення до інфікованої системи. Часто Rootkit використовується для підміни системних утиліт на шкідливі та приховання слідів їх діяльності [5].
- Ransomware: програма – вимагач, котра блокує доступ до користувацьких даних на комп’ютері або в мережі шляхом шифрування файлів і подальшого повідомлення про необхідну матеріальну винагороду для зняття блокування [4].
- Spyware: шпигунське програмне забезпечення, котре встановлюється без відома користувача і використовується для збору персональної інформації про нього. Отримані дані надсилаються до зловмисників [6].
- Botnet: шкідливе програмне забезпечення, котре здійснює контроль над групою інфікованих пристроїв. Використовується для виконання DDoS атак та розсилання спам повідомлень e-mail. Зазвичай користувач не підозрює про те, що його пристрій є носієм такого програмного забезпечення [9].

- Adware: шкідливе програмне забезпечення, котре спрямоване на постійний показ реклами на інфікованому комп'ютері. Часто, adware розповсюджується разом з вільним програмним забезпеченням [7].

1.2. Моделі та методи інтелектуального аналізу даних

Для боротьби зі шкідливим програмним забезпеченням існують різні методи його виявлення.

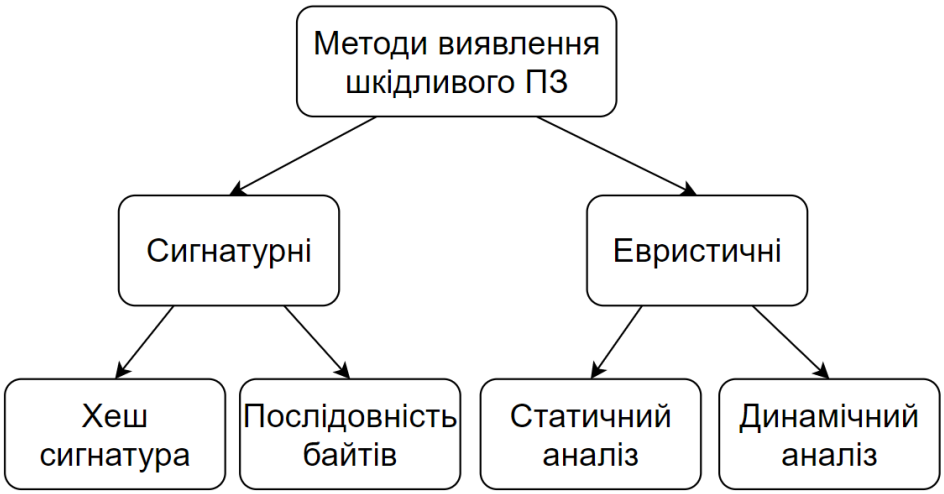

Рис 1.2.1 — Методи виявлення шкідливого ПЗ

Антивірусне програмне забезпечення зазвичай використовує сигнатурний метод для виявлення шкідливого програмного забезпечення. Такий метод полягає у виділенні характерних ознак з програмного коду вірусу і використання його для пошуку схожих зразків програмного забезпечення [8]. Характерними ознаками може бути послідовність байтів або хеш окремого файлу [10], це забезпечує розпізнавання з мінімальною помилкою першого роду. Перевагами такого підходу є відносна швидкодія алгоритмів при детектуванні відомого шкідливого програмного забезпечення, хоча істотним недоліком є низька ймовірність виявлення раніше невідомих зразків [13].

Основний підхід при сигнатурному методі полягає у використанні статичного аналізу файлів та програмного коду з метою виділення окремих послідовностей байтів [12].

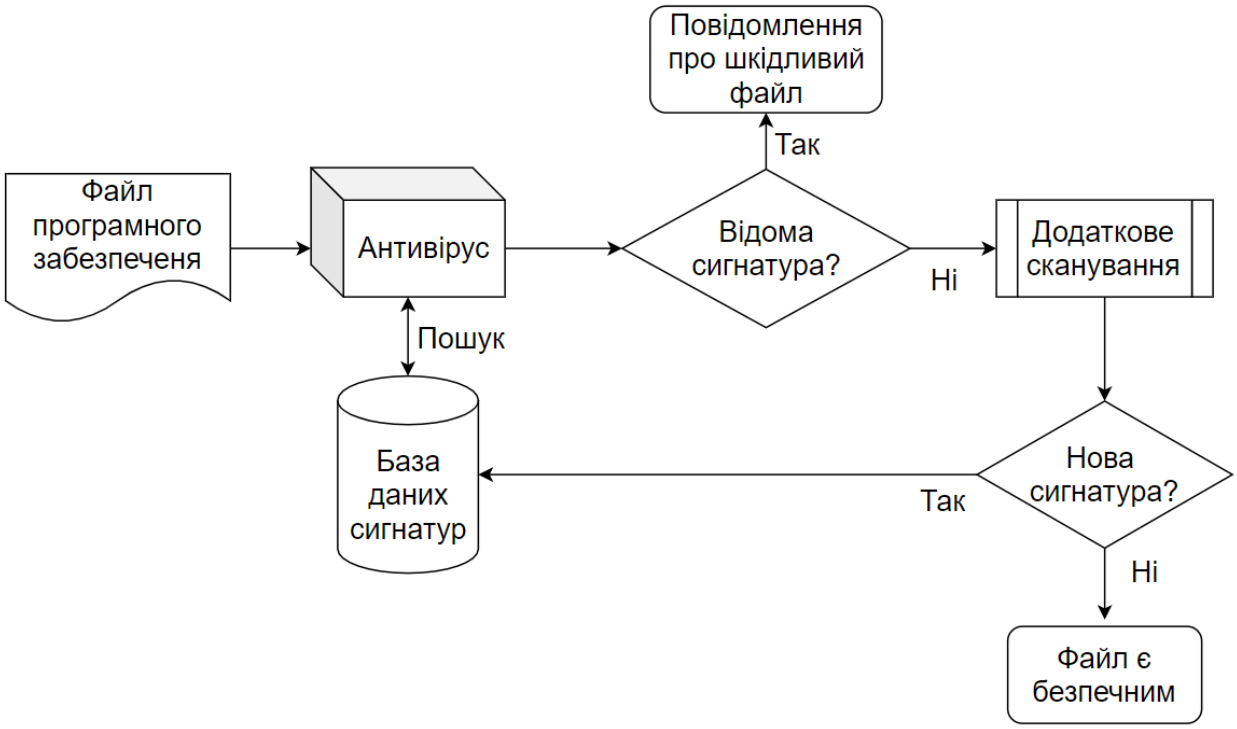

Рисунок 1.2.2 — Алгоритм роботи сигнатурного методу

Перешкодою до розпізнавання шкідливого програмного забезпечення для сигнатурних методів є використання технік обфускації програмного коду щодо вже відомих зразків. Такі техніки включають в себе: вставку "недіючого коду", перейменування регістрів, заміна інструкцій, ін’єкція коду [11]. Наведемо коротке визначення і опис кожної з технік:

- Перейменування регістрів: зміна імен регістрів у програмному коді або заміни їх між собою. Наприклад: перейменування цільового регістру для збереження змінних EAX на регістр EBX у архітектурі x86.
- Заміна інструкцій: при такій техніці, інструкції, що мають схожий функціонал замінюються між собою. Наприклад, заміна інструкцій MOV на інструкцію POP в архітектурі x86.
- Перестановка функцій: проста техніка зміни порядку функцій у файлах програмного коду.
- Вставка "недіючого коду": проста техніка обфускації, при якій здійснюється вставка програмного коду, який не змінює загального функціоналу програми, але змінює сигнатуру. Це може

бути додавання інструкцій NOP (англ. no operation performed), або вставка інструкцій PUSH/POP, котрі працюють з фіктивними змінними.

- Ін'єкція коду: вбудовування коду шкідливого програмного забезпечення до коду інших програм. Для цього проводиться процедура декомпіляції цільової програми і здійснюється процес ін'єкції шкідливого коду в неї [14]. Така техніка обфускації є однією з найбільш складних, що значно зменшує ймовірність розпізнання сигнатурними методами.

Альтернативою сигнатурним методам є евристичні методи, тобто такі, що розпізнають поведінку шкідливого програмного забезпечення. При використанні такого підходу на етапі навчання проводиться аналіз діяльності програмного забезпечення у контрольованому середовищі. Отримані дані в подальшому використовуються на етапі екзамену і в залежності від результату розпізнавання зразку присвоюється статус шкідливого або безпечного [8]. Таким чином, методи, що засновані на аналізі поведінки зразків програмного забезпечення можуть виявляти раніше невідомі загрози, якщо їх поведінка збігається з уже відомими, а також розпізнавати вже відомі зразки, до яких були використані техніки обфускації програмного коду. На противагу, недоліками таких методів є відносно висока обчислювальна складність, тому і відносно тривала процедура виконання аналізу, а також відносно висока помилка першого роду [10]. Детальний аналіз діяльності шкідливого програмного забезпечення, великої кількості отриманих ознак та пошук схожих зразків між собою можуть покращити ймовірність виявлення загроз нульового дня [15], [16]. Для розуміння поведінки шкідливих зразків евристичні методи спираються на методи аналізу даних та машинного навчання, серед яких: метод опорних векторів, градієнтний бустінг, нейронні мережі, баєсів класифікатор, дерева рішень, випадковий ліс та ін.

При аналізі шкідливого програмного забезпечення здійснюється дослідження файлів з метою вивчення їх поведінки, еволюції та визначення можливих цілей для атаки [17]. Загалом існують три основні техніки для аналізу шкідливих зразків: статичний, динамічний та гібридний аналіз. Також набуває популярності аналіз на основі сканування оперативної пам’яті пристроїв.

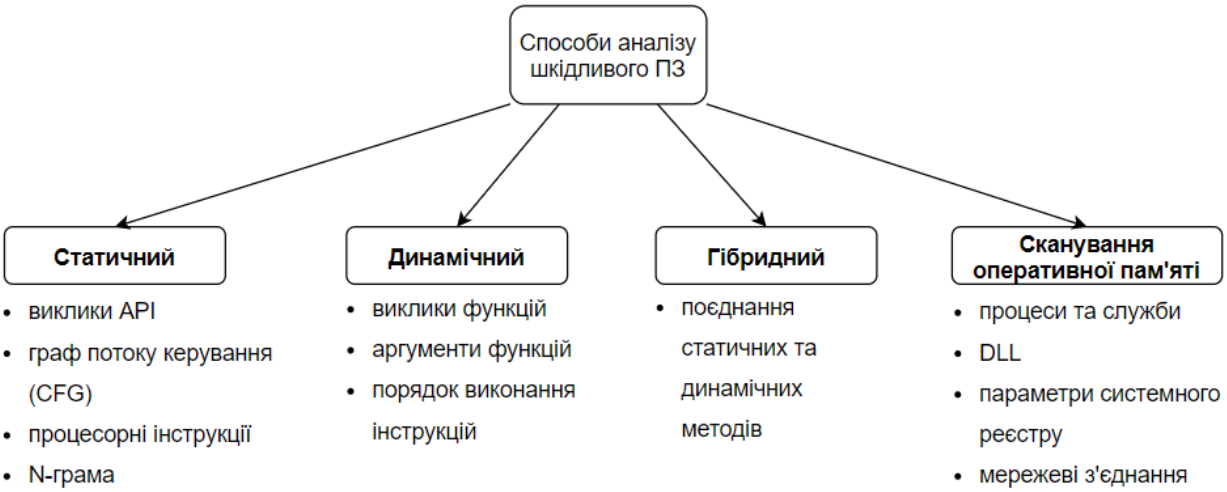

Рисунок 1.2.3 — Методи аналізу шкідливого ПЗ

Розглянемо кожен зі способів аналізу окремо.

Техніка статичного аналізу базується на дослідженні виконуваних (.exe) файлів без їх запуску. Зазвичай автори шкідливого програмного забезпечення використовують упаковщики виконуваних файлів для того щоб уникнути процедуру аналізу [6]. Упаковка шкідливих файлів у більшості випадках супроводжується подальшим шифруванням та видозміною наявного коду для унеможливлення детектування системами, що використовують на сигнатурні методи. Таким чином виконуваний файл майже завжди повинен бути розпакований та декомпільований. Для процедури декомпіляції застосовується такий інструмент як дизасемблер. З його допомогою можливо отримати службову інформацію про шкідливий файл, дослідити програмний код та визначити його характерні ознаки, які можна використати для виявлення шкідливого ПЗ у подальшому. Характерними ознаками можуть

виступати: виклики Windows API, граф потоку керування (англ. control flow graph), процесорні інструкції, байтові N–грами [20].

Майже всі програми використовують виклики Windows API для взаємодії з операційною системою Windows. Наприклад, OpenFile є функцією Windows API, котра міститься в Kernel32.dll і яка створює новий файл або відкриває вже існуючий. Таким чином, виклики API показують поведінку програми і можуть бути використані для виявлення шкідливого ПЗ. Наприклад, послідовність викликів WriteProcessMemory, LoadLibrary та CreateRemoteThread з великою ймовірністю є процедурою ін'єкція шкідливої DLL у процес операційної системи, бо рідко зустрічається у безпечних файлах.

Також ознакою для розпізнавання шкідливих файлів може бути текст в програмному коді шкідливого ПЗ, котрий може бути виведений на екран [6]. Деяка текстова інформація може видати наміри зловмисника, наприклад рядок "This program cannot be run in DOS mode" котрий міститься поза заголовком виконуваного файлу скоріш за все вказує на шкідливий файл котрий є одним з типів троянських програм.

Граф потоку керування (англ. contol flow graph) — направлений граф котрий дає інформацію про виконання програми, у графі вершинами є окремі блоки коду а ребра — переходи між ними. У задачі виявлення шкідливого ПЗ граф потоку керування використовується для вивчення поведінки виконуваних файлів та визначення структури програмного коду [17].

Процесорні інструкції визначають операції, котрі будуть виконані центральним процесором. Інструкції складаються з операційного коду та декількох аргументів та мають такий вигляд: addl %eax %ebx, subl $8 %edi. Операційний код інструкцій може бути використаний як ознака розпізнавання шкідливого ПЗ шляхом проведення частотного аналізу або обчислення схожості між послідовностями кодів.

N–грама це всі послідовності довжини N, що входять у початкову [19]. Наприклад, слово "програмне" містить в собі наступні N–грами довжини 4: "прог", "рогр", "огра", "грам", "амне". Використання N–грам можливе при

аналізі викликів API, процесорних інструкцій та довільних послідовностей, що можуть бути ознаками шкідливого ПЗ.

Окрім вищезазначених ознак, існують ще декілька, які використовуються при статичному аналізі серед них: розмір файлів, довжина функцій, мережеві порти TCP/UDP, IP адреси джерела та отримувача, запити HTTP [17].

Одне з найбільш масштабних досліджень на тему уникнення сигнатурних методів було здійснене вченими Dhilung Kirat та Giovanni Vigna [18]. Був проведений аналіз 2810 шкідливих зразків та отримано 78 унікальних технік, котрі дозволяють шкідливому ПЗ уникнути виявлення сигнатурними методами.

Вченими Ali Hamzeh та Sattar Hashemi був використаний оригінальний підхід, котрий виділяє послідовності інструкцій програмного коду та будує графічні зображення на їх основі. Потім ознаки отриманого зображення обробляються алгоритмом комп'ютерного зору — методом локальних бінарних шаблонів (англ. local binary pattern). На фінальному етапі отримані дані обробляються методами машинного навчання для виявлення шкідливого ПЗ. Така техніка мала точність виявлення на рівні 91,9% [21]. Вчені Bander Ali Saleh Al-rimy та Syed Zainudeen Mohd Shaid також використали схожий підхід для відображення шкідливого ПЗ. Основою стала послідовність викликів API та перетворення їх на зображення. Отримані зображення також використовувались для побудови моделей виявлення [23].

Zahra Salehi та інші [22] виконували побудову моделей на основі послідовності викликів API з виконуваних файлів та використовували отримані послідовності для навчання класифікатора. Як результат було визначено 3 можливі ознаки: "аргументи API", "список викликів API" та "API та список аргументів", всі 3 множини були використані окремо та порівняні. Результат порівняння вказав на перевагу при використанні API та списку аргументів, що мали точність розпізнавання на рівні 98,4% та помилки першого роду 3%.

Також вчені Yixuan Cheng та інші [24] аналізували виклики API з використанням інструменту WinDbg та використали метод опорних векторів для виявлення шкідливого шелл–коду. Незважаючи на доволі малий розмір навчальної вибірки отримана точність становила 94.37%, хоча помилка другого роду становила 44.44%.

Kyoung Soo Han та інші [25] досліджували виклики API з таблиці адрес імпорту (англ. import address table) методами статичного аналізу. Було здійснене порівняння різних послідовностей для виявлення схожості та класифікації різних типів шкідливого ПЗ. Схожість серед одного типу становила близько 40% поряд з помилкою першого роду у 16%.

Таблиця 1 — Характеристики систем виявлення на основі статичних методів

Автор / рік	Ознака для виявлення	Класифікатор	Навчальна вибірка (шкідливі / безпечні зразки)	Точність виявлення	Помилка першого роду
Salehi 2014 [22]	Системні функції, аргументи функцій	DT, NB	826/395	98.4%	3%
Hashemi 2018 [21]	Операційний код інструкцій	KNN	3100/3100	91.9%	-
Santos 2013 [29]	Послідовність операційних кодів	DT, KNN, BN, SVM	1000/1000	97.5%	6%
Cheng 2017 [24]	Послідовність системних функцій	SVM	18/72	94.4%	1.4%

Динамічний аналіз базується на дослідженні поведінки шкідливого ПЗ. Процедура виявлення включає в себе виконання та моніторинг файлів у контрольованому середовищі — віртуальній машині, симуляторі або емуляторі [7]. Важливим аспектом при дослідженні методами динамічного аналізу є приховання ознак віртуального середовища, так як деякі зразки шкідливого ПЗ можуть детектувати факт того, що вони виконуються у штучному середовищі та маскувати або зупиняти свою діяльність.

Ефективність методів динамічного аналізу є вищою в порівнянні з статичним аналізом, тому що відсутня необхідність розпаковки, дешифрування та декомпіляції виконуваних файлів для їх дослідження. Ще одною перевагою є можливість розпізнавання раніше невідомого шкідливого ПЗ, поліморфних та зразків до яких були застосовані техніки обфускації. Хоча в порівнянні з методами статичного аналізу динамічні вимагають більше часу на виконання та обчислювальних ресурсів. [6]

Динамічний аналіз передбачає використання різних ознак для дослідження, серед них – виклики функцій, аргументи функцій, послідовності та потік виконання інструкцій. Також дослідники широко використовують і інші ознаки для виявлення шкідливого ПЗ — файлову систему, системний реєстр Windows та мережеві ознаки. [20]

Вчений David Mohaisen та інші вивчали можливість класифікації шкідливого ПЗ на основі методів машинного навчання [30]. Для навчання класифікатора використовувались такі ознаки як системний реєстр, файлова система та мережеві підключення. При наявних даних у 1980 зразків троянських програм була досягнена точність 95%. У подальших роботах була запропонована AMAL – автоматизована система аналізу та класифікації шкідливого ПЗ на основі дослідження поведінки. Така система складається з двох частин, перша аналізує шкідливі зразки з допомогою файлової системи, системного реєстру та мережевої активності, а друга частина виконує класифікацію на основі отриманих даних про активність. Система AMAL

показала точність виявлення 99% на тестовій вибірці з 115000 зразків шкідливого ПЗ. [28]

У своїй роботі Qi Chen та David Bridges [26] вивчали відомий вірус WannaCry на основі системних лог файлів. Текстова інформація аналізувалась на основі показників TF-IDF, були визначені характерні текстові ознаки з великою частотою появи у лог файлах.

Найбільш часто дослідники використовують виклики API у якості індикатора шкідливого ПЗ. У своїй роботі Guanghui Liang та інші [27] використали техніку класифікації на онові перехоплення викликів API та побудови залежності між ними. Отримана інформація використовувалась для обчислення схожості між зразками шкідливого ПЗ та подальшої їх класифікації. Eunjin Kim та інші [31] запропонували метод обробки на основі обробки послідовності API викликів та застосуванні алгоритму множинного вирівнювання послідовностей (MSA). Після отриманих даних про схожі послідовності був здійснений пошук найдовших спільних підпослідовностей. Такий підхід мав точність у 99.8% та нульовою помилкою першого роду.

Hisham Shehata Galal та інші також використали перехоплення викликів API разом з їх аргументами. Потім, схожі за семантикою виклики групувались у послідовності, котрі в подальшому класифікувались з використанням дерева рішень. Найкращу точність, що вдалось досягти становила 97.17% [32]. Chun-I Fan та інші [33] виконували перехоплення API для дослідження функцій, які приховуються шкідливим ПЗ. Були проаналізовані як API користувача так і недокументований системний API. Отримана система, на основі 80 API показала точність виявлення у 95% з використанням дерев рішень на баєсівського класифікатора.

Таблиця 2 — Характеристики систем виявлення на основі динамічного аналізу

Автор /рік	Ознака для виявлення	Класифікатор	Навчальна вибірка (шкідливі / безпечні зразки)	Точність виявлення	Помилка першого роду
Liang / 2016 [27]	виклики API	DT, ANN, SVM	12199/-	91.3%	-
Mohaisen / 2013 [30]	файлова система, системний реєстр, мережа	DT, SVM, KNN	1980/-	95%	5%
Mohaisen / 2015 [28]	файлова система, системний реєстр, мережа	DT, SVM, KNN	115000/-	99%	-
Ki / 2015 [31]	послідовність викликів API	-	23080/-	99.8%	0%
Fan / 2015 [33]	API користувача, системний API	J48, NB, SVM	773/253	95.9%	5%
Galal / 2017 [32]	послідовність викликів API	DT, SVM	2000/2000	97.2%	-

В динамічному аналізі шкідливе ПЗ виконується у ізольованому середовищі щоб уникнути його дії. Тому існує декілька типів таких середовищ: емулятори, дебагери, симулятори та віртуальні машини. Коротко розглянемо кожен тим окремо.

Емулятор використовується для контролю над виконанням шкідливих файлів. Повна емуляція системи має контроль над центральним процесором, файловою системою та іншими ресурсами. Зазвичай емулятори дозволяють досліджувати мережеву активність, оперативну пам'ять, системні функції, процеси та виклики API. Нажаль, більшість шкідливого ПЗ має можливості для виявлення того, що воно виконується у контрольованому середовищі. Для цього воно здійснює активність по визначенню та порівнянні ключових ознак в операційній системі та виконання операцій поза емулятором [36].

Дебагер — ще один тип контрольованого середовища, котрий дозволяє проводити аналіз виконання файлів іноді на рівні процесорних інструкцій. Таке середовище також може бути виявлене досить простими способами. Функції Windows API такі як "IsDebuggerPresent", "OutputDebugString", "CheckRemoteDebuggerPresent" можуть вказувати на присутність дебагера в системі. Ще одною ознакою наявності можуть бути ключі системного реєстру, файли та директорії файлової системи. Ще одною мірою ускладнення роботи дебагера є навмисне використання механізмів виключних ситуацій (англ. exception) тільки у випадку його наявності [35].

Наступним типом контрольованого середовища є симулятор. Це комп'ютерна програма, котра імітує дії користувача без безпосередньої його участі. Основні дії що дозволяє симулятор це перехоплення викликів функцій, API, симуляцію всієї операційної системи та мережевих підключень [36]. Шкідливе ПЗ може протидіяти симуляторам, для цього може бути проведене сканування реєстру, файлів та процесів для визначення його присутності. Також індикатором наявності симулятора може бути час виконання програм, так як він у більшості випадків є більшим ніж той, котрий виміряний при виконанні програм поза контрольованим середовищем. [34]

Найпопулярнішим типом контрольованого середовища є віртуальні машини. Таке програмне забезпечення виконує імітацію окремої операційної системи але є ізольованою від первинної системи, тому програми всередині не можуть взаємодіяти з оригінальною системою. Шкідливе ПЗ також намагається знайти індикатори наявності віртуальних машин у системі засобами сканування системного реєстру, файлової системи запущених процесів. Також можлива перевірка на наявність шляхом виконання спеціальних інструкцій, які можуть бути виконані тільки поза віртуальним середовищем [35]. Встановлення віртуальних машин часто супроводжується встановленням спеціалізованих драйверів та інструментів, що також можуть показати її наявність у системі [34].

Гібридний аналіз шкідливого ПЗ поєднує в собі використання методів статичного та динамічного, внаслідок цього можливо отримати переваги обох методів [37]. Статичний аналіз є швидшим безпечнішим за динамічний, на противагу, динамічний є більш надійним, може визначати шкідливе ПЗ попри використання технік обфускації а також раніше невідомі шкідливі зразки у випадку схожості поведінки з вже існуючими.

Rafiqul Islam та інші [38] виділили дві ознаки для статичного аналізу — частота довжин функцій та символьні рядки. Ознаки для динамічного аналізу — виклики API та аргументи функцій. На отриманих даних для їх класифікації були застосовані алгоритми випадкового лісу, котрий показав точність виявлення 99.8% на застарілих зразках шкідливого ПЗ та 97.1% для нових.

Вчені Shijo P. V. та Salim A. [39] запропонували техніку виявлення та класифікації невідомих файлів на основі статичного аналізу символьних рядків разом з динамічним аналізом викликів API. Отримана система мала точність 95.8% при використанні статичного аналізу, 97.1% при використанні динамічного та 98.7% при їх комбінації. Для машинного навчання був використаний метод опорних векторів. Igor Santos та інші [42] розробили інструмент виявлення шкідливого ПЗ, котрий аналізує процесорні інструкції для статичного аналізу та системні функції та виключні ситуації (англ.

exceptions) для динамічного аналізу. Такий інструмент показав точність в 95.9% при використанні статичного аналізу, 77.26% для динамічного та 96.6% при їх комбінації. Для машинного навчання був використаний метод опорних векторів.

Таблиця 3 — Характеристики систем виявлення на основі гібридних методів

Автор /рік	Ознака для виявлення	Класифікатор	Навчальна вибірка (шкідливі / безпечні зразки)	Точність виявлення	Помилка першого роду
Shijo / 2016 [39]	Системні функції	SVM	1368/456	98.7%	-
Santos / 2013 [42]	Системні функції та виключення	DT, KNN, NB, SVM	13189/13000	96.6%	3%
Ma / 2016 [40]	Системні функції	DT, SVM, RF, IB1	2,939/-	97%	5.1%
Islam / 2013 [38]	Системні функції, довжина функцій	DT, SVM	2939/-	97%	5.1%

Аналіз оперативної пам’яті пристроїв стає все більш популярним методом для виявлення шкідливого ПЗ, так як доводить свою ефективність та точність. Така техніка дає можливість визначати різноманітні перехоплення [41] та стан системних функцій, що були змінені шкідливим ПЗ [44]. Основним

джерелом досліджень є образ оперативної пам'яті з інформацією про запущені програми, операційну систему та загальний стан пристрою.

Процес проведення аналізу оперативної пам'яті складається з двох етапів. Перший етан — отримання образу оперативної пам'яті цільового пристрою з використанням спеціалізованих інструментів. Другий етап — безпосереднє її дослідження на шкідливу активність.

Численні дослідження на тему аналізу оперативної пам'яті показали різноманіття підходів отримання інформації про шкідливе ПЗ. Вчені Adi Hayon та Tomer Teller представили техніку аналізу на основі отримання образу пам'яті по настанню окремих системних подій: викликів API, стану швидкодії системи, використанні програмних інструментів [43]. Отримана інформація може описати сам процес виконання шкідливого файлу, а не тільки фінальний результат. У подальшому вчений P. Choi та інші представили модифікацію такого способу [45], реалізувавши отримання образу оперативної пам'яті внаслідок довільно обраного виклику API.

Робота Ahmed Zaki та Benjamin Humphrey [5] була присвячена вивченню ефектів дій шкідливих програм типу rootkit на рівні ядра операційної системи: драйверів, таблиці системних сервісів ядра (англ. system service descriptor table), таблиця адрес переривань (англ. interrupt descriptor table), функцій зворотного виклику. Отримані результати свідчили, що найбільш використовуваними шляхами для атак є функції зворотного виклику, модифіковані драйвери та сторонні підключені пристрої.

В роботі Rayan Mosli та інших [48] було проведено дослідження трьох ознак, виділених з образу оперативної пам'яті: використаних програмних бібліотек, активності у системному реєстрі та викликів API. Для даних з системного реєстру та використанням методу опорних векторів була досягнута точність розпізнавання у 96%. У наступному дослідженні [49] Rayan Mosli та інші використали аналіз дескрипторів процесів для дослідження зразків ПЗ та перевірки їх діяльності. Таким чином були виявлені основні

дескриптори, котрі використовуються шкідливим ПЗ: дескриптори сегментів, дескриптори процесів, примітиви синхронізації.

Таблиця 4 — Характеристики систем виявлення на основі методів аналізу оперативної пам'яті

Автор /рік	Ознака для виявлення	Класифікатор	Навчальна вибірка (шкідливі / безпечні зразки)	Точність виявлення	Помилка першого роду
Mosli / 2016 [48]	Системний реєстр, програмні бібліотеки, виклики API	SVM, DT, KNN	400/100	96%	-
Mosli / 2017 [49]	Системні дескриптори	KNN, SVM	3130/1157	91.4%	-
Zaki / 2014 [5]	Драйвери, перехоплення, функції зворотного виклику	-	-	-	-

Ще одними об'єктами для дослідження при аналізі оперативної пам'яті є дослідження перехоплень API, ін'єкція DLL та приховані процеси.

Windows API зазвичай використовується для обміну інформацією між системними ресурсами: файлами, процесами, реєстром, мережевими підключеннями [46]. Шкідливе ПЗ може використовувати спосіб перехоплення API, тобто воно може змінювати системні функції, додаючи в

них шкідливий код. Далі приведені основні типи перехоплень системного API [47] [50]:

- перехоплення inline API: така техніка передбачає запис до пам'яті окремого процесу. Наприклад, додання інструкцій переходу (англ. JMP) у цільову функцію, котра здійснює перехід до області пам'яті, де міститься код шкідливого ПЗ.
- перехоплення таблиці адрес імпорту (англ. import address table): виконувані файли зберігають інформацію про системні функції у таблиці адрес імпорту. Шкідливе ПЗ виконує заміну адреси системної функції на власну, таким чином процес буде викликає функцію зловмисника замість системної.
- перехоплення таблиці системних сервісів (англ. system service descriptor table): таблиця системних сервісів містить в собі адреси функцій рівня ядра. З використанням SSDT шкідливе ПЗ може маскувати свою активність. Наприклад, перехоплюючи адресу системної функції "NTTerminateProcess" можливо прибрати можливість завершити окремий процес.
- перехоплення таблиці адрес переривань (англ. interrupt descriptor table): таблиця адрес переривань містить у собі адреси функцій, котрі виконують обробку виключних функцій та переривань. Таким чином підміна адреси може вести до виконання шкідливого коду у випадку появи виключних ситуацій у процесі.
- перехоплення IRP–пакету: I/O request packet — структура даних ядра Windows, котра забезпечує обмін даними між драйвером та окремою програмою. Використання перехоплення дозволяє виконувати шкідливі функції, такі як реєстрування введення із клавіатури або моніторинг доступів до файлової системи.

Ін'єкція DLL — процес вбудовування шкідливого коду до процесу. Внаслідок виконання такої процедури відбувається передача контролю

виконання до адресного простору шкідливого ПЗ [51]. Основні техніки ін’єкцій DLL наступні:

- класична ін’єкція DLL до процесу: шкідливе ПЗ виконує запис адреси бібліотеки DLL до адресного простору іншого процесу та створює новий потік виконання для забезпечення процедури завантаження DLL. Першим кроком є виклик системної функції "VirtualAllocEx" для виділення пам’яті у цільовому процесі, далі відбувається запис шляху DLL до процесу через виклик "WriteProcessMemory". Наступним кроком є завантаження DLL внаслідок дії функції "LoadLibrary". Останнім крок — запуск окремого потоку виконання для запуску DLL, для цього можуть використовуватись такі системні функції: "NtCreateThreadEx", "RtlCreateUserThread" або "CreateRemoteThread" [51].
- ін’єкція DLL через подію вікон: програмний код програм Windows виконується на основі системних подій. Шкідливе ПЗ може здійснювати ін’єкцію своїх бібліотек внаслідок виконання таких подій. Функція "SetWindowsHookEx" може завантажувати код у послідовність виконання. Таким чином, такий код буде виконуватись кожен раз при настанні будь-якої події у цільовому процесі [52].
- ін’єкція DLL через системний реєстр: відбувається модифікація значення ключа реєстру "Appinit_DLL". Адреса шкідливої DLL додається до вже наявних значень шляхом викликів "RegCreateKeyEx" та "RegSetValueEx". Для застосування змін необхідно виконати перезавантаження пристрою. [54]

Індикатором існування шкідливого програмного забезпечення також може бути наявність прихованих процесів, файлів та мережевих підключень [53]. Приховання процесів зазвичай здійснюється через rootkit, котрий модифікує код програми. Іншим методом є перехоплення викликів між

програмами та ядром операційної системи через модифікацію системних таблиць, структур даних, бібліотек та системних функцій API [54]. Деякі зразки шкідливого ПЗ можуть здійснювати маніпуляцію об'єктів ядра (англ. direct kernel object manipulation), тим самим здійснюючи приховання свого існування у системі. [56].

Також для перешкоджання аналізу оперативної пам'яті існують наступні техніки:

- блокування доступу до пам'яті: виконується перешкоджання доступу до пам'яті через відключення процесів, модифікації драйвера або маніпуляції об'єктами ядра [55].
- приховання пам'яті: шкідливе ПЗ намагається вилучити певну область пам'яті від аналізу через її приховання у операційній системі.
- перешкоджання виконанню: створення надмірної кількості об'єктів в пам'яті з метою збільшення витраченого часу на її аналіз.

Таким чином, існує декілька характерних рис розвитку майбутнього шкідливого програмного забезпечення. Перша риса — автоматична розробка шкідливих зразків. Зловмисники також досліджують перспективні методи виявлення, і для їх протидії створюються інструменти для автоматичного розроблення шкідливого програмного забезпечення. Друга риса — постійна зміна структури та функціональності шкідливих зразків. Так, як здебільшого навчання моделей розпізнавання здійснюється на вже відомих зразках, то навіть незважаючи на відмінні показники готових моделей, вони можуть показати незадовільну точність на нових даних. Третя риса — збільшення складності шкідливого програмного забезпечення, використання нових методів шифрування та обфускації коду для унеможливлення проведення задачі його виявлення. Тому, зважаючи на описані риси, майбутній розвиток шкідливого програмного забезпечення та процедури щодо його протидії

мають тренд на подальше ускладнення методів в обох областях, що є викликом для дослідників та компаній у області захисту інформації.

Машинне навчання вплинуло на широкий спектр прикладних задач, найбільш продуктивним є застосування відповідних методів при розпізнаванні образів та обробці природної мови. Останнім часом саме штучні нейронні мережі є найбільш використовуваними моделями для роботи з зображеннями. Серед численних архітектур та підходів був розроблений окремий тип — згорткові нейронні мережі, котрі мають можливість виділяти як прості (кути, лінії) так і більш складні (фігури, обличчя) образи із вхідних зображень. Порівняно з іншими архітектурами штучних нейронних мереж саме вдало побудовані згорткові показують найбільш високі результати у задачах розпізнавання зображень [58].

Коротко опишемо методи та архітектуру, що застосовуються при побудові згорткових нейронних мереж. Як відомо, входом до повнозв'язної нейронної мережі є одновимірний вектор даних.

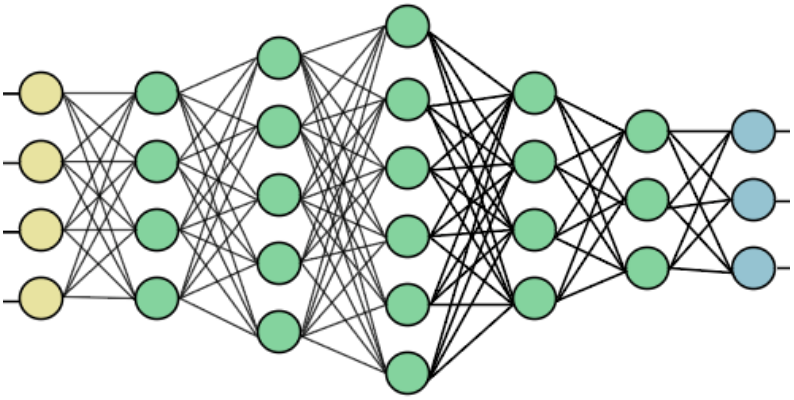

Рисунок 1.2.4 — Архітектура повнозв'язної нейронної мережі

Так, як зображення є двовимірним об'єктом, то виникає проблема коли необхідно подати його на вхід нейронній мережі.

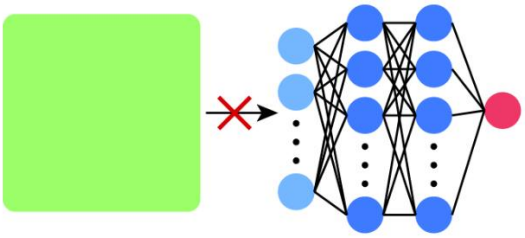

Рисунок 1.2.5 — Неможливість подання 2D зображень на вхід повнозв'язної нейронної мережі

Вирішенням такої проблеми може стати перетворення зображення у одновимірний вектор шляхом перебудови даних і розміщення рядків або колонок один за одним. Проблема такого перетворення у тому, що відбувається втрата важливої просторової інформації щодо зв'язків між пікселями сусідніх рядків або колонок, котрі після перетворення можуть вже не розміщуються близько один від одного. Для вирішення такої проблеми був створений окремий вид мереж — згорткові нейронні мережі.

У процесі згортки окремого зображення використовується ядро, котре застосовується до вхідного зображення методом ковзного вікна. Результатом згортки є ще одне зображення, в якому пікселі з високими значеннями яскравості знаходяться у зонах, які схожі на шаблон ядра, а пікселі з низькими значеннями яскравості сигналізують про відсутність шаблону у певній області зображення. Ядро — це невелика матриця розміру зазвичай 3×3 або 5×5 що містить в собі той шаблон, котрий ми намагаємось виділити із зображення.

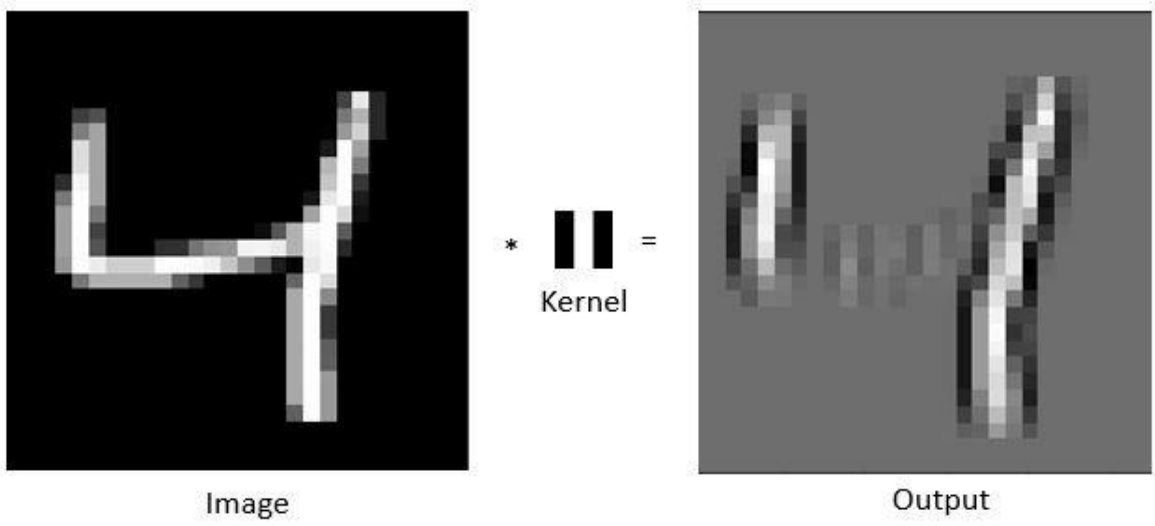

Рисунок 1.2.6 — Приклад застосування ядра до вхідного зображення

На Рисунку 1.2.6 ядром є шаблон вертикальної лінії розміром 3×3, результатом операції згортки отримуємо інше зображення (карту ознак) з котрого можливо визначити, в яких місця вхідного ображення існують вертикальні лінії, котрі збігаються з шаблоном. Таким чином відбувається

екстракція ознак з вхідних даних, змінюючи вигляд ядра, можливо виділяти довільні ознаки із зображення.

Опишемо процедуру обчислення згортки на прикладі зображення розмірності 10×10 та ядра розмірності 3×3.

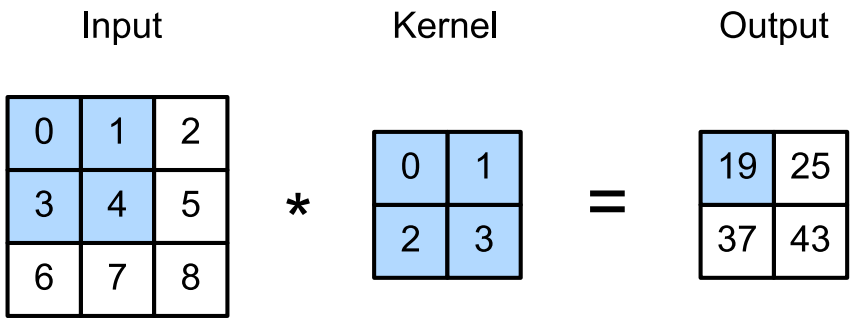

Рисунок 1.2.7 — Обчислення значень пікселів при операції згортки

При даній операції здійснюється поелементне множення ядра та частини вхідної матриці, яка відповідає розмірності ядра з подальшою сумою всіх результуючих значень. Обчислена сума є значенням яскравості пікселя для вихідної карти ознак. В подальшому описана процедура обчислення відбувається методом ковзного вікна, проходячи по всій площі вхідного зображення.

При виконанні процедури згорки розмір вихідного зображення є меншим за розмір вхідного, так як для частини пікселів біля краю неможливо обчислити поелементне множення, без врахування пустих значень, що знаходяться поза зображенням. Якщо позначити ширину вхідного зображення як W_i, а ширину H_i, ширину та висоту ядра відповідно як W_c та H_c, то результуюче зображення після процедури згортки буде мати наступні розміри:

$$W_r = (W_i - W_c + 1),$$

$$H_r = (H_i - H_c + 1),$$

де W_r та H_r ширина та висота результуючого зображення.

Якщо існує необхідність отримання вихідного зображення з розмірністю вхідного то можливо використати процедуру доповнення вхідного зображення

додатковою кількістю пікселів на межі перед виконанням процедури згортки. Зазвичай додаються пікселі з нульовими значеннями яскравості.

Для роботи з кольоровими 2D зображеннями необхідно розділити одне вхідне зображення на окремі канали, де кожен з них буде містити інформацію про окремий колір зображення. У випадку моделі RGB, існує 3 канали зображення, кожен з яких відповідає за червоний, зелений та синій колір відповідно.

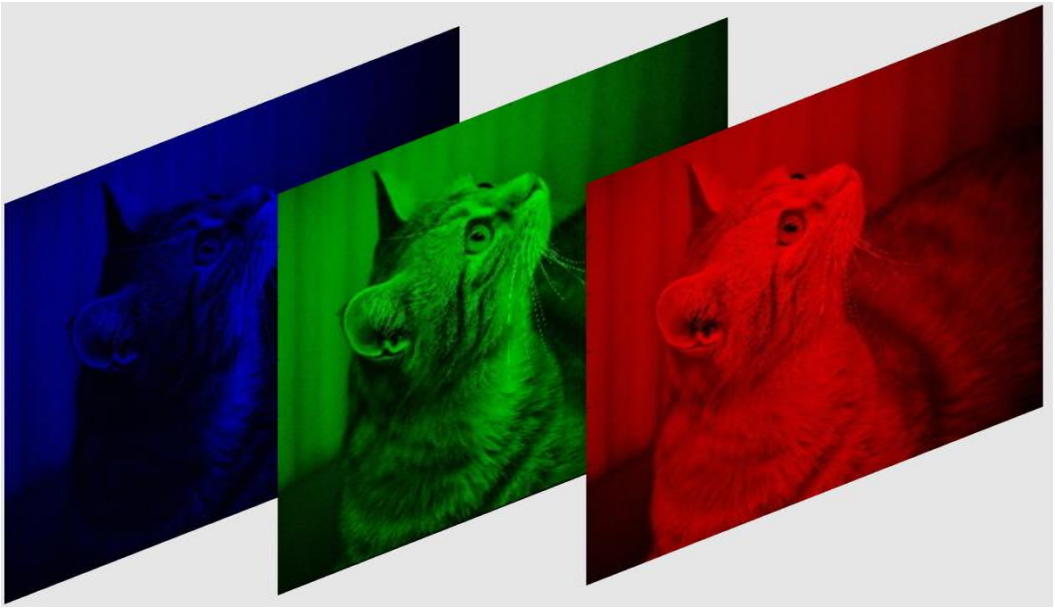

Рисунок 1.2.8 — Представлення вхідного зображення як сукупності RGB каналів

Розглянемо основні архітектурні елементи згорткових нейронних мереж, такі мережі містять у собі спеціалізовані згорткові шари, вхідними даними для яких є окремі зображення. Так, як кольорове вхідне зображення містить у собі 3 канали, котрі відповідать червоному, зеленому та синьому кольору, розмір вхідної матриці наступний: $W \times H \times 3$, де W та H — ширина та висота зображення. Для його обробки ядро також повинно містити 3 канали, котрі будуть виконувати екстракцію окремо з кожного каналу. Процедура обчислення карти ознак аналогічна до описаної вище, з доданням кроку обчислення суми трьох значень згортки, отриманих з кожного каналу зображення. У згортковому шарі є можливість вибору довільної кількості ядер для екстракції ознак з вхідного зображення. В результаті операції згортки ми

отримуємо нове зображення, котре є результатом пошуку шаблону, закодованому у ядрі. Таким чином при застосування n ядер буде отримано n нових карт ознак, дані яких можливо інтерпретувати як значення відповідності шаблонам у ядрах. Можливо об'єднати отримані n карт ознак у нове n-канальне зображення, так як кількість ядер у згортковому шарі визначає кількість каналів вихідного зображення, котрі в свою чергу будуть слугувати входами для подальших згорткових шарів. Таким чином, додаючи нові згорткові шари до мережі ми можемо проводити екстракцію більш складних шаблонів із вхідного зображення.

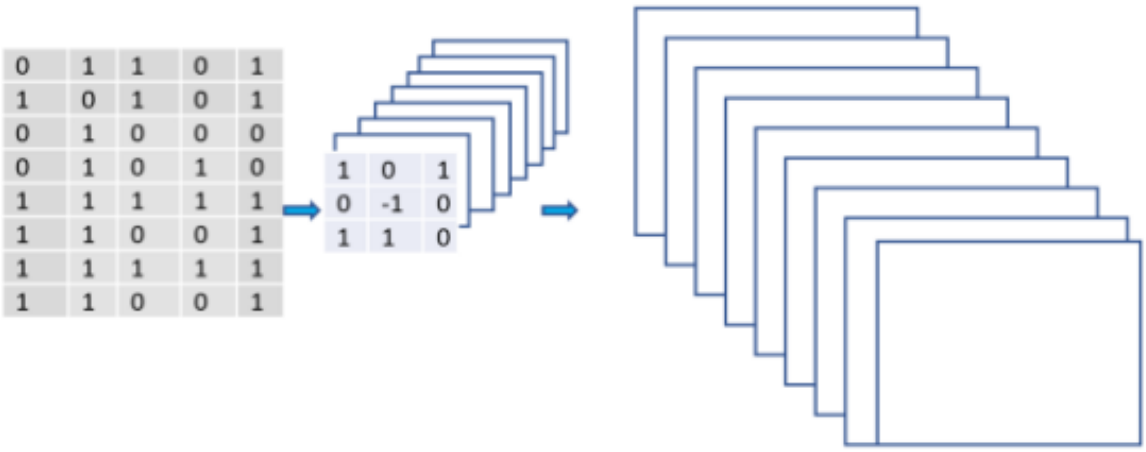

Рисунок 1.2.9 — Зміна розмірності вхідних даних після застосування операції згортки

Причиною для додання великої кількості згорткових шарів є необхідність виділення складних шаблонів із вхідних зображень, незважаючи на невеликий розмір ядер та їх можливість виділяти лише базові ознаки, їх застосування на комбінації багатьох карт ознак, як на комбінації попередньо виділених ознак, дозволяє агрегувати базові ознаки, формуючи з них складні. В результаті додання великої кількості згорткових шарі, ті які є останніми мають можливість виділяти такі ознаки як окреме обличчя або певний об'єкт.

Крім згорткових шарів, у мережі також існують агрегувальні шари, задача яких полягає у зменшенні простору ознак вхідного зображення. Операція агрегування полягає у виборі розміру ковзного вікна та вибору максимального значення в ньому, два сусідніх вікна не повинні перетинатись.

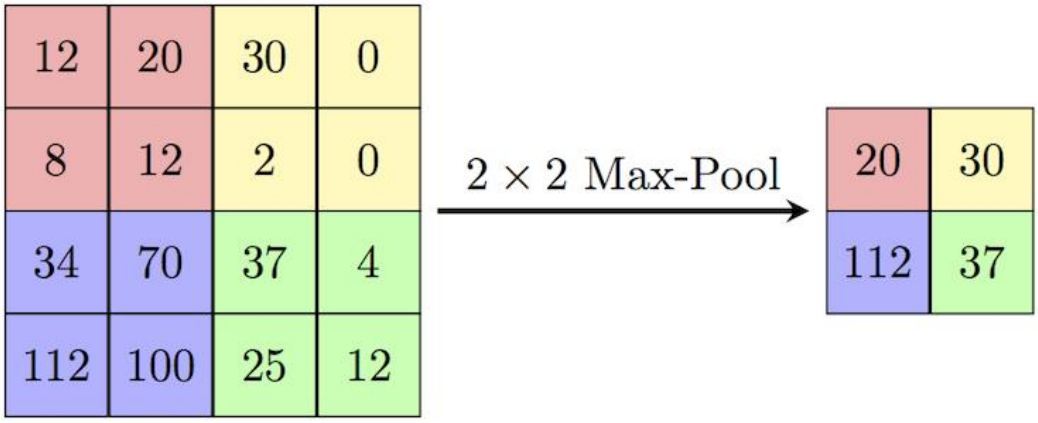

Рисунок 1.2.10 — Приклад операції агрегування з вікном розміру 2×2

Так, як в результаті операції згортки відносно великі значення пікселів відповідають місцям знаходження шаблонів, то такі пікселі можна вважати важливими ознаками зображення. Виконання агрегування покликано концентрувати важливі ознаки із отриманих карт ознак при одночасному зменшенні їх вихідних розмірів. Шари агрегації знаходяться у мережі не так часто як згорткові, зазвичай на 3-4 згорткові шари існує 1 шар агрегації.

Останньою ланкою у згортковій нейронній мережі є окрема повнозв'язна нейронна мережа, на вхід якої подається звичний одновимірний вектор. Так, як в останніх шарах мережі карти ознак мають відносно невелику одну з розмірностей, то з використанням операції агрегування можливо перетворити їх в одновимірний вектор. Інформацію останніх карт ознак можливо інтерпретувати як виділення окремих високорівневих ознак, високі значення окремих пікселів сигналізують про наявність ознаки і навпаки, тому такі дані можливо в подальшому обробити з використанням повнозв'язної нейронної мережі. Її конфігурація здійснюється залежно від задачі, котру необхідно виконати, наприклад, для задачі класифікації вихідний шар такої нейронної мережі буде мати логістичну функцію активації та кількість вихідних нейронів буде відповідати кількості класів для класифікації.

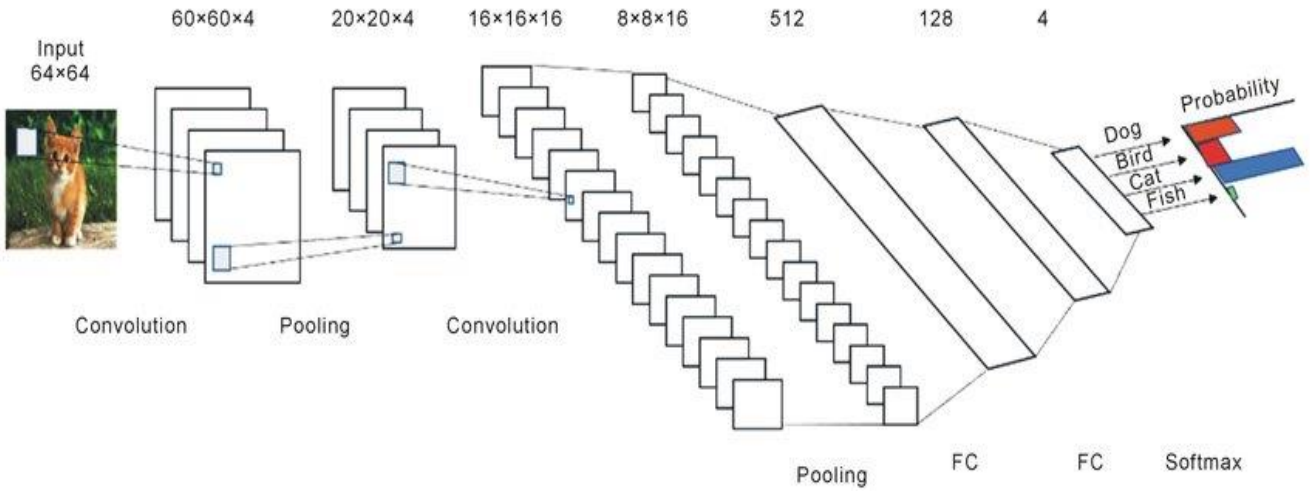

Рисунок 1.2.11 — Приклад архітектури згорткової нейронної мережі

Навчання згорткової нейронної мережі здійснюється аналогічно до звичайної, тобто на вихідному шарі обчислюється функція втрат і застосовується метод стохастичного градієнтного спуску для налаштування параметрів мережі, така процедура повторюється на всьому об'ємі даних . Так, як кожне ядро має вплив на результат роботи мережі, то параметри ядер також змінюються для детектування ознак, котрі часто з'являються у вхідних даних можуть бути ключовими для правильної класифікації.

1.3. Постановка задачі

Метою роботи є створення інтелектуальної системи розпізнавання шкідливого програмного забезпечення на основі згорткових нейронних мереж з використанням кватерніонних елементів у реалізації. Необхідно дослідити та оптимізувати модель ознакового опису мережі, її методи навчання та розпізнавання. На основі вхідної вибірки з описом набору реалізацій шкідливих та безпечних зразків програмного забезпечення необхідно здійснити їх аналіз та класифікацію. Отримана система повинна забезпечити оптимальні показники точності виявлення та класифікації шкідливих зразків.

Для досягнення поставленої задачі в роботі необхідно вирішити наступні завдання:

1. Визначення методу екстракції ознакового опису спостережень;

2. Вибір методу класифікаційного аналізу для прогнозування впливу шкідливого програмного забезпечення та побудова вирішальних правил;
3. Побудова моделі та методу навчання моделі розпізнавання шкідливого програмного забезпечення;
4. Дослідження особливостей функціонування та реалізації кватерніонних шарів нейронної мережі;
5. Програмна реалізація алгоритмів інтелектуальної системи розпізнавання;
6. Формування даних для навчальної матриці з подальшим аналізом та класифікацією шкідливого програмного забезпечення;
7. Тестування та оцінювання ефективності розробленого програмного забезпечення, аналіз результатів;

2. ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ РОЗПІЗНАВАННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Методи екстракції ознакового опису спостережень

Файли шкідливого програмного забезпечення зазвичай розповсюджуються у вигляді виконуваних бінарних файлів або файлів бібліотек з розширеннями .exe, .dll, .bin відповідно. Особливістю таких файлів є окремі структурні компоненти у їх змісті. Серед них: сегменти text, rdata, data, rscr. Текстовий сегмент містить безпосередні інструкції програмного коду. У частині rdata знаходяться дані, доступні лише для читання з програмного коду, такі як строки тексту, константи, інформацію про директорію для відладки програми. Дані сегменту data можуть бути змінені при безпосередньому виконанні файлу. Rscr частина містить посилання на ресурси та сторонні директорії і бібліотеки, що використовуються.

Для більш продуктивного дослідження можливо перевести вміст виконуваного файлу у зображення, так як наявність певних шаблонів всередині файлу стає видимою наочно. Екстракцію ознак з таких зображень можливо провести з використанням машинного начання [64]. Окремі шаблони у виконуваних файлах можуть бути індикатором окремих видів шкідливого ПЗ. Для рішень задачі розпізнавання підходять згорткові нейронні мережі, котрі через операії згортки, агрегування та інших обчислень проводять екстракцію ознак з довільних зображень. Шкідливе ПЗ може бути перетворене у зображення при застосуванні перетворення послідовностей бінарних даних у значення яскравостей окремих пікселей вихідного зображення.

Загалом, виконуваний файл може розглядатись як послідовність бінарних нулів та одиниць. Тому першим кроком до екстракції ознакового опису є перетворення виконуваного файлу у вигляд строки одиниць та нулів. Для подальшого перетворення у зображення необхідно розділити строку у послідовності довжиною у 8 симолів, які в подальшому будуть інтерпретовані як значення восьмибітного числа — яскравість окремого пікселя.

Восьмибітна бінарна послідовність може бути представлена у діапазоні $0000\,0000_{bin}$ (0_{dec}) до $1111\,1111_{bin}$ (256_{dec}), сукупність таких значень можливо об'єднати у чорно-біле зображення. На вхід пропонована кватерніонна мережа для обробки приймає триканальні зображення, тому слід додатково створити відсутні розмірності.

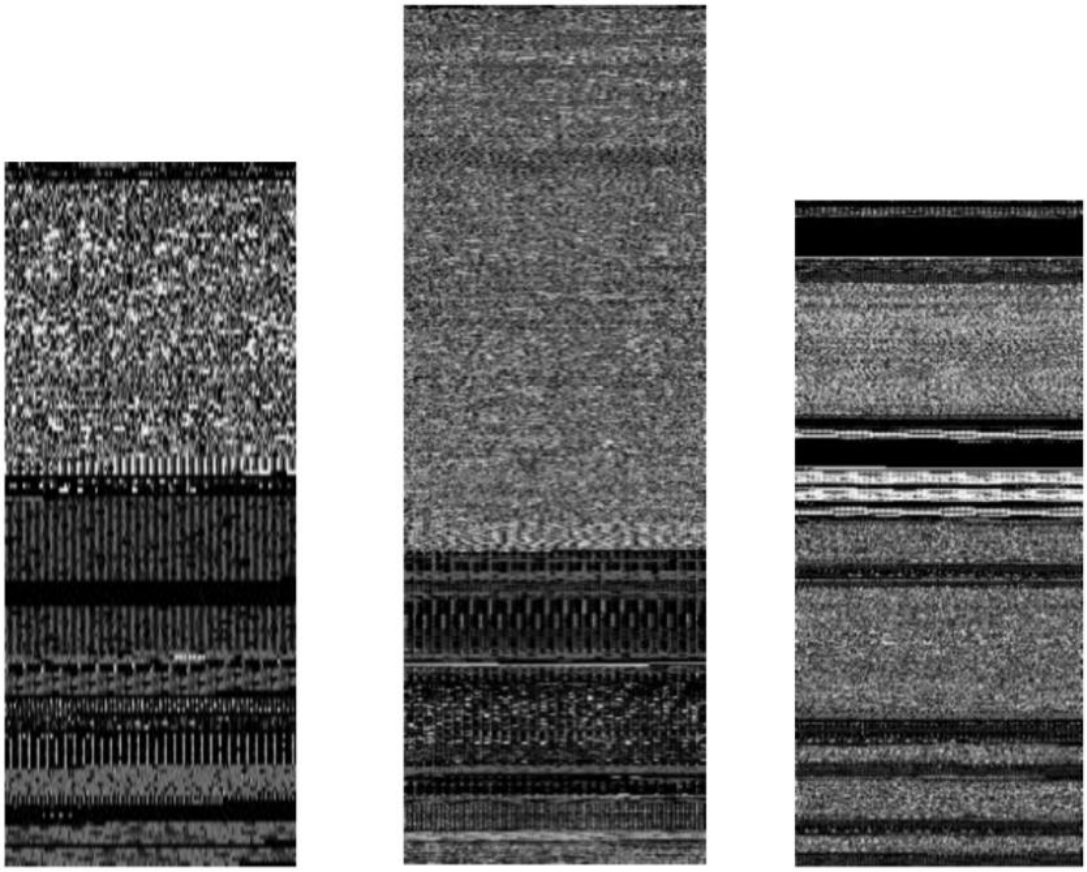

Рисунок 2.1.1 — Приклад зображень виконуваних файлів шкідливого ПЗ

2.2. Модель кватерніонної нейромережі для розпізнавання шкідливого програмного забезпечення

Ключовою особливістю згорткових нейронних мереж є екстракція ознак з багатовимірних даних використовуючи набір ядер у згорткових шарах. При обробці багатоканальних вхідої інформації, таких як кольорові зображення, відбувається сума результатів операцій згортки щодо різних каналів і вихідними даними слугує один канал на кожне ядро.

Незважаючи на те, що такий спосіб обробки показує високі результати у практичних задачах розпізнавання, можливо виділити деякі особливості,

котрі можливо покращити. Так, як кожне ядро проводить операцію суми результів згортки з всіх каналів, не враховується зв'язок, що потенційно може існувати між ними. Тому можлива втрата структурної інформації про колір та як результат не оптимальне представлення кольорового зображення [59]. Іншою проблемою є те, що операція суми може внести можливість перенавчання попри застосування регуляризації щодо значень матриць ядер.

Зважаючи на вищезазначені проблеми, можливо застосувати модель кватерніонної згорткової мережі, де кожен піксель зображення представлений у вигляді окремого кватерніону замість використання декількох окремих матриць для каналів кольору. Беручи матрицю кватерніонів як вхідні дані можливо розробити наступні компоненти: кватерніонний згортковий шар та повнозв'язний кватерніонний шар. На відміну від згорткових нейронних мереж з дійсними значеннями, котрі виконують лише операцію «масштабування» вхідних даних, кватерніонні нейронні мережі, завдяки їх властивостям додають операцію повороту в просторі кольору, що може дати додаткову структурну інформацію про зображення.

2.2.1. Алгебра кватерніонів

Кватерніон це різновид гіперкомплексного числа, який описав Сер Вільям Ровен Гамільтон у 1848 році, таке число можливо інтерпретувати як точку у тривимірному просторі. Таким чином кватерніон Q можливо представити у вигляді:

$$\hat{q} = r + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} \tag{1}$$

де числа $r, x, y, z \in \mathrm{R}$.

Уявні одиниці i, j, k задовольняють наступній властивості:

$$i^2 = j^2 = k^2 = ijk = -1, \tag{2}$$

N-вимірний кватерніон може бути представлений у вигляді вектора:

$$\hat{q} = [\hat{q}_1, \cdots, \hat{q}_N], \tag{3}$$

Подібно до дійсних чисел для кватерніонів визначені такі операції:

- додавання

$$\hat{p} + \hat{q} = (r_p + r_q) + (x_p + x_q) \cdot i + (y_p + y_q) \cdot j + (z_p + z_q) \cdot k, \quad (4)$$

- множення на скаляр

$$\alpha \cdot \hat{q} = \alpha \cdot r + \alpha \cdot x\boldsymbol{i} + \alpha \cdot y\boldsymbol{j} + \alpha \cdot z\boldsymbol{k}, \quad (5)$$

- Гамільтонове множення

$$\begin{aligned} \hat{p}\hat{q} = {} & (r_p \cdot r_q - x_p \cdot x_q + y_p \cdot y_q + z_p \cdot z_q) \\ & + (r_p \cdot x_q - x_p \cdot r_q + y_p \cdot z_q + z_p \cdot y_q) \cdot \boldsymbol{i} \\ & + (r_p \cdot y_q - x_p \cdot z_q + y_p \cdot r_q + z_p \cdot x_q) \cdot \boldsymbol{j} \\ & + (r_p \cdot z_q - x_p \cdot y_q + y_p \cdot x_q + z_p \cdot r_q) \cdot \boldsymbol{k}, \end{aligned} \quad (6)$$

- спряження

$$\hat{q}^* = r - x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k}, \quad (7)$$

- норма кватерніона

$$\|\hat{q}\| = \sqrt{r^2 + x^2 + y^2 + z^2}, \quad (8)$$

Кватерніон також можливо визначити через матрицю дійсних чисел:

$$\hat{q} = \begin{bmatrix} r & -x & -y & -z \\ x & r & -z & y \\ y & z & r & -x \\ z & -y & x & r \end{bmatrix}, \quad (9)$$

Операції (4, 5, 6, 7) можуть бути використані при поворотах векторів у тривимірному просторі. Наприклад, якщо необхідно здійснити поворот вектора $q = [q_1\ q_2\ q_3]$ на довільний кут θ відносно осі $w = [w_1\ w_2\ w_3]$, де ${w_1}^2 + {w_2}^2 + {w_3}^2 = 1$, з результуючим вектором $p = [p_1\ p_2\ p_3]$, то можливо використати наступну формулу:

$$\hat{p} = \hat{w}\hat{q}\hat{w}^*, \quad (10)$$

де $\hat{q} = 0 + q_1 i + q_2 j + q_3 k$ та $\hat{p} = 0 + p_1 i + p_2 j + p_3 k$ представлені у вигляді кватерніонів, також $\hat{w} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}(w_1 i + w_2 j + w_3 k)$. В зв'язку з зручністю представлення поворотів 3D векторів, кватерніони широко використовуються в механіці та фізиці. В області комп'ютерного зору та обробки зображень методи, що базуються на використанні кватерніонів можуть бути використані при вирішенні деяких задач, та показують

перспективні результати в задачах класифікації кольорових зображень [60]. В задачах зменшення цифрового шуму та підвищення роздільної здатності зображень методи, що базуються на використанні кватерніонів зберігають більше інформації про зв'язки між каналами кольору і можуть відтворювати більшу роздільну здатність [61].

2.2.2. Кватерніонні згорткові шари

Входом пропонованої моделі кватерніонної згорткової нейронної мережі є матриця 2D кватерніонів, що представляє кольорове зображення, позначене як $\widehat{\boldsymbol{A}} = [\hat{a}_{nn'}] = 0 + \boldsymbol{R}i + \boldsymbol{G}j + \boldsymbol{B}k$, де $\boldsymbol{R}, \boldsymbol{G}, \boldsymbol{B}$ представляють значення червоного, зеленого та синього каналів зображення розмірів $N \times N$. Якщо маємо ядро кватерніонів $\widehat{\boldsymbol{W}} = [\hat{w}_{ll'}]$ розмірами $L \times L$, то необхідно визначити операцію згортки між входом $\widehat{\boldsymbol{A}}$ та ядром $\widehat{\boldsymbol{W}}$. Така операція повинна застосовувати поворот та масштабування векторів кольору для пошуку оптимального представлення у просторі, мати такий самий результат як і згорткові мережі з дійсними значеннями для чорно-білих зображень. Для досягнення вищезазначених властивостей можливо представити операцію згортки у наступному вигляді, котра використовує особливості повороту кватерніонної алгебри, значення елементів кватерніону будуть визначатись як:

$$\hat{w}_{ll'} = s_{ll'} \left(cos\frac{\theta_{ll'}}{2} + sin\frac{\theta_{ll'}}{2}\mu\right), \tag{11}$$

де $\theta_{ll'} \in [-\pi, \pi]$ та $s_{ll'} \in R$, μ — вісь чорно-білого зображення з одиничною довжиною $\left(\frac{\sqrt{3}}{3}(i + j + k)\right)$. Таким чином, згорка буде визначатись:

$$\widehat{\boldsymbol{A}} \otimes \widehat{\boldsymbol{W}} = \widehat{\boldsymbol{F}} = [\hat{\boldsymbol{f}}_{\boldsymbol{kk'}}] \tag{12}$$

з відповідною розмірністю $(N - L + 1) \times (N - L + 1)$,

$$\hat{\boldsymbol{f}}_{\boldsymbol{kk'}} = \sum_{l=1}^{\boldsymbol{L}} \sum_{l'=1}^{\boldsymbol{L}} \frac{1}{s_{ll'}} \hat{w}_{ll'} \hat{a}_{(k+l)(k'+l')} \hat{w}_{ll'}^{*} \tag{13}$$

Набір таких ядер і формує згортковий шар. На відміну від згорткового шару мережі з дійсними значеннями, в котрому здійснюється множення дійсних

чисел, операція в згортковому шарі кватерніонної мережі здійснює поворот та масштабування вхідної матриці $\hat{a}_{nn'}$ на кожному кроці. Вісь повороту встановлюється як $(\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3})$, тобто вісь чорно-білого зображення у просторі кольору, кут повороту та значення масштабування визначаються як $\theta_{ll'}$та $s_{ll'}$ відповідно. Перевага такого визначення полягає у визначенні пікселя як окремого кватерніону та застосування до нього лише двох операцій повороту та масштабування, що передбачають меншу кількість ступенів свободи на відміну від мереж з дійсними значеннями, що зменшує ймовірність перенавчання при процедурі згортки ядрами. Також кватерніонна мережа зберігає можливу інформацію зв'язку між різними каналами кольору на відміну від традиційних згорткових мереж, де формується лише одна карта ознак з декількох вхідних каналів кольору.

Така кватерніонна мережа може застосовуватись і для обробки чорно-білих зображень, для таких даних можливо вважати канали кольору однаковими. Так, як всі вектори пікселей чорно-білого зображення є паралельними вісі повороту, то така мережа буде виконувати функцію звичайної згорткової мережі з дійсними значеннями.

Якщо представити кожен елемент матриці $\hat{a}_{nn'}$ у вигляді 3D вектора $\hat{a}_{nn'} = [a_1\ a_2\ a_3]$, то можливо замінити операцію (14) на множення матриць.

$$\hat{\boldsymbol{f}}_{\boldsymbol{kk'}} = \sum_{\boldsymbol{l=1}}^{\boldsymbol{L}} \sum_{\boldsymbol{l'=1}}^{\boldsymbol{L}} s_{ll'} \begin{bmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{bmatrix} \hat{a}_{(k+l)(k'+l')}, \qquad (14)$$

де $\hat{\boldsymbol{f}}_{\boldsymbol{kk'}}$— представлення кватерніона, де

$$f_1 = \frac{1}{3} + \frac{2}{3}\cos\theta_{ll'}, f_2 \frac{1}{3} - \frac{2}{3}\cos(\theta_{ll'} + \frac{\pi}{3}), f_3 = \frac{1}{3} - \frac{2}{3}\cos(\theta_{ll'} + \frac{\pi}{3})$$

Так, як вхідні і вихідні дані для таких згорткових шарів є матриці кватерніонів, то такі шари можуть бути об'єднані аналогічно до шарів у традиційних згорткових мережах, і є сумісними з ними.

2.2.3. Кватерніонні повнозв'язні шари

Для збереження виділених ознак з використанням кватерніонної мережі доцільно вихідні дані згорткового шару подавати на вхід повнозв'язного шару з кватерніонних елементів. Зважаючи на можливість представлення повнозв'язних шарів як різновиду згорткових шарів, де ядра мають однакову розмірність з вхідними даними, то таку мережу можливо побудувати із кватерніонних елементів. Якщо входом є N-вимірний вектор кватерніон $\hat{a} = [\hat{a}_i], i = 1, 2, 3, \cdots, N$, то застосовуючи M одновимірних ядер $\widehat{w}^m = [\widehat{w}_i^m], i = 1, \cdots, M$ можливо отримати значення $\hat{b} = \left[\hat{b}_i\right]$ у вузлі мережі:

$$\hat{b} = \sum_{i=1}^{N} \frac{1}{s_i} \widehat{w}_i^m \hat{a}_i \widehat{w}_i^{m*}, \qquad (15)$$

де s_i норма вектора $\widehat{w}_i^m$.

Обислення значення $\hat{b}$ також можливо представити у вигляді множення матриць, аналогічно до обчислення у згортковому шарі (15).

2.2.4. Шари агрегації

Агрегувальні шари та функції активації дозволяють отримувати нелінійні вихідні залежності між вхідними та вихідними даними. В шарі агрегації, який усереднює значення проводить усереднення уявних частин кватерніона, для шару, що вибирає максимальне значення можливо визначити декілька операцій, таких як проекція на вісь чорно-білого зображення або знаходження норми вектора. Для даної моделі використана операція знаходження максимуму з трьох уявних одиниць для шару максимізаційного агрегування. Функції активації, що використовуються є аналогічними до стандартних у нейронних мережах. Для ReLU у випадку повороту у від'ємну частину для RGB каналів відбувається встановлення значення до найближчої точки у просторі кольору. Для логістичної функції активації необхідно розділити кватерніон на окремі значення і подати на вхід повнозв'язних мереж для навчання класифікаторів.

Таку кватерніонну мережу можливо поєднувати з згортковими мережами з дійсними значеннями, так для зв'язку зі згортковим шаром вихід

кватерніонної мережі може бути переведений у 3 чороно-білих карти ознак, котрі відповідають каналам кольору. В подальшому такі карти ознак можуть бути об'єднані або окремо подаватись на вхід згорткової мережі з дійсними значеннями. Для з'єднання з повнозв'язним шаром можливо сформувати одновимірний вектор з елементів кватерніону і розмістити їх один за одним у розмірності $(N \times 3) \times 1$, де N — кількість кватерніонів у мережі.

2.3. Метод навчання моделі розпізнавання шкідливого програмного забезпечення

2.3.1. Ініціалізація ваг

Правильна ініціалізація ваг моделі є ключовою для її ефективного процесу навчання. Можливо вважати значення параметру s схожим до параметру ваги окремого ядра згорткової мережі з дійсними значенням, котрий «масштабує» вектор що трансформується. Додатковий параметр θ слугує кутом повороту трансформованого вектора відносно певної заданої. Також при додаванні трансформованих векторів результуюча норма вектора залежить від параметра θ, хоча проекція на вісь чорно-білого кольору не змінюється з вищезазначеною архітектурою. Тому доцільно ініціалізувати ваги та кут повороту за нормальним розподілом задля збереження однакової дисперсії градієнтів впродовж процедури навчання [62]. Таким чином, для кожного k шару кут повороту θ та вага s ініціалізується випадковим значенням з нормальним розподілом:

$$s_k = \cup\left[-\frac{\sqrt{6}}{\sqrt{n_k + n_{k+1}}}, \frac{\sqrt{6}}{\sqrt{n_k + n_{k+1}}}\right], \theta = \cup\left[-\frac{\pi}{2}, \frac{\pi}{2}\right], \quad (16)$$

де $\cup$ позначає нормальний розподіл та n_k — розмірність k-ї вхідної матриці кватерніонів.

2.3.2. Градієнтний спуск

Градієнтний спуск є основою навчання нейронної мережі, котрий базується на ланцюговому правилі диференціювання складної функції,

результатом якого є оновлення параметрів моделі, що навчається. Якщо позначити дійсну функцію втрат L для навчання кватерніонної згорткової мережі. $\hat{p} = r_p + x_p \cdot i + y_p \cdot j + z_p \cdot k$ та $\hat{q} = r_q + x_q \cdot i + y_q \cdot j + z_q \cdot k$ є 2 кватерніонні числа, і так як операція повороту (10) може бути представлена у вигляді множення матриць, то аналогічне перетворення можливо застосувати до градієнту кватерніону.

Якщо маємо наступне визначення градієнта:

$$\frac{\partial L}{\partial q} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial q}, \qquad \frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial \theta}, \qquad \frac{\partial L}{\partial s} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial s}, \tag{17}$$

де $q = [q_1\ q_2\ q_3]$ та $p = [p_1\ p_2\ p_3]$ вектори, що відповідають $\hat{p}$ та $\hat{q}$. У випадку коли q та p довільні елементи карт ознак $\hat{a}_{nn'}$ та ядер $\hat{w}_{ll'}$ часткові похідні градієнта виглядають наступним чином:

$$\begin{aligned}
\frac{\partial p}{\partial q} &= s\begin{bmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{bmatrix}, \\
\frac{\partial p}{\partial \theta} &= s\begin{bmatrix} f_1' & f_2' & f_3' \\ f_3' & f_1' & f_2' \\ f_2' & f_3' & f_1' \end{bmatrix}\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}, \\
\frac{\partial p}{\partial s} &= \begin{bmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{bmatrix}\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix},
\end{aligned} \tag{18}$$

де f_i визначені як (8), матриця f_i' має той самий вид що і вираз (7), змінюється лише порядок множення матриць, тобто градієнтний спуск можна інтерпретувати як поворот відносно осі на обернений кут θ.

2.3.3. Функції втрат та активації

Функції активації та функції втрат повинні бути диференційованими для можливості обчислення градієнту та його подальшого розповсюдження по мережі. Для кватерніонної мережі бідь яка функція, яка є диференційованою і застосовується до кожного компоненту кватерніону [63] дозволяють використати ланцюгове правило і можуть використовуватись як функції активацій або втрат. Конкретний вибір залежить від задач, котрі вирішуються

при використанні мережі. Так, для класифікації об'єктів, де останнім шаром є повнозв'язний з дійсними значеннями, де потрібно перетворити вихідні кватерніони на одновимірний вектор функція втрат матиме вигляд категорійної крос-ентропії. У задачах регресії, коли виходи кватерніонів слід перевести у триканальні зображення, то фунціями активації можуть бути середньоквадратична помилка або подібні.

3. РЕАЛІЗАЦІЯ АЛГОРИТМІВ РОЗПІЗНАВАННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Формування навчальних та тестових даних

Перетворення файлів шкідливого ПЗ у зображення можливо виконати шляхом формування вектору з восьмибітних цілих чисел, отриманих безпосередньо з їх бінарного вмісту. Кожне зі значень буде кодувати яскравість окремого пікселя у діапазоні [0, 255]. Отриманий вектор слід перевести у двовимірну матрицю (чорно біле зображення) довільних розмірів. З файлів шкідливого програмного забезпечення оброблених в такий спосіб був сформований набір тестових даних [65], котрий використовувався для навчання мережі. Вибірка містить 9339 зображень зразків шкідливого програмного забезпечення, котрі розподілені за 25 класами:

- Adialer.C: застарілий вірус, котрий поширюється у мережах Dial-up, та виконує дзвінки на довільні номери за рахунок інфікованого комп'ютера.
- Agent.FYI: троянська програма, виконує відключення антивірусних служб та завантаження програм backdoor-ів для подальшого контролю системи.
- Allaple.A: мережевий хробак, розповсюджується шляхом численного копіювання у файловій системі, кожна копія якого є зашифрованою і відрізняється від інших. Використовується для проведення DDoS атак.
- Allaple.L: видозмінена версія мережевого хробака Allaple.A.

- Alureon.gen!J: троянська програма, що виконує зміну налаштувань DNS таблиць на мережевому обладнанні: маршрутизаторах, комутаторах.
- Autorun.K: хробак, котрий поширюється шляхом копіювання на флеш-накопичувачі або будь-які переносні носії інформації.
- C2LOP.gen!g: троянська програма для зміни налаштувань браузера, показу реклами.
- C2LOP.P: видозмінена версія троянської програми C2LOP.gen!g.
- Dialplatform.B: троянська програма, створена для мережі Dial-up і виконує виклики на окремі привілейовані номери.
- Dontovo.A: троянська програма для завантаження довільних шкідливих файлів.
- Fakerean: вірус, котрий вбудовує себе у код антивірусних програм. Видозмінює себе в залежності від операційної системи.
- Instantaccess: троянська програма, здійснює несанкціоновану демонстрацію відео на інфікованому ПК.
- Lolyda.AA1: троянська програма, що надсилає інформацію про ігрові акаунти, що існують на зараженому комп'ютері.
- Lolyda.AA2: видозмінена версія троянських програм сімейства Lolyda
- Lolyda.AA3: видозмінена версія троянських програм сімейства Lolyda.
- Lolyda.AT: видозмінена версія троянських програм сімейства Lolyda, здійснює викрадення логінів та паролів ігрових акаунтів.
- Malex.gen!J: троянська програма, дозволяє контролювати комп'ютер без відому користувача.
- Obfuscator.AD: троянська програма.
- Rbot!gen: backdoor для віддаленого контролю інфікованого комп'ютера.
- Skintrim.N: троянська програма для завантаження шкідливого ПЗ та його оновлень з заздалегідь визначених веб-ресурсів.
- Swizzor.gen!E: троянська програма для завантаження додаткового ПЗ з веб-ресурсів.

- Swizzor,gen!I: видозмінена версія Swizzor.gen!E.
- VB.AT: вірус для зараження окремих програм на мові Visual Basic.
- Wintrim.BX: троянська програма для завантаження додаткового шкідливого ПЗ.
- Yuner.A: мережевий хробак для платформи Windows.

Вибірка була розділена на навчальну та тестову у пропорції 3:1, тобто 6226 зразків використовувались для навчання та 3113 для тестування отриманої мережі.

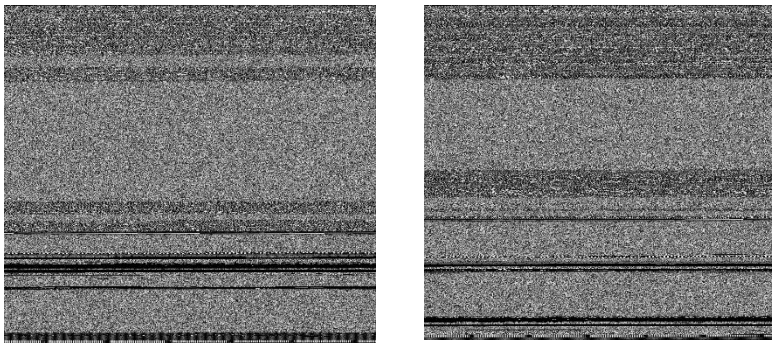

Рисунок 3.1.1 — Приклади зразків шкідливого ПЗ класу C2LOP.gen!g

3.2. Короткий опис програмного забезпечення

3.2.1. Бібліотека TensorFlow

Tensorflow це популярна бібліотека для глибокого машинного навчання, котра розроблена компанією Google та використовується при побудові і тренуванні нейронних мереж для різних прикладних задач.

Основними елементами нейронних мереж є 3 компоненти: шар входу, приховані шари та шар виходу. Шари входу та виходу є відповідно початковими і кінцевими частинами мережі. Вхідний шар містить початкові дані, що будуть використовуються для задачі розпізнавання, регресії, тощо, вихідний шар буде містити значення, котрі були отримані внаслідок обробки вхідної інформації прихованими шарами мережі. Зазвичай у мережах для глибинного навчання існує велика кількість прихованих шарів між вхідним та вихідним. Такі шари містять у собі значення ваг і виконують обчислення з використанням вхідних даних доки не будуть отримані значення для вихідного

шару. Існують деякі параметри, котрі можливо змінювати у прихованих шарах, серед них:

- кількість нейронів: вибране значення буде визначати кількість ваг, яку буде мати окремий шар.
- функція активації: зазвичай обирають однакову функцію активації для всіх прихованих шарів та окрему функцію для вихідного шару, залежно від задачі, що вирішується.
- параметри регуляризації: для запобігання перенавчанню моделі при її побудові можливо застосувати методи регуляризації ваг або виключення (англ. dropout).

Окремі типи шарів можуть мати додаткові, характерні для нього параметри, так згортковий шар буде мати значення розміру ядра та їх кількості.

Кожен шар нейронної мережі виконує певні обчислення з використанням даних, котрі надходять до нього з попереднього, обчислені значення відповідно слугують входами для наступного шару. Такий процес описаний через тензори (англ. tensors) та операції (англ. operations) у бібліотеці TensorFlow, котрі є базовими для опису нейронних мереж з її використанням.

Тензори представлені як багатовимірні масиви, котрі містять дані окремого типу (цілі числа, числа з рухомою комою, тощо). Всі числові дані у мережі представлені тензорами. Кожен тензор може мати різну кількість вимірів, але всередині мережі він буде мати конкретний фіксований розмір. В окремих ситуаціях тензор може бути пустий і приймати на вхід дані ззовні, таким тензором є вхідний, він приймає початкові дані для подальшого навчання прихованих шарів нейромережі.

Операції визначаються як певні обчислення, що дають певне результуюче значення. Так, операції приймають на вхід тензори, виконують обчислення над ними та повертають результат також у вигляді тензора. Тому,

операції пов’язують всі тензори нейромережі у вигляді певного конвеєра обчислень від вхідного тензора до тензора вихідного шару.

В нейронній мережі, котра побудована з використанням бібліотеки TensorFlow кожен шар буде містити тензор значень нейронів та тензор ваг, операція обчислення значень нейронів приймає на вхід ці два тензори і повертає тензор значень нейронів для наступного шару. Сама операція представляє собою множення матриць ваги та значень нейронів і застосування функції активації до результату. Таким чином, тензор поточного шару є результатом операції попереднього шару і є входом для операції наступного шару. Своєрідний "потік" (англ. flow) тензорів через операції від вхідного шару мережі до вихідного виправдовує назву бібліотеки TensorFlow.

3.2.2. Бібліотека Keras

Бібліотека TensorFlow надає можливість представлення елементів мережі у вигляді тензорів та операцій над ними, бібліотека Keras є своєрідною абстракцією над бібліотекою Tensorflow, де можливо побудувати нейромережу використовуючи окремі шари. Кожен шар є сукупністю декілкох об’єктів: вхідний тензор зі значеннями нейронів, тензор зі значеннями ваг зв’язків, внутрішня операція котра обчислює результуючий тензор, котрий є вхідним для наступного шару.

При побудові нейронної мережі з використанням Keras необхідно описувати окремі шари а також зв’язки між створеними шарами. Для всіх шарів вхідною інформацією слугує вихід попереднього шару, крім вхідного. Keras надає 2 моделі для побудови мереж: послідовне (англ. sequential) та

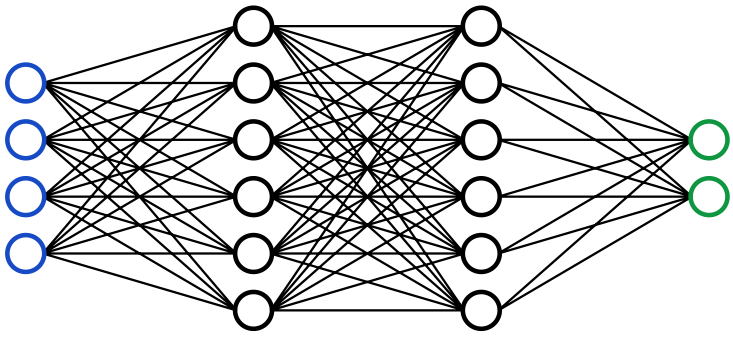

Рисунок 3.2.1 — Приклад архітектури мережі для реалізації

```
model = Sequential()
model.add(Dense(4, input_shape = (4,), activation = 'relu')
model.add(Dense(6, activation = 'relu'))
model.add(Dense(6, activation = 'relu'))
model.add(Dense(2, activation = softmax)
```

Рисунок 3.2.2 — Опис мережі на Рисунку 3.2.1 з використанням Keras

функціональне (англ. functonal). Якщо бажана мережа може бути послідовно побудована, тобто існує одна послідовність від вхідного до вихідного шару і окремі шари завжди мають тільки один вхід, то таку модель можливо побудувати з використанням sequential підходу.

Таким чином, процес побудови sequential моделі досить простий і включає два етапи: ініціалізація моделі та додання нових шарів через метод add().

Для більш складних моделей, коли можливе існування декількох вхідних або вихідних шарів, або кількох зв'язків між шарами доцільно використовувати functional метод побудови мережі. Відмінністю від послідовної побудови є те, що необхідно явно вказувати зв'язок між окремими створеними шарами мережі, таких зв'язків може бути декілька.

Після того як модель побудована з використанням одного з вищезазначених методів необхідно скомпілювати модель для подальшого її навчання. Процес компіляції передбачає додання до моделі декількох параметрів: функцію втрат для вихідного шару, метод оптимізації майбутньої нейронної мережі, метрики для оцінки якості моделі.

```
model.compile(loss = 'binary_crossentropy', optimizer = 'rmsprop', metrics = ['accuracy'])
```

Рисунок 3.2.3 — Приклад компіляції моделі

На Рисунку 3.2.3 описана компіляція моделі з використанням логістичної функції втрат, методом оптимізації RMSProp та метрики якості — точність (відношення правильно спрогнозованих тестових об'єктів до всіх існуючих). У випадку декількох вихідних шарів і необхідності різних функцій втрат слід

вказувати список функцій у правильному порядку, в якому знаходяться вихідні шари.

3.2.3. Архітектура нейронної мережі

Для реалізації нейронної мережі була використана послідовна модель побудови бібліотеки Keras. Модель мережі має наступні додані шари:

- початковий кватерніонний згортковий шар QConv2D з кількістю фільтрів — 32, розмірністю ядра 3×3, вхідною розмірністю зображення $64 \times 64 \times 3$ та фукцією активації випрямлений лінійний вузол (англ. rectified linear unit, ReLU)
- кватерніонний згортковий шар QConv2D з кількістю фільтрів — 32, розмірністю ядра 3×3 та фукцією активації ReLU
- шар максимізаційного агрегування з розмірністю ядра 2×2
- регуляризація виключенням (англ. dropout) для 50% нейронів попереднього шару
- два послідовних кватерніонних згорткових шари QConv2D з кількістю фільтрів — 64, розмірністю ядра 3×3 та фукцією активації ReLU
- шар максимізаційного агрегування з розмірністю ядра 2×2
- регуляризація виключенням (англ. dropout) для 50% нейронів попереднього шару
- шар зменшення розмірності зображення до одновимірного вектору Flatten
- повнозв'язний шар QDense, що складється с 512 нейронів та функцією активації ReLU,
- регуляризація виключенням (англ. dropout) для 50% нейронів попереднього шару
- вихідний повнозв'язний шар QDense з кількістю нейронів, що дорівнює кількості класів для розпізнавання та логістичною функцією втрат softmax.

Як метод оптимізації параметрів мережі був обраний оптимізатор Adam, котрий об'єднує в собі властивості методів інерції та RMSProp. Обчислення інерції виконується за наступною формулою:

$$m_{new} = \beta_1 * m_{old} - (1 - \beta_1) * \frac{\partial(Loss)}{\partial(W_{old})}, \quad (19)$$

де m_{new} — значення інерції для поточної ітерації навчання, m_{old} — значення інерції отримане на попередньому кроці, коефіцієнт β_1 слугує для обмеження швидкості навчання моделі, $\frac{\partial(Loss)}{\partial(W_{old})}$ — похідна функції втрат відносно ваг моделі.

Для динамічної зміни коефіцієнта швидкості навчання (англ. learning rate) необхідно обчислювати значення кешу за наступним виразом:

$$cache_{new} = \beta_2 * cache_{old} - (1 - \beta_2) * \left(\frac{\partial(Loss)}{\partial(W_{old})}\right)^2, \quad (20)$$

де $cache_{new}$ — значення кешу для поточної ітерації навчання мережі, $cache_{old}$ — значення кешу отримане на попередньому кроці, коефіцієнт β_2 слугує для плавного оновлення коефіцієнту швидкості навчання моделі, дозволяючи проводити тренування довший період, порівняно з оптимізатором Adagrad, $\left(\frac{\partial(Loss)}{\partial(W_{old})}\right)^2$ — квадрат похідної функції втрат відносно ваг моделі.

$$W_{new} = W_{old} - \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * m_{new}, \quad (21)$$

де W_{new} — значення ваг для поточної ітерації навчання мережі, W_{old} — значення ваг отримане на попередньому кроці, коефіцієнт α — коефіцієнту швидкості навчання моделі, $cache_{new}$ — значення кешу для поточної ітерації навчання мережі, ϵ — значення для запобігання ділення на 0, m_{new} — значення інерції для поточної ітерації навчання.

Для значень β_1, β_2 та ϵ використовувались рекомендовані значення 0.9, 0.99 та $1 \cdot 10^{-8}$ відповідно.

Для оцінки якості моделі використовувався критерій точності, тобто відношення кількості правильно класифікованих тестових зразків до всієї кількості тестових зразків.

Функція втрат моделі — категорійна крос-ентропія, при якій спочатку до значення кожного вихідного нейрона мережі застосовується логістична функція softmax:

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=0}^{K} e^{z_j}}, \tag{22}$$

де z_i — значення окремого вихідного нейрону мережі, K — кількість вихідних значень мереже, що дорівнює кількості класів розпізнавання моделі.

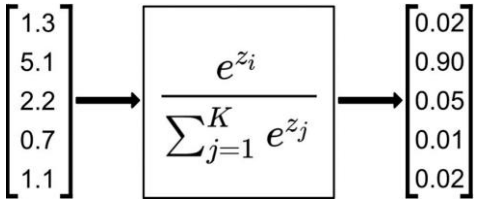

Рисунок 3.2.4. Приклад обчислення функції softmax

Вихідні значення є ймовірностями належності окремого зразку до одного з відомих класів розпізнавання. Наступним кроком є застосування логистичної функції до всіх значень ймовірності та сума результатів:

$$Loss = \sum_{j=1}^{N} (Y_j)(-\log(Y_{pred_j})) + (1 - Y_j)(-log(1 - Y_{pred_j})), \tag{23}$$

де $Loss$ — значення функції втрат моделі, Y_j — бажане значення ймовірності належності зразка до класу розпізнавання, Y_{pred_j} — значення ймовірності належності спрогнозоване мережею.

3.3. Результати машинного навчання

В результаті застосування тестової вибірки даних з 9339 зразків шкідливого програмного забезпечення як вхідних даних побудованої нейронної мережі з використанням кватерніонних шарів до були отримані графіки точності та графіки втрат моделі за епохами навчання на навчальних та тестових даних відповідно.

Точність для навчальної вибірки даних змінювалась від значення 0.4465 до максимального — 0.9419 на 25-й епосі навчання. Для тестової вибірки мінімальне і максимальне значення точності розпізнавання мали значення 0.6630 та 0.9561 відповідно. Отримані значення свідчать про високу точність моделі розпізнавання.

Значення функцій втрат для навчальної вибірки змінювались від максимального значення — 1.8802 до мінімального — 0.1719. Для тестової вибірки максимальне — 1.0375 та мінімальне — 0.1445 значення функцій втрат відповідно.

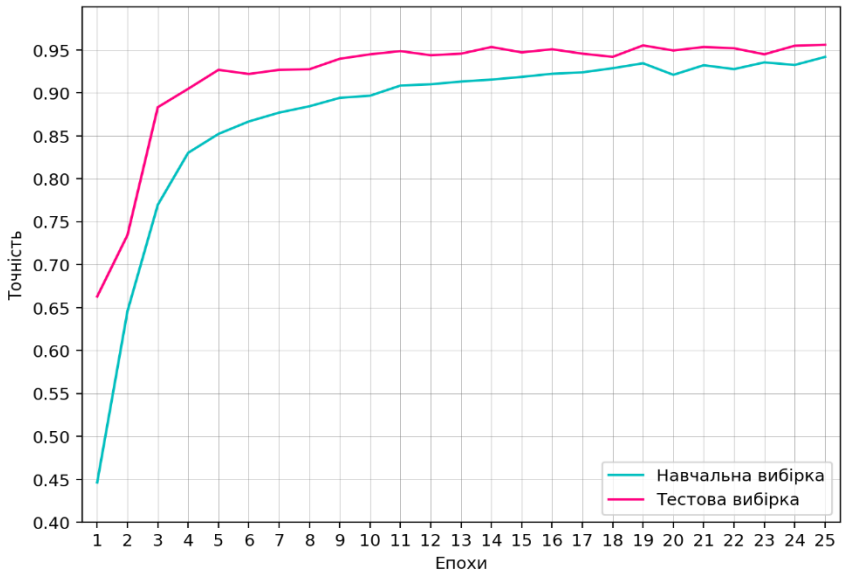

Рисунок 3.3.1 — Залежність точності розпізнавання моделі від епох навчання

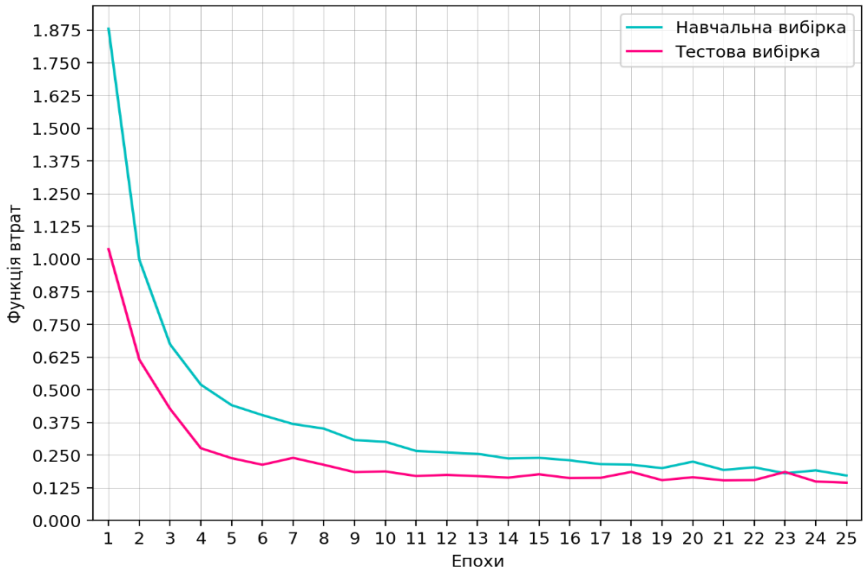

Рисунок 3.3.2 — Залежність функції втрат мережі від епох навчання

Графіки демонструють підвищення якості моделі зі збільшенням кількості епох для навчання моделі. Подальший ріст точності є можливим, але існує ризик перенавчання (англ. overfitting) мережі, при якому значення функцій втрат для тестових даних вже почнуть зростати зі збільшенням епох.

ВИСНОВКИ

Дана робота присвячена проблемі дослідження та класифікації шкідливого програмного забезпечення з використанням штучних нейронних мереж. Зважаючи на існуючі досягнення у області інтелектуальних систем розпізнавання зображень, методом для аналізу стали згорткові нейронні мережі, що мали кватерніонні елементи у своїй реалізації.

У процесі дослідження екстракції ознакових спостережень був обраний метод, що дозволяє перетворювати бінарні файли програмного забезпечення у чорно-білі графічні зображення, які у подальшому слугували вхідними даними для згорткової нейронної мережі.

Для програмної реалізації алгоритмів була використана мова Python та окремі бібліотеки для машинного навчання TensorFlow та Keras. Застосовуючи послідовну модель побудови була отримана нейронна мережа, що містила кватерніонні згорткові та повнозв'язні шари, шари максимізаційного агрегування та зменшення розмірності даних, для регуляризації використовувалось виключення частки нейронів окремих частин мережі.

Сформована вибірка даних для розпізнавання інтелектуальною системою загалом містила 9339 зразків зображень шкідливого програмного забезпечення, розділених на 25 класів розпізнавання. Процес тренування здійснювався впродовж 25-ти епох. В результаті навчання створеної мережі була отримана точність 0.9561 на тестових даних обсягом 3113 зразків.

Таким чином задача забезпечення оптимальних показників точності виявлення та класифікації шкідливих зразків була виконана, тому побудова інтелектуальної системи розпізнавання шкідливого програмного забезпечення може вважатись успішною.

СПИСОК ЛІТЕРАТУРИ

1. “The AV-TEST Security Report”, 2019/2020.: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2019-2020.pdf., AV-TEST, 2019
2. X. Wang, D. Xuan, X. Fu, A. Champion, and W. Yu, “Detecting worms via mining dynamic program execution”, Proc. 3rd, SecureComm, 2007.
3. M. Karresand, “Separating Trojan horses, viruses, and worms - A proposed taxonomy of software weapons”, IEEE, 2003
4. N. Scaife, K. R. B. Butler, P. Traynor, and H. Carter, “CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data”, Proc. – Int. Conf. Dist. Comp. Sys., 2016.
5. A. Zaki and B. Humphrey, “Unveiling the kernel : Rootkit discovery using selective automated kernel memory differencing”, Virus Bulletin, 2014.
6. Y. Ye, S. S. Iyengar, D. Adjeroh, and T. Li, “A Survey on Malware Detection Using Data Mining Techniques”, ACM Comp. Survey, 2017.
7. N. B. Ithnin and G. A. N. Mohamed, “Survey on Representation Techniques for Malware Detection System”, American Journal of Applied Science, 2017.
8. A. Damodaran, T. H. Austin, F. C. A. Visaggio, Di Troia, and M. Stamp, “A comparison of static, dynamic, and hybrid analysis for malware detection”, J. Comp. Vir. Hack. Tech., 2017.
9. M. Chowdhury and A. Rahman, “Malware Analysis and Detection Using Data Mining and Machine Learning Classification”, 2018.
10. Z. Bazrafshan, S. M. H. Fard, H. Hashemi, and A. Hamzeh, “A survey on heuristic malware detection techniques”, IKT 2013.
11. I. You and K. Yim, “Malware obfuscation techniques: A brief survey”, BWCCA 2010.
12. R. Hosseini and A. Souri, “A state-of-the-art survey of malware detection approaches using data mining techniques”, Hum. Comp. Inf. Scie., 2018.

13. M. Alazab, P. Watters, and S. Venkataraman, “Towards understanding malware behaviour by the extraction of API calls”, CTC 2010.
14. W. Wong and M. Stamp, “Hunting for metamorphic engines”, Journal of Computer Virology., 2006.
15. S. N. Das, P. K. Vijayaraghavan and M. Mathew, “An Approach for Optimal Feature Subset Selection using a New Term Weighting Scheme and Mutual Information”, Int. J. Adv. Sci. Eng. Inf. Tech., 2011.
16. M. Hafiz, M. R. Mokhtar, and M. Yusof, “A Review of Predictive Analytic Applications of Bayesian Network”, Int. J. Adv. Sci. Eng. Inf. Tech., 2016.
17. D. Ucci, R. Baldoni and L. Aniello, “Survey on the Usage of Machine Learning Techniques for Malware Analysis”, arXiv1710.08189, 2018.
18. D. Kirat and G. Vigna, “MalGene”, Proceedings of the CCS 2015.
19. T. Abou-Assaleh, R. Sweidan, V. Keselj, and N. Cercone, “N-grambased detection of new malicious code”, Proc. COMPSAC 2004., 2004.
20. 20E. Gandotra, S. Sofat, and D. Bansal, “Malware Analysis and Classification: A Survey”, Journal of Information Security, 2014.
21. H. Hashemi and A. Hamzeh, “Visual malware detection using local malicious pattern”, J. Comp. Vir. Hack. Tech., 2018.
22. Z. Salehi, M. Ghiasi, and A. Sami, “Using feature generation from API calls for malware detection”, Computer Fraud and Security, 2014.
23. S. Z. M. Shaid and M. A. Maarof, “Malware behaviour visualization”, Journal of Technology, 2014.
24. Y. Cheng, J. An, W. Huang, and W. Fan, “A Shellcode Detection Method Based on Full Native API Sequence and Support Vector Machine”, IOP, 2017.
25. K. S. Han, E. G. Im, and I. K. Kim, “Malware classification methods using API sequence characteristics”, Lect. Notes Elec. Eng., 2012.
26. Q. Chen and R. A. Bridges, “Automated Behavioral Analysis of Malware A Case Study of WannaCry Ransomware”, arXiv Prepr. arXiv1709.08753, 2017

27. G. Liang, C. Dai, and J. Pang, “A Behavior-Based Malware Variant Classification Technique”, International Journal of Information and Education Technology, 2016.
28. A. Mohaisen, M. Mohaisen, and O. Alrawi, “AMAL: High-fidelity, behavior-based automated malware analysis and classification”, Computer Security, 2015.
29. I. Santos, P. G. Bringas, X. Ugarte-Pedrero, and F. Brezo “Opcode sequences as representation of executables for data-mining-based unknown malware detection”, Information Science, 2013.
30. O. Alrawi and A. Mohaisen, “Unveiling Zeus: automated classification of malware samples”, Proceedings of the 22nd International Conference on World Wide Web companion, 2013.
31. Y. Ki, H. K. Kim, and E. Kim, “A novel approach to detect malware based on API call sequence analysis”, International Journal of Distributed Sensor Networks, 2015.
32. H. S. Galal, M. A. Atiea, and Y. B. Mahdy, “Behavior-based features model for malware detection”, Journal of Computer Virology and Hacking Techniques, 2016.
33. C.-I. Fan, Y.-F. Tseng, C.-H. Chou, and H.-W. Hsiao, “Malware Detection Systems Based on API Log Data Mining”, 2015 IEEE, 2015.
34. X. Chen, Z. Morley Mao, J. Andersen, J. Nazario, and M. Bailey, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware”, Proc. Int. Conf. Depend. Syst. Netw., 2008.
35. M. Sikorski and A. Honig, Practical malware analysis: the hands-on guide to dissecting malicious software. No Starch Press. 2012.
36. M. Egele, C. Kruegel, E. Kirda, and T. Scholte, “A survey on automated dynamic malware-analysis techniques and tools”, ACM Computer Survey, 2012.
37. M. Eskandari, S. Hashemi, and Z. Khorshidpour, “HDM-Analyser: a hybrid analysis approach based on data mining techniques for malware detection”, Journal of Computer Virology and Hacking Techniques, 2013.

38. R. Islam, S. Versteeg, L. M. Batten, and R. Tian, “Classification of malware based on integrated static and dynamic features”, J. Net. Comp. Apps., 2013.
39. P. V. Shijo and A. Salim, “Integrated static and dynamic analysis for malware detection”, Procedia Computer Science, 2015
40. X. Ma, J. Jiang, W. Yang, and Q. Biao, “Using multi-features to reduce false positive in malware classification”, Proc. ITNEC 2016.
41. C. Rathnayaka and A. Jamdagni, “An efficient approach for advanced malware analysis using memory forensic technique”, Proc. IEEE International Conference on Embedded Software Systems, 2017.
42. I. Santos, F. Brezo, P. G. Bringas, J. Nieves, and J. Devesa, “OPEM: A static-dynamic approach for machine-learning-based malware detection”, Adv. Int. Syst. Comp., 2013.
43. A. Hayon and T. Teller, “Enhancing Automated Malware Analysis Machines with Memory Analysis Report”, Black Hat USA, 2014.
44. M. Cohen and J. Stüttgen “Anti-forensic resilient memory acquisition”, Digital Investigation, 2013.
45. K.-W. P. Choi, Yu-Seong Kim, and Sang-Hoon, “Toward Semantic Gap-less Memory Dump for Malware Analysis”, 2016.
46. G. Willems, F. Freiling, and T. Holz, “Toward automated dynamic malware analysis using CWSandbox”, IEEE, 2007.
47. M. H. Ligh, M. Richard, B. Hartstein, and S. Adair, Malware analyst’s cookbook and DVD: tools and techniques for fighting malicious code., 2011.
48. R. Mosli, Y. Pan, B. Yuan, and R. Li, “Automated malware detection using artifacts in forensic memory images”, IEEE HST 2016.
49. R. Mosli, Y. Pan, B. Yuan, and R. Li, “A behavior-based approach for malware detection”, IFIP 2017.
50. Adlice Software, “Rootkits hooks”, 2014: https://www.adlice.com/.
51. A. Hosseini, “Ten Process Injection Techniques: A Technical Survey Of Common And Trending Process Injection Techniques”, 2017.

52. S. Kim, K. Lee, J. Park, K. Yim, and I. You, “A Brief Survey on Rootkit Techniques in Malicious Codes”, J. In. Serv. Inf. Sec., 2012.
53. J. Butler, J. Pinkston, and J. L. Undercoffer, “Hidden processes: The implication for intrusion detection”, IEEE Systems Workshop, 2003.
54. J. Berdajs and Z. Bosnic, “Extending applications using an advanced approach to DLL injection and API hooking”, Software: Practice and Experience, 2010.
55. K. Lee, K. Kim, B. N. Noh, and H. Hwang, “Robust bootstrapping memory analysis against anti-forensics”, Digital Investigation, 2016.
56. A. Schuster, “Searching for processes and threads in Microsoft Windows memory dumps”, Digital Investigation, 2006.
57. R. H. Arpaci-Dusseau, S. T. Jones, and A. C. Arpaci-Dusseau, “VMM-based hidden process detection and identification using Lycosid”, Proc. VEE 2008.
58. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, “Deep residual learning for image recognition”, in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016.
59. Xu, Y., Zhang, Yu, L., H., Nguyen, Xu, H., T.: Vector sparse representation of color image using quaternion matrix analysis. IEEE Tr. Img. Proc., 2015
60. Yu, L., Xu, H., Xu, Y., Zhang, H.: Quaternion-based sparse representation of color image. IEEE International Conference 2013
61. Gai, S., L., Yang, Wang, G., Yang, P.: “Sparse representation based on vector extension of reduced quaternion matrix for multiscale image denoising”, IET 2016
62. Bengio, Glorot, X. Y.: Understanding the difficulty of training deep feedforward neural networks. J. Mach. Lear. Res., 2010
63. T. Parcollet, M. Morchid., G Linares, “A survey of quaternion neural networks”, Springer Nature B.V., 2019
64. Z. Wang, J. Gu, J. Kuen, B. Shuai, L. Ma, Wang, G. A. Shahroudy, T. Liu, X. Wang, and J. Cai, “Recent advances in convolutional neural networks”, Pattern Recognition, 2018.

65. Kalash, M., Rochan, M., Mohammed, N., Bruce, N. D. B., Wang, Y., & Iqbal, F. (2018). Malware Classification with Deep Convolutional Neural Networks. 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS).

ДОДАТКИ

Додаток А

Реалізація архітектури згорткової мережі

```
from __future__ import print_function
from keras.models import Sequential
from keras.layers import MaxPooling2D, Dense, Flatten, Activation, Dropout, Conv2D
from keras.optimizers import adam
from sklearn.model_selection import train_test_split
from quaternion_layers import QConv1D, QConv2D, QConv3D, QDense
from keras.preprocessing.image import ImageDataGenerator
import pickle, os
import numpy as np

pathRoot = "/malimg_paper_dataset_imgs"
genData = ImageDataGenerator().flow_from_directory(target_size=(64, 64), directory=pathRoot)
images, lbs = next(genData)
XtrainData, XtestData, YtrainData, YtestData = train_test_split(images/255., lbs, test_size=0.30)
numOfClasses = 25
eps = 25

reluActivation = Activation('relu')
threeByThreeConv = (3, 3)
dropOut = Dropout(0.25)
model = Sequential()
model.add(QConv2D(32, threeByThreeConv,  padding='same', input_shape=(64,64,3)))
model.add(reluActivation)
model.add(QConv2D(32, threeByThreeConv))
model.add(reluActivation)
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(dropOut)
model.add(QConv2D(64, threeByThreeConv, padding='same'))
model.add(reluActivation)
model.add(QConv2D(64, threeByThreeConv))
model.add(reluActivation)
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(dropOut)
model.add(Flatten())
model.add(QDense(512))
model.add(reluActivation)
model.add(Dropout(0.5))
model.add(Dense(numOfClasses))
model.add(Activation('softmax'))
model.compile(metrics=['accuracy'], optimizer='adam', loss='categorical_crossentropy')
model.fit(XtrainData, YtrainData,
          validation_data=(XtestData, YtestData),
          epochs=eps,
          shuffle=True)

modelScores = model.evaluate(XtestData, YtestData, verbose=1)
```

Додаток Б

Реалізація кватерніонних згорткових шарів QConv та QConv2D

```
class QConv(Layer):

    def __init__(self, rank,
                 filters,
                 kernel_size,
                 strides=1,
                 padding='valid',
                 data_format=None,
                 dilation_rate=1,
                 activation=None,
                 use_bias=True,
                 bias_initializer='zeros',
                 kernel_regularizer=None,
                 bias_regularizer=None,
                 activity_regularizer=None,
                 kernel_constraint=None,
                 bias_constraint=None,
                 **kwargs):
        super(QConv, self).__init__(**kwargs)
        self.rank = rank
        self.filters = filters
        self.kernel_size = conv_utils.normalize_tuple(kernel_size, rank, 'kernel_size')
        self.strides = conv_utils.normalize_tuple(strides, rank, 'strides')
        self.padding = conv_utils.normalize_padding(padding)
        self.data_format = 'channels_last' if rank == 1 else conv_utils.normalize_data_format(data_format)
        self.dilation_rate = conv_utils.normalize_tuple(dilation_rate, rank, 'dilation_rate')
        self.activation = activations.get(activation)
        self.use_bias = use_bias
        self.bias_initializer = initializers.get(bias_initializer)
        self.kernel_regularizer = regularizers.get(kernel_regularizer)
        self.bias_regularizer = regularizers.get(bias_regularizer)
        self.activity_regularizer = regularizers.get(activity_regularizer)
        self.kernel_constraint = constraints.get(kernel_constraint)
        self.bias_constraint = constraints.get(bias_constraint)
        self.input_spec = InputSpec(ndim=self.rank + 2)

    def build(self, input_shape):
        if self.data_format == 'channels_first':
            channel_axis = 1
        else:
            channel_axis = -1
        if input_shape[channel_axis] is None:
            raise ValueError('The channel dimension of the inputs '
                             'should be defined. Found `None`.')
        input_dim = input_shape[channel_axis] // 3
        self.kernel_shape = self.kernel_size + (input_dim , self.filters)

        kern_init = QInit(
            kernel_size=self.kernel_size,
            input_dim=input_dim,
            weight_dim=self.rank,
            nb_filters=self.filters,
        )

        self.kernel = self.add_weight(
            self.kernel_shape,
            initializer=kern_init,
            name='kernel',
            regularizer=self.kernel_regularizer,
            constraint=self.kernel_constraint,
            trainable = True
        )

        if self.use_bias:
            bias_shape = (3 * self.filters,)
            self.bias = self.add_weight(
                bias_shape,
                initializer=self.bias_initializer,
                name='bias',
                regularizer=self.bias_regularizer,
                constraint=self.bias_constraint,
                trainable = True
            )

        else:
            self.bias = None

        self.input_spec = InputSpec(ndim=self.rank + 2,
                                    axes={channel_axis: input_dim * 3})
        self.built = True
```

```
def call(self, inputs):
    channel_axis = 1 if self.data_format == 'channels_first' else -1
    input_dim    = K.shape(inputs)[channel_axis] // 3
    if self.rank == 1:
        f_phase   = self.kernel[:, :, :self.filters]
        f_modulus   = self.kernel[:, :, self.filters:]

    elif self.rank == 2:
        f_phase   = self.kernel[:,:, :, :self.filters]
        f_modulus   = self.kernel[:, :, :, self.filters:]

    elif self.rank == 3:
        f_phase   = self.kernel[:,:,:, :, :self.filters]
        f_modulus   = self.kernel[:,:,:, :, self.filters:]

    f_phase1 = tf.cos(f_phase)
    f_phase2 = tf.sin(f_phase)*(3**0.5/3)
    convArgs = {"strides":       self.strides[0]       if self.rank == 1 else self.strides,
                "padding":       self.padding,
                "data_format":   self.data_format,
                "dilation_rate": self.dilation_rate[0] if self.rank == 1 else self.dilation_rate}
    convFunc = {1: K.conv1d,
                2: K.conv2d,
                3: K.conv3d}[self.rank]


    f1 = (K.pow(f_phase1,2)-K.pow(f_phase2,2))*f_modulus
    f2 = (2*(K.pow(f_phase2,2)-f_phase2*f_phase1))*f_modulus
    f3 = (2*(K.pow(f_phase2,2)+f_phase2*f_phase1))*f_modulus
    f4 = (2*(K.pow(f_phase2,2)+f_phase2*f_phase1))*f_modulus
    f5 = (K.pow(f_phase1,2)-K.pow(f_phase2,2))*f_modulus
    f6 = (2*(K.pow(f_phase2,2)-f_phase2*f_phase1))*f_modulus
    f7 = (2*(K.pow(f_phase2,2)-f_phase2*f_phase1))*f_modulus
    f8 = (2*(K.pow(f_phase2,2)+f_phase2*f_phase1))*f_modulus
    f9 = (K.pow(f_phase1,2)-K.pow(f_phase2,2))*f_modulus

    f1._keras_shape = self.kernel_shape
    f2._keras_shape = self.kernel_shape
    f3._keras_shape = self.kernel_shape
    f4._keras_shape = self.kernel_shape
    f5._keras_shape = self.kernel_shape
    f6._keras_shape = self.kernel_shape
    f7._keras_shape = self.kernel_shape
    f8._keras_shape = self.kernel_shape
    f9._keras_shape = self.kernel_shape
    f_phase1._keras_shape = self.kernel_shape
    f_phase2._keras_shape = self.kernel_shape

    matrix1 = K.concatenate([f1, f2, f3], axis=-2)
    matrix2 = K.concatenate([f4, f5, f6], axis=-2)
    matrix3 = K.concatenate([f7, f8, f9], axis=-2)
    matrix = K.concatenate([matrix1, matrix2, matrix3], axis=-1)
    matrix._keras_shape = self.kernel_size + (3 * input_dim, 3 * self.filters)
    output = convFunc(inputs, matrix, **convArgs)

    if self.use_bias:
        output = K.bias_add(
            output,
            self.bias,
            data_format=self.data_format
        )

    if self.activation is not None:
        output = self.activation(output)

    return output

def compute_output_shape(self, input_shape):
    if self.data_format == 'channels_last':
        space = input_shape[1:-1]
        new_space = []
        for i in range(len(space)):
            new_dim = conv_utils.conv_output_length(
                space[i],
                self.kernel_size[i],
                padding=self.padding,
                stride=self.strides[i],
                dilation=self.dilation_rate[i])
            new_space.append(new_dim)
        return (input_shape[0],) + tuple(new_space) + (3 * self.filters,)
```

```
        if self.data_format == 'channels_first':
            space = input_shape[2:]
            new_space = []
            for i in range(len(space)):
                new_dim = conv_utils.conv_output_length(
                    space[i],
                    self.kernel_size[i],
                    padding=self.padding,
                    stride=self.strides[i],
                    dilation=self.dilation_rate[i])
                new_space.append(new_dim)
            return (input_shape[0],) + (3 * self.filters,) + tuple(new_space)

    def get_config(self):
        config = {
            'rank': self.rank,
            'filters': self.filters,
            'kernel_size': self.kernel_size,
            'strides': self.strides,
            'padding': self.padding,
            'data_format': self.data_format,
            'dilation_rate': self.dilation_rate,
            'activation': activations.serialize(self.activation),
            'use_bias': self.use_bias,
            'bias_initializer': initializers.serialize(self.bias_initializer),
            'kernel_regularizer': regularizers.serialize(self.kernel_regularizer),
            'bias_regularizer': regularizers.serialize(self.bias_regularizer),
            'activity_regularizer': regularizers.serialize(self.activity_regularizer),
            'kernel_constraint': constraints.serialize(self.kernel_constraint),
            'bias_constraint': constraints.serialize(self.bias_constraint)
        }
        base_config = super(QConv, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))

class QConv2D(QConv):

    def __init__(self, filters, kernel_size,
                 strides=(1, 1), padding='valid',
                 data_format=None,
                 dilation_rate=(1, 1),
                 activation=None,
                 use_bias=True,
                 bias_initializer='zeros',
                 kernel_regularizer=None,
                 bias_regularizer=None,
                 activity_regularizer=None,
                 kernel_constraint=None,
                 bias_constraint=None,
                 seed=None,
                 **kwargs):
        super(QConv2D, self).__init__(
            rank=2, filters=filters,
            kernel_size=kernel_size,
            strides=strides, padding=padding,
            data_format=data_format,
            dilation_rate=dilation_rate,
            activation=activation,
            use_bias=use_bias, bias_initializer=bias_initializer,
            kernel_regularizer=kernel_regularizer,
            bias_regularizer=bias_regularizer,
            activity_regularizer=activity_regularizer,
            kernel_constraint=kernel_constraint,
            bias_constraint=bias_constraint,
            **kwargs)

    def get_config(self):
        config = super(QConv2D, self).get_config()
        config.pop('rank')
        return config
```

Додаток В

Реалізація кватерніонних повнозв'язних шарів QDense

```
class QDense(Layer):

    def __init__(self, units,
                 activation=None,
                 use_bias=True,
                 bias_initializer='zeros',
                 kernel_regularizer=None,
                 bias_regularizer=None,
                 activity_regularizer=None,
                 kernel_constraint=None,
                 bias_constraint=None,
                 **kwargs):
        if 'input_shape' not in kwargs and 'input_dim' in kwargs:
            kwargs['input_shape'] = (kwargs.pop('input_dim'),)
        super(QDense, self).__init__(**kwargs)
        self.units = units
        self.activation = activations.get(activation)
        self.use_bias = use_bias
        self.bias_initializer = initializers.get(bias_initializer)
        self.kernel_regularizer = regularizers.get(kernel_regularizer)
        self.bias_regularizer = regularizers.get(bias_regularizer)
        self.activity_regularizer = regularizers.get(activity_regularizer)
        self.kernel_constraint = constraints.get(kernel_constraint)
        self.bias_constraint = constraints.get(bias_constraint)
        self.input_spec = InputSpec(ndim=2)
        self.supports_masking = True

    def build(self, input_shape):

        assert len(input_shape) == 2
        assert input_shape[-1] % 2 == 0
        input_dim = input_shape[-1] // 3
        data_format = K.image_data_format()
        kernel_shape = (input_dim, self.units)
        fan_in, fan_out = initializers._compute_fans(
            kernel_shape,
            data_format=data_format
        )
        s = np.sqrt(1. / fan_in)

        def init_phase(shape, dtype=None):
            return np.random.normal(
                size=kernel_shape,
                loc=0,
                scale=np.pi/2,
            )

        def init_modulus(shape, dtype=None):
            return np.random.normal(
                size=kernel_shape,
                loc=0,
                scale=s
      phase_init = init_phase
      modulus_init = init_modulus

      self.phase_kernel = self.add_weight(
          shape=kernel_shape,
          initializer=phase_init,
          name='phase_kernel',
          regularizer=self.kernel_regularizer,
          constraint=self.kernel_constraint
      )
      self.modulus_kernel = self.add_weight(
          shape=kernel_shape,
          initializer=modulus_init,
          name='modulus_kernel',
          regularizer=self.kernel_regularizer,
          constraint=self.kernel_constraint
      )
```

```
        if self.use_bias:
            self.bias = self.add_weight(
                shape=(3 * self.units,),
                initializer=self.bias_initializer,
                name='bias',
                regularizer=self.bias_regularizer,
                constraint=self.bias_constraint
            )
        else:
            self.bias = None

        self.input_spec = InputSpec(ndim=2, axes={-1: 3 * input_dim})
        self.built = True

    def call(self, inputs):
        input_shape = K.shape(inputs)
        input_dim = input_shape[-1] // 3
        phase_input = inputs[:, :input_dim]
        modulus_input = inputs[:, input_dim:]

        f_phase   = self.phase_kernel
        f_phase1 = tf.cos(f_phase)
        f_phase2 = tf.sin(f_phase)*(3**0.5/3)
        f_modulus   = self.modulus_kernel

        f1 = (K.pow(f_phase1,2)-K.pow(f_phase2,2))*f_modulus
        f2 = (2*(K.pow(f_phase2,2)-f_phase2*f_phase1))*f_modulus
        f3 = (2*(K.pow(f_phase2,2)+f_phase2*f_phase1))*f_modulus
        f4 = (2*(K.pow(f_phase2,2)+f_phase2*f_phase1))*f_modulus
        f5 = (K.pow(f_phase1,2)-K.pow(f_phase2,2))*f_modulus
        f6 = (2*(K.pow(f_phase2,2)-f_phase2*f_phase1))*f_modulus
        f7 = (2*(K.pow(f_phase2,2)-f_phase2*f_phase1))*f_modulus
        f8 = (2*(K.pow(f_phase2,2)+f_phase2*f_phase1))*f_modulus
        f9 = (K.pow(f_phase1,2)-K.pow(f_phase2,2))*f_modulus

        matrix1 = K.concatenate([f1, f2, f3], axis=-1)
        matrix2 = K.concatenate([f4, f5, f6], axis=-1)
        matrix3 = K.concatenate([f7, f8, f9], axis=-1)
        matrix = K.concatenate([matrix1, matrix2, matrix3], axis=0)

        output = K.dot(inputs, matrix)

        if self.use_bias:
            output = K.bias_add(output, self.bias)
        if self.activation is not None:
            output = self.activation(output)

        return output

    def compute_output_shape(self, input_shape):
        assert input_shape and len(input_shape) == 2
        assert input_shape[-1]
        output_shape = list(input_shape)
        output_shape[-1] = 3 * self.units
        return tuple(output_shape)

    def get_config(self):
        config = {
            'units': self.units,
            'activation': activations.serialize(self.activation),
            'use_bias': self.use_bias,
            'bias_initializer': initializers.serialize(self.bias_initializer),
            'kernel_regularizer': regularizers.serialize(self.kernel_regularizer),
            'bias_regularizer': regularizers.serialize(self.bias_regularizer),
            'activity_regularizer': regularizers.serialize(self.activity_regularizer),
            'kernel_constraint': constraints.serialize(self.kernel_constraint),
            'bias_constraint': constraints.serialize(self.bias_constraint),
        }
        base_config = super(QDense, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))
```