Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Виявлення шкідливих стеганографічних інструкцій у файлах зображень з використанням ШІ
(тема)

Виконав:
здобувач другого року навчання,
групи СШІм-24-2
Олександр ЛІСІН
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Системи штучного інтелекту
(повна назва освітньої програми)
Керівник проф. Євгеній БОДЯНСЬКИЙ
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Лариса ЧАЛА
(підпис) (власне ім'я, прізвище)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Лісіну Олександру Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Виявлення шкідливих стеганографічних інструкцій у файлах зображень з використанням ШІ

затверджена наказом університету від 6 квітня 20 26 р. № 269Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 22 травня 20 26 р.

3. Вихідні дані до роботи науково-технічні публікації за тематикою стеганографії, стегоаналізу та виявлення шкідливого програмного забезпечення; відкриті набори даних зображень для задач стегоаналізу; фреймворки PyTorch, NumPy, OpenCV та Matplotlib; документація бібліотек глибокого навчання; методи LSB-, DCT- та DWT-стеганографії; архітектури CNN та моделі з механізмом уваги; засоби розробки PyCharm та Python; відкриті дослідження у сфері кібербезпеки та цифрової мімікрії

4. Перелік питань, що потрібно опрацювати в роботі
1) Аналіз предметної галузі та постановка задачі
2) Опис інструментарію для розробки та датасет
3) Опис архітектурних рішень

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / термін виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	06.04.2026	виконано
2	Дослідження предметної області	10.04.2026	виконано
3	Дослідження інструментів глибокого навчання	12.04.2026	виконано
4	Створення датасету	15.04.2026	виконано
5	Розробка архітектури	18.04.2026	виконано
6	Експеримент №1	19.04.2026	виконано
7	Експеримент №2	20.04.2026	виконано
8	Розробка нової архітектури для спрощеної задачі	22.04.2026	виконано
9	Експеримент №3	23.04.2026	виконано
10	Дослідження результатів експерименту №3	24.04.2026	виконано
11	Експеримент №4	25.04.2026	виконано
12	Дослідження результатів експерименту №4	26.04.2026	виконано
13	Адаптація до ускладненої задачі	27.04.2026	виконано
14	Експеримент №5	28.04.2026	виконано
15	Експеримент №6	28.04.2026	виконано
16	Підсумок результатів	29.04.2026	виконано
17	Оформлення пояснювальної записки	18.05.2026	виконано
18	Захист кваліфікаційної роботи	22.05.2026	

Дата видачі завдання 6 квітня 20 26 р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. Євгеній БОДЯНСЬКИЙ
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 84 с., 48 рис., 9 табл., 1 дод., 26 джерел.

ЗГОРТКОВИЙ ЕНКОДЕР, ЗОБРАЖЕННЯ, КІБЕРБЕЗПЕКА, МЕХАНІЗМ УВАГИ, СТЕГАНОГРАФІЯ, CNN, COMMAND & CONTROL, LSB.

Об’єкт дослідження – системи виявлення замаскованого шкідливого програмного забезпечення у непідозрілих файлах з використанням глибоких нейронних мереж.

Предмет дослідження – методи адаптації моделей штучного інтелекту для виявлення корисного навантаження, замаскованого методами цифрової мімікрії.

Мета роботи – розробка моделі глибокого штучного інтелекту та її адаптація до задач виявлення зашифрованих шкідливих бітових послідовностей у файлах зображень.

Методи дослідження – теоретичні (аналіз методів використання стеганографії для шифрування, аналіз існуючих методів боротьби з цифровою мімікрією), та практичні (розробка моделі виявлення заражених методами стеганографії зображень).

Результатом дослідження є ідея архітектури моделі класифікації файлів зображень для відокремлення зображень з корисним навантаженням від звичайних чи нешкідливих.

ABSTRACT

Master’s thesis contains: 84 pp., 48 fig., 9 tabl., 1 ann., 26 references.

ATTENTION, CNN, COMMAND & CONTROL, CONVOLUTIONAL ENCODER, CYBERSECURITY, IMAGE, LSB, STEGANOGRAPHY.

Object of research are systems dedicated for detecting concealed malicious software in unsuspected files using deep neural networks.

Subject of research are methods aimed for adapting artificial intelligence models to detect payloads concealed by digital mimicry instruments.

Purpose of the work is development of a deep artificial intelligence model and its adaptation to the task of detecting encrypted malicious bit sequences in image files.

Research methods are theoretical (analysis of steganography methods used for encryption, analysis of existing methods to combat digital mimicry) and practical (development of a detection model for images infected by steganographic methods).

Result of the research is the concept of an AI model architecture dedicated for classification of image files to separate images containing payloads from ordinary or harmless ones.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект;

CNN – Convolutional Neural Network – згорткова нейромережа;

C2 – Command & Control – командно-контрольний сервер;

DCT – Discrete Cosine Transform – дискретне косинусне перетворення;

DWT – Discrete Wavelet Transform – дискретне вейвлет-перетворення;

LSB – Least Significant Bit – найменш значущий біт;

OLE – Object Linking and Embedding – фреймворк для забезпечення підтримки одного об’єкта декількома різними застосунками;

SRM – Spatial Rich Model – просторово-насичена модель.

ВСТУП

У сучасну епоху кіберзагроз, коли антивіруси, брандмауери та інші різноманітні системи ефективно блокують відомі віруси, зловмисники продовжують дедалі частіше залучати методи цифрової мімікрії. Стеганографія є одним із найнебезпечніших та витончених інструментів у цій справі.

Антивіруси традиційно перевіряють файли за сигнатурами відомих загроз, тоді як зображення з прихованими повідомленнями можуть виглядати цілком нормально і проходити перевірку. Крім того, браузери та переглядачі зображень іноді інтерпретують вміст файлу не так, як очікується, що відкриває вразливості.

Використання моделей глибокого навчання для боротьби зі стеганографією є чи не найперспективнішим шляхом розвитку сучасного стегоаналізу. Нейронні мережі здатні самостійно вивчати складні ієрархічні ознаки зображення. Вони помічають мікроскопічні порушення зв’язків між сусідніми пікселями, які неможливо описати однією формулою, як при використанні класичного стегоаналізу. Без ШІ сучасний стегоаналіз фактично приречений, бо саме здатність нейромереж знаходити закономірності в хаосі робить їх головним інструментом

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Огляд предметної галузі

Стеганографія це практика приховування інформації шляхом вбудовування секретних повідомлень у звичайний непомітний файл носія так, щоб ніхто не підозрював навіть саме існування повідомлення. На відміну від криптографії, яка робить повідомлення недоступним, але не приховує його існування, стеганографія націлена насамперед на те, щоб зробити факт передачі повідомлень непомітним. Насправді, стеганографія використовується ще з прадавніх часів, єдина відмінність – це носії: від записів воском на дошці та невидимих чорнил до приховування даних у зображеннях, аудіо чи відео.

Для прихованої передачі даних перш за все вибирається покривний носій – файл, що буде, не викликаючи підозр, вміщувати повідомлення, тобто, вище згадані зображення, аудіо, відео, текст або мережеві пакети. Перед вбудуванням дані опціонально шифруються, що додає додатковий захист, якщо дані вдасться витягнути.

Серед найпопулярніших методів вбудування:

– LSB (Least Significant Bit). Ідея полягає у зміні найменш значущого біта, наприклад, біт кольору R,G,B пікселя. Ці зміни зазвичай непомітні в зображенні або дуже мало помітні в аудіо. Метод простий у реалізації та пропонує велику місткість для корисного навантаження [20];

– частотна область. Полягає у модифікації коефіцієнтів DCT або DWT [13]. Тобто данні вбудовуються не в сирі пікселі, а в частотні компоненти, наприклад, після DCT, яку використовує формат JPEG;

– bit-plane. Полягає у модифікація вищих або нижчих бітів пікселів у відповідних площинах. Цей метод пропонує більшу стійкість але при цьому більш помітний [19], [18];

1.2 Актуальність дослідження

Використання стеганографії у зловмисних цілях досі є суттєвою проблемою, через її ефективність. Розглянемо використання стеганографії на прикладі атаки типу Command & Control (C2). Все починається зі звичайного зараження жертви, найчастіше, через фішинг, вивантажується невеликий скрипт, який сам по собі не класифікується антивірусами як шкідливий. Після цього він осідає в системі і чекає на подальші інструкції. Саме для їх передачі зловмисники використовують звичайні растрові зображення, вони служать прихованим каналом зв'язку для передачі команд вже зараженим машинам. Більш того, мережеві фільтри часто ігнорують зображення і не перевіряють їх так ретельно, як виконувані файли, і цим активно користуються для обходу захисту. Для цієї стратегії необхідне непідозріле місце зберігання зображень з інструкціями. Тримати власний сервер для цього – недоцільно, бо його легко заблокувати. Натомість зображення завантажуються на легітимні платформи типу Twitter чи imgur. І це має сенс, адже трафік до цих сервісів не викличе підозри ні у користувача, ні у корпоративного фаерволу. Таким чином заражена машина за розкладом звертається до певної платформи, завантажує зображення, зчитує приховані дані, розшифровує команду і виконує її. Назовні ж це виглядає як звичайний HTTP-запит до сервісу.

Такий спосіб використання стего-об'єктів можна спостерігати у відносно нещодавно задокументованих справах зловживання стеганографією для C2. До прикладу можна взяти Witchetty APT, Stegobot або сучасні кампанії Remcos, AgentTesla чи AsyncRAT.

1.2.1 Witchetty APT

Кампанія Witchetty, згідно з розслідуванням, застосовувала багатофазну модель атаки. На ранніх етапах використовувалися два основні

компоненти: бекдор X4 та LookBack. В межах розширення набору інструментів групи було виявлено додатковий компонент, ідентифікований як Backdoor.Stegmap, який наочно ілюструє використання стеганографії у розгортанні повторного завантаження шкідливого коду.

Актори використовували растрове зображення, що візуально відповідло застарілому логотипу Microsoft Windows. Такий файл був вибраний через його непримітність для розміщення на публічному хостингу. DLL-завантажувач скачував це зображення безпосередньо з репозиторію GitHub. Використання довіреного публічного сервісу знижувало ймовірність виклику підозри через мережеві червоні прапорці, оскільки завантаження з GitHub частіше сприймається як легітимне. У самому файлі було сховане корисне навантаження, що не було розміщене у вигляді окремого виконуваного файлу, а вбудоване у структуру зображення, шляхом вставлення даних у масив пікселів або в області, що не впливають на візуальне відтворення зображення (вище згаданий метод LSB). Після скачування DLL-завантажувач проходив процедуру витягання прихованих даних із бітів зображення. Відомо, що передані дані розблоковуються за допомогою XOR-ключа. Тобто приховані інструкції були додатково зашифровані симетричною операцією XOR, що унеможливлює просте витягнення навіть у разі доступу до бінарного носія [22].

1.2.2 Stegobot

Ще один чудовий приклад використання стеганографії описаний у дослідженні Stegobot. Це ботнет, який будується поверх соціальної мережі, таких як Facebook, де вузлами є користувачі, а ребрами – їхні соціальні зв’язки (дружба, листування тощо). Ключова ідея полягає в тому, що всі комунікації між ботами відбуваються лише між користувачами, які вже мають соціальний зв’язок, тобто без потреби створення нових, штучних каналів зв’язку.

Пропаганда ботів відбувається через соціальні атаки, наприклад, зараження через електронні листи або повідомлення у соцмережах, які надходять від друзів або колег. Після зараження бот починає вбудовувати приховані дані у зображення, які користувачі зазвичай завантажують. Розсилка повідомлень у ботнеті відбувається за допомогою простого алгоритму «restricted flooding» – бот відправляє приховані повідомлення всім сусідам у соціальній мережі в межах певної кількості кроків. Схема зв'язків ботнету представлена на рисунку 1.1.

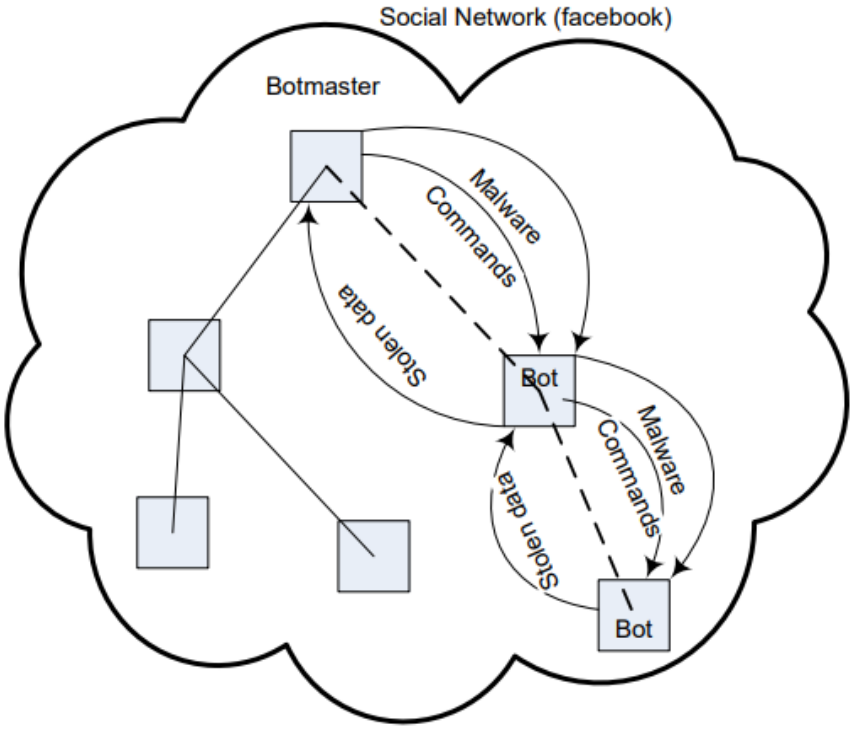

Рисунок 1.1 – Ботнет Stegobot

Для передачі команд і крадених даних Stegobot використовує все ту ж саму стеганографію в зображеннях, які користувачі зазвичай обмінюються у соціальних мережах. Таким чином, ботнет маскує свою активність під звичайний обмін фотографіями, що робить його комунікацію практично непомітною для систем виявлення. Схема процесу передачі прихованих повідомлень Stegobot представлена на рисунку 1.2.

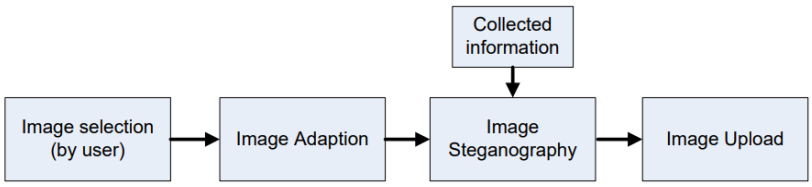

Рисунок 1.2 – Схема передачі повідомлень

У Stegobot використовується jpeg-стеганографія, зокрема схема YASS, яка дозволяє приховувати дані у фотографіях, що завантажуються користувачами у соціальну мережу. Коли заражений користувач завантажує зображення, бот вбудовує у нього зашифровані дані: команди або викрадену інформацію. Коли інший заражений користувач переглядає це зображення, бот вилучає приховану інформацію. Цей процес повторюється по ланцюжку соціальних зв'язків, поки дані не дійдуть до власника ботнету (ботмастера). Використання стеганографії робить таку комунікацію практично непомітною, оскільки трафік виглядає як звичайний обмін фотографіями, а не як підозріла передача даних.

У цьому прикладі стеганографія є ключовим елементом, що забезпечує непомітність комунікації ботнету. Вона дозволяє ботам передавати інформацію, не створюючи нових мережевих з'єднань і не викликаючи підозр у системах моніторингу трафіку. Використання соціальної мережі як каналу передачі зображень додає додатковий рівень маскування, оскільки трафік виглядає як звичайна соціальна активність. Експериментальна оцінка показала, що стеганографічний канал є достатньо надійним і має прийнятну пропускну здатність для передачі значних обсягів інформації [8].

1.2.3 Сучасні кампанії

Навіть в одних із найнещодавніших багатоступеневих атаках – Remcos та AsyncRAT, стеганографія використовується як механізм

доставляння носія (того ж зображення) в який маскується наступна стадія шкідливого коду. Це все ще дозволяє приховати бінарні або текстові артефакти від простих сигнатурних і сигнально-орієнтованих засобів захисту, антивірусів, проксі, а також ускладнює ручний аналіз, оскільки носій виглядає як звичайний мультимедіа-файл. У розглянутому випадку носієм виступає jpeg-зображення для приховування у ньому коду, закодованого у Base64 або в подібній формі, яке стає другою або третьою стадією інфікування після початкового фішингу.

Стандартна схема такої атаки має подібний вигляд: фішинговий лист – зловмисний документ (Excel з OLE-об'єктом чи експлойт) – запуск .hta чи .vbs – завантаження скрипта з інтернету – завантаження jpg-файла – витяг із jpg закодованого блоку – отримання і розпакування завантажувача – завантаження кінцевого корисного навантаження. Схема ланцюжка зараження представлена на рисунку 1.3.

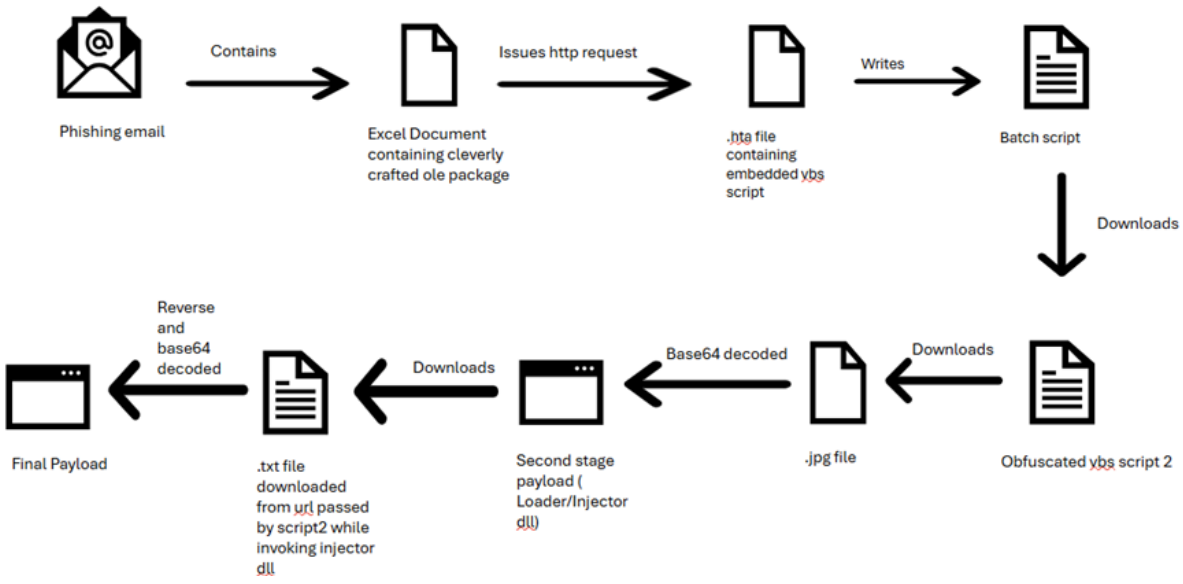

Рисунок 1.3 – Схема зараження Remcos та AsyncRAT

В цьому випадку також використовувалась стеганографія, щоб приховати завантажувач всередині JPG. Це робить перехід від скрипт-макроса до бінарного виконуваного компоненту менш помітним в

мережевому трафіку й знижує ймовірність виявлення традиційними перевірками контенту, що орієнтуються на основні сигнатури виконуваних файлів. Сам jpg містив у собі текстовий блок, який був обрамлений маркерами початку-кінця: <<BASE64 START>> та <<BASE64 END>>. Між цими мітками зберігався Base64-припасований (іноді у зворотному порядку) бінарний контент, тобто фактично закодований виконуваний файл або DLL. Завантажувальний скрипт типу PowerShell, отримавши jpg за HTTP-запитом, виконував просту текстову фільтрацію. Він знаходив маркери, витягував їх вміст, декодував Base64 (передбачалася додаткова операція реверсування байтового/текстового порядку у випадку reversed base64) і таким чином відновлював бінарну форму завантажувача.

Нарешті на зараженій машині створюється легітимний процес в призупиненому стані, відбувається відтинання його існуючого образу (операція, схожа на Unmap), виділення пам’яті у віддаленому процесі шляхом VirtualAllocEx, запис туди зловмисного коду, налаштування контексту потоку через SetThreadContext та відновлення виконання. Ця техніка, відома як process hollowing або process injection, дозволяє завантаженому коду виконуватися в облісті пам’яті легітимного процесу, уникаючи прямого запуску підозрілих exe-файлів на диску [12].

1.3 Інструменти глибокого навчання для виявлення стеганографії

Враховуючи вище описані приклади, можна сказати, що зловмисники часто використовують саме файли зображення для передачі шкідливих повідомлень закодованих методом LSB. Тож для виявлення аномалій у LSB зображення майже ідеально підходять глибокі нейромережі. Як прототип для такої моделі можна взяти архітектуру розроблену для виявлення шкідливого коду у APK-файлах з використанням Трансформера. Згідно з цією архітектурою, для виявлення шкідливого ПЗ іноді застосовують підхід, коли бінарні файли або байтові послідовності перетворюють у зображення.

Ці зображення потім аналізують за допомогою CNN, які добре підходять для вилучення просторових ознак із візуальних даних. Щоб мережа фокусувалась на найважливіших регіонах, додається механізм уваги. Це допомагає мережі ігнорувати менш важливі або шумові ділянки і концентруватися на характерних патернах, які є ознаками шкідливого ПЗ.

1.3.1 Згорткова нейромережа

Так як в цьому випадку ми також маємо справу з зображеннями, застосування CNN є найбільш очевидним рішенням. Можна розглянути архітектуру, розроблену спеціально для виявлення стеганографічних змін у зображенні. Вона передбачує використання фіксованого шару згортки з високочастотними SRM-фільтрами для підсилення стеганографічного шуму і послаблення контенту зображення, після чого вихід підхоплюється послідовністю згорткових шарів з нелінійними функціями, batch normalization, absolute value, pooling і, у деяких варіантах, skip-зв’язки для витягнення ієрархічних ознак. Для самої класифікації використовуються повнозв’язні шари або глобальний average pooling із softmax для бінарного рішення.

Така архітектура моделі дуже чутлива до передобробки вхідних діних, тому значну роль тут відіграють саме SRM-фільтри. Це високочастотні детектори, які за замовчуванням налаштовані на виявлення статистичних відхилень, що породжуються вбудовуванням даних. Схема однієї з подібних архітектур представлена на рисунку 1.4.

Впровадження SRM, як першого згорткового шару, дозволяє цій моделі CNN виділяти стеганографічний шум ще на ранній стадії. Цей шар послаблює вплив великих однорідних областей на зображенні і підсилює локальні зміни, де програми часто вбудовують бітові зміни. При цьому нормалізація значень самих SRM-ядер (наприклад множення на 1/12 або

інші масштаби) досить істотно впливає на кінцеву точність, що свідчить про те, що процес передобробки критично важливий.

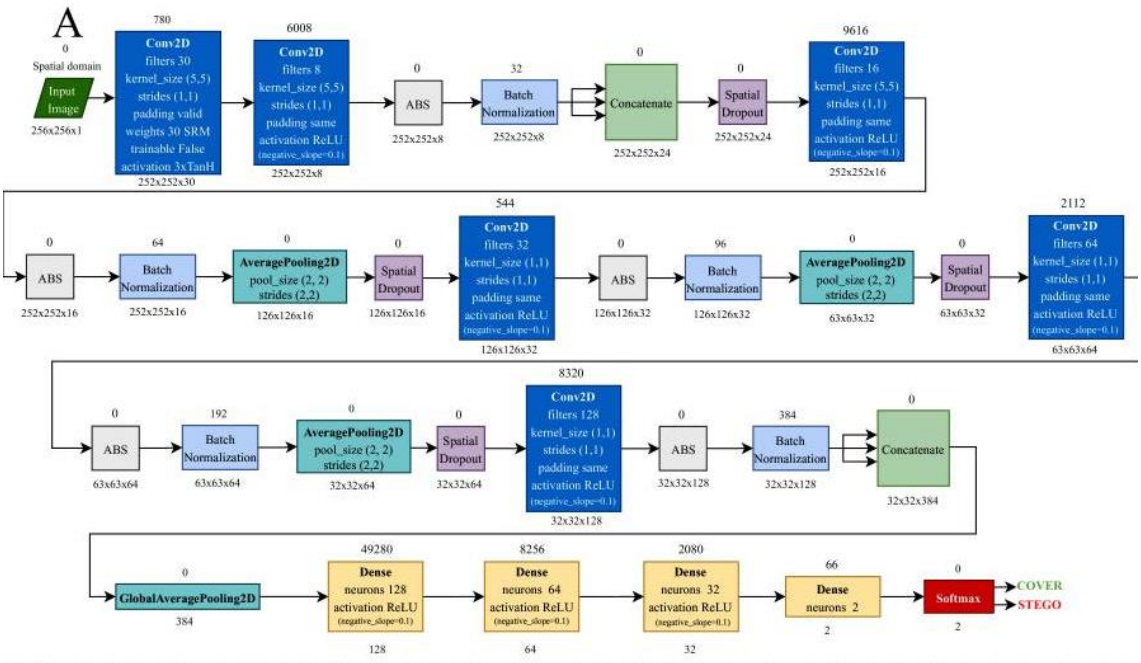

Рисунок 1.4 – Архітектура CNN

На ранніх шарах, після SRM, функції активації підкреслюють локальні деталі, де стеганографічні зміни найбільш помітні. Порівняння значень функцій активації для чистих зображень та змінених показує локальні відмінності: в мережі вони виглядають як підвищена активація у тих, в яких було вбудовано дані.

На глибших рівнях модель формує багаторівневі абстракції статистичних шаблонів стеганографії, які менш залежні від конкретного візуального контенту; ці узагальнені ознаки дозволяють класифікувати зображення.

CNN бачать стеганографію як сукупність статистичних відхилень, підсилених SRM-передобробкою і агрегованих на різних рівнях абстракції. Натренована мережа вчиться розпізнавати ці багаторівневі сигнатури й виносить бінарне рішення. Однак успішність цього процесу вельми чутлива

до передобробки даних, нормалізації фільтрів, вибору нелінійних функцій та розподілу навчальних і тестових даних [5].

1.3.2 Механізм уваги

Відрізнити чисте зображення від зміненого стеганографією може бути недостатньо, адже не виключена імовірність хибного позитивного результату, якщо повідомлення, заховане у зображення, не є шкідливим. Саме тому, з завданням відрізнити приховані інструкції від звичайних повідомлень найкраще впорається механізм уваги.

Трансформери вже чудово справляються з виявленням шкідливого коду. Моделі на їх основі здатні фокусуватися на найбільш значущих частинах вхідних даних, що є особливо важливим при аналізі складних послідовностей, таких як вихідний код програм або метадані додатків. У випадку виявлення шкідливого коду це означає, що модель може виділяти ключові ознаки, які свідчать про наявність шкідливих дій або патернів у коді, ігноруючи менш важливу інформацію.

Чудово демонструє потужність механізму уваги модель на основі BERT, створена для виявлення та класифікації шкідливого ПЗ, замаскованого в APK-файлах додатків для смартфонів на базі Android. Згідно з архітектурою, вихідний код або метадані додатку перетворюються у текстовий формат, який розбивається на токени (словники, ключові слова, дозволи, виклики API тощо). Ця послідовність токенів подається на вхід трансформеру. Механізм уваги трансформера дозволяє моделі одночасно аналізувати різні ознаки цієї послідовності, звертаючи увагу на різні її частини. Це дає змогу виявити складні залежності між різними елементами послідовності, які можуть бути ознаками шкідливого коду. Завдяки двонаправленій природі моделей типу BERT, трансформер враховує контекст як зліва, так і справа від кожного токена, що дозволяє краще розуміти структуру послідовності і виявляти приховані патерни шкідливих

інструкцій. На основі отриманих контекстуальних представлень модель виконує класифікацію додатків або фрагментів коду на шкідливі та нешкідливі, а також може розподіляти їх за категоріями: трояни, шпигуни, рекламне ПЗ тощо.

Графічне представлення архітектури моделі на основі BERT для виявлення та класифікації шкідливого коду представлено на рисунку 1.5

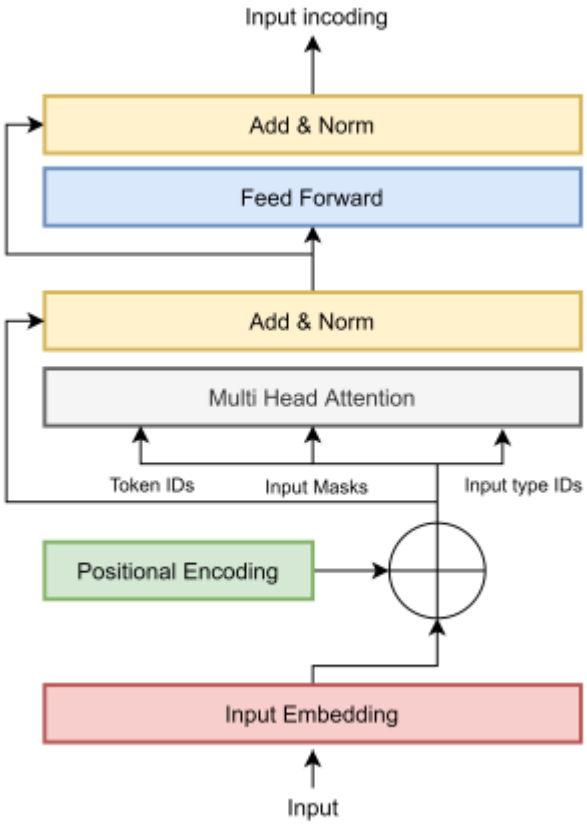

Рисунок 1.5 – Архітектура моделі BERT

Слід зауважити, що архітектура цієї моделі від початку не була створена для класифікації шкідливого ПЗ. Трансформерна модель BERT була лише донавчена на спеціально підготовленому наборі даних, що містить як шкідливі, так і нешкідливі додатки. Механізм уваги таким чином адаптувався до виділення ключових ознак у вихідному коді та метаданих, що дозволило ефективно класифікувати додатки за відповідними класами.

Методологія донавчання моделі BERT з метою класифікації Android додатків представлена на рисунку 1.6.

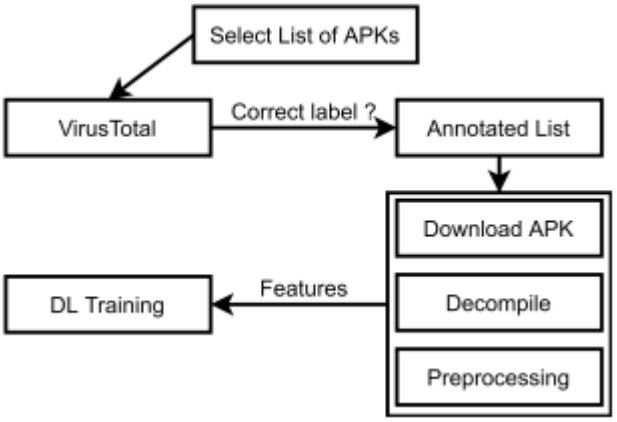

Рисунок 1.6 – Метод донавчання BERT

Вже тільки це дає зрозуміти наскільки трансформери є ефективним засобом для виявлення шкідливого коду завдяки здатності моделі виділяти значущі патерни у складних послідовностях, враховувати довготривалі залежності та контекст, що значно підвищує точність класифікації [16].

1.4 Постановка задачі

Метою кваліфікаційної роботи є дослідження випадків неправомірного використання методів стеганографічного вбудування корисного навантаження у легітимні файли-носії та знаходження можливого засобу протидії. Це передбачає розробку експериментальних архітектур моделей штучного інтелекту для виявлення шкідливих послідовностей. Результат сприятиме можливому підвищенню ефективності протидії кібератакам з використанням стеганографії та зменшенню спрямованих на це витрат обчислювальних потужностей.

При аналізі предметної галузі виділено ряд завдань на виконання в ході кваліфікаційної роботи:

– проведення аналізу задокументованих випадків неправомірного використання стеганографії;

– виявлення найпоширеніших методів використання стеганографії для кібератак;

– аналіз існуючих засобів виявлення стеганографічних інструкцій та їх протидії;

– розробка експериментальної архітектури моделі для виявлення та класифікації стеганографічних повідомлень;

– зафіксувати результати навчання моделі, її ефективність відносно витрат часу та ресурсів.

Успішне виконання поставлених завдань дозволить прискорити впровадження нових методів захисту публічних ресурсів від несвідомого зберігання небезпечних елементів.

2 ОПИС ІНСТРУМЕНТАРІЮ ДЛЯ РОЗРОБКИ ТА ДАТАСЕТ

2.1 Вибір основного фреймворку

Коли мова йде про розробку моделей штучного інтелекту на думку одразу спадають два найпопулярніші фреймворки: PyTorch та TensorFlow. Ці дві бібліотеки стали лідерами не випадково. вони виникли в період революції у глибокому навчанні і отримали підтримку найпотужніших технологічних гігантів. TensorFlow створювався Google як цілісна інфраструктура, в той час як PyTorch розроблений Meta орієнтуввся на наукову спільноту. Для поточної задачі доцільно використовувати саме PyTorch, на що є низка причин.

Головною перевагою є те, що PyTorch використовує динамічний обчислювальний граф, що створюється під час виконання програми. Це дає дослідникам можливість змінювати структуру моделі на ходу, що значно спрощує експерименти, налагодження та розробку нових архітектур. TensorFlow, хоча і має підтримку динамічних графів через TensorFlow 2.x і Eager Execution, традиційно базується на статичних графах, що ускладнює швидке прототипування.

Завдяки його динамічному графу, PyTorch дозволяє використовувати стандартні інструменти налагодження Python, такі як pdb, що робить процес виявлення помилок більш інтуїтивним і швидким. TensorFlow, особливо у своїх ранніх версіях, вимагав складніших підходів до налагодження через статичний граф.

PyTorch є домінуючим фреймворком у науковому співтоваристві. близько 85% нових наукових публікацій пов’язаних з глибоким навчанням використовують саме його. Це означає, що більшість нових алгоритмів і моделей першими з’являються у вигляді відкритого коду на PyTorch, що дає дослідникам перевагу у швидкому впровадженні інновацій. Він має одну з найбільш активних спільнот у сфері машинного навчання, що забезпечує

швидку підтримку, численні навчальні матеріали, бібліотеки та розширення. Це особливо важливо для дослідників, які часто стикаються з унікальними задачами.

PyTorch краще інтегрується з науковими бібліотеками Python наприклад, ті самі NumPy чи SciPy, що полегшує проведення комплексних експериментів, аналізу даних і візуалізації. Це ж робить його більш природним для розробників і дослідників, особливо тих, хто вже знайомий з Python. Це знижує поріг входження і пришвидшує розробку.

Враховуючи вище зазначене, для експериментальних досліджень у сфері штучного інтелекту PyTorch має суттєві переваги завдяки своїй гнучкості, простоті налагодження, швидкому доступу до нових наукових розробок тощо. Саме ці фактори роблять PyTorch найпопулярнішим вибором серед дослідників у 2026 році. TensorFlow, у свою чергу, більше орієнтований на промислові застосування та масштабованість, але в академічному середовищі PyTorch залишається лідером [15].

2.2 Вибір середовища розробки

Для написання коду вибране інтегроване середовище розробки PyCharm, розроблене компанією JetBrains, що спеціалізується на програмуванні на мові Python. Воно широко використовується як у промисловій розробці, так і в наукових дослідженнях, має потужний функціонал, зручний інтерфейс та широкий спектр інструментів для кодування, налагодження, тестування та управління проєктами.

PyCharm має глибоку інтеграцію з Python і підтримує всі основні бібліотеки для машинного навчання, включно з PyTorch. Це дозволяє безперешкодно імпортувати, запускати та налагоджувати моделі, а також працювати з науковими пакетами типу NumPy, SciPy, Matplotlib тощо. А також відзначається одним із найкращих у галузі механізмів автозаповнення коду, який глибоко розуміє синтаксис Python і контекст коду. Це дозволяє

значно зменшити кількість помилок під час написання і пришвидшує розробку, особливо у великих і складних проєктах.

Також в ньому присутній потужний відладчик, який підтримує покрокове виконання коду, встановлення точок зупину, перегляд змінних у реальному часі. Це особливо корисно при роботі з динамічними графами PyTorch, де важливо відслідковувати стан моделі та даних під час тренування.

Середовище пропонує максимально простий та інтуїтивно зрозумілий інтерфейс з багатьма корисними інструментами: рефакторинг, пошук по коду, інтеграція з базами даних, тестування, виведення графіків, що робить робочий процес більш продуктивним. Приклад інтерфейсу середовища розробки представлено на рисунку 2.1.

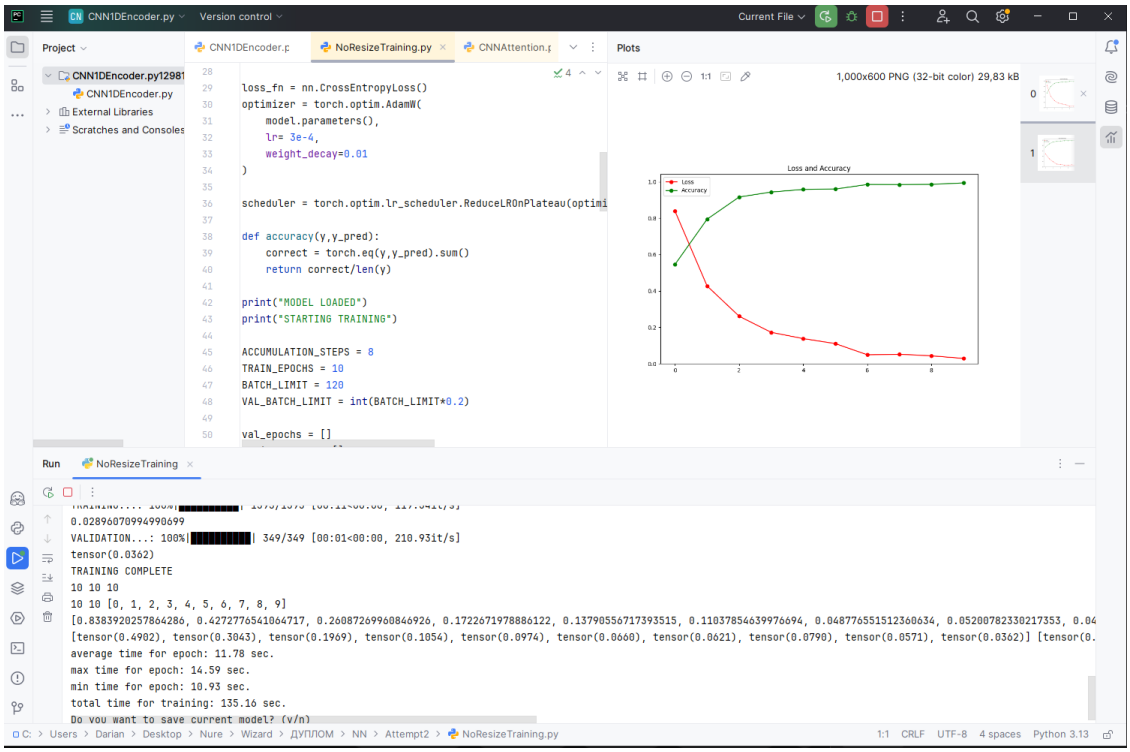

Рисунок 2.1 – Інтерфейс PyCharm

Головні риси які стали ключовими для вибору PyCharm відносно інших IDE це саме його глибина взаємодії з Python, потужними

інструментами налагодження, підтримкою наукових бібліотек і зручним управлінням проєктами. Це робить його одним із найкращих виборів для розробки мовою Python, що включає експерименти зі штучним інтелектом, особливо для дослідників, які цінують продуктивність і комфорт роботи. Водночас, варто враховувати, що PyCharm є більш ресурсоємним і частково платним, що може бути фактором при виборі середовища [11].

2.3 Вибір датасету

У дослідженнях глибокого навчання якість навчальних даних є фундаментальним фактором, що визначає успішність побудованої моделі. І хоча вже існує безліч датасетів у вільному доступі, створення власного датасету для поточної задачі стає необхідним і виправданим кроком для подальших експериментів, в першу чергу тому, що специфіка задачі та контекст дослідження не можуть бути адекватно представлені існуючими наборами даних. Доступні датасети не відображають унікальні особливості досліджуваної предметної області та завдання. Власний датасет дозволить зібрати дані, які максимально відповідають цільовим умовам, що підвищить релевантність і якість навчання моделі.

Тож для реалізації методів виявлення різних типів корисного навантаження прийнято рішення створити кастомний датасет, що складається з прикладів чистих зображень, зображень з закодованими нешкідливими повідомленнями та зображень з вбудованими послідовностями коду.

У якості корисного навантаження що репрезентує звичайне повідомлення прийнято рішення використати сценарій до фільму. Повідомлення формується через алгоритм, який зчитує текстовий файл та ділить його на батчі по 20–25 рядків кожен. Ці сукупності рядків і виступають в якості шифрованого повідомлення. Зразок змісту звичайних повідомлень представлено на рисунку 2.2.

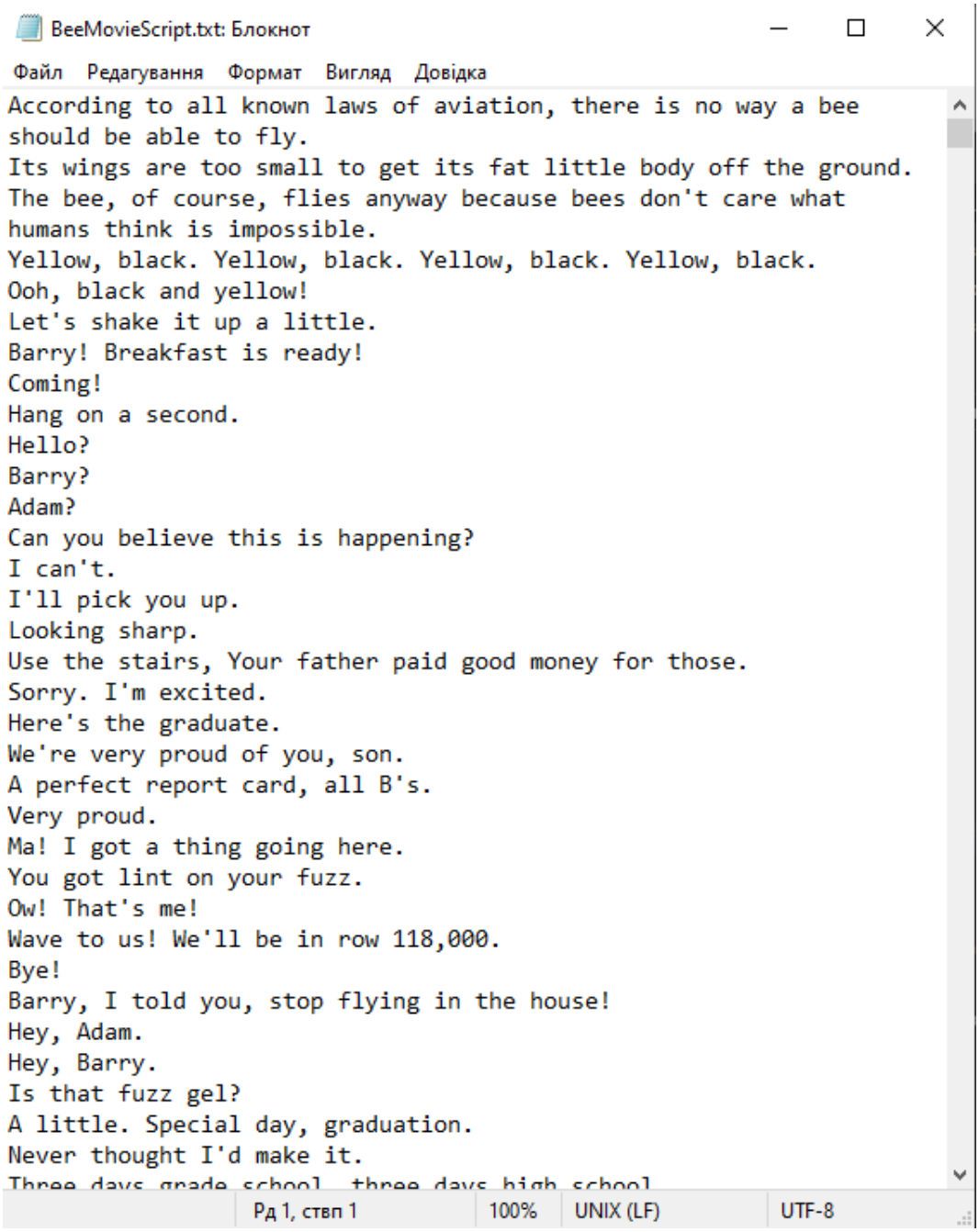

Рисунок 2.2 – Зразок рядків звичайного повідомлення

У якості корисного навантаження що репрезентує шкідливе повідомлення прийнято рішення використати зразки Python коду. При цьому алгоритм вбудування залишається ідентичним до звичайного повідомлення.

Зразок змісту повідомлень з кодом представлено на рисунку 2.3.

Code.txt: Блокнот

Файл Редагування Формат Вигляд Довідка

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import ReduceLROnPlateau
import random
import os
import matplotlib.pyplot as plt
from tensorflow.python.keras.utils.version_utils import callbacks


class TextGeneratorLSTM:
    def __init__(self, level='char'):
        self.level = level  # 'char' or 'word'
        self.tokenizer = Tokenizer(char_level=(level == 'char'))
        self.model = None
        self.sequence_length = 0
        self.vocab_size = 0

    def load_text(self, file_path):
        with open(file_path, 'r', encoding='utf-8') as f:
            return f.read()

    def prepare_data(self, text, sequence_length):
        self.tokenizer.fit_on_texts([text])
        total_words = self.tokenizer.texts_to_sequences([text])[0]
        self.vocab_size = len(self.tokenizer.word_index) + 1

        input_sequences = []
        for i in range(sequence_length, len(total_words)):
            seq = total_words[i - sequence_length:i + 1]
            input_sequences.append(seq)

        input_sequences = np.array(input_sequences)
        self.sequence_length = sequence_length
```

Рд 1, ствп 1 | 100% | Windows (CRLF) | UTF-8

Рисунок 2.3 – Зразок рядків повідомлення з кодом

Як вже було зазначено у дослідженні, метод LSB є одним із найпростіших і найпоширеніших способів приховування даних у растрових зображеннях. Він базується на зміні найменш значущого біта кожного пікселя, що мінімально впливає на візуальне сприйняття зображення. Реалізація методу вбудування корисного навантаження представлено на рисунку 2.4.

```
def encode_lsb_jpg_to_png(input_jpg_path, output_png_path, message):

    img = Image.open(input_jpg_path)
    img = img.convert("RGB")
    data = np.array(img)
    flat = data.flatten()
    unique_filename = str(uuid.uuid4()) + ".png"
    output_path = output_png_path/unique_filename

    message_bytes = message.encode("utf-8")
    message_bits = ''.join(format(b, '08b') for b in message_bytes)
    length_bits = format(len(message_bits), '032b')
    full_bits = length_bits + message_bits

    if len(full_bits) > len(flat):
        warnings.warn(f"Data overload! Shorten the message! {len(full_bits)} > {len(flat)}")
        return

    for i, bit in enumerate(full_bits):
        flat[i] = (flat[i] & 0xFE) | int(bit)

    modified_data = flat.reshape(data.shape)
    stego_img = Image.fromarray(modified_data.astype("uint8"), mode: "RGB")
    stego_img.save(output_path, format="PNG")
```

Рисунок 2.4 – Метод вбудування корисного навантаження

Алгоритм відкриває вхідне JPEG зображення і конвертує його в RGB формат. Це важливо тому що JPEG може мати різні колірні простори, а явна конвертація гарантує що зображення має рівно три канали червоний, зелений і синій в правильному порядку [24]. Після цього зображення перетворюється в NumPy масив для побітових операцій, і цей масив розгортається в одновимірний вектор методом flatten. Тобто пікселі йдуть послідовно, спочатку всі значення першого рядка зліва направо по каналах R, G, B для кожного пікселя, потім другий рядок і так далі.

Вхідне повідомлення конвертується в байти через UTF-8, де кожен байт перетворюється в 8-бітний бінарний рядок. Всі ці біти конкатенуються в один довгий бінарний рядок який і стане корисним навантаженням. Перед самим повідомленням додається 32-бітний заголовок який містить довжину повідомлення в бітах. Наприклад якщо повідомлення займає 1000 біт то

заголовок міститиме число 1000, записане як 32-бітне ціле 0…01111101000. Це дозволить декодеру знати скільки біт необхідно зчитати для відтворення повідомлення.

Після формування послідовності алгоритм перевіряє чи вистачає пікселів у зображенні щоб вмістити всі біти. Якщо для зображення 64 на 64 пікселя з трьома каналами маємо 12288 значень пікселів, що дозволить вмістити максимум 12288 біт або приблизно 1536 символів тексту, включно з заголовком.

Етап вбудування корисного навантаження є центральною частиною представленого алгоритму. Для кожного біта повідомлення береться відповідний елемент розгорнутого масиву пікселів і його молодший біт замінюється на біт повідомлення. Для цього використовується операція “& 0xFE”, що є побітовою операцією AND з двійковим числом 11111110 у якості маски. Вона обнуляє молодший біт пікселя залишаючи всі інші біти незміненими. Потім операція “| int(bit)” встановлює молодший біт в потрібне значення, 0 або 1.

Вбудовування відбувається послідовно з першого елемента масиву пікселів, що ключовою особливістю представленого алгоритму. Спочатку заповнюється заголовок (перші 32 позиції), потім одразу за ним іде повідомлення. Решта пікселів після останнього біта повідомлення залишається незміненою.

Отриманий масив повертається до оригінальної форми зображення методом reshape і зберігається як PNG файл з унікальним UUID іменем. Варто зазначити, що PNG обраний свідомо, бо це формат без втрат, тобто при збереженні пікселі не стискаються з втратами як у JPEG, і вбудовані біти залишаються незміненими [14].

Цей метод використовується для енкодингу одного єдиного зображення. Якщо використовувати тільки його, генерація датасету, який налічує понад 1500 зображень займе суттєву кількість часу. Це зумовлено тим, що метод потребує ручного заповнення аргументів, таких як шлях до

картинки для вбудування чи саме повідомлення, яке може налічувати сотні символів. Робити все це вручну недоцільно, тому для конвеєрної генерації екземплярів тренувального датасету використано алгоритм представлений на рисунку 2.5.

```
def encode_batch(msg_path, imgs, output_png_path):

    with open(msg_path, 'r', encoding='utf-8') as f:
        lines = [line.strip() for line in f if line.strip()]

    if not lines:
        print("File empty!")
        return

    current_index = 0
    total_lines = len(lines)

    for img in tqdm(imgs):

        count = random.randint( a: 160,  b: 170)
        batch = []
        for _ in range(count):
            batch.append(lines[current_index])
            current_index += 1
            if current_index >= total_lines:
                current_index = 0

        msg = "\n".join(batch)
        LSB.encode_lsb_jpg_to_png(img, output_png_path, msg)
```

Рисунок 2.5 – Метод вбудування батчу

Наведений алгоритм відкриває текстовий файл за шляхом, що вказаний користувачем у конфігураційному файлі і зчитує всі рядки одразу. Кожен з цих рядків очищується від пробільних символів на початку і кінці, і порожні рядки відфільтровуються. В результаті отримується список рядків який цілком зберігається в пам'яті. Якщо файл виявився порожнім після фільтрації, алгоритм виведе повідомлення і аварійно завершить роботу. Приклад налаштувань вказаних у конфігураційному файлі представлено на рисунку 2.6.

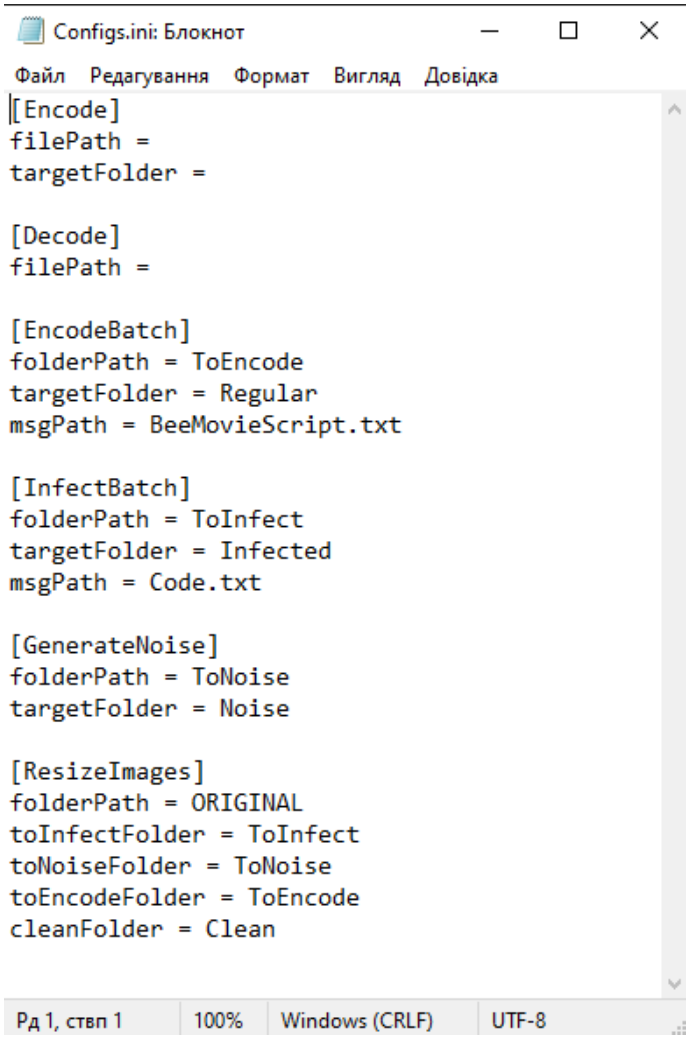

Рисунок 2.6 – Конфігураційні налаштування алгоритму вбудування

Перед початком обробки зображень ініціалізується глобальний лічильник. Він являється спільним для всіх зображень у пакеті і не скидається між зображеннями, а продовжує рухатись вперед через весь список рядків. Це дозволяє кожному наступному зображенню в циклі отримувати продовження тексту з того місця, де закінчилось попереднє.

У самому циклі для кожного зображення поданого на вхід виконується певна послідовність дій. Спочатку генерується випадкове число від A до B, в даному прикладі від 160 до 170, що визначатиме кількість рядків які будуть вбудовані в поточне зображення. Випадковий діапазон використано, щоб додати невеликої варіативності довжині повідомлення між зображеннями.

На основі вибраного числа формується батч рядків для поточного зображення. Для цього послідовно беруться рядки з загального списку починаючи з поточного індексу лічильника. Після кожного використаного рядка лічильник збільшується на одиницю. Якщо лічильник досягає кінця списку, то він скидається в нуль і алгоритм починає ітерацію списку з початку. По факту це кільцева буферизація, де список рядків представляється як нескінченна циклічна послідовність.

Зібрані рядки об'єднуються в одне повідомлення через символ нового рядка. Тобто якщо було взято 161 рядок, повідомлення являє собою відповідно 161 рядок тексту розділених переносами. Сформоване повідомлення передається в зазначений раніше базовий алгоритм LSB вбудовування разом з поточним зображенням і шляхом для збереження результату.

Слід уточнити, що оригінальні зображення мають неоднорідну форму, і для того, щоб зображення могли сформуватись у батчі це повинно бути виправлене. Для цього програмно реалізовано алгоритм, що приводить зображення до однорідного розміру. Згаданий алгоритм наведено на рисунку 2.7.

```
def resize(image, output_path, resize:bool=True, output_size=(64,64)):
    unique_filename = str(uuid.uuid4()) + ".png"
    output_path = output_path / unique_filename
    image = Image.open(image)
    if resize:
        image = image.convert("RGB")
        image = image.resize(output_size, Image.Resampling.LANCZOS)
    image.save(output_path, format="PNG")
```

Рисунок 2.7 – Алгоритм перетворення зображень

Перше що робить метод це генерує унікальне ім'я для вихідного файлу на основі UUID4. По суті UUID4 є випадково згенерованим ідентифікатором з 122 бітами ентропії, що практично виключає колізії

навіть при обробці великих датасетів. До рядка UUID додається розширення .png, і формується повний шлях до вихідного файлу шляхом об'єднання вихідної директорії з цим іменем.

Зображення відкривається через бібліотеку PIL з вхідного шляху. На цьому етапі зображення ще не завантажується повністю в пам'ять бо PIL використовує ліниве завантаження, тобто пікселі читаються лише коли це дійсно необхідно.

Якщо параметр resize має значення True виконуються два послідовні перетворення. Спочатку зображення конвертується в RGB колірний простір. Це необхідно тому що оригінальні зображення можуть мати різні формати типу RGBA з альфа каналом, з каналом для відтінків сірого, у форматі для палітрових чи друкарських зображень. Явна конвертація в RGB гарантуватиме що результат завжди матиме рівно три канали однакової структури незалежно від формату оригіналу [9].

Потім зображення масштабується до цільового розміру, за замовчуванням 64×64 пікселів, методом LANCZOS. Цей метод є алгоритмом інтерполяції заснованим на функції sinc яка мінімізує артефакти при зменшенні розміру. Він обчислює значення кожного нового пікселя як зважену суму сусідніх пікселів оригіналу з вагами визначеними функцією Ланцоша. Порівняно з простішими методами як NEAREST або BILINEAR він краще зберігає чіткі границі і дрібні деталі при даунсемплінгу. Це не гратиме важливої ролі, але для чистоти експерименту варто не впускати потенційно важливих деталей.

Якщо ж опція resize не задіяна конвертація і масштабування пропускаються, і зображення зберігається як є, змінюючи тільки формат на PNG та ім'я файлу. Це передбачено для більш гнучкого контролю над процесом генерації зразків зображень, бо у подальших експериментах може виникнути необхідність застосування зображень оригінальних розмірностей.

Приклад вигляду згенерованого датасету для одного з класів представлено на рисунку 2.8.

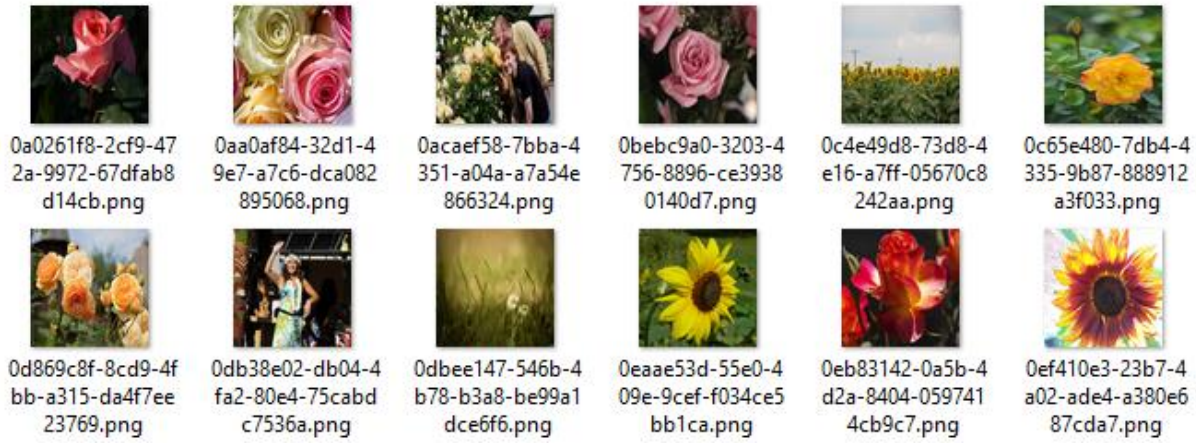

Рисунок 2.8 – Приклад зображень

У результаті отримано сукупність екземплярів зображень одного розміру з закодованими повідомленнями, що готові до використання у навчанні моделі класифікатора.

Додатково до зображень з корисним навантаженням прийнято рішення додати картинки з випадковим шумом. Це допоможе визначити чи здатна буде модель відсіяти дані, які не несуть у собі якогось смислового навантаження. Для цього додатково до алгоритму вбудування повідомлень реалізовано алгоритм обробки зображень з додаванням Гаусівського шуму, пердставлений на рисунку 2.9.

Алгоритм ітерується по вхідному списку зображень, для кожного з яких виконується повний цикл обробки незалежно від інших. Зображення звично відкривається через метод бібліотеки PIL і примусово конвертується в RGB за тією самою причиною, щоб гарантувати три канали незалежно від формату джерела.

Потім зображення перетворюється в NumPy масив з типом float32. Використання float32 замість uint8 є принциповим, при додаванні шуму

значення пікселів можуть тимчасово виходити за межі 0-255, і float32 дозволяє коректно зберегти ці проміжні значення без переповнення.

Готовим методом генерується масив гаусівського шуму тієї самої форми що й зображення з окремим значення шуму для кожного пікселя кожного каналу. Розподіл шуму визначається показником математичного очікування та стандартного відхилення.

```
def gaussian_noise(imgs, output_path, sigma=0.3):

    for img in tqdm(imgs):

        img = Image.open(img)
        img = img.convert("RGB")
        data = np.array(img, dtype=np.float32)
        noise = np.random.normal(loc=random.randint(-1, b: 1), scale=sigma, size=data.shape)

        unique_filename = str(uuid.uuid4()) + ".png"
        out_path = output_path / unique_filename

        noisy_data = data + noise
        final_data = np.clip(noisy_data, a_min: 0, a_max: 255).astype(np.uint8)

        result_img = Image.fromarray(final_data)
        result_img.save(out_path, format: "PNG")
```

Рисунок 2.9 – Алгоритм додавання шуму

Математичне очікування визначено випадковим цілим числом з рівномірного розподілу між -1 і 1 включно, тобто приймає одне з трьох значень: -1, 0 або 1. Це дозволить внести невеликий випадковий зсув яскравості всього зображення. Воно може стати трохи темнішим, залишитись без зміни або стати трохи світлішим.

Стандартне відхилення визначено фіксованим значенням за замовчуванням 0.3. Відносно діапазону від 0 до 255 це дуже малий рівень шуму, настільки, що переважна більшість значень буде в межах від -0.9 до 0.9. Це означає що шум вплине переважно на молодші біти пікселів, що ідеально підходить для експерименту [23].

Шум додається до масиву зображення простим поелементним додаванням. Так як результуючі значення можуть виходити за межі допустимого діапазону 0-255, застосовується метод clip, який обрізає всі значення до визначеного діапазону 0-255, після чого масив конвертується назад в uint8.

Для збереження зашумленого зображення генерується унікальне UUID ім'я і результат зберігається як PNG без втрат.

2.4 Методи препроцесингу датасету

Для реалізації подання сформованого датасету на навчання моделі було програмно реалізовано декілька методів препроцесингу сирих даних. Метод завантаження даних у пам’ять представлено на рисунку 2.10.

```
data_dir = "../dataset"
transform = transforms.Compose([
    transforms.ToTensor(),
])
dataset = datasets.ImageFolder(
    root=data_dir,
    transform=transform
)
```

Рисунок 2.10 – Завантаження даних

Для завантаження датасету використовується звичайний метод наданий бібліотекою torch. При цьому єдину трансформацію, яку проходять зображення датасету, це перетворення у тензори. Будь яка інша трансформація спотворить вбудоване корисне навантаження.

Так як для подальших експериментів в директорії датасету може знаходитись стохастична кількість класів для різних цілей, було модифіковано оригінальний алгоритм формування навчальних та валідаційних вибірок. У першу чергу, через невизначену кількість класів у

директорії програмно реалізовано метод вирішення проблеми перемапінгу міток, представлений на рисунку 2.11.

```
class TargetTransformSubset(Dataset):
    def __init__(self, subset, mapping):
        self.subset = subset
        self.mapping = mapping
    def __getitem__(self, index):
        img, label = self.subset[index]
        return img, self.mapping[label]
    def __len__(self):
        return len(self.subset)
```

Рисунок 2.11 – Перемапінг

Ця реалізація є обгорткою над стандартним класом Subset бібліотеки torch яка вирішує цю конкретну проблему. Стандартний Subset зберігає оригінальні мітки датасету, але після фільтрації класів мітки можуть мати прогалини. Наприклад якщо після фільтрації залишились тільки мітки класів 0 і 2, то мітка 1 буде відсутня, що спричинить проблеми при навчанні.

Кастомний клас TargetTransformSubset приймає словник mapping і при кожному зверненні до елементу автоматично перетворює оригінальну мітку на нову через цей словник. Метод len відповідно здійснює підрахунок довжини до внутрішнього subset [1].

Для підготовки датасету до експерименту з задачею мультикласової класифікації реалізовано алгоритм наведений на рисунку 2.12.

Функція представленого алгоритму приймає словник, де ключами є імена директорій датасету а значеннями є цільові мітки класів. Це дозволить гнучко конфігурувати як групування так і перемаппінг міток.

Спочатку алгоритм визначає які оригінальні індекси класів в основній директорії відповідають запитаним іменам через перебір класів датасету. Потім будується словник що відображає оригінальний індекс класу на новий індекс з вхідного словника. Далі з повного списку індексів відбираються тільки ті індекси зображень чиї мітки входять до списку

вибраних для експерименту. Відфільтрований subset і цільовий мапінг передаються в TargetTransformSubset, і результат передається у функцію формування остаточних вибірок для тренування.

```
def get_dataset_wsplit(class_groups: dict = None, bs:int=None):
    if class_groups is None:
        raise ValueError("No class groups provided.")
    required_orig_indices = {
        i for i, cls_name in enumerate(dataset.classes)
        if cls_name in class_groups
    }
    target_mapping = {
        i: class_groups[cls_name]
        for i, cls_name in enumerate(dataset.classes)
        if cls_name in class_groups
    }
    indices = [
        i for i, label in enumerate(dataset.targets)
        if label in required_orig_indices
    ]
    filtered_dataset = Subset(dataset, indices)
    new_dataset = TargetTransformSubset(filtered_dataset, target_mapping)
    return form_dataset(new_dataset, bs)
```

Рисунок 2.12 – Формування датасету для мультикласової класифікації

Для підготовки датасету до експерименту з задачею бінарної класифікації реалізовано алгоритм наведений на рисунку 2.13.

```
def get_binary_dataset(classes:[str]=None, bs:int=None):

    if classes is None:
        raise ValueError("No classes provided.")
    if len(classes)!=2:
        raise ValueError("Only two classes are supported.")
    required_classes = classes
    class_to_idx = {cls: i for i, cls in enumerate(dataset.classes) if cls in required_classes}
    target_mapping = {orig_idx: new_idx for new_idx, orig_idx in enumerate(class_to_idx.values())}
    indices = [i for i, label in enumerate(dataset.targets)
               if dataset.classes[label] in required_classes]
    filtered_dataset = Subset(dataset, indices)
    new_dataset = TargetTransformSubset(filtered_dataset, target_mapping)
    return form_dataset(new_dataset, bs)
```

Рисунок 2.13 – Формування датасету для бінарної класифікації

Представлена функція бінарного датасету працює інакше ніж мультикласового. Вона приймає список рівно з двох імен класів відповідно, інакше виводиться помилка.

Логіка перемаппінгу в цій функції трохи відрізняється. Спочатку будується словник, у якому ключем є ім'я потрібного класу а значенням є його оригінальний індекс в основній директорії. Потім цільовий мапінг будується інвертуванням цього словника, де новий індекс визначається порядком ключів у вхідному списку класів вибраних для експерименту, а не алфавітним порядком оригіналу. Фільтрація індексів відбувається через перевірку імені класу на відміну від функції мультикласового датасету, де перевіряється числовий індекс. Нарешті так само створюється кастомний subset та відправляється на формування вибірок.

Після визначення subset одним із двох представлених методів на його основі будуються тренувальна та валідаційна вибірки. Програмна реалізація функції формування вибірок представлена на рисунку 2.14.

```
def form_dataset(new_dataset, bs):
    train_size = int(0.8 * len(new_dataset))
    val_size = len(new_dataset) - train_size

    train_dataset, val_dataset = random_split(
        new_dataset,
        lengths: [train_size, val_size]
    )
    train_loader = DataLoader(
        train_dataset,
        batch_size=bs,
        shuffle=True
    )
    val_loader = DataLoader(
        val_dataset,
        batch_size=bs,
        shuffle=False
    )
    return train_loader, val_loader
```

Рисунок 2.14 – Формування вибірок

Представлена функція приймає вже підготовлений одним з попередньо представлених методів датасет і бажаний розмір батчу. У ній датасет розділяється у співвідношенні 80 на 20 на тренувальну і валідаційну вибірки методом random_split. Тренувальний об’єкт DataLoader має shuffle=True, отже екземпляри перемішуватимуться перед кожною епохою. Валідаційний об’єкт має shuffle=False, для нього порядок фіксований для можливості відтворення результатів.

2.5 Системні вимоги до фреймворку

Тренування моделей штучного інтелекту за визначенням дуже ресурсозатратне. Відповідно бібліотека PyTorch має певні системні вимоги щодо її використання. Поточна конфігурація системи, використаної для експериментів наведено нижче:

– CPU: AMD Ryzen 7 5800X, 8 ядер, 3.8 GHz (турбо вища частота);

– GPU: NVIDIA GeForce RTX 4070 Super, 12 GB VRAM;

– ОЗП: 32 GB RAM;

– ОС: Windows 10 (64-bit);

– Диск: SSD 1 TB.

Відомості про мінімально необхідні системні налаштування, вказані на загальнодоступних ресурсах, передбачають що для базових задач достатньо сучасних багатоядерних CPU, таких як Intel Core i5-i7 або AMD Ryzen 5-7. Для більш вимогливих задач рекомендовано Intel Core i9 або AMD Ryzen 9. Також рекомендована наявність підтримки інструкцій типу AVX для прискорення лінійних/матричних операцій.

Поточний процесор Ryzen 7 5800X належить до категорії Ryzen 7 та відповідає і перевищує мінімальні вимоги, вважаючись близьким до робочого класу, придатного для багатьох серйозних робіт з навчання моделей. Він підтримує сучасні векторні інструкції AVX та AVX2 і має

достатню кількість ядер для ефективної підготовки даних, багатопотокових завдань і паралельних завантажувачів даних.

При роботі з використанням графічних процесорів перевага віддається NVIDIA через підтримку CUDA. Мінімальна вимога для CUDA обчислювальна потужність приблизно 3.7 або більше. Рекомендовано використовувати відеокарти з принаймні 8 GB відеопам’яті як у моделях GeForce RTX 3060/3070/3080 або вищих.

Наявна модель відеочіпа RTX 4070 Super є сучасною картою від NVIDIA із 12 GB відеопам’яті, що значно перевищує рекомендований мінімум 8 GB. Це забезпечить помітну перевагу при навчаннях середніх і, можливо, великих експериментальних моделей, дозволяючи використовувати більші батчі та складніші архітектури. Оскільки PyTorch, згідно з доступними даними, орієнтується на CUDA версії 11.x, зокрема CUDA 11.7 і cuDNN 8.5, поточна модель RTX 4070 Super сумісна з релізами CUDA 11.x і пізнішими збірками PyTorch, отже графічний процесор підходить для GPU-прискорення без обмежень.

Вимогою до обсягу оперативної пам’яті є мінімум 8 GB для базових задач. Для тренувань з великими датасетами і мультитаскінгу рекомендовано 16 GB або більше.

Наявні 32 GB значно перевищують рекомендований рівень, що дає достатній запас для одночасного виконання процесів підготовки даних, CPU-операцій або запуску кількох середовищ одночасно. Також це зменшує ризик ботлнеків пов’язаних з пам’яттю при роботі з великими датасетами або при запуску кількох моделей одночасно.

У якості пристрою зберігання рекомендовано SSD з мінімум 256 GB місця. Для великих датасетів і попередньо навчених моделей рекомендовано 512 GB і більше.

Поточний пристрій зберігання SSD розміром в 1 TB відповідає і значно перевищує рекомендовані вимоги. Це забезпечить швидке

завантаження даних, швидкий доступ до великого репозиторію моделей і швидкі операції з кешем і знімками моделей.

Серед операційних систем підтримуються Windows 10 (64-bit), Linux (Ubuntu 18.04+ рекомендовано для GPU-тренувань), macOS 10.14+ (з обмеженою GPU-підтримкою).

Windows 10 у наявній конфігурації офіційно підтримується згідно з рекомендаціями. Однак наголошується, що більшість користувачів віддає перевагу Linux для GPU-базованого тренування через кращу підтримку CUDA, драйверів та інструментів. Отже, на Windows 10 матиметься повна функціональність, але для максимального контролю та оптимізації, особливо у масштабних дослідницьких сценаріях, може бути розглянуто використання dual-boot або контейнер чи віртуалізацію з Linux.

Огляд вимог показав, що поточна конфігурація як програмно-апаратно, так і щодо обсягу оперативної пам’яті та пам’яті накопичувача загалом перевищує рекомендовані вимоги для комфортної роботи з PyTorch у більшості сценаріїв дослідження [6].

3 ОПИС АРХІТЕКТУРНИХ РІШЕНЬ

3.1 Архітектура на базі CNN + Attention

Так як повідомлення закодовані у растрові зображення, прийнято рішення побудови архітектури на базі згорткової нейронної мережі, що призначена для роботи з такого роду даними. До того ж корисне навантаження є послідовністю, що має певну закономірність, тож для реалізації мультикласової класифікації доцільним рішенням буде реалізація в архітектурі механізму уваги. Графічне представлення архітектури моделі наведено на рисунку 3.1

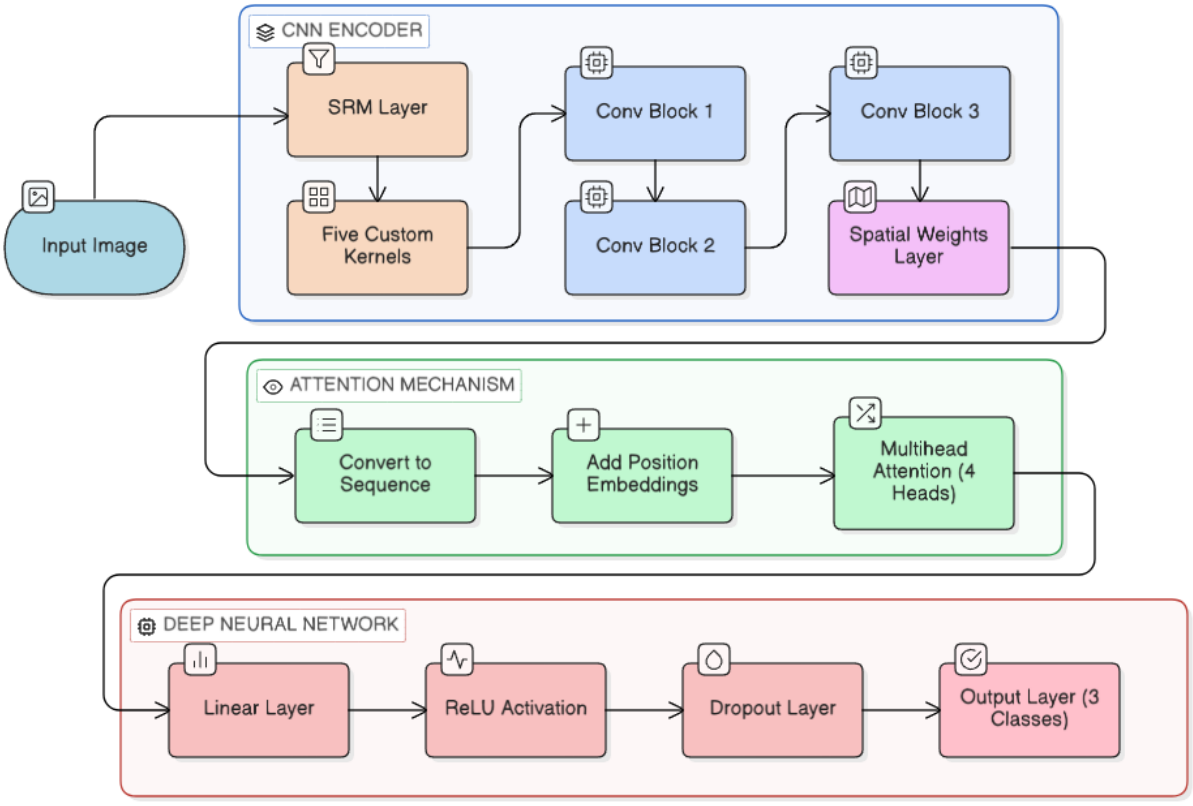

Рисунок 3.1 – Архітектура моделі CNN+Attention

Ідея представленої архітектури полягає у використанні SRM в якості препроцесингу. У розглянутих раніше системах стегоаналізу це бувв

найважливіший крок.. Фактично цей шар представляє собою набір фільтрів з фіксованими значеннями, які шукають залишкові сигнали між сусідніми пікселями. Звичайні зображення складаються з низькочастотного сигналу, який утворюють плавні переходи кольорів, зображені об’єкти, риси обличчя тощо, та високочастотного шуму. Стеганографія, як правило, ховається саме у цьому високочастотному шумі. SRM-фільтри побудовані так, що сума їхніх коефіцієнтів дорівнює нулю. Коли фільтр проходить по однорідній ділянці, результат дорівнює 0. Але при цьому, коли він проходить по ділянці зі стеганографічним шумом, він підсилює аномальні зміни значень пікселів.

Для нейромережі зміст самої картинки є завадою. Якщо подати до моделі просто фото, згорткові шари будуть вираховувати ознаки картинки, а не повідомлення. SRM-фільтри перетворюють зображення на карту залишків, що змусить модель фокусуватися виключно на статистичних аномаліях, які не мають нічого спільного з тим, що зображено на фото.

Для поточної архітектури застосовано 5 різних фільтрів для підсилення певних типів шуму. Математичне представлення кожного з 5 фільтрів наведено нижче:

– $\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ – фільтр горизонтального краю, призначений для виявлення різких горизонтальних змін у пікселях;

– $\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ – фільтр вертикального краю, призначений для виявлення вертикальних переходів;

– $\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ – фільтр діагонального краю, призначений для виявлення діагональних слідів при складних алгоритмах вбудування;

– $\begin{bmatrix} -1 & 2 & -1 \\ 2 & -4 & 2 \\ -1 & 2 & -1 \end{bmatrix}$ – фільтр, чутливий до адитивного шуму, призначений для порівняння центрального пікселя з усім оточенням;

– $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ – класичний фільтр Лапласа, призначений для виділення високочастотного шуму [26].

При цьому згортковий шар, що обробляє зображення цими фільтрами налаштований так, щоб застосовувати кожен з цих фільтрів до кожного з каналів зображення окремо. Це дозволить запобігти розмиттю чутливих даних. Програмна реалізація SRM-енкодера представлена на рисунку 3.2.

```
class SRMEncoder(nn.Module):  1 usage
    def __init__(self):

        super().__init__()

        filters = [
            [[0, 0, 0], [0, -1, 1], [0, 0, 0]],
            [[0, 0, 0], [0, -1, 0], [0, 1, 0]],
            [[0, 0, 0], [0, -1, 0], [0, 0, 1]],
            [[-1, 2, -1], [2, -4, 2], [-1, 2, -1]],
            [[0, -1, 0], [-1, 4, -1], [0, -1, 0]]
        ]

        kernels = torch.tensor(filters, dtype=torch.float).unsqueeze(1)
        kernels = kernels.repeat(3, 1, 1, 1)

        self.srm_conv = nn.Conv2d( in_channels: 3,
                                   out_channels: 15,
                                   kernel_size=3,
                                   padding=1,
                                   groups=3,
                                   bias=False,
                                   padding_mode='reflect')
        self.srm_conv.weight = nn.Parameter(kernels, requires_grad=False)

    def forward(self, x):
        return self.srm_conv(x)
```

Рисунок 3.2 – SRM-енкодер

Після обробки вхідного зображення цими фільтрами, карта залишків передається послідовності звичайних згорткових шарів. Тут є важливий

нюанс. Шари пулінгу, що зазвичай використовуються для зменшення витрат пам’яті шляхом стиснення карти ознак, майже гарантовано зітруть важливу інформацію. Якщо ж цього не зробити модель просто вдавиться колосальною кількістю даних.

Це ще раз підтверджує перевагу використання власно створеного набору навчальних даних. Якщо всі зображення датасету від початку будуть малого розміру це вирішить одночасно і проблему втрати інформації і нестачу пам’яті для обчислень.

Сам згортковий енкодер складається з трьох звичайних згорткових шарів, за кожним з яких слідує шар нормалізації батчу, функція активації LeakyReLU(20%) та Dropout(20%).

BatchNorm дозволить використовувати вищий коефіцієнт темпу навчання, можливо трохи його пришвидшить, та підійде у якості легкої регуляризації. LeakyReLU допоможе запобігти проблемі мертвого ReLU, коли нейрон потрапляє в зону від'ємних значень, і його градієнт стає рівним нулю, вимикаючи його назавжди. Також, оскільки модель зосереджена на роботі із шумом, де від'ємні відхилення від середнього так само важливі як і додатні, звичайний ReLU може знищити левову частку корисної інформації [17].

Dropout завадить нейронам ставати занадто залежними один від одного. Це допоможе моделі не заучувати текстуру конкретних зображень, а шукати загальні закономірності вбудовування коду чи тексту.

Після формування карти ознак до неї додається ще один канал з просторовими вагами. Він служить для того, щоб згортка самостійно вчилась де шукати корисний сигнал. У результаті згортковий енкодер надає карту ознак, що повністю відповідає розміру вхідного зображення без втрати будь-яких даних. Отримані ознаки готові до подачі на обробку до механізму уваги.

Програмна реалізація згорткового енкодера представлена на рисунку 3.3.

```
class CNNEncoder(nn.Module):

    def __init__(self):
        super().__init__()

        self.SRMEncoder = SRMEncoder()
        self.spatial_weight = nn.Parameter(torch.ones(1, 1, 4096))
        self.encoder = nn.Sequential(

            nn.Conv1d( in_channels: 15,  out_channels: 45,
                       kernel_size: 5, padding=2, groups=15, stride=1, padding_mode='reflect'),
            nn.BatchNorm1d(45),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.1),

            nn.Conv1d( in_channels: 45,  out_channels: 60,  kernel_size: 3,
                      padding=1, stride=1, padding_mode='reflect'),
            nn.BatchNorm1d(60),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.1),

            nn.Conv1d( in_channels: 60,  out_channels: 75,  kernel_size: 3,
                      padding=1, stride=1, padding_mode='reflect'),
            nn.BatchNorm1d(75),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.1)
        )

    def forward(self, x):
        x = self.SRMEncoder(x)
        b, c, h, w = x.shape
        x = x.view(b, c, h * w)
        weight = torch.sigmoid(self.spatial_weight)
        return self.encoder(x*weight)
```

Рисунок 3.3 – CNN-енкодер

Використана архітектура механізму уваги має класичну структуру multihead attention. Перед цим вхідні дані перетворюються на валідну послідовність, після чого до них додається позиційне кодування. Підготовлена послідовність може бути відправлена на вхід attention.

Після обробки даних механізмом уваги наступним кроком буде остаточна класифікація через лінійні шари. Вхідний лінійний шар виконує роль прихованого за яким слідує звичайна активація ReLU та Dropout(20%). Останній лінійний шар наприкінці визначає до якого з трьох доступних класів належить зображення. Програмна реалізація механізму уваги представлена на рисунку 3.4.

```python
class CNNAttentionModel(nn.Module):

    def __init__(self, num_classes, cnn_encoder, input_size=(3, 64, 64)):
        super().__init__()

        self.encoder = cnn_encoder
        dummy_input = torch.zeros(1, *input_size)
        with torch.no_grad():
            dummy_output = self.encoder(dummy_input)
        _, self.embed_dim, self.seq_len = dummy_output.shape
        # self.seq_len = h * w
        self.pos_embedding = nn.Parameter(torch.randn(1, self.seq_len, self.embed_dim))
        self.attention = nn.MultiheadAttention(
            embed_dim=self.embed_dim,
            num_heads=3,
            batch_first=True,
            dropout=0.1
        )
        self.classifier = nn.Sequential(
            nn.Linear(self.embed_dim,  out_features: 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear( in_features: 128, num_classes)
        )

    def forward(self, x):

        features = self.encoder(x)
        features = features.permute(0, 2, 1)
        features = features + self.pos_embedding
        attn_output, _ = self.attention(
            features,
            features,
            features
        )
        pooled = attn_output.mean(dim=1)
        out = self.classifier(pooled)
        return out
```

Рисунок 3.4 – Attention з класифікатором

3.1.1 Експеримент №1

Розроблену модель було натреновано на 1500 зображеннях. З них 500 чисті з невеликим зашумленням, 500 з корисним навантаженням у вигляді тексту та 500 з корисним навантаженням у вигляді коду. Для тренування розробленої моделі використовувалась стандартна стратегія навчання. Налаштування гіперпараметрів моделі представлено у таблиці 3.1.

Таблиця 3.1 – Гіперпараметри моделі CNN+Attention

Гіперпараметр	Функція/Значення
блок CNN	
Кількість згорткових шарів	3
Кількість фільтрів	45, 60, 75
Розмір ядра	5, 3, 3
Stride	1, 1, 1
Padding	2, 1, 1
Активація	LeakyReLU(0.2)
Нормалізація	BatchNorm1d
Dropout	Dropout(0.1)
Пулінг	–
блок Attention	
Тип уваги	Self-attention
Розмір ембедингу	75
Кількість голів	4
Attention dropout	0.1
блок класифікації	
Тип пулінгу	mean
Кількість прихованих вимірів	128
Кількість шарів	2
Dropout	0.3
Активація	ReLU
тренування	
Оптимізатор	AdamW
Швидкість навчання	3e-4
Згасання ваг	0.01
Scheduler	ReduceLROnPlateau
Розмір батчу	8
Кількість епох	100
Функція втрат	CrossEntropyLoss

Вказані вище гіперпараметри є остаточною версією моделі поточної архітектури для цього експерименту. Процес навчання адаптовано таким чином, щоб робити зміни та покращення, без потреби у додатковій логіці. Програмна реалізація циклу навчання наведена на рисунку 3.5.

Представлений цикл є стандартним повторенням процесу навчання протягом певної кількості епох, що вказана у таблиці вище. На початку кожної епохи модель переводиться у режим навчання. Разом з тим

ініціалізуються списки для накопичення значень функції втрат, прогнозів та справжніх міток.

```
for epoch in range(TRAIN_EPOCHS):
    print(f"EPOCH {epoch}")
    model.train()
    buffered_losses = []
    predictions = []
    labels = []
    for i, (image, label) in enumerate(tqdm(train_loader,
                                            file=sys.stdout,
                                            total=BATCH_LIMIT,
                                            desc="TRAINING..."):

        image = image.to(device)
        label = label.to(device)
        logits = model(image)
        predictions.append(torch.argmax(logits, dim=1).detach().cpu())
        labels.append(label.detach().cpu())
        loss = loss_fn(logits, label)
        buffered_losses.append(loss.item())
        loss.backward()
        clip_grad_norm_(model.parameters(), max_norm: 1.0)
        optimizer.step()
        optimizer.zero_grad()
        if i >= BATCH_LIMIT-1: break

    train_losses.append(sum(buffered_losses)/len(buffered_losses))
    train_accuracies.append(accuracy(torch.cat(labels),torch.cat(predictions)))
    print(train_losses[-1])
```

Рисунок 3.5 – Цикл навчання

Дані подаються батчами через ітератор з відстежуванням прогресу, при цьому додано можливість обмеження кількості оброблюваних батчів для економії ресурсів або тестових проходів. Дані зображень та відповідних міток з батчу переноситься на обчислювальний пристрій з підтримкою CUDA для апаратного прискорення. Модель генерує логіти, з яких формуються передбачення шляхом вибору класу з максимальною ймовірністю. Базуючись на передбаченнях, обчислюється точність моделі та значення втрат за допомогою відповідної функції, вказаної у таблиці

вище, і обидва зберігаються як значення метрик. Виконується зворотне поширення помилки для обчислення градієнтів, після чого застосовується обрізка градієнтів для стабілізації навчання. Алгоритм оптимізатора, вказаний у таблиці вище, оновлює параметри моделі, а градієнти обнуляються для наступного кроку. Після обробки визначеної кількості пакетів цикл переходить до обчислення середнього значення функції втрат за епоху та точності навчання, які також зберігаються в якості метрик[21].

Цикл валідації моделі виконується після кожної епохи навчання з метою оцінки здатності моделі до генералізації на наборі даних відсутньому у навчальній вибірці. Програмна реалізація циклу валідації представлено на рисунку 3.6.

```
model.eval()
val_epochs.append(epoch)
with torch.inference_mode():
        val_avg_loss = []
        val_labels = []
        val_predictions = []
        for i, (vimage, vlabel) in enumerate(tqdm(val_loader,
                                                  file=sys.stdout,
                                                  total=VAL_BATCH_LIMIT,
                                                  desc="VALIDATION..."):

            vimage = vimage.to(device)
            vlabel = vlabel.to(device)
            logits = model(vimage)
            val_avg_loss.append(loss_fn(logits, vlabel).detach().cpu())
            val_predictions.append(torch.argmax(logits, dim=1).detach().cpu())
            val_labels.append(vlabel.detach().cpu())
            if i >= VAL_BATCH_LIMIT-1: break

        val_loss = sum(val_avg_loss) / len(val_avg_loss)
        scheduler.step(val_loss)
        val_losses.append(val_loss)
        val_accuracies.append(accuracy(torch.cat(val_labels), torch.cat(val_predictions)))
        print(val_losses[-1])
```

Рисунок 3.6 – Цикл валідації

Одразу після циклу навчання модель переводиться у режим оцінки, що вимикає специфічні для навчання механізми, як Dropout і подібні. Потім фіксується номер поточної епохи для виведення метрик. Для зменшення

витрат обчислень та пам’яті використовується режим передбачення, який відключає обчислення градієнтів.

Набори даних для валідації подаються батчами через ітератор з аналогічним обмеженням кількості оброблюваних батчів. Кожен батч, так само як і при тренуванні, переноситься на обчислювальний пристрій. Модель генерує логіти, на основі яких обчислюється функція втрат, значення якої накопичуються для метрик. На основі отриманих прогнозів вираховується точність моделі та зберігається для вирахування метрик.

Після обробки вказаної кількості батчів обчислюється середнє значення функції втрат за цикл валідації. Це значення передається планувальнику навчання, функція якого вказана у таблиці вище, для коригування темпу навчання залежно від якості моделі. Середнє значення втрат та точність валідації, обчислена на основі усіх прогнозів і міток, зберігаються як метрики.

Результати навчання експериментальної моделі CNN+Attention наведено у вигляді графіків точності та втрат при тренуванні представлених на рисунках 3.7–3.8 відповідно

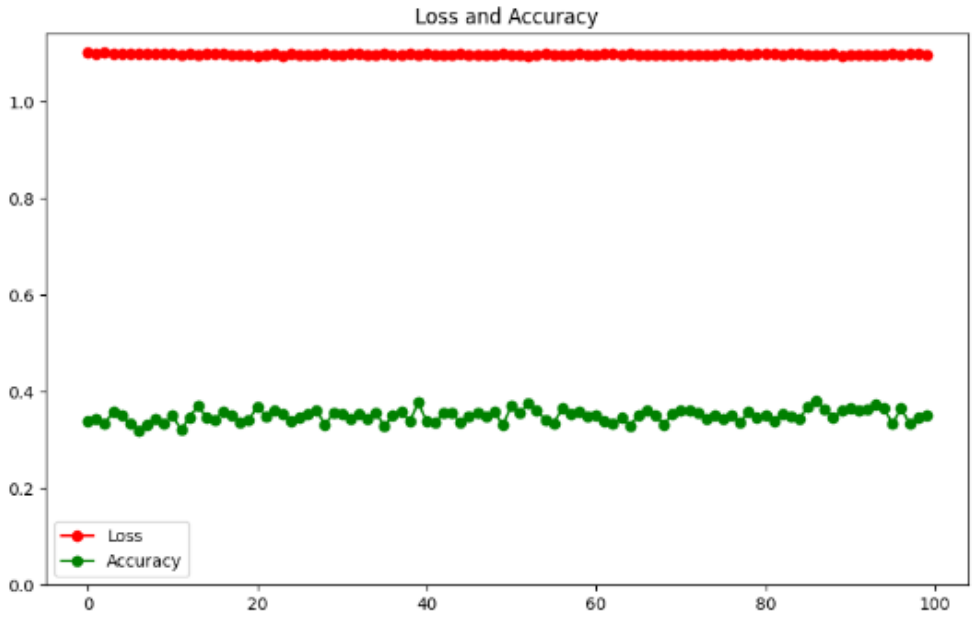

Рисунок 3.7 – Точність та втрати при навчанні

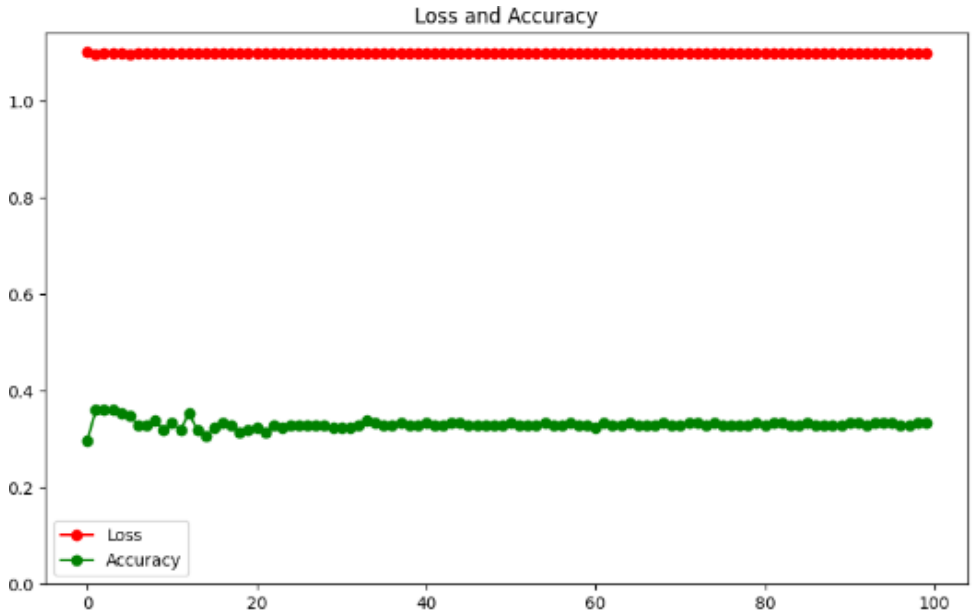

Рисунок 3.8 – Точність та втрати при валідації

Не зважаючи на добре продуману архітектуру, яка в теорії повинна була забезпечити безвідмовний успіх моделі, результати показали, без перебільшення, жахливу успішність. Як видно на графіку валідації, значення втрат протягом всього навчання залишаються максимально високими, а точність ані трохи не покращується. Це свідчить про те, що модель не здатна до генералізації. Більш того графік навчання висвічує майже ідентичні значення. Це означає що модель навіть не здатна до перенавчання. Тобто вона апріорі не зможе нічого вивчити.

Ретельне дослідження отриманих результатів експерименту виявило декілька можливих причин незадовільної роботи моделі. Серед них:

– етап препроцесингу (SRM-шар);

– одновимірні згорткові шари;

– слабкий градієнт;

– нерозрізнимі класи.

Найбільш вірогідною причиною можна вважати саме SRM-шар. Так як різниця між чистим зображенням і зміненим це один біт в одному пікселі, на початку вважалося що SRM зможе це виявити. Але схоже, що сигнал був

надзвичайно слабкий. Настільки, що він загубився серед природного шуму зображення, де різниці значень сусідніх пікселів можуть бути 10 чи 50. Глибше дослідження дає підставу припустити, що SRM добре спрацював би для DCT стеганографії, де зміни розподілені по частотних коефіцієнтах і створюють систематичні просторові аномалії. У поточному ж дослідженні алгоритм LSB змінює пікселі незалежно без просторової структури, що зводить нанівець усі потуги фільтрів [3].

Менш імовірна причина – використання 1D згорток замість 2D. Щоб поточна згортка змогла обробити зображення, після SRM-шару воно перетворюються на послідовність, через що втрачається зв’язок між верхніми та нижніми пікселями. Але так як використаний LSB алгоритм вбудовує повідомлення у зображення так само послідовно, ця проблема вважається некритичною.

Проблема слабкого градієнта складніша. Завчасно відомо, що для поточної моделі найлегший спосіб підвищити спроможність виявляти закономірності у даних, це звертати увагу на перші 100 позицій послідовності, у яких закодована довжина стеганографічного повідомлення. Але дослідження проблеми показало, що при вирахуванні градієнта, після проходження через mean pooling він ділиться рівномірно на велику кількість позицій послідовності. Лише для механізму уваги щоб сфокусуватись на перших позиціях послідовності потрібно пройти багато кроків навчання, не кажучи вже про згортку, що знаходиться ще далі.

Проблема нерозрізнимих класів це більш фундаментальна проблема. Якщо візуалізувати біти звичайного повідомлення і коду до вбудування то побачити різницю між ними можна. Але при кодуванні повідомлення у LSB все змінюється. Після вбудовування різниця все ще присутня, але якщо дані представити у вигляді статистики, структура для обох класів виявиться майже однаковою.

3.1.2 Експеримент №2

Зважаючи на виявлені вади архітектури прийнято рішення змінити її складові. Перш за все, як вже визначено, SRM-фільтри не здатні підсилити сигнал стеганографії вбудованої методом зміни найменшого біта, тож логічним рішенням буде його прибрати. Разом із тим варто спробувати використати двовимірні згорткові шари замість одновимірних. Програмну реалізацію оновленого CNN-енкодера представлено на рисунку 3.9.

```
class CNNEncoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d( in_channels: 3,  out_channels: 15,  kernel_size: 3,
                      padding=1,
                      groups=3,
                      padding_mode='reflect'),
            nn.BatchNorm2d(36),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.1),
            nn.Conv2d( in_channels: 15,  out_channels: 45,  kernel_size: 3,
                      padding=1,
                      groups=3,
                      padding_mode='reflect'),
            nn.BatchNorm2d(45),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.1),
            nn.Conv2d( in_channels: 45,  out_channels: 60,  kernel_size: 3,
                      padding=1,
                      groups=3,
                      padding_mode='reflect'),
            nn.BatchNorm2d(60),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.1)
        )
    def forward(self, x):
        return self.encoder(x)
```

Рисунок 3.9 – Оновлений CNN-енкодер

При заміні одновимірних згорток на двовимірні було переосмислено відповідні гіперпараметри. Варто відзначити найважливіше, що для запобігання перемішуванню даних з трьох вхідних каналів аж до самого виходу енкодера . на всіх згорткових шарах ознаки буде розділено на три відповідні групи. Оновлені гіперпараметри наведено у таблиці 3.2.

Таблиця 3.2 – Змінені гіперпараметри моделі CNN+Attention

Гіперпараметр	Функція/Значення
блок CNN	
Кількість згорткових шарів	3
Кількість фільтрів	15, 40, 60
Розмір ядра	3, 3, 3
Stride	1, 1, 1
Padding	1, 1, 1
Padding mode	reflect
Groups	3, 3, 3
Активація	LeakyReLU(0.2)
Нормалізація	BatchNorm2d
Dropout	Dropout(0.1)
Пулінг	–
блок Attention	
Тип уваги	Self-attention
Розмір ембедингу	75
Кількість голів	4
Attention dropout	0.1
блок класифікації	
Тип пулінгу	mean
Кількість прихованих вимірів	128
Кількість шарів	2
Dropout	0.3
Активація	ReLU
тренування	
Оптимізатор	AdamW
Швидкість навчання	3e-4
Згасання ваг	0.01
Scheduler	ReduceLROnPlateau
Розмір батчу	8
Кількість епох	100
Функція втрат	CrossEntropyLoss

Як видно, загалом гіперпараметри не змінились, не враховуючи згортковий енкодер. Так як ядра у згорткових шарах замінено на двовимірні, то вихідна карта ознак вже не буде послідовністю. Тобто єдину зміну яку необхідно внести у блок механізму уваги, це явний процес конвертації карти ознак у послідовність. В іншому механізм уваги залишено без змін.

Результати навчання експериментальної моделі на базі CNN+Attention наведено у вигляді графіку точності та втрат при тренуванні на рисунку 3.10.

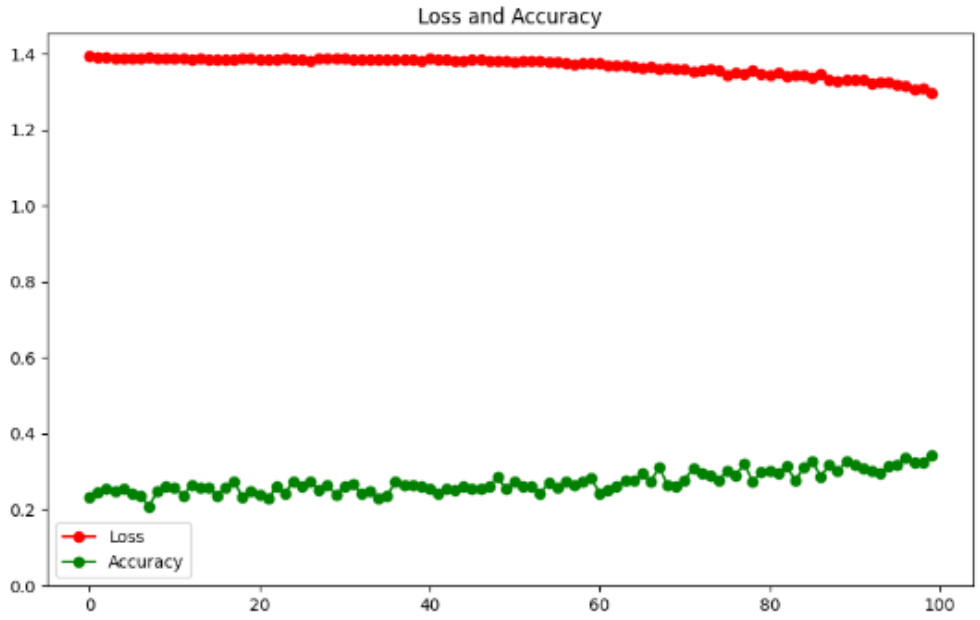

Рисунок 3.10 – Втрати та точність при тренуванні

З представлених результатів чітко видно, що високі втрати поступово почали йти на спад, при цьому точність від неймовірно низької так само повільно йде вгору. Це вказує на те, що модель при поточній архітектурі та конфігурації почала проявляти ознаки поступового навчання. Але, зважаючи на те скільки епох було витрачено на такий повільний прогрес і скільки на це було витрачено часу, результати не дуже задовільні.

Результати навчання експериментальної моделі на базі CNN+Attention у вигляді графіку точності та втрат при валідації наведено на рисунку. 3.11.

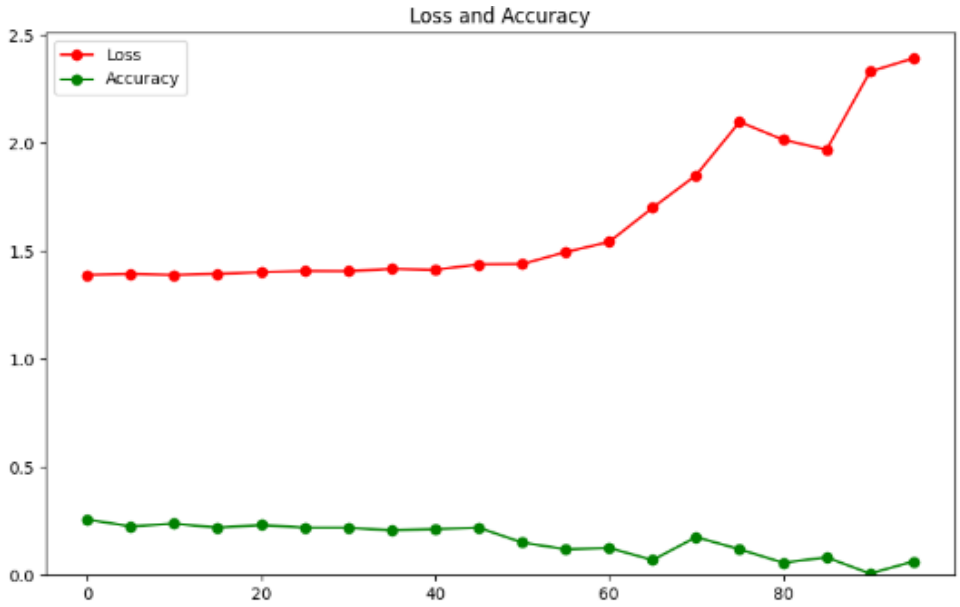

Рисунок 3.11 – Втрати та точність при валідації

З представленого графку втрат та точностей при валідації доволі чітко можна зрозуміти причину поступового падіння значення втрат та поступове підвищення точності у циклі тренування моделі. Різке підвищення значень втрат та зниження точності на валідації в момент їх протилежної зміни при тринуванні є явною ознакою перенавчання [25].

Модель, не виявивши нічого спільного між картинками одного класу та нічого відміного між картинками різних класів, почала зазубрювати ознаки наявні тільки у тренувальній вибірці. А так як дані валідаційної вибірки вона ніколи не бачила, то і правильно їх класифікувати не зможе. Зважаючи на вище зазначне, експеримент визнано провальним.

3.2 Спрощення задачі. Архітектура на базі CNN

Зважаючи на критичні вади попередньої моделі, прийнято рішення спростити задачу до бінарної класифікації. Разом із тим спрощено і саму архітектуру моделі класифікатора до звичайної згорткової нейронної мережі. Нова архітектура моделі представлена на рисунку 3.12.

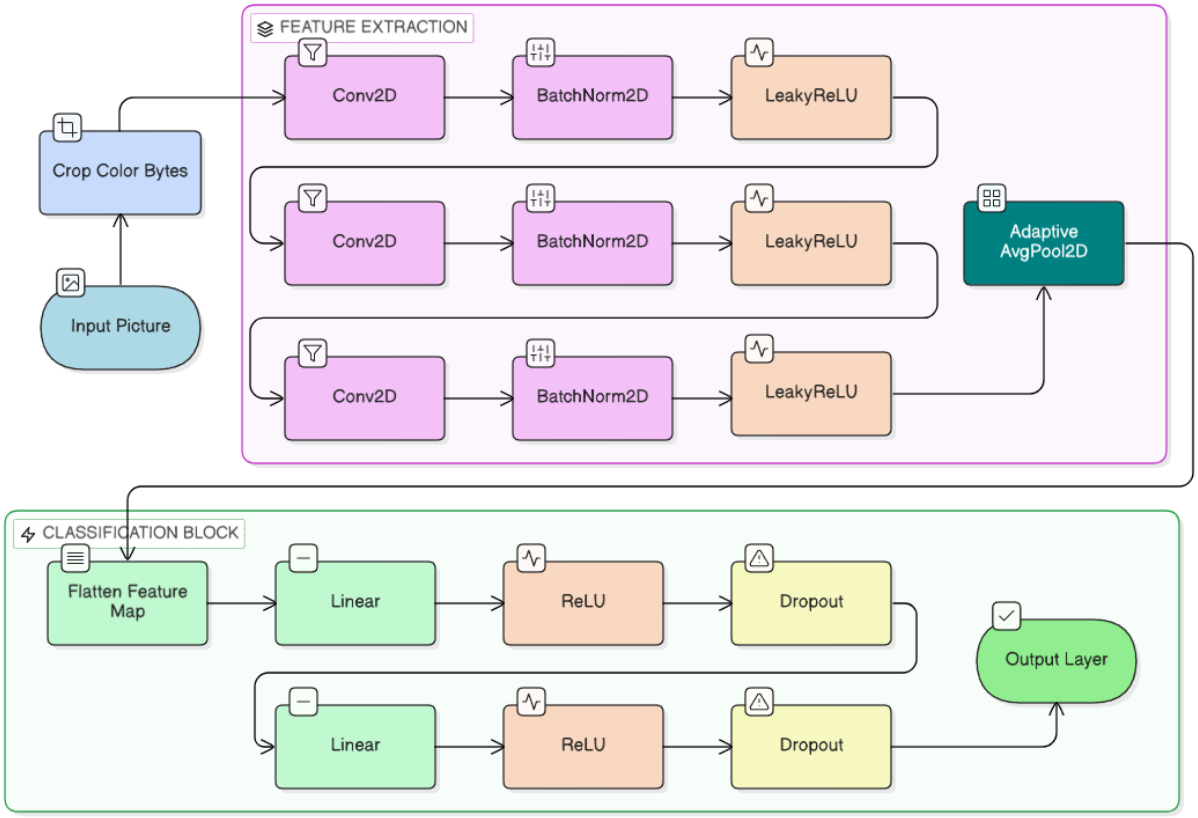

Рисунок 3.12 – Архітектура на базі CNN

Головна особливість представленої архітектури полягає у тому, що перед процесом витягнення ознак байти вхідного зображення обрізаються, залишаючи лише останній біт кожного каналу. Це дозволить моделі зосередитись на закодованому повідомленні, відкидаючи увесь сторонній шум.

3.2.1 Експеримент №3

За логікою архітектури, карта бітів захованого повідомлення, отримана після обрізання, оброблятиметься кількома згортковими шарами, навчаючись витягати з нього відповідні закономірності. Після кожної згортки розміщений шар нормалізації та шар активації. Отримана карта ознак одразу передається у класифікаційний блок, що складається з

декількох прихованих повнозв'язних шарів, за якими слідують функції активації та регуляризації.

Програмна реалізація архітектури моделі представлена на рисунку 3.13.

```
class LSBEncoder(nn.Module):  1 usage
    def __init__(self):
        super().__init__()

        self.encoder = nn.Sequential(
            nn.Conv2d( in_channels: 3,  out_channels: 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2),
            nn.Conv2d( in_channels: 32,  out_channels: 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),
            nn.Conv2d( in_channels: 64,  out_channels: 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.AdaptiveAvgPool2d((4, 4))
        )
    def forward(self, x):
        lsb = (x * 255).round().to(torch.int32) & 1
        lsb = lsb.float()
        return self.encoder(lsb)

class LSBDetector(nn.Module):
    def __init__(self, num_classes=None):
        super().__init__()

        self.lsb_encoder = LSBEncoder()
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 4 * 4,  out_features: 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear( in_features: 256,  out_features: 64),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear( in_features: 64, num_classes)
        )
    def forward(self, x):
        features = self.lsb_encoder(x)
        return self.classifier(features)
```

Рисунок 3.13 – Програмна реалізація архітектури

За своєю суттю модель є звичайною згортковою нейронною мережею з усіма виходящими. Тренування такої моделі прийнято на початку залишити стандартним. Опис визначених гіперпараметрів моделі представлено у таблиці 3.3.

Таблиця 3.3 – Гіперпараметри CNN моделі

Гіперпараметр	Значення/Функція
блок CNN	
Кількість згорткових шарів	3
Кількість фільтрів	32, 64, 128
Розмір ядра	3, 3, 3
Stride	1, 1, 1
Padding	1, 1, 1
Нормалізація	BatchNorm2d
Активація	LeakyReLU(0.2)
Dropout	–
Пулінг	AdaptiveAvgPool2d((4, 4))
блок класифікації	
Кількість лінійних шарів	3
Ширина шарів	256, 64, 1
Активація	ReLU
Dropout	0.2, 0.2, –
тренування	
Оптимізатор	AdamW
Швидкість навчання	3e-4
Згасання ваг	0.01
Scheduler	ReduceLROnPlateau
Розмір батчу	16
Кількість епох	10
Функція втрат	BCEWithLogitsLoss

Датасет для навчання переналаштовано для бінарної класифікації звичайних зображень без корисного навантаження та з вбудованим кодом програми. В інших аспектах датасет залишено без змін.

У якості реалізації циклу навчання для поточної моделі вирішено використати цикл навчання попередньої архітектури, так як логіка навчання

залишилась майже без змін. При цьому перед ініціалізацією навчання прийнято рішення про відстеження додаткових часових метрик, таких як:

– середній час проходження епохи;

– максимальний час на проходження епохи;

– мінімальни час на проходження епохи;

– загальний час навчання.

Програмна реалізація циклу для тренування представлена на рисунку 3.14.

```
for epoch in range(TRAIN_EPOCHS):
    epoch_time = time.time()
    print(f"EPOCH {epoch}")
    model.train()
    buffered_losses = []
    buffer_accuracies = []
    labels = []
    for i, (image, label) in enumerate(tqdm(train_loader,
                                            file=sys.stdout,
                                            desc="TRAINING...")):
        image = image.to(device)
        label = label.to(device).float()
        logits = model(image)
        predictions = torch.round(torch.sigmoid(logits)).detach().cpu()
        buffer_accuracies.append(accuracy(label.detach().cpu().unsqueeze(1), predictions))
        loss = loss_fn(logits, label.unsqueeze(1))
        buffered_losses.append(loss.item())
        loss.backward()
        clip_grad_norm_(model.parameters(), max_norm: 1.0)
        optimizer.step()
        optimizer.zero_grad()
        # if i >= BATCH_LIMIT-1: break
    end_time = time.time()
    train_losses.append(sum(buffered_losses)/len(buffered_losses))
    train_accuracies.append(sum(buffer_accuracies)/len(buffer_accuracies))
    time_for_epoch.append(time.time()-epoch_time)
    print(train_losses[-1])
```

Рисунок 3.14 – Цикл тренування

Представлений цикл навчання майже нічим не відрізнається від того, що був використаний при тренуванні моделі з архітектурою CNN+Attention. Серед очевидних відмінностей згадані вище методи відстежування часових

метрик. Для цього використовується стандартна бібліотека time у Python. На початку кожної ітерації тренування фіксується час її початку. Після проходження усіх ітерацій по наявним батчам таймер фіксує час завершення поточної епохи тренування. В кінці вираховується фактичний обсяг часу витраченого на епоху.

Поміж цього було прибрано опцію обмеження кількості оброблюваних батчів. Так як поточна архітектура моделі використовує в рази менше навчених параметрів потреба у подібних обмеженнях вже не є актуальною. Цикл валідації так само як і цикл тренування також прийнято рішення не змінювати від попереднього експерименту. Програмна реалізація циклу для валідації представлена на рисунку 3.15.

```
model.eval()
val_epochs.append(epoch)
with torch.inference_mode():
    val_avg_loss = []
    buffer_val_accuracies = []
    for i, (vimage, vlabel) in enumerate(tqdm(val_loader,
                                              file=sys.stdout,
                                              desc="VALIDATION...")):
        vimage = vimage.to(device)
        vlabel = vlabel.to(device).float()
        logits = model(vimage)

        val_avg_loss.append(loss_fn(logits, vlabel.unsqueeze(1)).detach().cpu())
        val_predictions = torch.round(torch.sigmoid(logits)).detach().cpu()
        buffer_val_accuracies.append(accuracy(vlabel.detach().cpu().unsqueeze(1),val_predictions))
        # if i >= VAL_BATCH_LIMIT-1: break

    val_loss = sum(val_avg_loss) / len(val_avg_loss)
    scheduler.step(val_loss)
    val_losses.append(val_loss)
    val_accuracies.append(sum(buffer_val_accuracies)/len(buffer_val_accuracies))
    print(val_losses[-1])
```

Рисунок 3.15 – Цикл валідації

Для циклу валідації так само відключено обмеження кількості оброблюваних батчів, через недоцільність. При цьому час затрачений на валідацію в даному випадку не вираховується. В момент завершення епох

навчання час фіксується та використовується в метриках як загальний час навчання моделі.

Результати навчання експериментальної моделі на базі CNN наведено у вигляді графіків точності та втрат при тренуванні наведених на рисунках 3.16–3.17 відповідно.

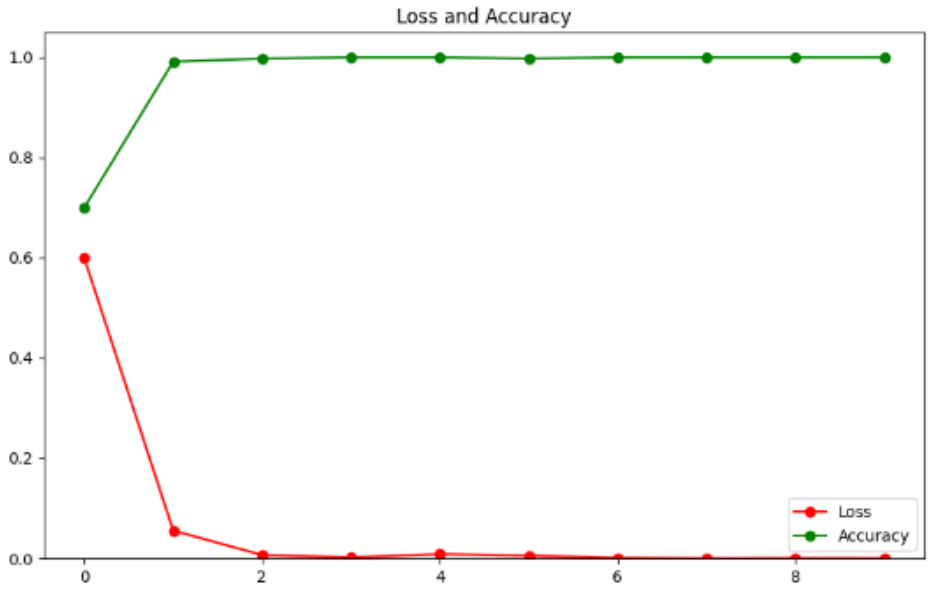

Рисунок 3.16 – Втрати та точність при тренуванні

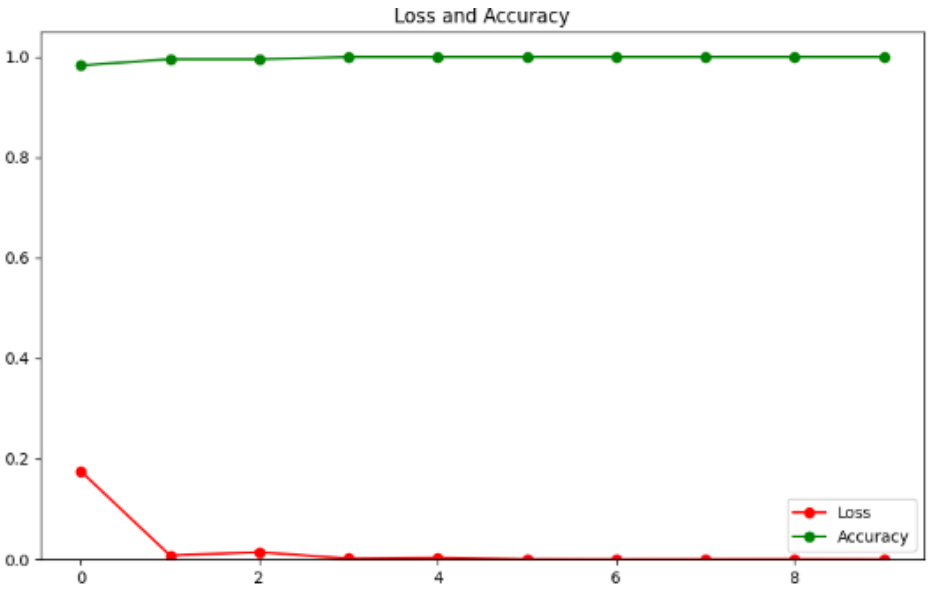

Рисунок 3.17 – Втрати та точність при валідації

Як видно з результатів, модель з поточною архітектурою бездоганно впоралась з поставленою задачею. Втрати при навчанні знизились майже до 0 на першій же епосі, так само точність виросла до 100%. Така сама картина спостерігається і на графіку валідації, що означає беззаперечну здатність моделі до генералізації. Часові показники характеристики процесу навчання моделі для поточного експерименту наведено у таблиці 3.4.

Таблиця 3.4 – Часові показники поточного експерименту

Показник	Значення, сек.
average time for epoch:	0.59
max time for epoch:	0.86
min time for epoch:	0.56
total time for training:	6.94

З часових показників видно, що модель повністю навчилась за лічені секунди, що в сотні разів ефективніше за попередню архітектуру, модель якої безрезультатно витратила години, так нічому і не навчившись. Таким чином спроба навчити модель для розрізнення чистих зображень та видозмінених за допомогою стеганографії вважається успішною.

Але головна проблема може виникнути при спробі класифікації саме двох видів корисного навантаження закодованих у зображення. Спершу варто перевірити чи взагалі відрізняється код від звичайного тексту на рівні біт. Бо по факту модель не здатна зчитувати суть повідомлення, тож припускається що вона шукає статистичні закономірності в бітах. Питання полягає в тому, чи вони взагалі там присутні.

3.2.2 Опис підґрунтя експерименту №3

Питання про різницю між класами виникає на підставі того, що у повідомленні і текст, і код це символи ASCII. Тобто вони мають схожий розподіл байтів, різниця між якими не статистична, а семантична. Але код, як правило, містить специфічні ASCII символи яких мало в звичайному

тексті. Це може створити різний статистичний розподіл у бітах навіть в межах ASCII. Варто спочатку перевірити статистичні дані між звичайним текстом та кодом[2]. Якщо значення між класами різняться, це може означати, що модель навчилась на статистиці символів конкретних файлів які були використані для генерації датасету. Програмна реалізація методу перевірки статистичних даних датасету представлено на рисунку 3.18.

```
for images, labels in train:
    lsb = (images * 255).round().to(torch.int32) & 1
    flat = lsb.view(images.shape[0], 3, -1)[:, 0, :]

    for cls in [1, 2]:
        mask = labels == cls
        if mask.sum() > 0:
            bits = flat[mask][:, 32:].float()
            print(f"Клас {cls} LSB mean (без заголовка): {bits.mean():.4f}")
            print(f"Клас {cls} LSB std:                  {bits.std():.4f}")
    break
```

Рисунок 3.18 – Метод статистичної перевірки

Представлений код імітує передобробку зображення в моделі обрізаючи все окрім останнього біта, щоб отримати тензор з яким і працює модель поточної архітектури. Тензор зображення перетворюється з формату зображення з каналами, довжиною та висотою на формат з каналами та прямою послідовністю пікселів.

Серед наявних каналів для перевірки обирається саме червоний канал, так як алгоритм вбудування повідомлення починає енкодинг саме з цього каналу. Перші 32 елемента повідомлення відповідають за його довжину і мають загальну структуру і їх статистика була б однаковою для обох класів. Тому заголовок вирішено пропустити щоб аналізувати виключно біти повідомлення.

З отриманої послідовності вираховуються стандартне відхилення та середнє арифметичне. Для ідеально випадкового розподілу нулів та одиниць у молодших бітах зображення середнє значення повинно дорівнювати 0.5. Стандартне відхилення при цьому також буде на рівні 0.5 і зменшуватись

при зміщенні середнього значення у будь-який бік.Результати статистичної перевірки наведено у таблиці 3.5.

Таблиця 3.5 – Статистичні показники закодованих зображень

Показник	Значення, сек.
Клас 1 LSB mean (без заголовка):	0.4823
Клас 1 LSB std	0.4997
Клас 2 LSB mean (без заголовка):	0.4942
Клас 2 LSB std	0.5000

Зі статистичних показників видно що середні значення та стандартні відхилення обох класів майже ідентичні і близькі до 0.5. Для розуміння, у необроблених фотографіях середнє значення та стандартне відхилення також зазвичай близькі до 0.5. Тобто молодші біти пікселів поводяться майже як випадковий шум.

Мало того, що зображення з корисним навантаженням статистично не відрізняються від звичайних, вони не відрізняються навіть між собою. Відповідно є висока вірогідність, що модель може бути не здатна розрізнити ці класи.

3.2.3 Експеримент №4

Не зважаючи на невтішні прогнози, прийнято рішення здійснити спробу навчання поточної моделі для бінарної класифікації зображень з вбудованим текстом та зображення з вбудованим кодом у якості корисного навантаження. Архітектуру моделі прийнято залишити без змін. Датасет сформовано з 1000 різних зображень, по 500 зображень на кожен клас. Гіперпараметри для циклу навчання залишено без змін.

Результати навчання експериментальної моделі на базі CNN для класифікації корисного навантаження у зображеннях наведено у вигляді графіків точності та втрат при тренуванні та при валідації на рисунках 3.19–3.20 відповідно.

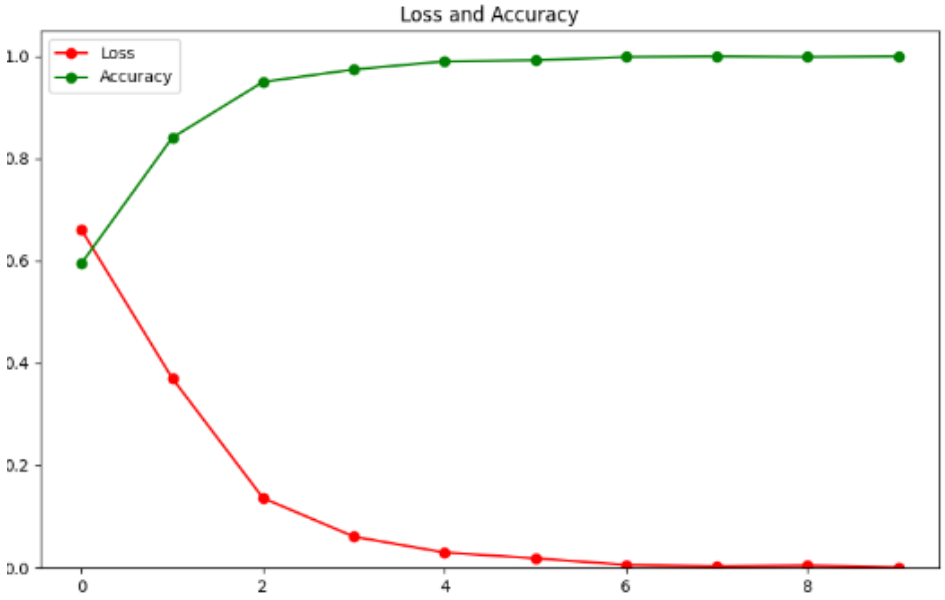

Рисунок 3.19 – Втрати та точність при тренуванні

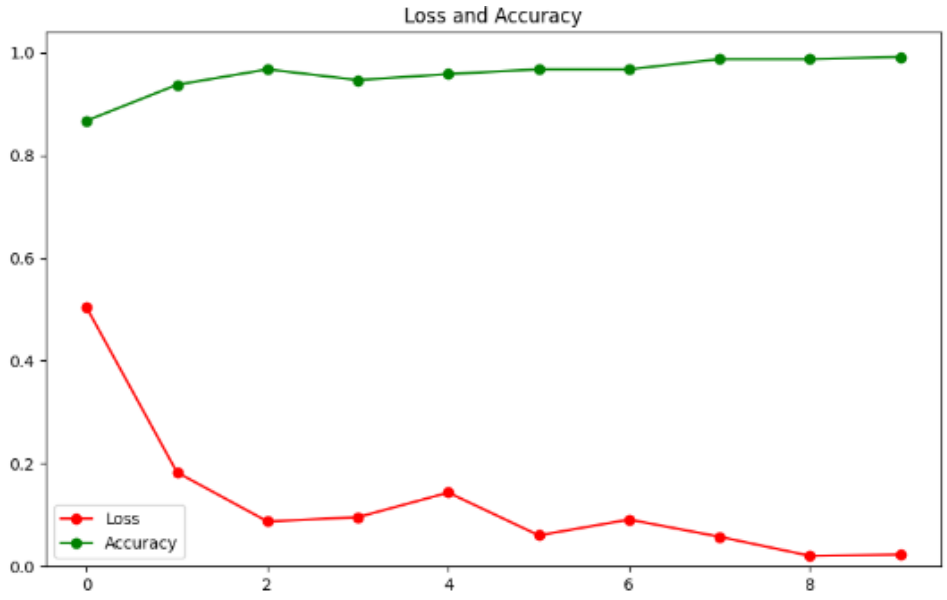

Рисунок 3.20 – Втрати та точність при валідації

Як видно з графіків, не зважаючи на усі шанси на провал, поточна модель бездоганно впоралася зі своєю задачею. При тренуванні втрати безперервно знижуються, а точність тренування росте. Разом із тим результати валідації вказують на здатність моделі до генералізації.

Також варто звернути увагу на часові показники представлені у таблиці 3.6.

Таблиця 3.6 – Часові показники моделі на основі CNN

Показник	Значення, сек.
average time for epoch:	0.81
max time for epoch:	3.08
min time for epoch:	0.55
total time for training:	9.27

З часових даних видно, що модель навчилася за лічені секунди, що в рази ефективніше за модель на основі CNN+Attention, яка пройшла сотню епох та тренувалась годинами безрезультатно.

Це неймовірний успіх, який дає відповідь на деякі питання. Модель вірогідно вчиться не на глобальній статистиці, а на локальних просторових патернах. Тобто модель навчилась розрізняти текст представлений у конкретному датасеті.

3.2.4 Перевірка валідності експерименту № 4

Якщо ставити задачу як розрізнення шкідливого коду від нешкідливого, то задача не має розв’язку. Це семантична задача замаскована під статистичну, яку CNN не може розв'язати принципово. Але задача і не полягає у тому, щоб розрізняти сенс коду, а в тому щоб розрізнити чи є вбудоване корисне навантаження звичайним текстом чи кодом. А тому з достатньою варіативністю навчальних даних модель могла б вивчити загальні відмінності між текстом і кодом.

Для закріплення результату прийнято рішення перевірити на практиці чи запам’ятовує модель статистичні дані. Для цього проведено експеримент, згідно з яким у 10 зображень для обох класів буде закодоване корисне навантаження, що складаються із символів та слів, які зустрічаються у звичайному тексті (для першого класу) та в коді програми (для другого

класу). Програмна реалізація алгоритму генерації корисного навантаження для цього експерименту представлено на рисунку 3.21.

```
def random_text(length=500):
    words = ['hello', 'world', 'test', 'message', 'random', 'text']
    return ' '.join(random.choices(words, k=length//5))

def random_code(length=500):
    templates = [
        "x = {}\n", "if x > {}:\n", "for i in range({}):\n",
        "def func_{}():\n", "    return {}\n"
    ]
    result = ""
    while len(result) < length:
        result += random.choice(templates).format(random.randint( a: 0,  b: 100))
    return result
```

Рисунок 3.21 – Втрати та точність при валідації

Для експерименту буде використана модель що вже була навчена при попередньому експерименті на прикладах тексту та коду. Тепер же її ефективність перевірятиметься на цих 20 тестових зображеннях. Програмна реалізація експерименту наведена на рисунку 3.22.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
data_dir = "DS"
transform = transforms.Compose([
    transforms.ToTensor(),
])
dataset = datasets.ImageFolder(
    root=data_dir,
    transform=transform
)
data = DataLoader(dataset, batch_size=1, shuffle=True)
loss_fn = CrossEntropyLoss()
model = InstanceCNNClassifier.LSBDetector(num_classes = 3)
state_dict = load_file("models/Multiclass_On_Unresized.safetensors")
model.load_state_dict(state_dict)
correct = []
losses = []
with torch.inference_mode():
    for (img,label) in data:
        logits = model(img)
        losses.append(loss_fn(logits,label).item())
        prediction = torch.round(torch.argmax(logits))
        if prediction == label:
            correct.append(1)
print("="*30)
print(f" Середня точність: {(sum(correct) / len(data))*100}%")
print(f" Середня величина втрат: {(sum(losses) / len(losses)):.2f}")
print("="*30)
```

Рисунок 3.22 – Програмна реалізація експерименту

Результати перевірки точності навченої моделі на зображеннях з вбудованими символами, характерними для тексту та коду преставлено на рисунку 3.23.

```
==============================
 Середня точність: 40.0%
 Середня величина втрат: 8.54
==============================
```

Рисунок 3.23 – Накопичення градієнта

Представлені результати можна вважати підтвердженням підозр про навчання моделі на конкретних прикладах корисного навантаження. Модель не вчиться відрізняти корисне навантаження за статистикою, вона вчиться розрізняти як повинен просторово виглядати текст і код. Але для мети дослідження цей результат є прийнятним. Зважаючи на вище зазначене, поточну архітектуру моделі прийнято вважати успішною.

3.3 Ускладнення задачі. Мультикласова класифікація.

Беззаперечний успіх моделі при бінарній класифікаціє дає підставу для продовження експерименту з моделлю на базі CNN, повернувшись до початкової задачі мультикласової класифікації. Для цього до датасету повернуто третій клас зображень без стеганографічних слідів. Тож тепер тренувальні дані знову складаються з чистих зображень, зображень з вбудованим текстом та зображення з вбудованим кодом програми.

Також, зважаючи на те, що в реальності для вбудування корисного навантаження можуть використовуватись зображення абсолютно будь-якого розміру, прийнято рішення відмовитися від зміни розміру зображень на однорідний щоб імітувати більш реалістичну задачу. Зовнішній вигляд зміненого датасету представлено на рисунку 3.24.

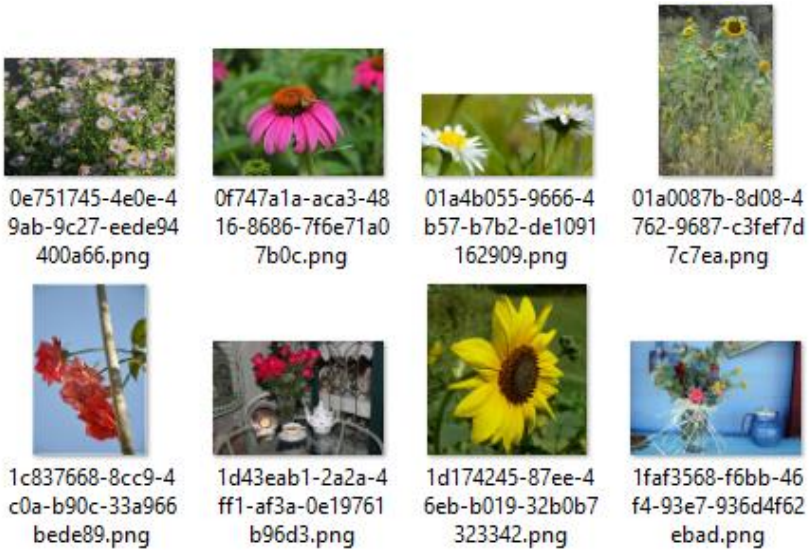

Рисунок 3.24 – Оновлений датасет

Приймаючи до уваги зміни у навчальних даних, слід передбачити реалізацію нестандартного циклу навчання. Різний розмір тензорів вхідних зображень унеможливлює розбиття навчальних даних на батчі, що спричинить низку критичних проблем. Саме тому прийнято рішення реалізувати навчання за стратегією накопичення градієнта. Схематичне представлення цієї стратегії наведено на рисунку 3.25.

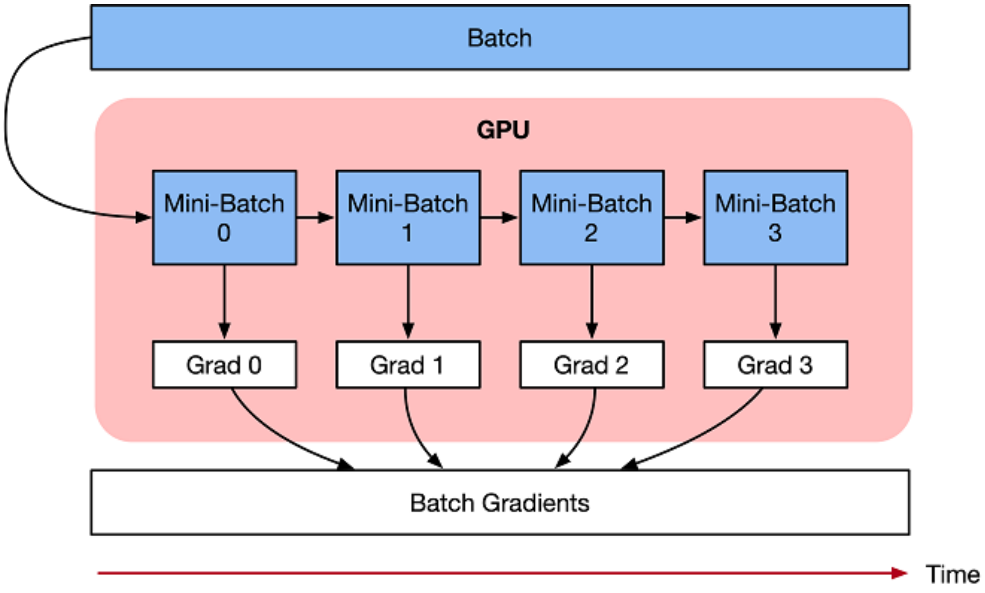

Рисунок 3.25 – Накопичення градієнта

У стандартному навчанні градієнти розраховуються для одного батчу і ваги оновлюються одразу. У данному випадку процес явно розбивається на декілька етапів у які вносяться деякі зміни.

При цій стратегії один екземпляр даних, як зазвичай, пропускається через модель і, базуючись на ньому, вираховуються втрати. Так як в поточному експерименті імітується батч розміром у 16 екземплярів, на цьому етапі критично необхідна нормалізація, тобто ділення значення втрат на кількість кроків накопичення. Це важливо для того, щоб градієнт від 16 картинок не став у 16 разів більшим за норму, що еквівалентно усередненню градієнта всередині батчу. Вирахуване значення втрат додається до градієнта, при цьому не перезаписуючи попередній. І тільки після 16 ітерацій ваги оновлюються та градієнт стирається [7].

3.3.1 Експеримент №5

Для чистоти експерименту архітектура моделі була залишена без змін. Результати навчання експериментальної моделі на базі CNN для задачі мультикласової класифікації при нестандартних даних наведено у вигляді графіків точності та втрат при тренуванні (рисунок 3.26) та при валідації (рисунок 3.27).

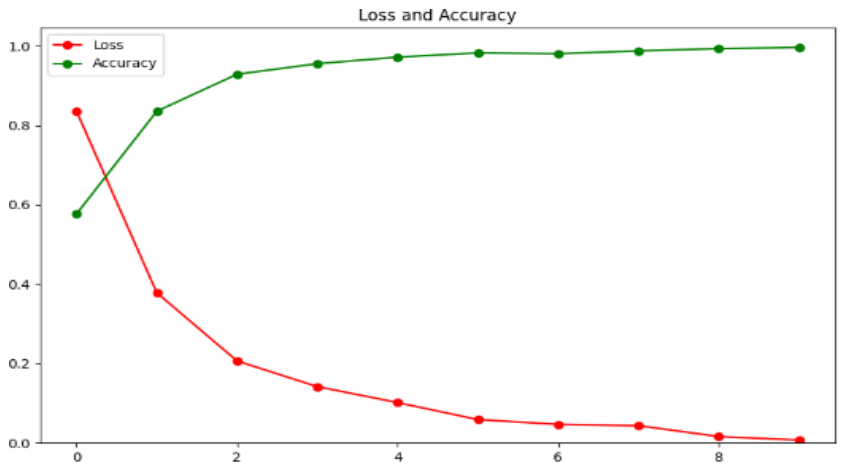

Рисунок 3.26 – Втрати та точність CNN при тренуванні

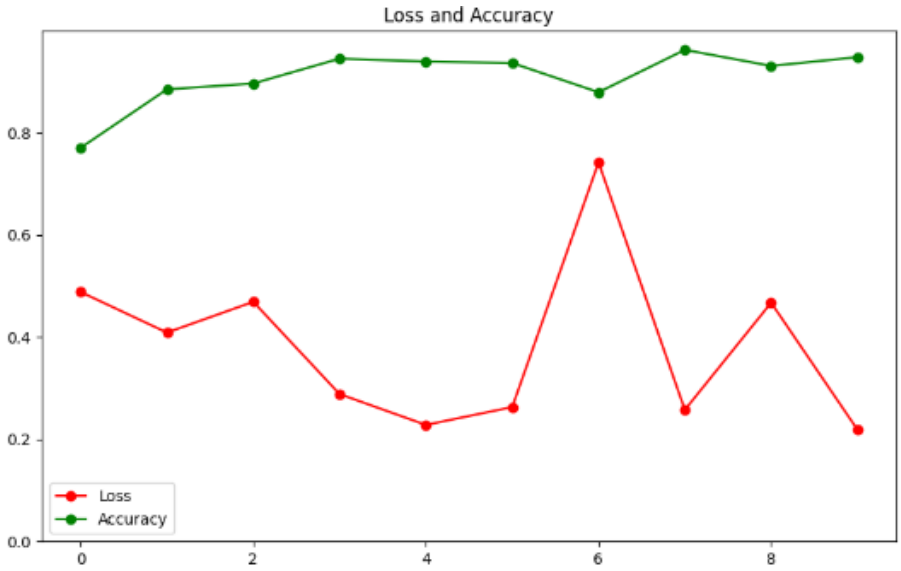

Рисунок 3.27 – Втрати та точність CNN при валідації

З результатів видно, що при навчанні втрати моделі стабільно знижуються, в той час як точність росте майже до 100%. Проте з валідацією спостерігаються проблеми. Модель вирізняє тренувальні дані, але вагається при класифікації нових даних. Часові показники для поточного експерименту наведено у таблиці 3.7.

Табл. 3.7 – Часові показники при ускладненні задачі

Показник	Значення, сек.
average time for epoch:	10.83
max time for epoch:	11.79
min time for epoch:	10.30
total time for training:	124.47

З часових показників видно, що час на навчання збільшився приблизно у 10 разів. Цього варто було очікувати, так як зображення що використовувались до цього мали дуже малий розмір (64x64) і, відповідно, містили менше даних. Зараз же датасет складається з зображень які мають розміри у діапазоні від 250 до 400 у обох вимірах.

3.3.2 Експеримент №6

Дивлячись на поточну архітектуру, можна припустити, що падіння якості генералізації пов’язане з неправильною нормалізацією. Поточні шари нормалізації розраховують статистику для всіх зображень у батчі одночасно для кожного каналу окремо. Якщо в модель подається батч із 16 зображень, функція нормалізації бере пікселі з усіх 16 картинок у межах одного каналу, знаходить їхнє спільне середнє і ділить на них. Але так як розбиття на батчі неможливе функція змушена розраховувати статистику лише по одній картинці. Це призвело до того, що на кожній ітерації уявлення моделі про дані радикально змінюється. Це вносить величезний шум, який і заважає втратам знижуватись.

В такому випадку необхідно розраховувати статистику окремо для кожного конкретного зображення та кожного каналу через нормалізацію за екземпляром. Потрібна функція що нормалізує значення лише в межах конкретного каналу для однієї конкретної картинки. Вона абсолютно не залежатиме від розміру батчу та ідеально підходитиме для поточної задачі [10]. Показники валідації після зміни функції нормалізації представлено на рисунку 3.28.

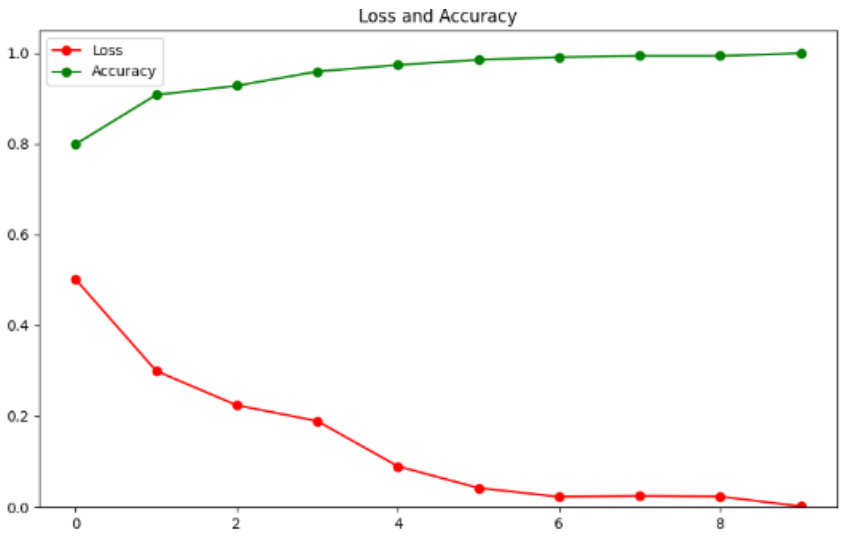

Рисунок 3.28 – Втрати та точність валідації після зміни нормалізації

Результат стало видно одразу. Тепер втрати не варіюються безконтрольно, а йдуть на спад, у той час як точність вже не застрягає на 80% а наближається до 100%, що свідчить про коректність прийнятого архітектурного рішення. Часові показники для поточної моделі з нормалізацією за екземпляром наведено у таблиці 3.8.

Таблиця 3.8 – Часові показники після зміни нормалізації

Показник	Значення, сек.
average time for epoch:	11.68
max time for epoch:	12.98
min time for epoch:	10.82
total time for training:	134.03

Хоча різниця невелика, але витрати часу на навчання збільшились приблизно на 10%. Цього варто було очікувати, бо через розрахунок статистики для кожного зображення окремо проводиться більше дрібних операцій. І хоч їх сумарна кількість саме математичних розрахунків залишається незмінною, додаткові витрати на керування ними компенсуються витраченим часом.

Зважаючи на вищезазначене, поточний експеримент прийнято вважати успішним.

3.4 Підсумки результатів експериментів

Протягом дослідження в рамках розробки методів з використанням штучного інтелекту для виявлення шкідливої стеганографії було розроблено дві архітектури на базі глибокого штучного інтелекту.

Початкова розроблена архітектура базувалась на препроцесингу зображень згортковою нейронною мережею та класифікації механізмом уваги. Ідея базувалась на припущенні, що згорткові шари зможуть вловити та підсилити стеганографічний сигнал. В свою чергу механізм уваги мав розпізнати у послідовності патерни, характерні для мови програмування.

Друга архітектура базувалась лише на звичайній згортковій нейронній мережі. Ідея полягала у відділенні самого зображення від його останніх бітів та спробі навчити згорткові ядра виділяти просторові патерни прихованого повідомлення.

Обидві архітектури було випробувано на спеціально згенерованому датасеті та протягом експериментів внесено необхідні корективи. Остаточні результати успішностей моделей у задачі виявлення стеганографічних повідомлень наведено у таблиці 3.9.

Таблиця 3.9 – Загальні результати

Експеримент (модель)	Архітектура	Задача	Час навч. (сек)	Точність %	Статус
1	CNN+ Attention	Мультикласова класифікація (3 класи)	4914.36	38	Провал
2	CNN+ Attention	Мультикласова класифікація (3 класи)	4322.59	4	Провал
3	CNN	Бінарна класифікація (чисті/код)	6.94	99	Успіх
4	CNN	Бінарна класифікація (текст/код)	9.27	99	Успіх
5	CNN	Мультикласова класифікація (3 класи), динамічний розмір зображення	124.47	91	Успіх
6	CNN	Мультикласова класифікація (3 класи), динамічний розмір зображення	134.03	99	Успіх

З представлених остаточних результатів видно, що перші моделі зі складною архітектурою проходили навчалися доволі довго, але так і не продемонстрували бажаного результату. Заради збереження витрат на обчислення експерименти з цією архітектурою були зупинені.

Натомість експерименти з другою архітектурою виявились більше ніж просто продуктивними. Моделі в них продемонстрували найвищу спроможність до навчання та здатність до генералізації, при цьому витративши на це мізерну кількість часу. Подальші експерименти з цією архітектурою показали здатність моделі до мультикласової класифікації, а також до навчання на нестандартному наборі даних. Потенційно модель може бути здатна відрізняти не тільки код та текст, а й мови програмування чи подібне.

Підсумовуючи вище зазначене, можна сказати що друга архітектура перевершує першу за усіма параметрами. Це доводить що найкраще рішення іноді є найпростішим.

ВИСНОВКИ

Під час виконання кваліфікаційного завдання було ретельно досліджено способи використання стеганографії у неправомірних цілях. Дослідження виявило проблеми стегоаналізу, які досі залишаються актуальними у сфері кібербезпеки. Серед них проблема впровадження глибокого навчання для виявлення шкідливої стеганографії.

Було проаналізовано ряд задокументованих, як старих так і нещодавніх, кібератак з використанням стеганографії на томі, чи іншому етапі. Одним з найпоширеніших та найнебезпечніших методів використання стеганографії виявлено вживлення шкідливих повідомлень у біти растрового зображення, який використовується й по сьогодні.

Аналіз існуючих методів машинного навчання показав, що наявні методи вирішення цієї проблеми мають завузьке коло застосувань, витрачають забагато обчислювальної потужності, застарілі чи не призначені для поточної задачі.

В результаті дослідження експериментальним шляхом було продемонстровано спроможність моделі глибокого штучного інтелекту до класифікації вбудованих у зображення стеганографічних повідомлень. Було визначено архітектуру та гіперпараметри оптимальні для ефективного навчання. Отриману модель можна використати як ідею для подальшого впровадження захисту від шкідливої стеганографії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Amit H. How to create a subset of dataloader in pytorch and why it is important. *Medium*. URL: https://medium.com/@heyamit10/how-to-create-a-subset-of-dataloader-in-pytorch-and-why-it-is-important-826ffd96ab9c (date of access: 14.05.2026).

2. ASCII table - Table of ASCII codes, characters and symbols. *ASCII code*. URL: https://www.ascii-code.com (date of access: 14.05.2026).

3. Ayub N., Selwal A. An improved image steganography technique using edge based data hiding in DCT domain. Journal of interdisciplinary mathematics. 2020. Vol. 23, no. 2. P. 357–366. URL: https://doi.org/10.1080/09720502.2020.1731949 (date of access: 09.05.2026).

4. Bieniasz J., Szczypiorski K. Steganography techniques for command and control (C2) channels. Botnets. 2019. P. 189–216. URL: https://doi.org/10.1201/9780429329913-5 (date of access: 09.05.2026).

5. Sensitivity of deep learning applied to spatial image steganalysis / R. Tabares-Soto et al. *PeerJ computer science*. 2021. Vol. 7. P. e616. URL: https://doi.org/10.7717/peerj-cs.616 (date of access: 14.05.2026).

6. PyTorch system requirements. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/python/pytorch-system-requirements/ (date of access: 14.05.2026).

7. Gradient Accumulation Algorithm. *MindSpore*. URL: https://www.mindspore.cn/tutorials/experts/en/r2.2/optimize/gradient_accumulation.html (date of access: 14.05.2026).

8. Houmansadr A., Piyawongwisal P., Borisov N. Stegobot: construction of an unobservable communication network leveraging social behavior. *arXiv*. URL: https://arxiv.org/abs/1107.2031 (date of access: 14.05.2026).

9. Image file type and format guide. *MDN Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Web/Media/Guides/Formats/Image_types (date of access: 14.05.2026).

10. Instance normalisation vs batch normalisation. *Stack Overflow*. URL: https://stackoverflow.com/questions/45463778/instance-normalisation-vs-batch-normalisation (date of access: 10.05.2026).

11. PyCharm: the only Python IDE you need. *JetBrains*. URL: https://www.jetbrains.com/pycharm/ (date of access: 09.05.2026).

12. Kshatriya K. New steganographic campaign distributing multiple malware variants. *Seqrite Labs*. URL: https://www.seqrite.com/blog/steganographic-campaign-distributing-malware (date of access: 14.05.2026).

13. Kumar V., Kumar D. A modified DWT-based image steganography technique. *Multimedia tools and applications*. 2017. Vol. 77, no. 11. P. 13279–13308. URL: https://doi.org/10.1007/s11042-017-4947-8 (date of access: 14.05.2026).

14. PNG (portable network graphics) specification. *W3C*. URL: https://www.w3.org/TR/REC-png-961001 (date of access: 09.05.2026).

15. Pytel R. ML engineer comparison of pytorch, tensorflow, JAX, and flax. *SoftwareMill*. URL: https://softwaremill.com/ml-engineer-comparison-of-pytorch-tensorflow-jax-and-flax/ (date of access: 09.05.2026).

16 Rahali A., Akhlouf M.A. MalBERT: using transformers for cybersecurity and malicious software detection. *arXiv*. URL: https://arxiv.org/abs/2103.03806 (date of access: 14.05.2026).

17 Rallabandi S. Activation functions: ReLU vs. Leaky ReLU. *Medium*. URL: https://medium.com/@sreeku.ralla/activation-functions-relu-vs-leaky-relu-b8272dc0b1be (date of access: 14.05.2026).

18. Steganography explained: the hidden art of secret messages. *Sangfor Technologies*. URL: https://www.sangfor.com/glossary/cybersecurity/steganography-explained (date of access: 14.05.2026).

19. Secure bit-plane based steganography for secret communication / C.-N. Bui et al. *IEICE transactions on information and systems*. 2010. E93-D, no. 1.

P. 79–86. URL: https://doi.org/10.1587/transinf.e93.d.79 (date of access: 14.05.2026).

20. Stoyanova V. T. Steganography system using LSB methods. *SSRN electronic journal*. 2018. URL: https://doi.org/10.2139/ssrn.3283729 (date of access: 14.05.2026).

21. Tam A. Creating a training loop for pytorch models. *Machine Learning Mastery*. URL: https://machinelearningmastery.com/creating-a-training-loop-for-pytorch-models/ (date of access: 14.05.2026).

22. Using steganography to hide malware - witchetty APT case study. *HawkEye*. URL: https://hawk-eye.io/2022/12/using-steganography-to-hide-malware-witchetty-apt-case-study (date of access: 14.05.2026).

23. Sophia S., Madheswaran M., Sasikumar S. Modulation recognition for software defined radio signal. *Handbook of research on mobile software engineering*. P. 398–412. URL: https://doi.org/10.4018/978-1-61520-655-1.ch023 (date of access: 14.05.2026).

24. Salvaggio C. What is inside a JPEG file. *Society for Imaging Sciences and Technology*. URL: https://www.imaging.org/IST/IST/Resources/Tutorials/Inside_JPEG.aspx (date of access: 14.05.2026).

25. What is overfitting? *Amazon Web Services, Inc.* URL: https://aws.amazon.com/what-is/overfitting/ (date of access: 14.05.2026).

26. Yatsura P., Progonov D. Influence of srm filters preprosesing on stego data localization in digital images. *Theoretical and applied cybersecurity*. 2025. Vol. 7, no. 3. URL: https://doi.org/10.20535/tacs.2664-29132025.3.346664 (date of access: 14.05.2026).