МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

Кваліфікаційна робота магістра

# ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ РОЗПІЗНАВАННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ВИКОРИСТАННЯМ НЕЙРОМЕРЕЖОВОЇ АРХІТЕКТУРИ ТРАНСФОРМЕРІВ

Здобувач освіти гр. ІК.мз-13с Денис ФРОЛОВ

Науковий керівник, кандидат технічних наук, доцент В’ячеслав МОСКАЛЕНКО

В.о. завідувача кафедри кандидат технічних наук, доцент Ігор ШЕЛЕХОВ

Суми 2023

*Сумський державний університет*
(назва вузу)

Факультет ___ *ЕлІТ* ___ Кафедра ___ *Комп'ютерних наук* ___

Спеціальність *«122 - Комп'ютерні науки»* ___

Затверджую:

в.о. зав.каф. Шелехов І.В.

"___" ___ 20___р.

# ЗАВДАННЯ
# НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

*Фролову Денису Івановичу*
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інформаційна технологія розпізнавання шкідливого програмного забезпечення з використанням нейромережової архітектури трансформерів*
затверджую наказом по інституту від "____" ___ 20 ___ р. № ___

2. Термін здачі студентом закінченого проекту (роботи) ___

3. Вхідні дані до проекту (роботи) ___

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
*1) Аналіз проблеми розпізнавання шкідливого програмного забезпечення. 2) Постановка задачі і формування завдань дослідження. 3) Огляд наукової літератури. 4) Опис інформаційної технології. 5) Програмна реалізація обраних моделей архітектур штучних нейронних мереж. 6) Аналіз результатів.*

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

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

| Розділ | Консультант | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |
| | | | |

7. Дата видачі завдання ______________________________________________

Керівник ________________________
(підпис)

Завдання прийняв до виконання ________________________
(підпис)

**КАЛЕНДАРНИЙ ПЛАН**

| № п/п | Назва етапів дипломного проекту (роботи) | Термін виконання проекту (роботи) | Примітка |
|---|---|---|---|
| *1* | *Аналіз проблеми розпізнавання шкідливого програмного забезпечення* | | |
| *2* | *Формалізована постановка задачі дослідження* | | |
| *3* | *Огляд наукової літератури* | | |
| *4* | *Опис інформаційної технології* | | |
| *5* | *Програмна реалізація обраних моделей архітектур штучних нейронних мереж* | | |
| *6* | *Аналіз результатів* | | |
| *7* | *Оформлення кваліфікаційної магістерської роботи* | | |

Студент – дипломник ________________________
(підпис)

Керівник проекту ________________________
(підпис)

# РЕФЕРАТ

**Записка:** 112 стор., 40 рис., 4 додатки, 62 джерела.

**Об'єкт дослідження** — процес розпізнавання шкідливого програмного забезпечення.

**Мета роботи** — підвищення ефективності розпізнавання шкідливого програмного забезпечення за рахунок використання нейромережової архітектури трансформерів.

**Методи дослідження** — аналіз літературних джерел, методи нейромережевого моделювання, методи подання програмного коду в вигляді 2D топологій, методи оцінювання ефективності моделей аналізу даних.

**Результати** — проведений огляд наукової літератури з конвертації бінарних файлів шкідливого програмного забезпечення в зображення у відтінках сірого, а також, з розвитку технології штучних нейронних мереж та, особливо, архітектури трансформерів; підібрано релевантний набір даних; проведено розпізнавання зображень шкідливого програмного забезпечення з використанням архітектур згорткової нейронної мережі, нейромережі Swin трансформерів (першої та другої версій) а також, гібридної нейронної мережі CoAtNet; за результатами проведеного аналізу, найкращу ефективність показала гібридна нейронна мережа CoAtNet.

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ, МАШИННЕ НАВЧАННЯ, ЗОБРАЖЕННЯ ШКІДЛИВИХ ПРОГРАМ, РОЗПІЗНАВАННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ЗГОРТКОВА НЕЙРОННА МЕРЕЖА, SWIN ТРАНСФОРМЕР, COATNET, PYTHON, KERAS, САМОУВАГА

# ЗМІСТ

ВСТУП …………………………………………………………………………… 7
РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ ……………..…. 11
1.1 Сучасний стан та тенденції розвитку систем кіберзахисту ……..…… 12
1.2 Моделі і методи розпізнавання шкідливого програмного забезпечення ………………………………………………………….……... 14
1.3 Формалізована постановка задачі …………………………,…….. 17
РОЗДІЛ 2 ПЕРЕТВОРЕННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ У ЗОБРАЖЕННЯ ……………………………………………………….……… 19
2.1 Шістнадцятковий формат файлу зловмисного програмного забезпечення ……………………………………………………..…………… 19
2.1.1 Байтовий файл ……………………………………………….…….. 19
2.1.2 Файл ASM ……………………………………………………….….. 20
2.2. Шкідливе програмне забезпечення як зображення …………………... 22
РОЗДІЛ 3 ТЕОРЕТИЧНІ ЗАСАДИ ГЛИБИННОГО НАВЧАННЯ …………..… 25
3.1 Штучний нейрон ……………………………………………………….... 26
3.2 Штучна нейронна мережа ……………………………………………… 28
3.2.1 Персептрон …………………………………………………….... 31
3.2.2 Алгоритми оптимізації ………………………………………… 34
3.2.3 Перенавчання ……………………………………………………… 37
3.3 Глибинне навчання ……………………………………………………… 38
3.4 Метрики оцінювання …………………………………………………… 42
РОЗДІЛ 4 АРХІТЕКТУРИ НЕЙРОННИХ МЕРЕЖ ……………………...…… 46
4.1 Згорткова нейронна мережа ..…………………………………………… 46
4.2 Трансформери …………………………………………………………… 51
4.2.1 Архітектура трансформеру зору ………………………….……… 55
4.2.2 Архітектура Swin трансформеру (V1) ……………………….... 56
4.2.3 Архітектура Swin трансформеру (V2) …………………………… 60
4.3 Гібридна нейронна мережа CoAtNet ……………………………….. 64

РОЗДІЛ 5 РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ РОЗПІЗНАВАННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ …………………..………. 66
5.1 Набір даних Malimg ……………………………………………………….. 66
5.2 Опис програмного забезпечення ……………………………………..… 69
5.3 Результати машинного навчання ……………………………………..… 71
5.3.1 Результати навчання згорткової нейронної мережі ……………… 72
5.3.2 Результати навчання Swin трансформер (V1) …………………….. 75
5.3.3 Результати навчання Swin трансформер (V2) ……………….…... 78
5.3.4 Результати навчання CoAtNet …………………………………….. 82
5.3.5 Порівняння результатів ……………………………………..…….. 85
ВИСНОВКИ ………………………………………………………………………... 88
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ……………………………………….. 90
ДОДАТОК А ……………………………………………………………………..…... 98
ДОДАТОК Б ………………………………………………………………………… 101
ДОДАТОК В ……………………………………………………….……………….... 108
ДОДАТОК Г ……………………………..……………………….…………………… 110

# ВСТУП

Наш світ змінюється на очах. Перебуваючи в Україні, ми бачимо це як ніхто інший.

Майже рік тому в Україні почалася війна. Для того, щоб завдати найбільшої шкоди нашій країні, зараз використовується різна зброя. Особливу роль в цьому відіграють кібератаки. При цьому в Україні вже кілька років фактично триває повномасштабна кібервійна. В зв’язку з цим, тематика дослідження з розпізнавання шкідливого програмного забезпечення є особливо актуальною в цей час.

Крім того, світова економіка перебуває в процесі цифровізації та еволюціонує від індустріального до інформаційного суспільства. Особливе місце в цьому процесі посідають технології машинного навчання [1]. Тому дані стали більш цінним ресурсом, ніж, наприклад, фінанси чи енергія. Як результат, вони також стали критичною мішенню для кіберзлочинців.

Однак, не всі кібер-цілі мають однакову цінність. Наприклад, медичні дані в 10-20 разів цінніші, ніж дані кредитної картки. Щоб захистити себе на ринку, де кількість кіберзлочинців стрімко зростає разом із кількістю атак, організації вкладають значні кошти в кібербезпеку. Очікується, що глобальний ринок кібербезпеки може перевищити 320 мільярдів доларів до 2027 року [2].

Аналіз стану глобальної інформаційної безпеки підприємств показує, що кількість кібератак, спрямованих на витік даних, продовжує зростати. Кібератаки, як відносно прості події, можуть бути об’єднані в сценарії, які, у свою чергу (залежно від тривалості, інтенсивності та мети), можуть формувати набагато складніші дії. Така глобальна активність може перетворитись у кібервійну.

Мета кібервійни полягає в тому, щоб руйнувати критично важливу інфраструктуру держави та організацій шляхом завдання шкоди комп’ютерам та інформаційним мережам. Внаслідок цього відбувається деградація енергетичної, фінансової, соціально-культурної та політичної сфер держави.

Швидке та значне зростання кількості кібератак призвело до значного збільшення кількості звернень із критичними станами та визнанням економічних збитків у всьому світі. Цей важливий фактор потенційно може призвести до збільшення критичної людської помилки, що згодом знизить ефективність захисту. Нещодавні аналітичні вдосконалення можуть допомогти знайти практичні рішення щодо термінової потреби в розробці автоматизованих платформ виявлення зловмисного програмного забезпечення, які можуть надавати прогностичну інформацію про еволюцію складних кібератак.

Точне розпізнавання зловмисного програмного забезпечення як основне завдання в процесі здійснення моніторингу критичної інфраструктури під час атаки (а також, подальшого прогнозування розвитку сценарію атаки) може відігравати критичну роль у захисті та збереженні ресурсів.

Системи виявлення шкідливих програм (malware database scanner / MDS) — це перша лінія захисту від зловмисних атак, тому для систем виявлення шкідливих програм важливо точно й ефективно виявляти шкідливі програми. Поточні MDS зазвичай використовують традиційні алгоритми машинного навчання, які потребують вибору та вилучення функцій, що займає багато часу та може викликати помилки.

Разом з цим, шкідливе програмне забезпечення є постійною загрозою, яка значно зросла за останні кілька років. Однак через більш просунуті та спеціалізовані атаки поточні методи виявлення надто повільні або неефективні для аналізу такого програмного забезпечення в реальному часі.

Крім того, література вказує на подальші можливості розробки високопродуктивних алгоритмів для точного прогнозування серйозності кібератак і подальшої діагностики зараженого обладнання на основі виявлених випадків. Успішні алгоритми використовували комбіновані підходи шляхом злиття даних спостережень та зображень шкідливих файлів. Ці дослідження спочатку об'єднували ознаки, отримані з зображень, з характеристиками даних і надавали результати з глибинних класифікаторів. Так, функції, отримані за результатами аналізу зображень, можна об'єднати з іншими доступними

функціями/даними для створення більш надійного та узгодженого набору даних, який може надати детальну інформацію для глибинної нейронної мережі з метою подальшого прогнозування серйозності атаки.

З часом, класифікація зловмисного програмного забезпечення продовжує ускладнюватись. Однією з ключових причин цього є введення мутацій для уникнення виявлення шкідливого програмного забезпечення. Це означає, що шкідливі файли з того самого сімейства зловмисного програмного забезпечення з однаковою поведінкою постійно змінюються або маскуються за допомогою різних технологій, щоб виглядати різними. Характеристики, отримані з необроблених бінарних файлів або розібраного коду, використовуються в існуючих алгоритмах категоризації шкідливих програм на основі машинного навчання. Різноманітність таких властивостей ускладнює розробку загальних методів класифікації зловмисного програмного забезпечення, які б добре працювали у різних робочих сценаріях.

Для ефективної оцінки та класифікації таких величезних обсягів даних з шкідливого програмного забезпечення необхідно розділити їх на групи та ідентифікувати відповідні родини на основі їх поведінки. Для цього бінарні файли зловмисного програмного забезпечення перетворюються у зображення в градаціях сірого. Це здійснюється завдяки можливості зафіксувати незначні зміни, зберігаючи глобальну структуру, що допомагає виявити варіації. В даній роботі запропоноване рішення з використанням машинного (глибинного) навчання для ефективної класифікації зловмисного програмного забезпечення на сімейства на основі колекції дискримінаційних шаблонів, отриманих із візуалізації бінарних файлів у вигляді зображень. Завдяки використанню інформаційних технологій нейронних мереж, зазначений підхід підвищує ефективність MDS створює більш широкий горизонт можливостей для таких систем.

Значний розвиток технологій машинного навчання (ML) у різних областях [3-6]  виглядає як інноваційний підхід до виявлення більш просунутих загроз з використанням технологій глибинного навчання. У цій роботі пропонується

оцінити швидший і більш ефективний підхід до розпізнавання шкідливих програм із використанням моделей машинного навчання (а саме, архітектур згорткової нейронної мережі, Swin трансформерів та гібридної мережі CoAtNet).

Для оцінки підходу в роботі використовувався набір даних Malimg, який був представлений у знаковій роботі [7] Nataraj з колективом співавторів о 2011 році, та, де-факто, став бенчмаркінг стандартом для дослідження ефективності використання різних методів і моделей машинного навчання.

**Мета** – підвищення ефективності розпізнавання шкідливого програмного забезпечення за рахунок використання нейромережової архітектури трансформерів.

**Об'єкт** – процес розпізнавання шкідливого програмного забезпечення.

**Предмет** – моделі і методи розпізнавання шкідливого програмного забезпечення.

**Гіпотеза:** модель машинного навчання, побудована на нейромережевій архітектурі трансформерів, буде розпізнавати образи шкідливого програмного забезпечення з кращою акуратністю (точністю) у порівнянні з моделлю, побудованою на архітектурі згорткової нейронної мережі.

**Обґрунтування новизни**: недослідженою є ефективність використання нейромережової архітектури трансформерів для розпізнавання образів шкідливого програмного забезпечення, що має дослідницький потенціал і надає роботі наукову новизну.

# РОЗДІЛ 1
# АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ

Проблема ідентифікації та розпізнавання шкідливих програм є дуже складним завданням, яке не має ідеального рішення. Індустрія зловмисного програмного забезпечення перетворилася на великий та добре організований ринок. Це призвело до масштабних інвестицій у технології та ресурси, створені для уникнення традиційного захисту, що представляє одну з найбільших проблем для спільноти безпеки. Наприклад, розвиток інформаційно-комунікаційних технологій та швидкий ріст кількості пристроїв Інтернету речей (IoT) став ще одним фактором, який призвів до широкого впровадження великої кількості різноманітних систем. Компанії змушені намагатися першими, ніж конкуренти запускати нові продукти.

Це обумовлює наявність на ринку великої кількості продуктів з недоліками програмного забезпечення, що призводить до збільшення вразливостей, а також, до збільшення векторів атак. Більшість із цих пристроїв мають або дозволяють мати доступ до значної кількості конфіденційних даних. Як результат, вони стають привабливою мішенню для зловмисників.

Разом з цим, відповідно до звіту Європейського агентства мережевої та інформаційної безпеки (ENISA) «Threat Landscape 2022» [5] минулий рік характеризувався новими викликами, наприклад:

– нова хвиля хактивізму спостерігається особливо з початком російсько-української війни;

– бізнес-модель "хакер як послуга" набирає обертів, зростаючи з 2021 року.

Така протизаконна діяльність може потенційно коштувати світовій спільноті більше 300 мільярдів доларів до 2027 року [2].

В поточному контексті війни кількість кібератак, як складова військових активностей проти України, спрямована насамперед на критичну інфраструктуру нашої держави та зростає щодня.

### 1.1 Сучасний стан та тенденції розвитку систем кіберзахисту

Шкідливе (зловмисне) програмне забезпечення — це загальний термін для будь-якого програмного забезпечення, яке перешкоджає роботі комп'ютера, збирає конфіденційну інформацію або отримує доступ до приватних комп'ютерних систем [9]. Він визначає різні типи шкідливого програмного забезпечення, наприклад рекламне, шпигунське, віруси, хробаки, трояни та програми-вимагачі, призначені для заподіяння шкоди або використання запрограмованих пристроїв, служб або мереж. З початку 1970-х років, коли з'явився вірус Creeper, існує загроза для окремих осіб і організацій у всьому світі.

Наразі виявлення шкідливого програмного забезпечення здійснюється переважно за допомогою методів на основі сигнатури та евристики, а також за допомогою аналізу поведінки. Однак, ці методи не встигають за еволюцією зловмисного програмного забезпечення.

На відміну від аналізу на основі сигнатур, коли використовуються бази даних сигнатур, які однозначно ідентифікують відоме зловмисне програмне забезпечення, методи на основі евристики покладаються на правила та відомі шаблони для виявлення відомих і нових форм зловмисного програмного забезпечення.

Поведінковий аналіз допомагає визначити та зрозуміти можливості зловмисного програмного забезпечення, відстежуючи поведінку та шукаючи підозрілу активність у системі. Коли алгоритм виявляє зловмисне програмне забезпечення, він поміщає підозрілий файл у карантин або згодом видаляє його.

Тоді як з боку зловмисного програмного забезпечення віруси поділяють спільну поведінку між своїми родинами (що дозволяє створити для них єдиний загальний підпис), кіберзлочинці намагаються бути на крок попереду антивірусного (anti-virus / AV) програмного забезпечення, розробляючи поліморфне та метаморфічне шкідливе програмне забезпечення, яке не збігається з відомими сигнатурами.

Тому постачальники AV, як правило, використовують гібридний аналіз, поєднуючи методи на основі сигнатур і евристики для боротьби з невідомим

шкідливим програмним забезпеченням. Разом з цим, вони також виконують поведінковий аналіз за допомогою статичного аналізу (досліджуючи код) або динамічного аналізу (через виконання в безпечному середовищі).

Метод статичного аналізу базується на аналізі коду або шаблонів для виявлення сигнатур, послідовностей байтів або частотного розподілу коду операції без виконання програми. Однак. отримання вихідного коду залежить від зворотного проектування. Динамічний аналіз базується на аналізі поведінки під час виконання шкідливої програми в контрольованому та безпечному середовищі (наприклад, на віртуальній машині, в емуляторі, пісочниці тощо). Порівняно зі статичним аналізом, він більш точний і не потребує аналізу коду.

Однак, коли динамічний аналіз виконується у великому масштабі, це вимагає більше часу та ресурсів. Крім того, контрольоване середовище відрізняється від реального середовища, де поведінка зловмисного програмного забезпечення може бути викликана лише за певних умов, таких як, конкретна система, команда або послідовність дій. Крім того, зловмисники впроваджують засоби захисту шкідливого програмного забезпечення для визначення середовища, в якому воно працює, шляхом демонстрації іншої поведінки та ускладнюючи його виявлення у віртуальному середовищі.

Першою лінією захисту від кібератак є система виявлення шкідливих програм (malware detection system / MDS). Вони розповсюджуються постачальниками антивірусних засобів, щоб перевірити, чи є файл шкідливим чи безпечним. Традиційні системи виявлення зловмисного програмного забезпечення базуються на алгоритмах неглибинного машинного навчання (термін «неглибинне машинне навчання» або «традиційне машинне навчання» відноситься до алгоритмів машинного навчання, які не є глибинним навчанням, наприклад, дерева рішень, опорні векторні машини та класифікатор Байєса). Продуктивність традиційних алгоритмів машинного навчання значною мірою залежить від якості ідентифікованих функцій. Однак, процес вибору і вилучення функцій забирає надзвичайно багато часу, схильний до помилок і вимагає високого рівня знань у цій галузі.

Наступне покоління алгоритмів машинного навчання (глибинне навчання) стало дуже популярним останнім часом завдяки його здатності автоматично отримувати складні характеристики високого рівня, які забезпечують високу точність

Сучасні системи виявлення зловмисного програмного забезпечення, засновані на глибинному навчанні, здебільшого базуються на повторюваних нейронних мережах (RNN) із викликами API та машинними інструкціями в якості вхідних даних і мають високу точність.

Машинне навчання є однією з найбільш швидкозростаючих галузей комп'ютерних наук, що використовує статистику та штучний інтелект (artificial intelligence / AI) для надання системам здатності автоматично навчатися та вдосконалюватися.

Наразі AI широко використовується в наукових додатках, які вимагають аналізу великих обсягів даних. В повсякденні людей оточують різні реалізації технологій машинного навчання, наприклад, додатки персональної допомоги з розпізнаванням голосу або ідентифікацією обличчя в смартфонах, пошукові системи (які навчаються надавати найкращі результати), антиспам рішення для фільтрації повідомлень електронної пошти або операції з кредитними картками (які забезпечені ПЗ, що спрямоване на виявлення шахрайства).

В останні роки, завдяки здешевленню обчислювальних потужностей, дослідники можуть вивчати ще більш складні моделі і застосовувати їх до більших наборів даних.

## 1.2 Моделі і методи розпізнавання шкідливого програмного забезпечення

Як було зазначено раніше, системи виявлення зловмисного програмного забезпечення здійснили низку трансформацій від раннього виявлення на основі сигнатур до сучасного виявлення на основі хмари. Далі кожен з них буде розглянуто в деталях.

Перші постачальники засобів захисту від шкідливих програм зазвичай використовували сигнатури для ідентифікації шкідливих програм. Сигнатура — це унікальна послідовність байтів, представлена відомими зразками зловмисного програмного забезпечення та витягнута експертами. Таким чином, характерні ознаки атаки або вірусу, що використовуються для їх виявлення [10]. Виявлення робиться шляхом простої перевірки, чи файл містить відомі сигнатури зловмисного програмного забезпечення. Процес вилучення сигнатури займає дуже багато часу та загрожує помилками. Через такі обмеження фокус ідентифікації зловмисного програмного забезпечення було зміщено на евристику.

Виявлення зловмисного програмного забезпечення на основі евристики є незначним покращенням виявлення на основі сигнатур. Ідеальний набір евристик складається з правил і шаблонів, які достатньо описують функції, що належать до всіх варіантів того самого типу зловмисного програмного забезпечення, але не хибно співставляються на доброякісному програмному забезпеченні. Вхідний файл зіставляється з набором шкідливих евристик, щоб визначити його зловмисність. Цей метод може охопити низку зразків зловмисного програмного забезпечення одним набором.

Евристичний метод значно зменшує робоче навантаження інженерів. Однак евристика не є безкоштовною і вимагає великої кількості ручної праці для вилучення, що стало неможливим через зростання наборів інструментів для створення шкідливих програм.

Набори інструментів для створення зловмисного програмного забезпечення розроблені таким чином, що навіть зловмисник без знань програмування може розробити власне шкідливе програмне забезпечення (наприклад, з використанням ChatGPT [11]. Це спричиняє швидке зростання кількості нових зразків шкідливих програм, що робить ручне вилучення евристик неможливим. Щоб системи виявлення зловмисного програмного забезпечення залишалися ефективними, потрібні більш масштабована архітектура та інтелектуальний алгоритм виявлення. Таким чином народилася хмарна система виявлення зловмисного програмного забезпечення.

Класифікація на стороні хмари/сервера, як правило, вимагає виділення функцій і класифікації. Екстрактор ознак витягує кілька ознак із файлу, а витягнуті ознаки є вхідними даними класифікатора. При цьому потрібне детальне дослідження етапу виділення ознак і класифікації.

Ознаки – це характеристики файлу, наприклад, необхідні дозволи або серійний номер. Хороший вибір ознак має демонструвати різницю між шкідливими та доброякісними зразками. Вибір надмірних ознак часто спричиняє переобладнання та вимагає більшої обчислювальної потужності, тому набір ознак має бути коротким, але описовим представленням файлу. Залежно від того, які ознаки вибираються та вилучаються, процес вилучення ознак можна класифікувати на дві основні категорії: статичний і динамічний аналіз.

І статичний, і динамічний аналіз мають свої переваги та недоліки. Статичний аналіз може виконувати всебічний аналіз зловмисного програмного забезпечення, досліджуючи всі шляхи коду, але фокус статичного аналізу на вихідному коді робить його вразливим до заплутування коду та шифрування. Динамічний аналіз стійкий до обфускації, зосереджуючись на поведінці файлу, але не в змозі перевірити крайові випадки, приховані в програмному забезпеченні. Через обмеження статичного та динамічного аналізу системи виявлення зловмисного програмного забезпечення на практиці часто поєднують обидва для виконання гібридного аналізу, який надає більш повний опис ознак.

Класифікація за допомогою машинного навчання відбувається після вилучення ознак із файлу зловмисного програмного забезпечення та безпечних файлів, коли алгоритми вивчають кожен зразок, щоб класифікувати невидимі файли. Процес вилучення ознак перетворює файл на точку в n-вимірному просторі ознак, де n – це кількість вилучених ознак, а процес класифікації має на меті сегментувати простір ознак на кілька розділів на основі навчальних даних, де кожен розділ містить максимально можливу кількість балів, що належать до однієї категорії. Популярні алгоритми класифікації - це найближчий сусід, наивний класифікатор Байєса та опорна векторна машину.

Звичайні хмарні системи виявлення шкідливих програм базуються на традиційних алгоритмах машинного навчання. Як зазначалось, продуктивність таких алгоритмів значною мірою залежить від вибраних і виділених ознак, але це вимагає значного обсягу знань предметної області та займає багато часу. Клас алгоритмів машинного навчання які не потребують розробленого вручну екстрактора ознак, забезпечуючи високу точність і швидкість класифікації, є глибинним навчанням.

Глибинне навчання зробило серйозний прорив у багатьох областях, таких як розпізнавання зображень, розпізнавання мови та обробка природної мови. В цій роботі будуть розглянуті приклади архітектур нейронних мереж, які застосовуються для виявлення зловмисного програмного забезпечення.

## 1.3 Формалізована постановка задачі

Ідентифікація та розпізнавання зловмисного програмного забезпечення є надзвичайно складною темою, де немає універсального рішення. У результаті постачальники AV зосереджуються на гібридних підходах, які поєднують звичайні методи на основі сигнатур, евристики та машинного навчання з людським аналізом.

У даній роботі буде висвітлено підхід до розпізнавання зловмисного програмного забезпечення шляхом перетворення його бінарних файлів у зображення у відтінках сірого та подальшого використання штучних нейронних мереж, побудованих на основі архітектури трансформерів.

Головна мета роботи полягає у визначенні найбільш ефективного рішення за результатами проведеного аналізу використання інформаційної технології розпізнавання шкідливого програмного забезпечення з нейромережевою архітектурою трансформерів в порівнянні зі згортковою нейронною мережею.

Для досягнення поставленої мети сформульовані наступні задачі:

– проаналізувати проблему розпізнавання шкідливого програмного забезпечення;

– зробити огляд наукової літератури з конвертації бінарних файлів шкідливого програмного забезпечення в зображення;

– здійснити огляд наукової літератури з розвитку технологій нейронних мереж (згорткових, трансформерів зору, гібридних);

– підібрати релевантний набір даних;

– впровадити інформаційну технологію розпізнавання шкідливого програмного забезпечення з використанням архітектур штучних нейронних мереж;

– дослідити та провести порівняльний аналіз отриманих результатів, зробити висновки.

Виконання поставлених задач буде здійснюватися через наступні основні етапи робіт:

– аналіз проблеми та огляд наукової літератури;

– вибір релевантного набору даних;

– навчання моделей глибинного навчання;

– обчислення метрик та побудова графіків;

– прогнозування;

– аналіз отриманих результатів та висновки.

За результатами проведеної роботи буде проаналізовано сучасний стан і тенденції розвитку методів виявлення шкідливого програмного забезпечення, розвиток технології глибинного навчання, розглянуто архітектури згорткової нейронної мережі, архітектури Swin трансформерів (першої та другої версій), а також, гібридної нейромережі CoAtNet.

# РОЗДІЛ 2
# ПЕРЕТВОРЕННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ У ЗОБРАЖЕННЯ

Останнім часом методи машинного навчання (ML) використовуються як підхід до виявлення та аналізу шкідливих програм. Зростання кількості випадків атак зловмисного програмного забезпечення (написаного як досвідченими злочинцями так і початківцями з допомогою, наприклад, ChatGPT ), зниження вартості процесорної потужності та прогрес, досягнутий у цій галузі, сприяли появі нових досліджень та пропозицій щодо покращення аналізу шкідливого програмного забезпечення.

## 2.1 Шістнадцятковий формат файлу зловмисного програмного забезпечення

Кожен зразок зловмисного програмного забезпечення має файл, що містить шістнадцяткове значення двійкового вмісту файлу (формат .HEX), а також файл метаданих, згенерований інструментом дизассемблера IDA з витягнутими даними з двійкового коду, такими як інструкції зі складання, виклики функцій, аргументи, використані змінні та регістри тощо.

### 2.1.1 Байтовий файл

Байтовий файл містить двійковий вміст зловмисного програмного забезпечення в необробленому шістнадцятковому вигляді (рис. 2.1.1). На наступному зображенні показано знімок байтового файлу на прикладі зловмисного програмного забезпечення Rammit.

```
0cH8YeO15ZywEhPrJvmj.bytes
70881000 55 89 E5 83 EC 18 8B 45 08 C7 44 24 08 04 A0 88
70881010 70 C7 44 24 04 00 A0 88 70 89 04 24 E8 8F 61 00
70881020 00 C9 C3 8D B6 00 00 00 00 8D BC 27 00 00 00 00
70881030 55 89 E5 83 EC 18 8B 45 08 C7 44 24 08 04 A0 88
70881040 70 C7 44 24 04 00 A0 88 70 89 04 24 E8 5F 61 00
70881050 00 C9 83 F8 01 19 C0 C3 90 8D B4 26 00 00 00 00
70881060 55 89 E5 53 83 EC 14 8B 15 00 A0 88 70 85 D2 74
70881070 34 8B 1D 04 A0 88 70 83 EB 04 39 DA 77 15 8B 03
70881080 85 C0 74 F3 FF D0 83 EB 04 8B 15 00 A0 88 70 39
70881090 DA 76 EB 89 14 24 E8 1D 61 00 00 C7 05 00 A0 88
708810A0 70 00 00 00 00 C7 04 24 00 00 00 00 E8 0F 61 00
708810B0 00 83 C4 14 5B 5D C3 89 F6 8D BC 27 00 00 00 00
708810C0 83 EC 04 60 BE 10 00 00 00 8D 54 76 10 52 B8 00
708810D0 04 00 00 6B C0 04 50 BE 01 00 00 00 C1 E6 0B 56
```

Рис. 2.1.1 – Зразок коду байтового файлу

Шістнадцятковий запис (hexadecimal record) складається з наступних 6 полів [12]:

– початковий код (start code), один символ, двокрапка ASCII ':', усі символи, що передують цьому символу у записі, ігноруються;

– кількість байтів (byte count) - дві шістнадцяткові цифри, які вказують кількість байтів у полі даних;

– адреса (address) - чотири шістнадцяткові цифри для представлення 16-бітового початкового зміщення адреси пам’яті;

– тип запису (record type) - дві шістнадцяткові цифри для визначення значення поля даних, подання від 00 до 05 (дані, кінець файлу або розширений сегмент, початковий сегмент, розширена лінійна адреса, початкова лінійна адреса);

– дані (data) - послідовність даних із n байтів, представлена 2n шістнадцятковими цифрами;

– контрольна сума (checksum) - дві шістнадцяткові цифри, обчислене значення, яке використовується для забезпечення відсутності помилок у записі.

### 2.1.2 Файл ASM

Файл ASM — це маніфест метаданих, який є журналом, що містить інформацію метаданих, таку як виклики функцій, розподіл пам’яті та маніпулювання змінними (рис. 2.1.2.1). На наступному зображенні показано вміст знімка файлу ASM байтового файлу Rammit.

```
OcH8YeO15ZywEhPrJvmj.asm
.text:70881060                  ; =============== S U B R O U T I N E =======================================
.text:70881060
.text:70881060                  ; Attributes: bp-based frame
.text:70881060
.text:70881060                  sub_70881060    proc near           ; CODE XREF: .text:70881155<0x19>p
.text:70881060
.text:70881060                  Memory          = dword ptr -18h
.text:70881060
.text:70881060 55                       push    ebp
.text:70881061 89 E5                        mov     ebp, esp
.text:70881063 53                       push    ebx
.text:70881064 83 EC 14                     sub     esp, 14h
.text:70881067 8B 15 00 A0 88 70                mov     edx, ds:Memory
.text:7088106D 85 D2                        test    edx, edx
.text:7088106F 74 34                        jz      short loc_708810A5
.text:70881071 8B 1D 04 A0 88 70                mov     ebx, ds:dword_7088A004
.text:70881077
.text:70881077                  loc_70881077:                       ; CODE XREF: sub_70881060+22<0x19>j
.text:70881077 83 EB 04                     sub     ebx, 4
.text:7088107A 39 DA                        cmp     edx, ebx
.text:7088107C 77 15                        ja      short loc_70881093
.text:7088107E
.text:7088107E                  loc_7088107E:                       ; CODE XREF: sub_70881060+31<0x19>j
.text:7088107E 8B 03                        mov     eax, [ebx]
```

Рисунок 2.1.2.1 – Зразок ASM файлу Rammit

Програма зборки складається з трьох основних розділів:

– розділ даних використовується для оголошення ініціалізованих даних або констант, які не змінюються під час виконання, таких як константні значення, імена файлів або розмір буфера;

– розділ bss використовується для оголошення змінних, наприклад неініціалізованих даних;

– у текстовій частині розміщено фактичний код.

Крім попередніх розділів, можуть бути додаткові розділи, такі як:

– rsrc - містить усі ресурси для програми;

– rdata - використовується для зберігання даних, які не належать до розділу data або bss. Це також дані, які доступні лише для читання та містять літеральні рядки, константи та інформацію про каталог налагодження;

– idata - містить інформацію про імпорт, наприклад, DLL програми, включно з каталогом імпорту та таблицею адрес імпорту;

– edata - містить інформацію про імена та адреси експортованих функцій, містить каталог експорту, який надає адресу та зміщення функцій програмам, які імпортують DLL;

– reloc - містить таблицю базових переміщень. Базове переміщення — це зміна інструкції або ініціалізованого значення змінної, яка потрібна, якщо завантажувач не може завантажити програму.

Інші частини також можуть з’явитися в результаті використання поліморфних або метаморфічних методів, щоб приховати фактичний код. У

деяких трансформаціях зловмисного програмного забезпечення можна побачити різні бінарні фрагменти, а секцію складання зловмисного програмного забезпечення можна ідентифікувати за різними текстурами на зображеннях. Далі (рис. 2.1.2.2) показано зразкове перетворення файлу на прикладі Rammit.

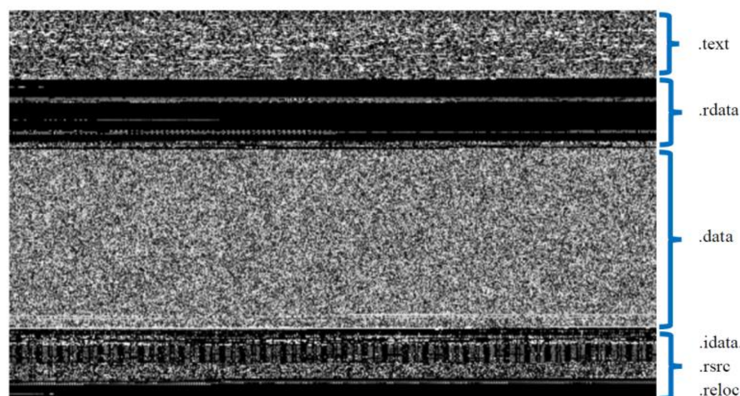



Рисунок 2.1.2.2 – Зображення бінарних фрагментів зразка Rammit

## 2.2 Шкідливе програмне забезпечення як зображення

На початкових етапах застосування машинного навчання до проблеми класифікації зловмисного програмного забезпечення до наборів даних застосовувалися алгоритми кластеризації. Ці підходи не дають змоги, коли в наборі даних є широкий спектр класів шкідливих програм. В опитуванні [13] дослідники обговорили різні алгоритми машинного навчання, які можна застосувати до набору даних файлів Windows PE. Вони витягли n-граму байтів, n-граму коду операції, виклики API та PE32 із виконуваного файлу Windows і застосували алгоритми на основі статистики (наївний Байєс), алгоритми SVM і K-nn, які згадувались в першому розділі роботи. Незважаючи на те, що вдалося отримати всі статистичні характеристики з виконуваного файлу, жоден із алгоритмів не зміг показати точність більше 90%.

У 2011 році в статті «Зображення шкідливого програмного забезпечення: візуалізація та автоматична класифікація» Nataraj з колективом співавторів [7], було запропоновано характеризувати та оцінювати шкідливе програмне забезпечення на основі його візуалізації у вигляді зображень відтінках сірого. Для аналізу в зазначеній статті використовувався набір даних Malimg.

Завдяки перетворенню двійкового коду шкідливе програмне забезпечення інтерпретується як 8-бітний масив. Біт є найбільш фундаментальною одиницею інформації в комп'ютерах і цифрових комунікаціях, оскільки він являє собою двійкове ціле число.

Група з 8 біт, або 1 байт, містить 28 різних значень, де діапазон цілих значень (які можуть бути записані у 8 бітах) змінюється залежно від обраного формату цілих чисел. Двома найпопулярнішими представленнями є діапазони від -128 (-1 × 27) до 127 (27 - 1) для представлення у вигляді доповнення двох, і від 0 до 255 (28 - 1) для представлення у вигляді беззнакового типу, що є тим самим діапазоном значень, в якому представлений піксель у відтінках сірого (рис. 2.2.1).

Переформувавши 8-бітний масив у матрицю та розглянувши його як зображення у градаціях сірого, вдалося виявити важливі візуальні кореляції в текстурі зображення шкідливих програм, що належать до того самого сімейства. Це може бути наслідком широко поширеного методу створення нових варіантів шкідливого ПЗ через повторне використання коду.

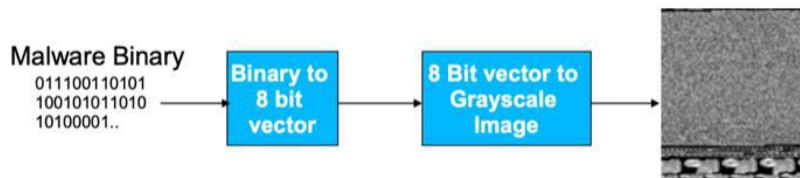



Рисунок 2.2.1 – Візуалізація зловмисного програмного забезпечення як зображення [7]

Для обчислення особливостей текстури в зображеннях зловмисного програмного забезпечення використовувався GIST [14], який використовує вейвлет-розкладання для вилучення особливостей із глобальної структури зображення. Ці елементи використовуються для порівняння з раніше ідентифікованими шкідливими шаблонами. Нарешті, алгоритм k-найближчих сусідів (kNN) з евклідовою відстанню виконав класифікацію шкідливих програм, досягнувши точності розпізнавання (accuracy) класифікації 97,18 відсотка для набору даних, що складається з 25 сімейств шкідливих програм.

Хоча зображення, відтворене на основі функцій глобальної структури, вразливе до структурних змін, кіберзлочинці, які знають про цю техніку, можуть

уникнути виявлення, перемістивши розділи коду або додавши фіктивні дані (наприклад, через обфускацію).

В зв’язку з цим було запропоновано інший підхід до протидії контрзаходам, які використовують кіберзлочинці, через використання згорткових нейронних мереж (CNN) для вилучення локальних і інваріантних характеристик із зображення, а також, знаходження шаблонів незалежно від їхнього положення [15]. Таким чином, це дозволяє нейронній мережі виявляти шаблони відомого шкідливого програмного забезпечення, присутнього на зображенні. Далі (рис. 2.2.2) показана структура згорткової нейронної мережі Гіберта (Gibert’s CNN), яка використовується для класифікації зловмисного програмного забезпечення, представленого у вигляді зображень у відтінках сірого.

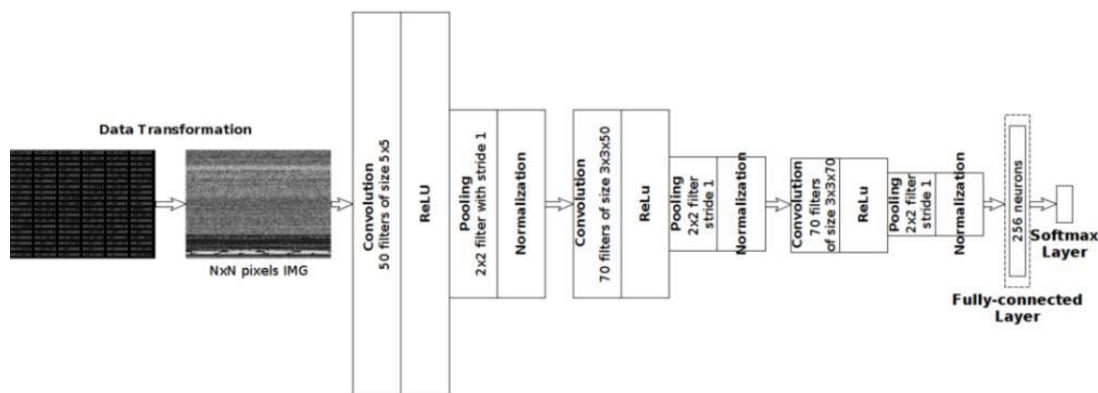



Рисунок 2.2.2 – Згорткова нейронна мережа Гіберта [15]

Так, за допомогою аналізу зображень у градаціях сірого, отриманих у результаті перетворення бінарного коду відомих зразків зловмисного програмного забезпечення, було зроблено висновок, що зображення з одного сімейства зловмисного програмного забезпечення схожі [7]. З введенням підходу щодо використання згорткових нейронних мереж (CNN) можна було отримати локальні та інваріантні особливості із зображення, знаходячи шаблони незалежно від їхнього положення. Таким чином, нейронна мережа давала змогу виявляти зразки відомого зловмисного програмного забезпечення на зображенні [7] [15].

Хоча запропоноване рішення має ряд переваг, які дозволяють виявляти шкідливі програми в контексті реального часу, ця стратегія має проблеми з певними зразками, які були стиснуті або зашифровані, а також з тим, які можуть мати зовсім іншу загальну структуру.

# РОЗДІЛ 3
# ТЕОРЕТИЧНІ ЗАСАДИ ГЛИБИННОГО НАВЧАННЯ

Нейронні мережі імітують роботу людського мозку, дозволяючи комп'ютерним програмам виявляти закономірності та вирішувати загальні проблеми в AI, ML і глибинному навчанні (deep learning / DL). Так, DL є провідною темою в спільнотах штучного інтелекту та машинного навчання та відноситься до набору методів, які використовуються для навчання в нейронних мережах. У цьому розділі описано основні концепції глибинного навчання.

Історію глибинного навчання можна простежити ще в 1940-х роках. Через обмежену обчислювальну потужність і недосвідчені стратегії навчання на той час, ранні алгоритми глибинного навчання, такі як штучні нейронні мережі, майже завжди приводили до локально оптимального рішення, яке не гарантувало конвергенцію. Лише в 2006 році було запропоноване зворотне розповсюдження (backward propagation) [16] як ефективний спосіб навчання та налаштування мережі, внаслідок чого алгоритми глибинного навчання набули корисності [17].

Успіх глибинного навчання значною мірою пояснюється відсутністю вручну розробленого екстрактора функцій. Алгоритми глибинного навчання здатні автоматично виявляти ознаки або уявлення, необхідні для класифікації, безпосередньо з необроблених даних. Іншими словами, екстрактор функцій є розроблений самим алгоритмом. Такі екстрактори ознак займають менше часу, менш схильні до помилок і, що найважливіше, здатні витягувати складні високовимірні характеристики, про які люди не можуть здогадатись [18].

Глибинне навчання досягло значних успіхів у багатьох областях, таких як розпізнавання зображень, розпізнавання мови, обробка природної мови та багато іншого [18]. Далі в роботі розглядаються дві найбільш перспективні архітектури нейронних мереж (згорткова та трансфомер), які можуть бути використані в системі виявлення зловмисного програмного забезпечення.

Штучний інтелект (artificial intelligence / AI) - це широка галузь комп'ютерних наук, зосереджена на створенні розумних комп'ютерних програм, здатних вирішувати завдання, які зазвичай асоціюються з розумними істотами.

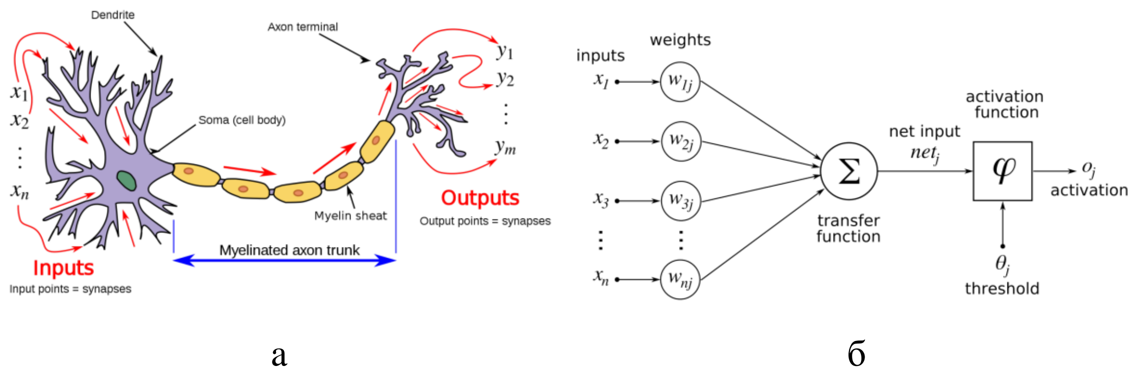
Машинне навчання (machine learning / ML) - це підгалузь штучного інтелекту, зосереджена на створенні комп'ютерних програм, які можуть навчитися вирішувати певні завдання і вдосконалюватися без наявного завдання і самовдосконалюватися без явного програмування. У галузі машинного навчання існує кілька різних методів для досягнення цієї мети.

Пізніші інновації в машинному навчанні значною мірою були зумовлені дослідженнями в галузі штучних нейронних мереж (artificial neural networks / ANN). Цей підхід натхненний тим, що мозок функціонує як мережа взаємопов'язаних нейронів. Штучні нейронні мережі закладають основу сучасних алгоритмів глибинного навчання та спочатку були натхненні моделюванням біологічної нейронної системи.

Першим фізичним застосуванням ANN, винайденим у 1958 році, була одношарова мережа під назвою "персептрон Mark 1". Він був побудований як машина, призначена для розпізнавання зображень [19]. Основним структурним елементом персептрона були лінійні порогові одиниці, які також називаються персептронами. Вперше вони були запропоновані Мак-Каллохом і Піттсом у 1943 році і заклали основи математичної моделі штучної нейцонної мережі [20]. Сьогодні ці одиниці відомі як штучні нейрони.

## 3.1 Штучний нейрон

Штучний нейрон залишився майже незмінним з моменту свого виникнення і є будівельним блоком багатьох сучасних архітектур ANN. Вважається, що ці нейрони є спрощеною математичною моделлю нейронів людського мозку. Незважаючи на те, що біологічний нейрон є значно більш досконалим, основні концепції залишаються тими ж самими як показано далі (рис. 3.1.1).



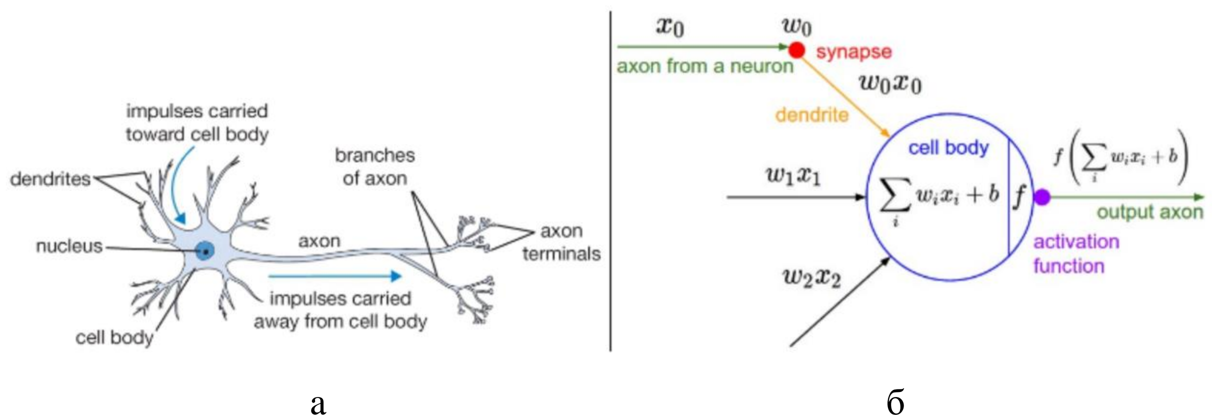

а б

Рисунок 3.1.1 – Порівняння нейронів:

а – спрощений біологічний нейрон [21]; б – штучний нейрон [22]

На рис.3.1.1 можна побачити схожу структуру штучного (а) та біологічного нейронів.

Нейрон, з'єднаний синапсами є основним будівельним блоком біологічної нейронної системи. Кожен нейрон отримує вхідні сигнали від своїх дендритів, потім вхідні дані обробляються клітинним ядром і виводять сигнали вздовж свого аксона через синапси. Після цього вихідний сигнал приймається дендритами інших нейронів. Обчислювальна модель нейрона є надзвичайно спрощеною версією біологічного нейрона: дендрити та аксони відображаються на вході та виході обчислювального нейрона, синапси є продуктами входу та ваг, а клітинне ядро моделюється за допомогою функції активації (рис. 3.1.2) [23].




а б

Рисунок 3.1.2 – Порівняння нейронів:

а – спрощений біологічний нейрон; б – обчислювальний нейрон [23]

Функціональність штучного нейрона полягає в тому, щоб приймати ряд дискретних вхідних сигналів, які є виходами інших нейронів.

Це схоже на те як працюють дендрити біологічного нейрона. Зважена сума цих вхідних сигналів обчислюється і передається нелінійній пороговій функції, також відомій як функція активації (activation function). Функція активації визначає, чи повинен нейрон надсилати вихідний сигнал, чи ні. Це можна порівняти з біологічним нейроном, в якому сома (цибулиноподібна, не відросткова частина нейрона або іншого типу клітин мозку, що містить клітинне ядро [24]) виконує функцію підсумовування та активації, змушуючи вихідні аксони посилати електричні сигнали, якщо електричний потенціал в сомі досягає певного порогу.

Математично штучний нейрон можна визначити як у рівнянні 3.1, де

– $y_j$ позначає вихід $j$-ого нейрона;

– $b$ – член зсуву, що навчається, який представляє порогове значення нейрона;

– $x_i$ – вихід $i$-ого попереднього нейрона з $w_i$, що позначає вагу, яка навчається та пов'язана з $x_i$;

– $\phi()$ представляє функцію активації.

$$y_j = \varphi\left(b + \sum_{i=1}^{n} w_{ij} x_i\right) \quad (3.1)$$

### 3.2. Штучна нейронна мережа

Штучні нейронні мережі (artificial neural networks / ANN) виникли як комп’ютерна технологія (натхненна концепцією людських біологічних нейронних мереж, таких як нервова система та мозок), яка обробляє інформацію та дозволяє машині навчатися на даних. Існують різні типи ШНМ, але лише три з них розглядаються в цій дипломній роботі: згорткова нейронна мережа (convolutional neural network / CNN), Swin трансформер (transformer) та гібридна нейронна мережа CoAtNet.

Основна одиниця кожної нейронної мережі називається нейроном або блоком обробки. Вони організовані в рівні, де кожен нейрон діє на основі

локальної інформації та передає свій вихід нейронам на тому самому рівні (внутрішньошарові з’єднання) на інший рівень (міжшарові з’єднання) або на внутрішній і проміжний шари для виконання завдання з розпізнавання.

Люди здатні створювати ментальні шаблони на своїх біологічних нейронних мережах з різних вхідних даних (наприклад, тексту, зображень, звуків) за допомогою сенсорних механізмів зору, звуку, дотику, нюху та смаку. Подібним чином нейронні мережі розпізнають шаблони даних у характеристиках вхідних даних.

Штучні нейрони можуть бути складені один на одного, з'єднані та розташовані шарами (рис. 3.2.1). У такій конфігурації всі входи передаються вперед через шари вузлів до кінцевих виходів без циклів. Це найпростіший тип ШНМ, відомий як мережа прямого поширення. Персептрон 1958 року був фактично одношаровою мережею прямого поширення. Шар у мережі прямого поширення (feed-forward network) часто називають повністю зв'язаним шаром (a fully-connected layer), оскільки вихід кожного нейрона з'єднаний з входом кожного нейрона в наступному шарі.

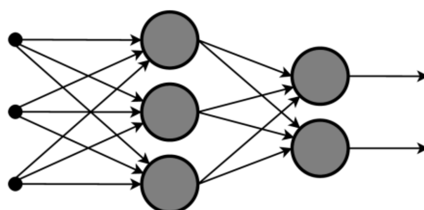


Рисунок 3.2.1 – Візуалізація штучної нейронної мережі прямого поширення [25]

Кожен шар нейронів у мережі прямого поширення може бути сформульований як математична функція. Враховуючи x як вхідні дані, а θ як набір параметрів, що навчаються для шару, тобто w і b, маємо y = f(x;θ).

Використовуючи ті ж самі позначення, можна визначити нейронну мережу як комбінацію декількох таких функцій, як показано в рівнянні 3.2.

$$\hat{y} = f(x;\theta) = f3(f2(f1(x;\theta))) \tag{3.2}$$

На відміну від біологічної нейронної системи, де нейрони згруповані та розташовані випадковим чином, нейрони в обчислювальній моделі організовані

шарами. Перший рівень — вхідний, а останній — вихідний, має розмір n, де n — кількість окремих категорій для класифікації. Усі шари між ними є прихованими (рис. 3.2.2, як представлення архітектури).

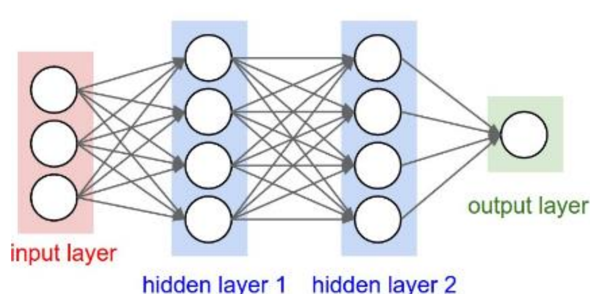



Рисунок 3.2.2 – Приклад нейронної мережі з 3 входами, 2 прихованими шарами з 4 нейронів і одним вихідним шаром [23]

Кожен нейрон повністю з'єднаний з усіма нейронами попереднього та наступного шару, але не з'єднаний з жодним із нейронів того самого шару. Нейрони в кожному шарі отримують вхідні дані (наприклад, $x_0$) від попереднього рівня, а синапс (наприклад, $w_0$) мультиплікативно взаємодіє з кожним вхідним сигналом (наприклад, $w_0x_0$). Потім нейрон обчислює суму всіх взаємодій синапсів, додає зсувний член і передає суму у функцію активації. Вихідні дані функції активації передаються вперед на вхід нейронів у наступні шари. Цей процес триває до вихідного рівня, де нейрон із найвищим виходом визначає категорію, до якої належить вхід. У процесі навчання вихід кожного зразка порівнюється з очікуваним результатом.

Різниця між очікуваними та фактичними значеннями кожного нейрона використовується для зворотного поширення (back propagation) [26], що незначно коригує ваги та зміщення кожного нейрона в прихованому шарі. Основна ідея штучних нейронних мереж полягає в тому, що зворотне поширення здатне дещо коригувати ваги та зміщення для кожної вибірки, і при достатній кількості вибірок, ваги та зміщення будуть збігатися, забезпечуючи оптимальну ефективність класифікації [23]. Як зауваження, біологічна модель є лише джерелом натхнення для нейронних мереж, обчислювальна модель є значно спрощеною версією, а поточні моделі нейронних мереж значно відхиляються від біологічної моделі [27].

Звичайні нейронні мережі погано масштабуються з практичних міркувань. Велика кількість вагових коефіцієнтів, ймовірно, перевищить дані [28].

Як було зазначено раніше, архітектура штучної нейронної мережі базується на взаємопов'язаних нейронах, де кожен нейрон складається з блоку підсумовування, за яким слідує блок виведення. Як і людина, блок підсумовування отримує вхідні дані від сенсорних механізмів (блоку), зважує кожне значення та обчислює зважену суму. Сумовий результат (значення активації) передається на вихідний блок для створення сигналу. Щоб зрозуміти, як працює нейронна мережа, важливо зрозуміти, що таке нейрони, як вони навчаються та передають знання один одному. В наступних розділах представлені два основних представлення нейрона: персептрон і сигмоїдна модель.

### 3.2.1 Персептрон

У 1958 році Розенблат запропонував штучне представлення нейронів — модель перцептрона. Це був найперший алгоритм керованого навчання бінарних класифікаторів, який є основною структурою для штучних нейронних мереж (ШМН).

Модель персептрона вводить регульовані ваги ($w_1$-$w_m$) у вхідні дані ШНМ, щоб мінімізувати помилку від фактичного двійкового виходу до бажаного (рис. 3.2.1.1). Ваги нейрона неявно визначають, які характеристики є більш значущими, і якщо функція є більш релевантною, це вплине на результат. Кожне нейронне з'єднання має власну вагу, яка починається з фіксованої або випадкової величини, а потім оновлюється під час навчання моделі. На вихід також впливає коефіцієнт зміщення (θ), який визначає чутливість сенсорних вхідних даних. Деякі типи даних можуть мати великий динамічний діапазон. Наприклад, об'єкт на зображенні має різні відображення залежно від тьмяного або яскравого світла. Необхідно визначити чутливість нейрона. Якщо він надто чутливий до менших значень вхідних даних, його вихідний сигнал буде насиченим для великих

вхідних значень. Однак, якщо він надто чутливий до великих значень вхідних даних, його значення активації стає нечутливим до малих значень вхідних даних.

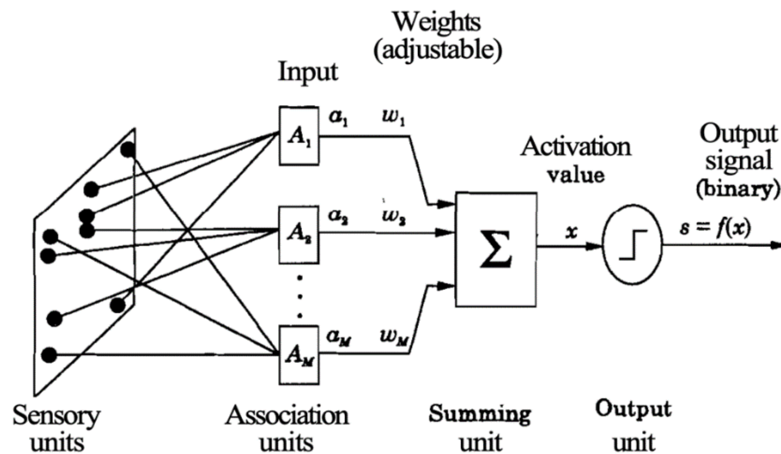



Рисунок 3.2.1.1 – Персептронна модель нейрона Розенблата [29]

Двійковий вихід персептрона, який веде до сигналу, є результатом f(x), який може бути 0 або 1, визначеним зваженою сумою x = $\Sigma w_i \alpha_i + \theta M_i = 1$ менше або більше 0.

$$Output = f(x)\begin{cases}1, if\ x > 0\\0, if\ x \leq 0\end{cases} \tag{3.3}$$

Завдяки двійковому виводу один шар персептрона обробляє лише лінійно розділені елементи в просторі. Для виконання будь-якого завдання класифікації зразків необхідний мультиперцептронний рівень. Насправді, штучні нейронні мережі (ШНМ) складаються з кількох шарів персептронів, тоді як перцептрон є представленням одного рівня ШНМ.

Основна складність багатошарової мережі персептронів полягає в тому, що дуже важко налаштувати ваги та зміщення. Невеликі зміни в них можуть кардинально змінити вихід, перемикаючись з 0 на 1 або навпаки, що вплине на поведінку мережі. З введенням сигмоїдних нейронів проблема була вирішена. Базуючись на моделі персептрона, функція виводу стала більш плавною. Замість того, щоб повертати двійковий вихід, він повертає дійсне значення від 0 до 1, яке можна інтерпретувати як ймовірність. Сигмоїдна функція визначається як:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

Як результат, відмінність між перцептроном і сигмоїдним нейроном полягає в функції активації, в якій вихід сигмоїдного нейрона:

$$Output = f(x) = \frac{1}{1 + e^{-(\sum_{i=1}^{M} w_i \alpha_i + \theta)}} \quad (3.5)$$

Таким чином, функції активації використовуються для стандартизації вихідних значень кожного нейрона. Найпоширенішими функціями активації є сигмоїда та випрямлена лінійна одиниця (ReLu) [30].

Мережі прямого зв'язку протікають лише в одному напрямку, від входу до виходу. Алгоритм зворотного поширення (backpropagation) інвертує потік, змінюючи кожну вагу в мережі. Він враховує частку загальної похибки, що дозволяє належним чином налаштувати та підігнати алгоритм. Кожна ітерація призводить до зменшення похибки ваги, що зрештою призведе до ваг, які дають хороші прогнози. Метод зворотного поширення помилки — метод навчання багатошарового перцептрону.

Для вимірювання продуктивності нейронної мережі за допомогою однієї навчальної вибірки використовується функція втрат (loss function). Тоді як середнім для всього навчального набору даних є функція вартості (value function). Функція вартості кількісно визначає похибку між прогнозом алгоритму та очікуваними значеннями в дійсному числі. Щоб мінімізувати функцію витрати/втрати, необхідно знайти баланс між вагами та зміщеннями за допомогою алгоритму оптимізації.

У контексті машинного навчання мірою помилки для задачі багатокласової класифікації є перехресна ентропія (cross-entropy). Зазвичай «істинний» розподіл (той, якому намагається відповідати алгоритм машинного навчання) виражається в термінах унітарного кодування (one-hot). Тоді як, в теорії інформації перехресна ентропія між двома розподілами ймовірності p та q над спільним простором подій вимірює середню кількість біт, необхідних для впізнання події з простору подій, якщо схема кодування, що використовується, базується на розподілі ймовірностей q, замість «істинного» розподілу p [31].

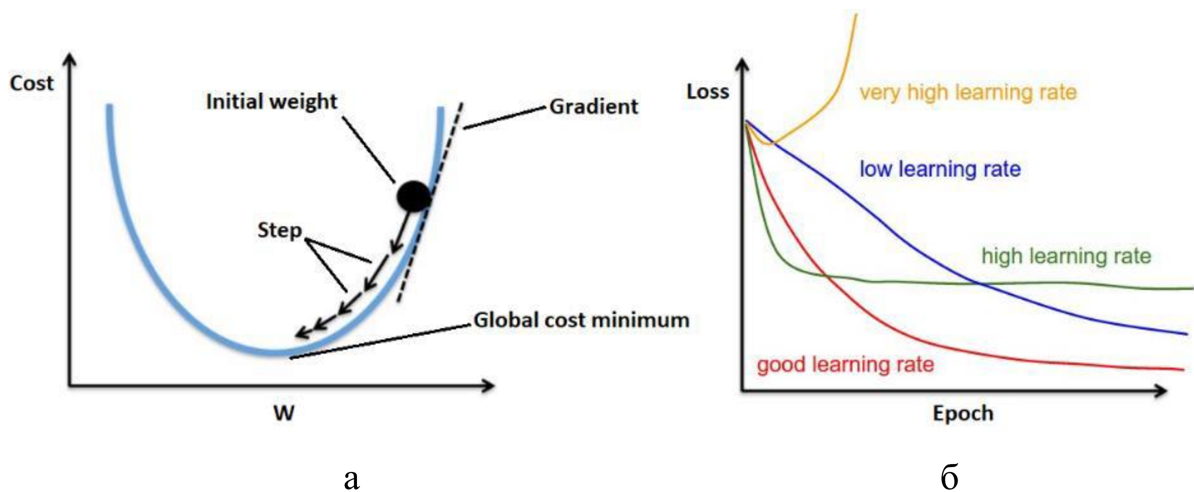
Втрати перехресної ентропії, або лог-втрати, вимірюють продуктивність моделі класифікації, вихідним результатом якої є значення ймовірності між 0 і 1. Втрати перехресної ентропії зростають, коли прогнозована ймовірність розходиться з фактичним значенням мітки.

### 3.2.2 Алгоритми оптимізації

Алгоритми оптимізації використовуються під час процесу навчання, щоб ітеративно оцінити параметри мережі, які призводять до мінімального значення функції вартості. Градієнтний спуск, середньоквадратичне поширення (RMSProp) і Адам (Adam) – це деякі з методів оптимізації, які обговорюються в цьому розділі. Градієнтний спуск — найпростіший метод оптимізації нейронної мережі, тоді як варіанти RMSProp і Adam зазвичай швидше зближуються до глобального мінімуму функції.

Популярним підходом до оптимізації для знаходження локального мінімуму або найменшого значення функції втрат, градієнт якої дорівнює нулю, є градієнтний спуск (gradient descent / GD) [32]. Алгоритм починається з випадкової ініціалізації ваги та зсуву в NN. Потім він ітеративно отримує градієнтний спуск на кожному кроці, поки не знайде нижню частину графіка. В результаті, він знаходить мінімум, де помилка найменша (рис. 3.2.2.1 а). Це означає, що модель налаштована на дані та здатна давати точніші прогнози.

Розмір кроку називається швидкістю навчання (рис. 3.2.2.1 б). Низька швидкість є більш точною, оскільки вона часто перераховує градієнт, але вона займає більше часу, тому знадобиться більше часу, щоб досягти дна. Високі показники навчання охоплюють більше кроків, але існує ризик перевищення найнижчої точки, оскільки схил пагорба постійно змінюється, застрягаючи на гірших значеннях втрат протягом епох.




а б

Рисунок 3.2.2.1 – Градієнтний спуск:

а – представлення градієнта, визначеного як нахил кривої та похідна функції активації; б – вплив різних темпів навчання

Залежно від обсягу даних, можливо, краще мати точне оновлення ваги замість коротшого часу для оновлення. Існує три варіанти градієнтного спуску залежно від кількості даних, які використовуються для вимірювання градієнта цільової функції:

– пакетний градієнтний спуск (batch gradient descent / BGD): обчислює середнє значення градієнтів для всього навчального набору даних, а потім використовує цей середній градієнт для оновлення ваг, виконуючи лише одне оновлення (крок) за одну епоху; однак, він є ненадійним для великих наборів даних, де BGD може бути повільним і складним, якщо не поміщається в пам’яті;

– стохастичний градієнтний спуск (stochastic gradient descent / SGD): обчислює градієнт для кожного прикладу в навчальному наборі даних і оновлює ваги; оскільки він розглядає лише один приклад за раз, вартість може коливатися в залежності від навчальних прикладів, що може ускладнити конвергенцію до точного локального мінімуму;

– міні-пакетний градієнтний спуск (mini-batch gradient descent / MBGD): пакетний градієнтний спуск збігається безпосередньо до локального мінімуму, а стохастичний градієнтний спуск швидше збігається для великих наборів даних; обидва методи мають свої обмеження - вони підходять лише для невеликих наборів даних або обчислюють один приклад за раз, відповідно; MBGD

використовує переваги обох підходів, базуючи обчислення на пакеті з фіксованою кількістю навчальних прикладів, що називається міні-пакетом; він обчислює середній градієнт міні-серії та часто оновлює вагові коефіцієнти, а також дозволяє векторизовану реалізацію для швидших обчислень.

Ще одним методом оптимізації, який широко використовується в машинному навчанні, є середньоквадратичне поширення (RMSProp). Його поведінка заснована на градієнтному спуску [32]. Він обмежує коливання у вертикальному напрямку, що збільшує швидкість навчання, і алгоритм може робити більші кроки в горизонтальному напрямку, швидше зближуючись до локального мінімуму [33].

Мінімальний або глобальний оптимум представлений точкою локального оптимуму показаний на рис. 3.2.2.2. Глобальний мінімум — це найкращий локальний мінімум/оптимум, якого можна досягти. Коли в точці «A» починається GD, після однієї ітерації лінія може закінчитися в позиції «B» (на іншій стороні еліпса). Тоді інший крок GD може бути зупинений у точці «C». У результаті досягнення мінімуму займе багато часу, збільшуючи ймовірність того, що градієнтний спуск застрягне в локальному оптимумі, а не досягне глобального оптимуму.

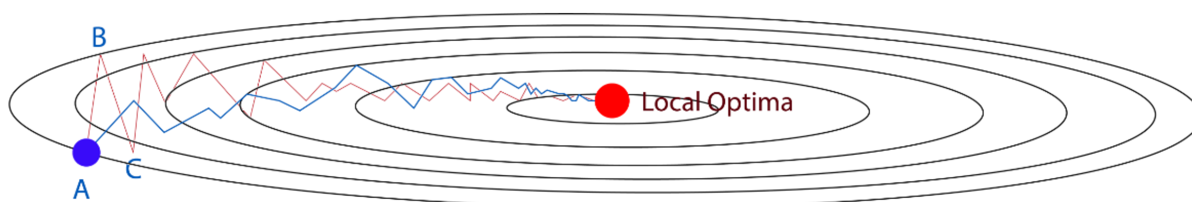



Рисунок 3.2.2.2 – Порівняння Gradient Descent і RMSProp [33]

Щоб не застрягти в локальному оптимумі, необхідне повільніше навчання у вертикальному напрямку та швидше навчання в горизонтальному напрямку. RMSProp використовує ідею експоненціально зваженого середнього (EWA) градієнтів, що дозволяє алгоритму забути ранні градієнти та зосередитися на нещодавно спостережених часткових градієнтах, знайдених під час пошуку.

Як показано на рис. 3.2.2.2 навчання у вертикальному напрямку сповільнюється, а досягнення локального мінімуму відбувається швидше.

Алгоритм Адам (Adam) поєднує всі попередні техніки в єдиний ефективний алгоритм. Як результат, цей алгоритм став популярним як один із найнадійніших і найефективніших алгоритмів оптимізації для глибинного навчання. Він ефективний з точки зору обчислень, потребує мало пам’яті, інваріантний до градієнтного діагонального масштабування та добре підходить для проблем із великою кількістю даних або параметрів. Метод також підходить для нестаціонарних цілей і проблем, пов’язаних із дуже шумними або розрідженими градієнтами, де мережа не може налаштувати свої ваги через відсутність сильних сигналів [34].

### 3.2.3 Перенавчання

Коли модель намагається передбачити шаблон у надто шумних даних відбувається перенавчання (overfitting) моделі. Це призводить до неточності такої моделі, оскільки зазначена тенденція не відображає реальності, наявної в даних. Модель, яка дає хороші результати на навчальному наборі даних, але погано працює на небачених даних, є ознакою того, що модель перенавчена. Зробити правильні припущення з набору навчальних даних для будь-яких інших даних із проблемної предметної області (яких не було раніше), є основною метою будь-якої моделі ML. Далі по тексту описано декілька найбільш поширених прийомів для  запобігання перенавчанню під час навчання нейронних мереж.

Почнемо з методу збільшення (augmentation) даних. Він полягає, відповідно, у збільшенні/розширенні розміру навчальних даних. Деякі з простих методів штучного розширення даних – це перевертання, переклад, обертання, масштабування та транспонування. Ці методи використовуються на найпоширеніших навчальних наборах даних для CNN, таких як CIFAR-10 [35]. Додавання більшої кількості даних змушує модель узагальнюватися, втрачаючи можливості перенавчити всі зразки.

Наступним методом запобігання перенавчанню моделі є регуляризація (regularization). Це техніка, яка додає член до функції втрат, щоб штрафувати або

накладати плату за зважування, змушуючи мережу вибирати матриці з меншою вагою, що призводить до простіших моделей, що зменшує перенавчання.

Ще одним методом є виключення (dropout) нейронів. Він базується на ідеї модифікації мережі в різні мережі шляхом випадкового виключення нейронів із нейронної мережі під час кожної взаємодії під час навчання, запобігаючи коадаптації нейронів. Як результат, різні мережі будуть переобладнані різними способами, що призведе до зменшення переобладнання як впливу на мережу. У кожній ітерації процесу певний відсоток нейронів випадково та тимчасово від'єднується, за винятком тих, які належать до вхідного та вихідного рівня. Потім вхідні дані поширюються через модифіковану мережу, а разом з ними і результат. Вагові коефіцієнти та зміщення оновлюються, і процес повторюється шляхом відновлення вилучених нейронів і вибору нової випадкової підмножини нейронів для відключення.

Нарешті, через зменшення коадаптації нейронів, нейрон не може покладатися на інший нейрон та змушений вивчати більше ознак у деталях, що корисно в поєднанні з іншими випадковими підмножинами нейронів. Вихідний результат процесу виключення можна розглядати як середнє значення великої кількості мереж.

### 3.3. Глибинне навчання

В останні роки, коли обчислювальні потужності стали більш доступними, кількість шарів у мережі прямого поширення значно збільшилася, що дозволило штучним нейронним мережам (ANN) моделювати більш складні функції. Галузь досліджень, пов'язана з багатошаровими (many-layered) ANN, відома як глибинне навчання.

Глибинне навчання (deep learning / DL) відноситься до набору методів, які використовуються для навчання в багатошарових нейронних мережах (deep neural networks / DNN). Таким чином, «глибина» глибинного навчання пов'язана з глибиною шарів у мережі та є підмножиною алгоритмів машинного навчання.

Спостерігаючи закономірності в даних, модель глибинного навчання потребуватиме більше точок даних для підвищення своєї точності, що означає потребу в більшої потужності комп'ютера, тоді як модель машинного навчання залежить від меншої кількості даних завдяки структурі даних, яка лежить в основі.

Глибоке навчання може бути керованим / контрольованим (supervised), неконтрольованим (unsupervised), напівконтрольованим (semi-supervised), самоконтрольованим (self-supervised) або з підкріпленням (reinforcement). В цій роботі для навчання моделей використовується метод керованого навчання.

У багатошарових нейронних мережах зникаючі градієнти (vanishing gradients) є однією з найбільших проблем, яка може призвести до непередбачуваної поведінки під час фази навчання. Це визначається нездатністю нейронної мережі поширювати інформацію про градієнт із вихідних даних назад до нижніх рівнів моделі. Градієнти контролюють скільки мережа вивчає під час фази навчання. Якщо градієнти близькі до нуля, модель покращується дуже повільно, що можливо призведе також до припинення навчання. Це може спричинити нездатність моделей із багатьма шарами навчатися на певному наборі даних або збігатися до низької ефективності прогнозування.

Проблему зникаючих градієнтів можна вирішити, використовуючи випрямлені лінійні вузли ReLU (rectified linear unit) як функцію активації. ReLU – це зрізаний/випрямлений лінійний вузол або випрямляч (rectifier). У контексті штучних нейронних мереж він є передавальною функцією, яка визначена наступним чином [30].

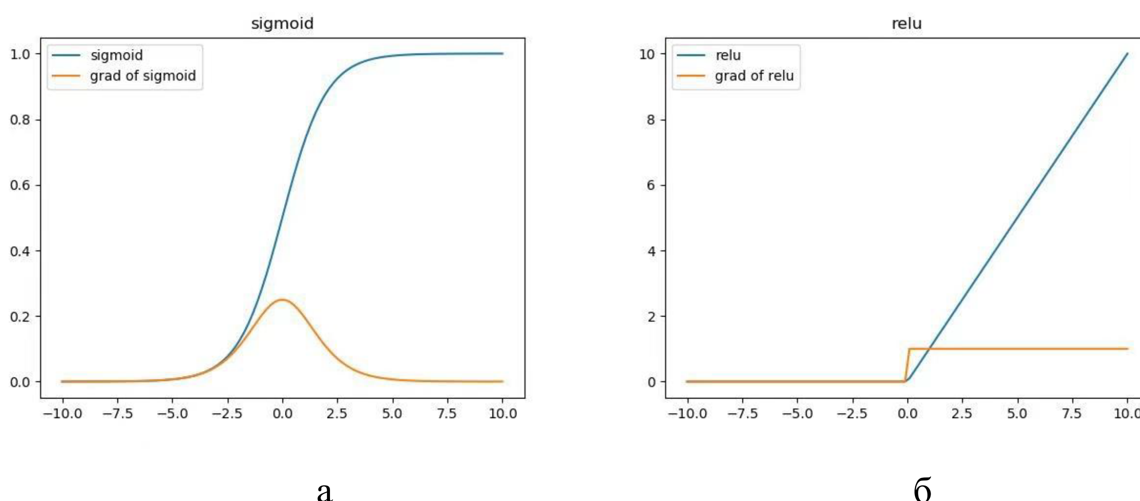
$$f(x) = x^{+} = \max(0, x) \tag{3.6}$$

Функція повертає 0, якщо вона отримує будь-які негативні вхідні дані, і сам вхідний сигнал для будь-яких позитивних вхідних даних, f(x) = max(0, x) (рис. 3.3.1).

Тоді як, градієнт сигмоїдної функції здавлює великий вхідний простір у малий вхідний простір між 0 і 1 (рис. 3.3.1. а). Тому велика зміна на вході

сигмоїдної функції призведе до незначної зміни на виході. Отже, похідна стає малою, що сприяє проблемі зникнення градієнта.

Градієнт ReLU може бути лише 0 або 1, який після багатьох шарів має тенденцію до стабілізації. Таким чином, загальний градієнт не надто малий або не надто високий (рис. 3.3.1. б), таким чином, контролюючи швидкість навчання. Крім того, його простіше обчислити, а похідна ReLU споживає менше обчислювальних ресурсів, ніж сигмоїдна функція [36].




а б

Рисунок 3.3.1 – Порівняння функції активації та похідної [27]:

а – сигмоїдна; б – ReLU

Функція сигмоїдної активації використовується для двокласової логістичної регресії, тоді як функція софтмакс (softmax) використовується для багатокласової логістичної регресії. Softmax нормалізує вхідне значення у вектор значень, які слідують за розподілом ймовірностей із загальною сумою одиниці. Вона використовується як функція активації, подібної до ReLU, але застосовується до останніх рівнів (а не до середніх) з метою нормалізації виходу моделей нейронної мережі для проблем багатокласової класифікації, де потрібна асоціація класу в більш, ніж двох мітках класу [37].

Стандартна структура моделі нейронної мережі з softmax як результатом показана на рис. 3.3.2.

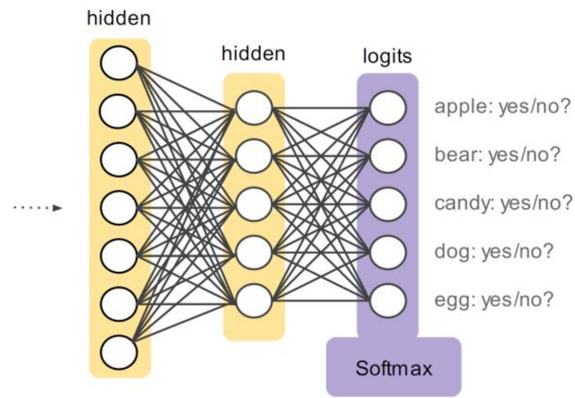



Рисунок 3.3.2 – Шар Softmax у нейронній мережі

Незбалансовані дані є поширеною проблемою під час навчання моделей машинного навчання, що означає, що класи набору даних представлені неоднаково. Існують два популярних підходи для вирішення зазначеної проблеми встановлення балансу. Як результат, кожен клас починає мати однакову вагу для моделі. Далі по тексту ці підходи описані більш детально:

– зважування класу (class weighting) – ваги класів безпосередньо впливають на функцію втрат (loss function), накладаючи більший (або менший) штраф на класи з більшою (або меншою) вагою - по суті, шляхом навмисного коригування зміщення моделі на користь більш точних прогнозів у вищій ваговій категорії, втрачаючи певну здатність прогнозувати нижчу вагову категорію (основний клас у незбалансованих наборах даних) [38];

– надмірна та недостатня вибірки (oversampling and undersampling) – методи надмірної вибірки (oversampling) по суті надають більшої ваги конкретним класам через дублювання спостережень, надаючи їм більшого впливу на підгонку моделі (однак, це може призвести до перенавчання моделі); методи недостатньої вибірки (undersampling), з іншого боку, видаляють вибірки з мажоритарного класу, що може призвести до втрати інформації важливої для моделі [38].

Далі по тексту коротко описані основні інструменти машинного навчання та бібліотеки, які використовуються для роботи над цією роботою: метрики оцінювання, архітектури глибинного навчання, а також, моделі штучних нейронних мереж.

### 3.4. Метрики оцінювання

Продуктивність алгоритмів машинного навчання можна виміряти різними методами. Метрична функція оцінки використовується для оцінки ефективності моделі. Далі про тексту розглядаються деякі з них – такі, як акуратність (accuracy), точність (precision), повнота (recall). Це відомі показники, що використовуються для вирішення проблем класифікації в машинному навчанні. Вони надзвичайно важливі, коли йдеться про статистичну перевірку гіпотез.

Маючи справу з проблемами класифікації, можна намагатись передбачити двійковий результат, наприклад, відношення окремого зображення зловмисного програмного забезпечення до зображень зловмисного програмного забезпечення в цілому, а також, виявити належність такого програмного зебезпечення до конкретного сімейства (класу). В таких випадках важливо враховувати кількість прогнозів, які помилково класифіковані як позитивні (false positive / FP) та помилково класифіковані як негативні (false negative / FN), особливо з огляду на контекст того, що саме передбачається прогнозувати. Тому правильно класифіковані зразки позначаються істинно позитивними (true positive / TP) або істинно негативними (true negative / TN), залежно від класу, до якого вони належать. Неправильно класифіковані зразки називаються хибно позитивними (FP), якщо фактичний клас є негативним, але прогнозований клас є позитивним, і хибно позитивними (FN) - в іншому випадку [39].

Акуратність або точність класифікації (accuracy) – частка правильно ідентифікованих прикладів. Це відношення правильних передбачень, поділених на загальну кількість виконаних передбачень. Вона має бути якомога вищою. Акуратність обчислюється за наступною формулою:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.7}$$

Точність (precision) – це частка правильно ідентифікованих позитивних прикладів від усіх позитивно класифікованих прикладів. Вона є співвідношенням між правильними прогнозами для даного класу та його кількістю зразків, що представляє відсоток релевантних результатів. Тобто, скільки з усіх класів було

передбачено вірно. Вона має бути якомога вищою. Формула точності наведена далі:

$$Precision = \frac{TP}{Actual\ Results}\ or\ \frac{TP}{TP + FP} \tag{3.8}$$

Повнота (recall) - також відома як чутливість або істинно позитивна частота (true positive rate / TPR). Це частка правильно ідентифікованих прикладів від усіх позитивних прикладів. Вона характеризується як відсоток релевантних результатів, які правильно класифіковані моделлю. А саме, визначається як відношення загальної кількості істинно позитивних результатів до загальної кількості позитивних прикладів, тобто усіх позитивних прикладів до загальної кількості позитивних прикладів, тобто, скільки було спрогнозовано вірно з усіх позитивних класів. Також, чутливість або істинно позитивна частота (TPR), визначається як частка правильно ідентифікованих прикладів від усіх позитивних прикладів. Вона характеризується як відсоток релевантних результатів, які правильно класифіковані моделлю. А саме - як відношення загальної кількості істинно позитивних результатів до загальної кількості позитивних прикладів, тобто усіх позитивних прикладів до загальної кількості позитивних прикладів, тобто, скільки було прогнозовано вірно з усіх позитивних класів. Має бути якомога вищою. Формула повноти – наступна:

$$Recall = \frac{TP}{Predicted\ Results}\ or\ \frac{TP}{TP + FN} \tag{3.9}$$

Ще одною метрикою є F1-міра (F1-score). Це середньозважене значення показників точності та повноти. Вона враховує як точність, так і запам’ятовування для вимірювання точності моделі. Хибно позитивні та хибно негативні результати можуть бути критичними залежно від того, що прогнозується, але вірно-негативні результати часто менш важливі. F1-міра намагається скоригувати це, призначаючи більшу вагу хибно негативним і хибно позитивним результатам, виключаючи з прогнозу велику кількість вірно негативних результатів. На точність впливають, головним чином, хибно негативні результати, на яких не слід зосереджуватись, тоді як на хибно

негативних і хибно позитивних результатах зазвичай зосереджуються. F1-міра може бути кращим показником для використання, якщо нам потрібно шукати баланс між точністю та пригадуванням, а також, якщо існує нерівномірний розподіл класів. Показник F1 досягає свого найкращого значення при 1, а найгіршого - при 0. Формула F1-міри – наступна:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (3.10)$$

Додатково до раніше зазначених метрик, для опису ефективності моделі статистичної класифікації використовується техніка під назвою матриця помилок (confusion matrix). Сама по собі точність класифікації може ввести в оману, якщо кількість спостережень у кожному класі неоднакова (наприклад, незбалансований набір даних) або якщо набір даних має більше двох класів.

По факту, матриця помилок – це таблиця, яка часто використовується для опису продуктивності моделі класифікації на наборі тестових даних, для яких відомі істинні значення. На виході може бути 2 або більше класів. Матриця показує, як класифікаційна модель плутається, коли вона робить прогнози.

Обчислення матриці помилок може забезпечити краще розуміння правильної поведінки моделі класифікації та допомогти виявити можливі помилки. Кількість точних і помилкових прогнозів підсумовується з підрахунками та ділиться на класи.

Представлення двійкової та багатокласової матриць помилок [40] показані на рис. 3.4.

| | | PREDICTED | |
|---|---|---|---|
| | Classes | Positive | Negative |
| ACTUAL | Positive | TP | FN |
| | Negative | FP | TN |

а

| | | PREDICTED classification | | | |
|---|---|---|---|---|---|
| | Classes | a | b | c | d |
| ACTUAL classification | a | TN | FP | TN | TN |
| | b | FN | TP | FN | FN |
| | c | TN | FP | TN | TN |
| | d | TN | FP | TN | TN |

б

Рисунок 3.4 – Матриці помилок [53]:

а – двійкова; б – багатокласова

Матриця помилок зазначає чотири типи результатів:

– TP (вірно позитивний результат): кількість правильно класифікованих позитивних випадків;

– TN (вірно негативний результат): кількість правильно класифікованих негативних випадків;

– FP (хибно позитивний результат): кількість помилково класифікованих негативних випадків як позитивних;

– FN (хибно негативний результат): кількість помилково класифікованих позитивних випадків як негативних.

# РОЗДІЛ 4
# АРХІТЕКТУРИ НЕЙРОННИХ МЕРЕЖ

За останні десятиліття сфера машинного навчання пережила прорив у вирішенні багатьох завдань. Винахід згорткових нейронних мереж (Convolutional Neural Networks / CNN) став важливою віхою у розвитку розпізнавання зображень. CNN є формою штучної нейронної мережі (ANN), яка імітує спосіб опрацювання зображень зоровою корою головного мозку. У 2015 році була запропонована ANN, яка перевершила людську продуктивність у задачі класифікації зображень ImageNet [41]. Альтернативна до CNN архітектура штучної нейронної мережі, названа трансформерами зору (vision transformers), була представлена у 2020 році [42]. Це ознаменувало собою нове сімейство штучних нейронних мереж, продуктивність яких може бути порівнянною з CNN. Далі по тексту буде наведений огляд обох цих архітектур.

## 4.1 Згорткова нейронна мережа

Згорткові нейронні мережі (CNN) – це тип прямої нейронної мережі, що є дуже ефективним у розпізнаванні зображень і подібних завданнях. Основною відмінністю між різними CNN є розташування нейронів у шарі. Згорткові нейронні мережі мають нейрони, розташовані в трьох вимірах (ширина, висота та глибина). Крім того, кожен нейрон всередині шару згортки є пов’язаним лише з невеликою областю на наступному шарі. Це організовано за трьома різними типами шарів (згортковий/конволюційний, об’єднаний та повністю підключений), що також відрізняється від прямої нейронної мережі. Використання згорткових нейронних мереж є одним з найбільш досліджених підходів до методів виявлення об'єктів.

CNN витягує ієрархічні локальні особливості із зразків даних незалежно від їх розташування, тому часто використовується для класифікації зображень. Також, можна зробити припущення, що CNN є кращим кандидатом для виявлення шкідливих програм порівняно з рекурентними нейронними мережами

(recurrent neural networks / RNN). У просторі зображень CNN можна навчити розпізнавати додавання надлишкових викликів API або машинних інструкцій перекладу або спотворенню функцій програмного забезмечення.

Згорткові нейронні мережі (CNN) розглядають кожен шар як тривимірний об’єм, а не як одновимірний масив у ANN. Це дозволяє кожному шару в мережі трансформувати об’єм у різні розміри та поступово зменшувати обсяг для економії обчислювальної потужності. CNN також має в основному два нових шари (шари згортки та шари об’єднання), які допомагають мережі класифікувати дані з високою точністю та швидкістю.

Згорткові нейронні мережі – це сукупність шарів згорток з нелінійними функціями активації типу ReLU, що застосовуються до виходів згорток. У повністю пов'язаних нейронних мережах кожен вхідний нейрон з'єднаний з кожним вихідним нейроном у наступних шарах (це також називається щільно пов'язаним шаром, або афінним шаром). У згорткових нейронних мережах цього не відбувається — використовуються згортки над вхідним шаром для обчислення вихідного. Це призводить до локальних зв'язків, де кожна область вхідних даних з'єднана з нейроном у вихідному шарі. Кожен шар застосовує різні фільтри, як правило, сотні або тисячі, та об'єднує їх результати.

Загалом реалізацію CNN можна визначити як таку, що включає наступний процес [43] (рис. 4.1.1):

– згортання кількох маленьких фільтрів на зображенні та застосування активації ReLU до матриці;

– виконання об’єднання для підвибірки та зменшення розмірності;

– повторення кроків 1 і 2, доки не буде стільки шарів згортки, скільки буде достатньо для високорівневих функцій;

– очищення вихідних даних та подання їх на повністю підключений рівень;

– виведення класу за допомогою функції активації, такої як сигмоїда (sigmoid) або софтмакс (softmax) для класифікації зображення.

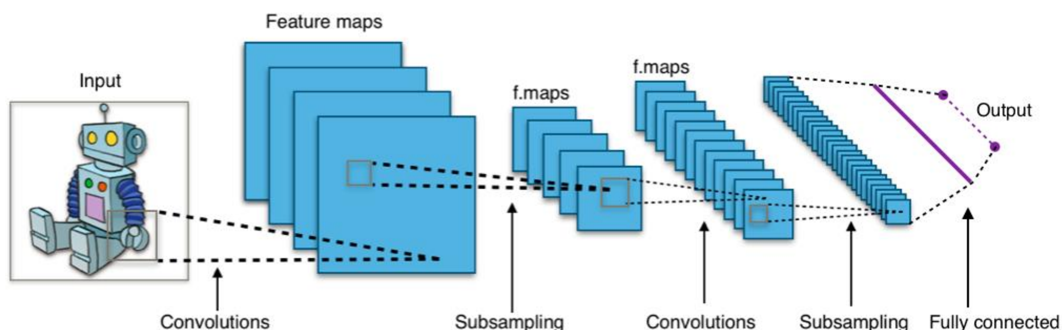



Рисунок 4.1.1 – Типова архітектура CNN [43]

Під час фази навчання CNN автоматично вивчає значення своїх фільтрів на основі завдання, яке потрібно виконати. Наприклад, при класифікації зображень штучна нейронна мережа може навчитися виявляти краї з необроблених пікселів у першому шарі, потім використовувати краї для виявлення простих форм у другому шарі, а потім використовувати ці форми для виявлення ознак більш високого рівня, таких як форми обличчя у вищих шарах. Останній шар є класифікатором, який використовує ці високорівневі ознаки.

CNN - це мережі прямого зв'язку, де нейрони активуються точно один раз під час кожної класифікації та не мають поняття часу та контексту. Така властивість дозволила згортковим нейромережам добре працювати з задачами класифікації зображень, де функції є просторово інваріантними, але - погано в задачах обробки природної мови та розпізнавання мовлення через те, що порядок і контекст слів мають велике значення.

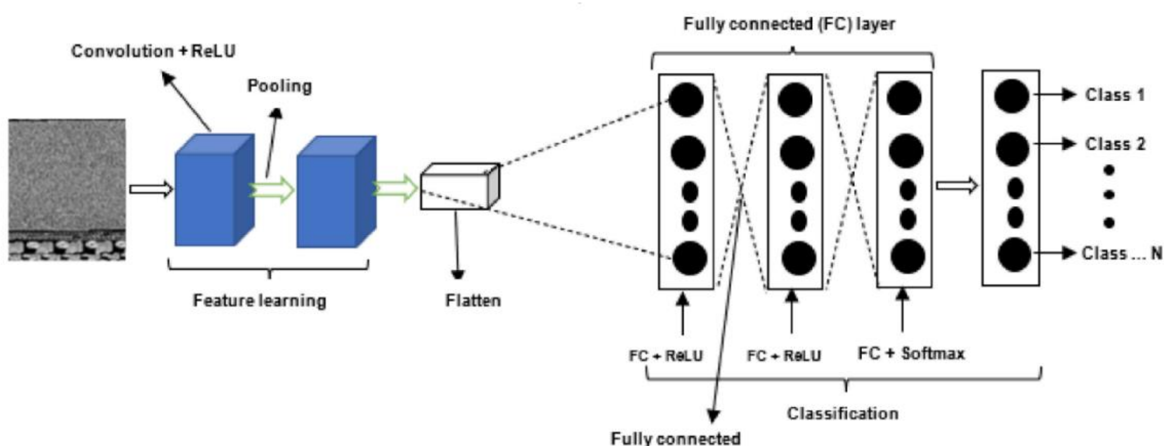
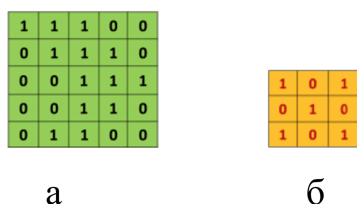



Рисунок 4.1.2 – Загальна архітектура згорткової нейронної мережі з розпізнавання шкідливого програмного забезпечення (перероблено з [44])

Рівень згортки є ядром CNN і є першим шаром (основним будівельним блоком), який витягує характеристики з вхідного зображення [45]. Рівень складається з набору доступних для навчання фільтрів, які вивчають особливості зображення за допомогою невеликих квадратів вхідних даних, створюючи двовимірну карту активації фільтра (рис. 4.1.1.1). У результаті карта активації запускається, коли виявляє деякі вивчені критерії через фільтри на деяку просторову позицію на вході.

Накопичування кількох шарів фільтрів дозволяє виконувати такі операції, як виявлення країв, розмиття, підвищення різкості, колір, орієнтація градієнта тощо. Це також покращує захоплення для низьких рівнів деталізації, хоча ціною більшої обчислювальної потужності.




а б

Рисунок 4.1.3 – Зображення [45]:

а – бінарне; б – матриця згортки

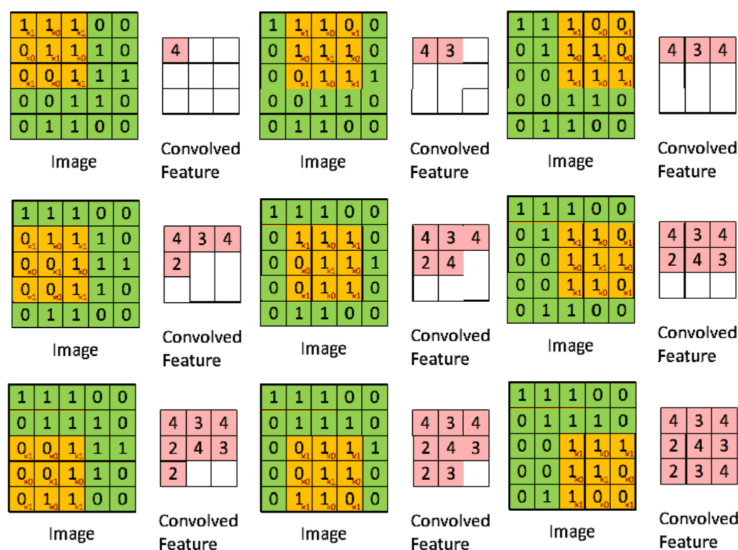



Рисунок 4.1.4 – Вихідні дані карти функцій [45]

Згорткові шари в CNN є локально з’єднаними на відміну від повністю з’єднаних, що одночасно є енергоефективним і дозволяє знаходити функції незалежно від положення.

Важливим аспектом CNN є об'єднуючі шари, які наносяться після згорткових шарів.

Роль шара об'єднання полягає в об'єднанні кількох менших об'єктів у більш загальний об'єкт [18], Рівень об'єднання не тільки зменшує розміри вхідного об'єму, але також запобігає перенавчанню, враховуючи більш загальне та абстрактне представлення [28]. Однак, було доведено, що шари об'єднання можна замінити згортковими шарами з більшим кроком без втрати точності [46].

Об'єднані шари зменшують гучність, щоб знаходити лише важливі шаблони та видаляти шум. Комбінація згорткового шару та шарів об'єднання діє як екстрактор ознак, класифікація виконується за допомогою повнозв'язаних шарів, подібних до штучних нейронних мереж із функціями, витягнутими зі згорткових шарів і шарів об'єднання як вхідних даних.

Об'єднання дозволяє зменшити розмір представлення і прискорити обчислення, а також зробити деякі ознаки, що виявляються, більш стійкими.

Агрегуючий (pooling) шар відповідає за субдискретизацію (subsampling) (або зниження якості (downsampling)) просторового розміру згорнутої риси [45]. Таке зменшення розміру кожної карти, що зберігає важливу інформацію, зменшує обсяг обчислень у мережі та контролює перенавчання. Існує декілька нелінійних функцій для реалізації агрегація, таких як мінімальне, максимальне та середнє, але найпоширенішим є максимальне або максі-агрегація

Найпоширенішим способом застосування об'єднання є максимальне об'єднання

Максимум або максимальна агрегація - це операція об'єднання, яка повертає найбільший елемент випрямленої частини карти ресурсів і використовує його для створення зменшеного (агрегуючого) ресурсу, що неявно зменшує обсяг даних і обчислень у мережі (рис. 4.1.5)

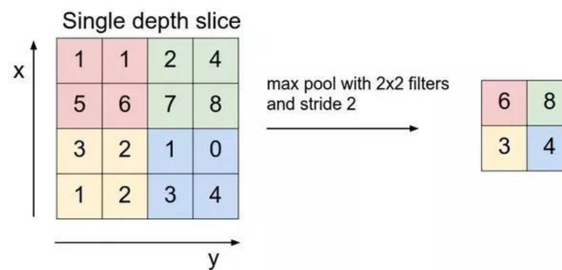



Рисунок 4.1.5 – Максимальна агрегація [45]

Було запропоновано багатовимірне масштабування (multidimensional scaling / MDS), яке виділяє набір ознак за допомогою гібридного аналізу та використовує CNN для класифікації виділених векторів ознак [47].

Шари згортки виводять високорівневі дані об’єктів, тоді як повністю зв’язаний рівень вивчає нелінійні комбінації цих об’єктів, які вирівнюються та з’єднуються з вихідними шарами. Таким чином, зведений вихід подається в нейронну мережу прямого зв’язку, а зворотне поширення (back propagation) застосовується до кожної ітерації навчання. Протягом певного періоду часу модель здатна розрізняти та класифікувати характеристики високого та низького рівня на зображеннях.

Застосування нульового заповнення називається широкою згорткою, а застосування нульового заповнення називається вузькою згорткою. Розмір кроку, який вказує на те, наскільки потрібно змістити фільтр на кожному кроці. Якщо розмір кроку дорівнює 1, наступні застосування фільтра будуть перекриватися. Більший крок призводить до меншої кількості застосувань фільтра і меншого розміру результату.

## 4.2. Трансформери

Архітектура трансформеру - це штучна нейронна мережа, створена для вивчення контексту і відстеження взаємозв'язків у послідовних даних.

Вперше вони були використані з великим успіхом у програмах машинного навчання, пов'язаних з мовою та перекладом [48]. Однак, представлення трансформеру зору у 2020 році [42] викликало великий інтерес до трансформерів у комп'ютерному зорі. Нещодавно було показано, що трансформери, які

застосовуються в задачах комп'ютерного зору, за наявності достатньої кількості навчальних даних демонструють ефективність, порівнянну з CNN, а в деяких випадках, навіть кращу за них [49]. Більшість програм, що використовують трансформери, навчаються на великих масивах даних.

Трансформери засновані на структурі кодер-декодер, тобто мережа розділена на дві частини, де кодер створює проміжне представлення вхідної послідовності, яке потім подається на декодер, що виробляє вихідну послідовність. Трансформер в основному складається з форми позиційного вбудовування (positional embedding), за яким слідують стеки шарів із самоувагою (self-attention) та повністю з'єднані шари з активаціями ReLU.

Огляд оригінальної структури трансформера можна побачити на рис. 4.2.1. Декодер приймає як вихід останнього кодерного блоку, так і вихід попереднього замаскованого шару з множинною увагою.

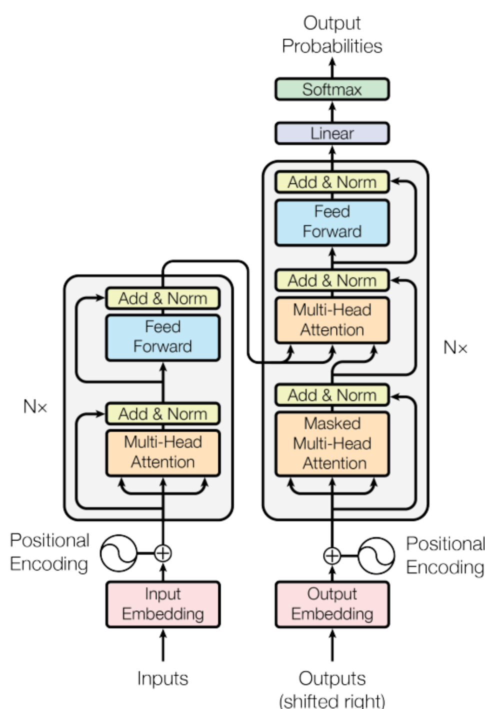



Рисунок 4.2.1 – Оригінальна структура трансформеру [48]

Як згадувалося раніше, трансформер приймає на вхід послідовності. В оригінальному трансформері послідовності представляли собою речення зі слів. Ці послідовності потрібно було розбити на набір токенів, тобто слів, які потім вбудовуються у вектори розмірності d за допомогою вивчених вбудовувань.

Потім у вектори додається позиційна інформація, яка дозволяє моделі використовувати порядок послідовності.

В оригінальному трансформері це позиційне кодування базується на синусі та косинусі, використовуючи формули, показані в рівняннях (4.1, 4.2), де p представляє позицію, а i - розмірність. Позиційне вбудовування має ту саму розмірність d, що й вбудовування, так що їх можна безпосередньо підсумувати.

$$Pos(p,2i) = sin(\frac{p}{10000^{2i/d}}) \quad (4.1)$$

$$Pos(p,2i+1) = cos(\frac{p}{10000^{2i/d}}) \quad (4.2)$$

Основний будівельний блок трансформера базується на механізмі, відомому як самоувага (self-attention). Увагу можна інтерпретувати як міру подібності або кореляції між елементами вхідної послідовності. Оригінальний трансформер використовував точкову увагу до продукту (dot-product attention), яка функціонує шляхом відображення запиту Q і пар ключ-значення K, V на вихід шляхом зваженого підсумовування. Ваги обчислюються як функція кореляції між запитом і відповідним ключем. Значення Q, K, V створюються матричним множенням між вхідним вкладом X з трьома різними ваговими матрицями WQ, $W_K$, $W_V$. Потім значення уваги обчислюються за допомогою рівняння 4.3 $\sqrt{d_k}$ додається як масштабний коефіцієнт, щоб запобігти збільшенню векторів.

$$\text{Attention}(Q,K,V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad (4.3)$$

Оператор softmax визначено у рівнянні 4.4, де exp() є стандартною експоненціальною функцією ex. Його можна інтерпретувати як масштабування вхідних даних у розподілі ймовірностей.

$$\text{Softmax}(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} \quad (4.4)$$

Обчислена матриця softmax ($\frac{QK^T}{\sqrt{d_k}}$) може бути інтерпретована як частина вбудовування вхідних даних, на яку слід звернути увагу, в той час як матриця V

представляє значення, які ми хочемо отримати на виході. Лінійна комбінація цих двох матриць дає кінцевий результат.

Можна розширити цю ідею, вводячи багатоголову увагу (multi-head attention). Замість того, щоб використовувати одне обчислення уваги з використанням Q, K, V розмірності d, можна обчислити h різних значень уваги з різними Qi , Ki , Vi розмірності $d_q$, $d_k$, $d_v$ . Результат кожного обчислення $Z_i$ називається головою. Вони не залежать одна від одної і тому можуть обчислюватися паралельно. Голови об'єднуютьсям (конкатенуються) і трансформуються за допомогою вагової матриці $W^O$ розміром $d$ $x$ $d$, де $d = h * d_k$. Процедура цього розрахунку наведена в рівняннях нижче.

$$Q_i = XW^{Qi} \tag{4.5}$$

$$K_i = XW^{Ki} \tag{4.6}$$

$$V_i = XW^{Vi} \tag{4.7}$$

$$Z_i = \text{Attention}(Q_i\ ,\ K_i\ ,\ V_i),\ i = 1\ ...\ \text{h} \tag{4.8}$$

$$\text{MultiHead}(Q,\ K,\ V) = \text{Concat}(Z_1,\ Z_2,\ .\ .\ .\ ,\ Z_h)W^O \tag{4.9}$$

Мотивація введення багатоголової уваги полягала в наступному: "Багатоголова увага дозволяє моделі спільно приділяти увагу інформації з різних підпросторів представлення з різних позицій. З однією головою уваги усереднення перешкоджає цьому" [48].

У декодері обчислення уваги першого рівня доповнюються маскуванням (masking). Вкладення, які були згенеровані до цього часу, маскуються маскувальною матрицею M, що складається з нулів і нескінченно малих значень для значень, які мають бути замасковані. Це призводить до усунення кореляції між замаскованими запитами та значеннями. Механізм маскованої уваги описано у рівнянні 4.10.

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}(\frac{QK^T + M}{\sqrt{d_k}})V \tag{4.10}$$

За кожним шаром трансформера слідує шар, позначений на рисунку 4.2.1 як "Додати і нормалізувати" (Add & Norm). Цей шар, по суті, додає залишкові зв'язки навколо попереднього підшару, а також нормалізує шар. Нормалізація шару

функціонує майже так само, як і пакетна нормалізація. Однак середнє значення і дисперсія обчислюються для каналів і векторів.

Серія повністю з'єднаних шарів зазвичай слідує за послідовністю кодер-декодер, щоб функціонувати як класифікаційна або регресійна голова.

### 4.2.1 Архітектура трансформери зору

Поява трансформера зору (vision transformer) викликала інтерес до трансформерів для виконання задач комп'ютерного зору.

Структура трансформера зору подібна до структури оригінального трансформера. Однак, у трансформері зору використовується лише кодер трансформера без декодера. Таким чином, він лише створює карту об'єктів, а не декодує її. Це робиться шляхом вставки голови на кінці трансформерного кодера. Завдяки цьому трансформер зору є гнучким і здатним створювати карти об'єктів для цілого ряду різних архітектур. Додатково, замість того, щоб перетворювати речення на послідовність слів, він розбиває зображення на послідовність ділянок. Ці фрагменти сплющуються і вбудовуються в d-вимірний простір за допомогою лінійної проекції, що навчається. Вони вводяться з позиційними кодами, які використовуються як вхідні дані для кодера.

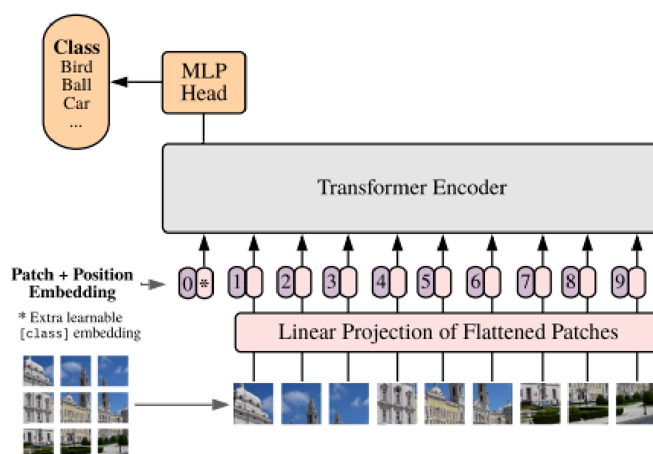



Рисунок 4.2.1.1 – Архітектура трансформера зору [42]

Всі шари прямого зв'язку трансформера зору, які називаються багатошаровими перцептронами (multilayer perceptrons / MLP), використовують

функцію активації, відому як лінійна одиниця гауссової похибки або GELU (Gaussian error linear unit).

Лінійна одиниця Гауссової похибки (GELU) – це функція активації, введена Хендриксом і Гімпелом [50], яка зазвичай використовується в більшості архітектур трансформерного типу. Це варіація ReLU, яка відрізняється від неї тим, що вентилює входи за їх значенням, а не за знаком. Оригінальні автори стверджують, що "підвищена кривизна і немонотонність можуть дозволити GELU легше апроксимувати складні функції, ніж ReLU [50].

Вона визначається рівнянням (16), де $\Phi$ позначає кумулятивну функцію розподілу для Гаусівського розподілу.

$$f(x) = \Phi(x)x \ (2.24) \qquad (4.11)$$

Стандартний трансформер зору здебільшого використовується в галузі класифікації зображень. Підвищена роздільна здатність, необхідна для аналізу зображень в порівнянні з перекладом тексту, виявилася проблемою при застосуванні трансформерів до області зображень.

Використання традиційних трансформерів зору для обробки зображень має квадратичну складність обчислень відносно розміру зображення через глобальні обчислення самоуваги. Одним з підходів до більш ефективного вилучення ознак із зображень є Swin трансформер [51].

Архітектуру Swin трансформеру буде більше детально розглянуто далі по тексту.

### 4.2.2 Архітектура Swin трансформеру (V1)

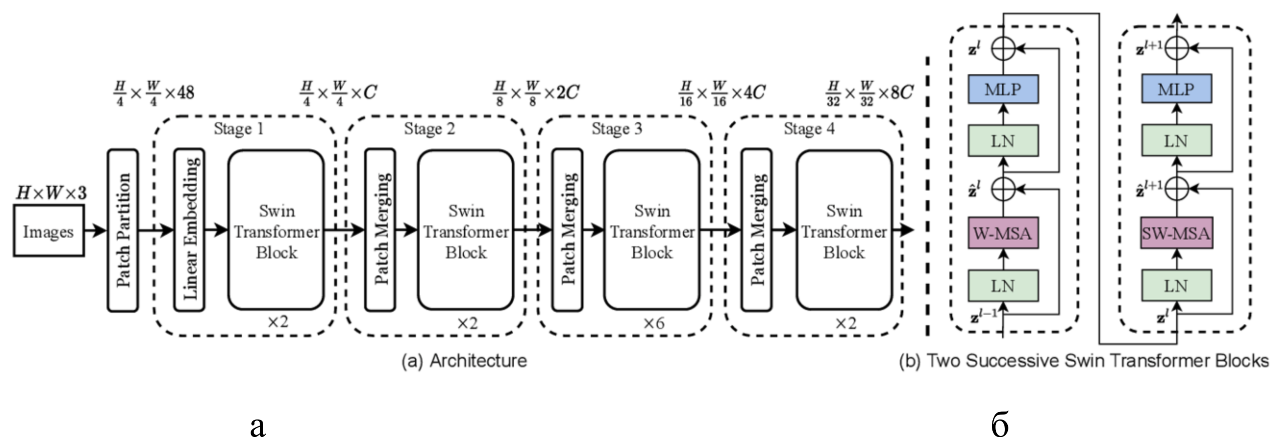
Swin Трансформер [51] — це трансформерна модель глибинного навчання з найсучаснішою продуктивністю в задачах зору. На відміну від трансформеру зору (ViT) [42], який йому передує, Swin Трансформер є високоефективним і має більшу точність. Завдяки цим бажаним властивостям Swin Трансформерs використовуються як основа в багатьох моделях сьогодні.

Swin трансформер є універсальною основою для задач комп'ютерного зору і демонструє високу точність у задачах виявлення об'єктів, таких як задача

виявлення COCO. Він використовує модифіковану версію MSA трансформера, яка називається віконною багатоголовою самоувагою (window based multi-head self-attention / W-MSA). Концепція W-MSA полягає в обчисленні уваги лише в локальних вікнах, а потім - в переміщенні вікон між шарами уваги. Він також об'єднує фрагменти зображення в глибоких шарах. Це дозволяє Swin трансформеру обчислювати увагу ієрархічно, без лінійної складності щодо розміру вхідних даних. Swin трансформер використовує GELU як функції активації.

Swin трансформер представив дві ключові концепції для вирішення проблем, з якими зіткнувся оригінальний трансформер зору – ієрархічні карти функцій і зміщене вікно уваги. Назва Swin трансформер походить від «Transformer Shifted Window».

Загальна архітектура Swin трансформеру показана далі на рис. 4.2.2.1.




а б

Рисунок 4.2.2.1 – Загальна архітектура Swin трансформеру [51]:

а – Архітектура Swin трансформеру (Swin-T); б – два послідовних блоки Swin трансформерів

Перше значне відхилення від трансормеру зору полягає в тому, що Swin Трансформер створює «ієрархічні карти функцій».

«Карти функцій» – це просто проміжні тензори, створені з кожного наступного шару. Під «ієрархічними» мається на увазі , що карти об’єктів об’єднуються від шару до шару, що фактично зменшує просторовий вимір (тобто

відбувається зменшення дискретизації) карт об'єктів з одного шару. до іншого (рис. 4.2.2.2).

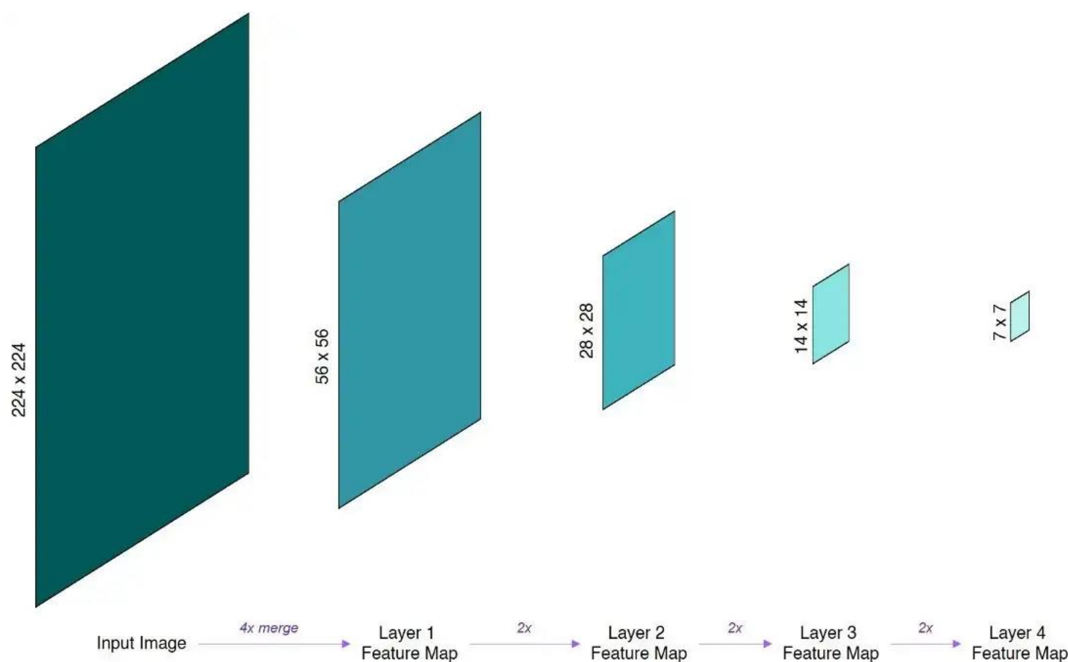



Рисунок 4.2.2.2 – Карти ієрархічних функцій у Swin трансформері (глибина карти функцій опущена з метою спрощення) [52]

Як зазначено на рис. 4.2.2.2 карти функцій поступово об'єднуються та зменшується після кожного шару, створюючи карти функцій з ієрархічною структурою. Просторова роздільна здатність цих ієрархічних карт функцій ідентична роздільній здатності в ResNet. Це було зроблено навмисно, щоб Swin трансформери можна було зручно замінити на мережі ResNet в існуючих методах для завдань комп'ютерного зору. Ці ієрархічні карти функцій дозволяють застосовувати Swin трансформер у тих областях, де потрібне детальне передбачення, наприклад у семантичній сегментації. Навпаки, трансформер зору використовує карти функцій єдиної низької роздільної здатності в усій своїй архітектурі [52].

Техніка зменшення дискретизації без згортки, яка використовується в Swin трансформер, відома як злиття патчів.

Трансформерний блок, який використовується в Swin трансформері, замінює стандартний багатоголовий модуль самоуваги (multi-head self-attention / MSA), який використовується у трансформері зору, на модуль MSA з вікном (window-

based multi-head self-attention / W-MSA) і модуль MSA зі зміщеним вікном (SW-MSA). Блок Swin трансформеру зображено на рис. 4.2.2.3.

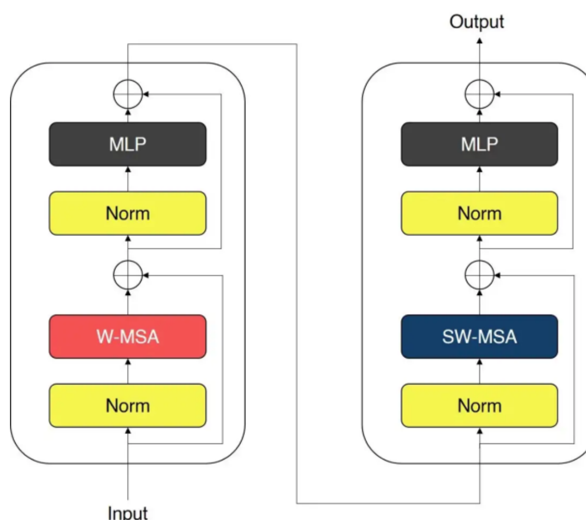



Рисунок 4.2.2.3 – Блок Swin Трансформер, з 2 під-блоками (W-MSA та SW-MSA) [52]

Блок Swin трансформеру складається з двох вузлів. Кожен підблок складається з рівня нормалізації, за яким слідує модуль уваги, за яким слідує інший рівень нормалізації та рівень MLP. Перший підрозділ використовує модуль Window MSA (WMSA), а другий підрозділ використовує модуль Shifted Window MSA (SW-MSA).

Стандартний MSA, який використовується у трансформері зору, виконує глобальний самоконтроль, а зв’язок між кожним патчем обчислюється з усіма іншими патчами. Це призводить до квадратичної складності щодо кількості патчів, що робить його непридатним для зображень з високою роздільною здатністю.

Щоб вирішити цю проблему, Swin трансформер використовує віконний підхід MSA.

Вікно — це просто набір патчів, і увага обчислюється лише в межах кожного вікна. Оскільки розмір вікна фіксований у всій мережі, складність вікна залежить від його складності. MSA є лінійним щодо кількості патчів (тобто розміру зображення), що значно покращує квадратичну складність стандартного MSA.

Однак, одним очевидним недоліком віконного MSA є те, що він обмежує самоувагу до кожного вікна, обмежуючи потужність моделювання мережі. Щоб вирішити цю проблему, Swin трансформер використовує модуль MSA зі зміщеним вікном (shifted window MSA / SW-MSA) після модуля W-MSA. Щоб запровадити міжвіконні з'єднання, SW-MSA зміщує вікна до нижнього правого кута з коефіцієнтом M/2, де M — розмір вікна.

Однак, це зміщення призводить до «загублених» патчів, які не належать жодному вікну, а також до вікон із неповними патчами. Swin трансформер застосовує техніку «циклічного зсуву», яка переміщує «загублені» патчі у вікна з неповними патчами. Після цього зсуву вікно може складатися з ділянок, які не є суміжними на вихідній карті об'єктів, тому під час обчислення застосовується маска, щоб обмежити увагу до суміжних ділянок.

Такий підхід зі зміщеним вікном вводить важливі перехресні зв'язки між вікнами та покращує продуктивність мережі.

Використовуючи ієрархічні карти функцій і зміщене вікно MSA, Swin трансформер вирішив проблеми, які були недоліком оригінального трансформеру зору. Як результат, Swin трансформер зазвичай використовується як базова архітектура в широкому діапазоні завдань зору, включаючи класифікацію зображень і виявлення об'єктів.

#### 4.2.3 Архітектура Swin трансформеру (V2)

У 2021 році з'явилась друга версія Swin трансформеру [53]. Першу версію було масштабовано до 3 мільярдів параметрів, що є найбільшою та найефективнішою моделлю щільного бачення на сьогодні. Крім того, адаптована версія використовує в 40 разів менше мічених даних і в 40 разів менше часу на навчання, ніж попередні роботи.

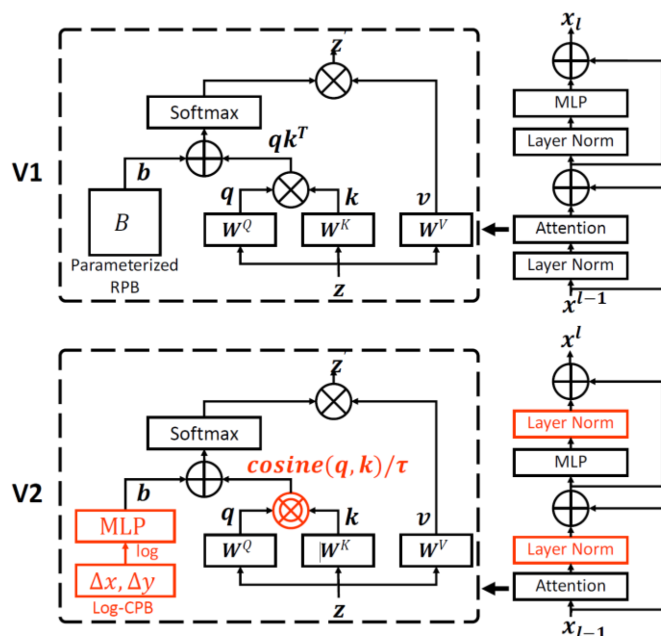



Рисунок 4.2.3.1 – Адаптації в архітектурі оригінального Swin трансформеру (виділені червоним) [53]

Для кращого збільшення ємності моделі та роздільної здатності вікна в оригінальну архітектуру Swin Трансформер (V1) внесено декілька адаптацій, а саме:

1) res-post-norm для заміни попередньої конфігурації pre-norm;

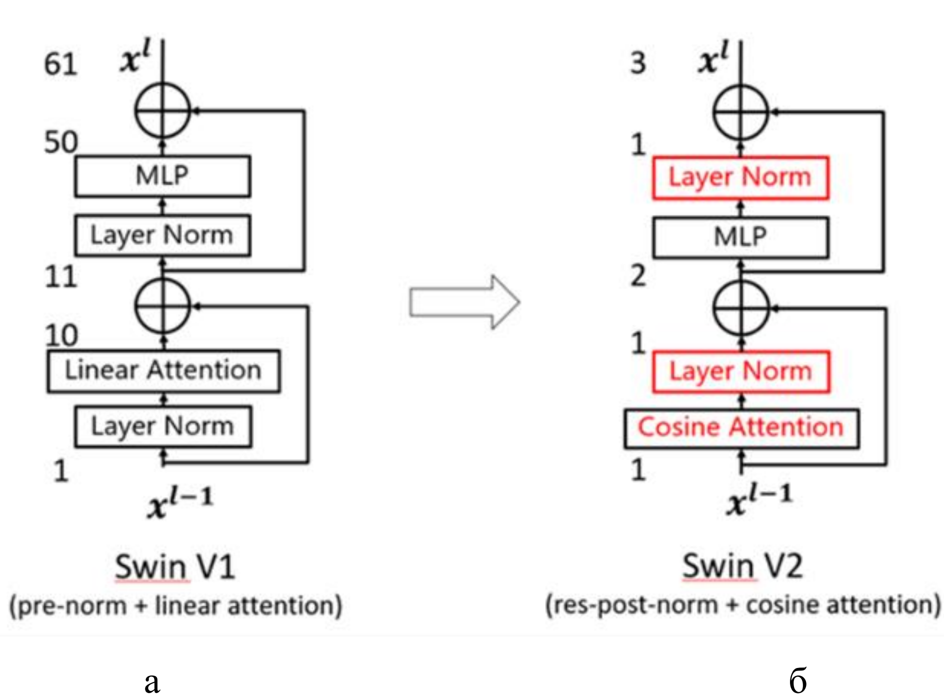
2) масштабована косинусна увага для заміни початкової уваги до скалярного добутку;

3) підхід безперервного зсуву відносного положення з логарифмічним інтервалом замінює попередній параметризований підхід.

Перші дві адаптації спрощують збільшення ємності моделі. Тоді як третя адаптація забезпечує більш ефективне перенесення моделі між роздільними здатностями вікон. Адаптована архітектура отримала назву Swin Трансформер V2 [53].

Як показано на рис. 4.2.3.2, після одного блоку трансформера значення характеристик на виході можуть збільшитися в 61 разів більше, ніж на вході. Щоб пом'якшити цю проблему,пропонуємо новий метод нормалізації під назвою залишкова пост-нормалізація. Як показано на рисунку 2 (праворуч), цей метод переміщує рівень нормалізації від початку до кінця кожної залишкової гілки, щоб вихідні дані кожної залишкової гілки нормалізувалися перед об'єднанням назад

у основну гілку. Таким чином, середня дисперсія ознак головної гілки не збільшується суттєво в міру поглиблення шарів. Експерименти показали, що цей новий метод нормалізації пом'якшує середню дисперсію ознак кожного шару в моделі.




а б

Рисунок 4.2.3.2 – Для розміщення більших моделей бачення конфігурацію нормалізації оригінального Swin трансформер було скориговано [53]: а – Swin V1; б – Swin V2

Як показано на рис. 4.2.3.2 (а), оригінальна архітектура використовує конфігурацію попередньої норми, у якій нормалізація відбувається на початку кожної залишкової гілки. Ця конфігурація призводить до збільшення значень функцій (від 1 до 61), що призводить до збою під час навчання. На рис. 4.2.3.2 (б) бачимо дві зміни, що було внесено до Swin V2:

1) нормалізацію переміщено в кінець залишкової гілки в новому методі під назвою residual-post-normalization, який бачить набагато м'якше збільшення значення (від 1 до 3);

2) лінійна функція скалярного добутку в одиниці уваги замінюється функцією косинуса.

Крім того, було виявлено, що, коли модель стає більшою, ваги уваги певних шарів мають тенденцію домінувати кількома конкретними точками в

початковому обчисленні самоуваги, особливо коли використовується залишкова пост-нормалізація. Щоб вирішити цю проблему, був запропонований механізм масштабованої косинусної уваги - для заміни звичайного скалярного добутку лінійної одиниці уваги. У новій масштабованій косинусній одиниці уваги обчислення самоуваги не залежить від вхідної величини, що призводить до менш насичених ваг уваги. Експерименти показали, що остаточна пост-нормалізація та механізм масштабованого косинусного фокусування не тільки стабілізують динаміку навчання великих моделей, але й підвищують точність.

Використовуючи log-spaced CPB, Swin трансформер V2 забезпечує плавний перехід між різними роздільними здатностями, дозволяючи використовувати меншу роздільну здатність зображення — 192 × 192 — без втрати точності в наступних завданнях порівняно зі стандартною роздільною здатністю 224 × 224, яка використовується під час попереднього навчання. Це прискорює навчання на 50 відсотків.

Масштабування ємності та роздільної здатності моделі призводить до надмірного споживання пам’яті GPU для існуючих моделей зору. Щоб вирішити проблему з пам’яттю, було об’єднано кілька важливих методів, у тому числі Zero-Redundancy Optimizer (ZeRO), контрольні точки активації та нову послідовну реалізацію самоконтролю. Як результат зазначених адаптацій, друга версія Swin трансформеру отримала методи масштабування до 3 мільярдів параметрів і можливості тренуватися із зображеннями з роздільною здатністю до 1536 × 1536 пікселів, включаючи res-post-norm і масштабований косинус (щоб полегшити масштабування моделі за ємністю), а також підхід безперервного зміщення відносного положення з логарифмічним інтервалом, який дозволяє ефективніше передавати модель між роздільними здатностями вікон [53].

Завдяки отриманим результатам [53], друга версія Swin трансформеру мотивує до додаткових досліджень.

### 4.3 Гібридна нейронна мережа CoAtNet

У дослідженні «CoAtNet: Поєднання згортки та уваги для всіх розмірів даних» [54] колектив авторів показав рішення проблеми гібридизації згортки та уваги з точки зору двох фундаментальних аспектів машинного навчання - узагальнення та потужності моделі. Назва CoAtNet походить від об'єднання слів Convolution та self-Attention.

Ця гібридна модель, яка поєднала ознаки згорткової мережі та трансформеру була, відповідно, названа CoAtNet і виділена у нове сімейство моделей.

Дослідження CoAtNet показує, що згорткові шари мають тенденцію до кращого узагальнення з більшою швидкістю збіжності завдяки значному попередньому індуктивному зміщенню, тоді як шари уваги мають вищу пропускну здатність моделі, що може отримати вигоду від навчання на великих наборах даних.

Ідея поєднання згортки та самоуваги для розпізнавання не нова. Хоча самоувага зазвичай покращує точність, вона часто пов'язана з додатковими обчислювальними витратами і тому часто розглядається як доповнення до згорткових мереж, подібно до модуля стиснення і збудження (squeeze-and-excitation) [55].

Інший популярний напрямок досліджень починається з кістяка (стовбура) трансформеру і намагається включити явну згортку або деякі бажані властивості згортки в цей кістяк [56, 57].

Зазначена робота також належить до цієї категорії досліджень, де показано, що відносна конкретизація уваги є природною сумішшю глибинної згортки та уваги, заснованої на змісті, з мінімальними додатковими витратами. Що ще важливіше, виходячи з перспективи узагальнення та ємності моделі, автори застосували системний підхід до проектування вертикальних шарів і показали, як і чому різні етапи мережі надають перевагу різним типам шарів. Таким чином, порівняно з моделями, які просто використовують готову згорткову мережу як стовбуровий шар (такими як ResNet-ViT [42]), CoAtNet також масштабує етап згортки (S2), коли загальний розмір збільшується. З іншого боку, порівняно з

моделями, що використовують локальну увагу [51, 58], CoAtNet послідовно використовує повну увагу для S3 та S4, щоб забезпечити пропускну здатність моделі, оскільки S3 займає більшу частину обчислень та параметрів.

Поєднання згорткових шарів та шарів уваги дозволяє досягти кращої узагальненості та пропускної здатності; однак ключовим викликом в зазначеній роботі було те, як ефективно поєднати ці ознаки, щоб досягти кращого компромісу між точністю та ефективністю. В зазначеній роботі було розглянуто два ключові висновки дослідження:

– широко використовувана згортка в глибину може бути ефективно об'єднана з шарами уваги за допомогою простої відносної уваги;

– просте накладання шарів згортки та уваги, при правильному підході, може бути напрочуд ефективним для досягнення кращого узагальнення та пропускної спроможності.

Ґрунтуючись на цих висновках, колективом авторів була запропонована проста, але ефективна мережева архітектура під назвою CoAtNet, яка має сильні сторони як згорткових мереж, так і трансформерів.

# РОЗДІЛ 5
# РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ РОЗПІЗНАВАННЯ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 5.1 Набір даних Malimg

На наступних зображеннях зловмисного програмного забезпечення можна виявити невеликі зміни (рис. 5.1.1.). В той саме час, у зразків, що належать до одного сімейства, загальна структура зберігається. Однак, вони візуально відрізняються від зразків зловмисних програм інших сімейств.

Рисунок 5.1.1 – Зразки зображень шкідливого програмного забезпечення по різних класах в наборі даних Malimg

Для реалізації завдань даної роботи було використано загальнодоступний набір даних Malimg, розміщенй на інтернет платформі Kaggle [59]. Даний набір даних містить 9340 зображень байт-плотів шкідливих програм з 25 різних сімейств.

Таблиця 5.1 – Перелік класів шкідливого програмного забезпечення в наборі даних Malimg

| № | Ярлик класу |
|---|---|
| 0 | Adialer.C |
| 1 | Agent.FYI |
| 2 | Allaple.A |
| 3 | Allaple.L |
| 4 | Alueron.gen!J |
| 5 | Autorun.K |
| 6 | C2LOP.P |
| 7 | C2LOP.gen!g |
| 8 | Dialplatform.B |
| 9 | Dontovo.A |
| 10 | Fakerean |
| 11 | Instantaccess |
| 12 | Lolyda.AA1 |
| 13 | Lolyda.AA2 |
| 14 | Lolyda.AA3 |
| 15 | Lolyda.AT |
| 16 | Malex.gen!J |
| 17 | Obfuscator.AD |
| 18 | Rbot!gen |
| 19 | Skintrim.N |
| 20 | Swizzor.gen!E |
| 21 | Swizzor.gen!I |
| 22 | VB.AT |
| 23 | Wintrim.BX |
| 24 | Yuner.A |

Оригінальний набір даних Malimg був створений в рамках проєкту з обробки сигналів для аналізу шкідливих програм на кафедрі електротехніки та комп'ютерної інженерії університету Каліфорнії (США).

Метою цього проєкту було дослідження методів обробки сигналів та зображень для аналізу шкідливого програмного забезпечення. Двійкові файли шкідливого програмного забезпечення були візуалізовані у вигляді зображень у сірій шкалі, при цьому спостерігалося, що для багатьох сімейств шкідливих програм зображення, які належать до одного сімейства, їх зображення виглядають дуже схожими за макетом і текстурою.

Як зазначалось раніше, більшість нових шкідливих програм є модифікаціями вже існуючих. Таким чином, варіанти такого ПЗ мають майже однаковий вміст.

Протягом реалізації вказаного проєкту було зроблено два основних спостереження:

1. Існує візуальна схожість у варіантах шкідливого програмного забезпечення в межах сімейств.
2. Існує візуальна несхожість між варіантами шкідливого програмного забезпечення різних сімейств.

Для подальшої роботи були використані ці візуальні подібності та відмінності і запропоновано функції, засновані на схожості зображень, для вирішення проблем класифікації, виявлення, пошуку шкідливого програмного забезпечення та інших завдань.

Одним із результатів проєкту в 2011 році стала публікація статті "Зображення шкідливого програмного забезпечення: Візуалізація та автоматична класифікація" [7], яка згадувалася раніше в цій роботі. Разом з цим, набір даних Malimg використовується вже більше десяти років в різних дослідженнях з розпізнавання шкідливого програмного забезпечення.

<BarContainer object of 25 artists>

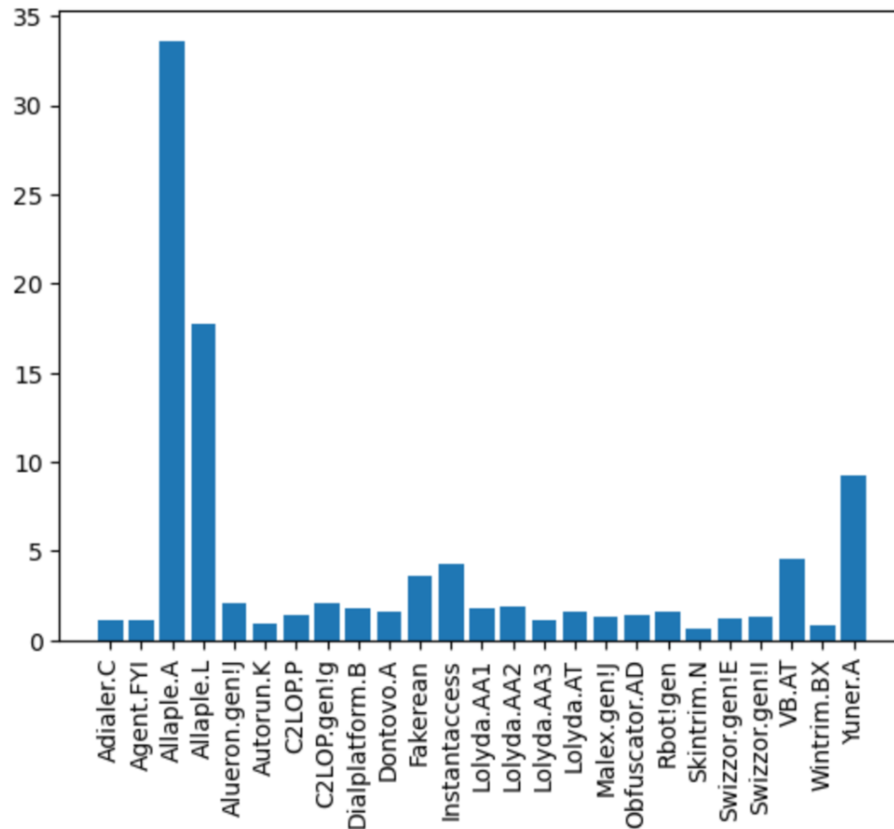



Рисунок 5.1.2 – Кількість зразків зображень по класах
в наборі даних Malimg

Загальна точність класифікації в даній роботі, досягнута CNN для набору даних із тими самими 25 сімействами зловмисного програмного забезпечення, була вищою, ніж в оригінальному підході Nataraj [7], досягнувши 98,33 відсотка точності.

## 5.2 Опис програмного забезпечення

Проблема ідентифікації та розпізнавання шкідливих програм є дуже складним завданням, яке не має ідеального рішення. Для її вирішення використовуються різні фреймворки з явними перевагами в конкретних підобластях глибинного навчання [60].

В даній роботі використовується мова програмування Python та Keras фреймворк.

Keras – це фреймворк нейронної мережі з відкритим кодом, створений для швидкого створення прототипів моделей глибинного навчання. Він був

розроблений саме для масштабування. Це високорівневий API нейронних мереж, написаний на Python. Він був створений на основі двох провідних фреймворків глибинного навчання Tensorflow і Theano (який зараз вже не підтримується). Відповідно, в даній роботі використовується Keras із бекендом Tensorflow. Keras полегшує реалізацію API, багаторівневого персептрона, згорткових нейронних мереж (1D, 2D, 3D), комірок довгої короткострокової пам'яті (LSTM).

Він також підтримує різні функції втрати, оптимізатори та функції активації, такі як Sigmoid, LeakyReLU, ReLU, Softmax, а також, Swin Transformer, які були описані раніше в даній роботі.

Побудований на основі TensorFlow з акцентом на взаємодії з користувачем, цей фреймворк легко масштабується завдяки паралелізму даних і, отже, може обробляти величезні обсяги даних, прискорюючи час навчання моделей [60]. Він вважається одним із найпопулярніших фреймворків, який пропонує послідовні та прості API, одночасно зменшуючи кількість дій користувача, необхідних для типових випадків використання. При цьому, Keras надає чіткі та дієві повідомлення про помилки.

TensorFlow – це платформа машинного навчання з відкритим вихідним кодом, написана на Python на основі механізму C/C++, що забезпечує високу продуктивність із великими наборами даних, була розроблена командою Google Brain Team [60]. Як результат, він добре підходить для розробки архітектур глибинного навчання та експериментування з моделями. TensorFlow отримав широке поширення, здебільшого тому, що він має Python API, який було покращено у другій версії.

Scikit-learn (Sklearn) – це бібліотека машинного навчання для мови програмування Python, яка надає прості та ефективні методи реалізації прогнозів, такі як статистичне моделювання, класифікація, регресія, кластеризація та зменшення розмірності [61]. Крім того, він включає в себе реалізацію численних традиційних методів машинного навчання. Цей пакет, написаний переважно на Python, базується на NumPy, SciPy та Matplotlib.

NumPy і SciPy – це бібліотеки, які разом забезпечують функціональність, подібну до MATLAB, у Python. Це програмне забезпечення з відкритим кодом, ліцензоване за ліцензією BSD [PVG+11]. Scikit-learn спочатку був започаткований Девідом Курнапо в 2007 році як проект Google Summer of Code. У 2010 році члени французького дослідницького інституту INRIA 16 взяли на себе керівництво та зробили перший публічний випуск. Scikit-learn безперервно розвивається, має надійну та швидку реалізацію і видатну документацію реалізованих алгоритмів. Бібліотека SciPy залежить від NumPy, яка забезпечує зручне та швидке маніпулювання n-вимірним масивом. Він надає класи для неконтрольованих і контрольованих підходів машинного навчання, таких як описані k-середні, дерева рішень і випадкові ліси. Він також забезпечує ефективні структури даних для оцінки та вдосконалення моделі.

### 5.3 Результати машинного навчання

В процесі реалізації інформаційної технології розпізнавання шкідливого програмного забезпечення на наборі даних Malimg було проведено навчання чотирьох моделей машинного навчання, а саме:

– згорткова нейронна мережа;

– трансформер Swin v.1;

– трансформер Swin v.2;

– гібридна згорткова нейронна мережа CoAtNet.

Для цілей навчання моделі оригінальний набір даних був розподілений наступним чином:

– тренінговий набір даних – 70% від оригінального набору даних;

– тестовий набір даних – 30% від оригінального набору даних: cx_train, x_test, y_train, y_test = train_test_split(imgs/255.,labels, test_size=0.3);

– валідаційний набір даних – 10% від тренінгового набору даних.

Кожна з чотирьох моделей була навчена і оцінена за умов, описаних нижче:

– train_loss / val_loss- перехресна ентропія використовувалася як функція втрат для моделей під час навчання;

– val_acc – акуратність (   ) на валідаційному наборі даних;

– train_acc – акуратність (точність класифікації) на тренінговому наборі даних;

– техніка динамічного зменшення швидкості навчання моделі глибинного навчання на плато, що використовується для підвищення точності класифікації за допомогою результуючої моделі: lr_reduction = ReduceLROnPlateau(monitor='val_accuracy',patience=4, verbose=1, factor=0.4, min_lr=0.0001);

– дострокова зупинка навчання моделі при досягненні порогу приросту акуратності (точності пролгнозування), що використовується як для усунення пастки перенавчання, так і для оптимізації часу навчання моделі: early_stop = EarlyStopping(monitor='val_accuracy', min_delta=0.00001, patience=8, mode='auto', restore_best_weights=True).

Отримані результати машинного навчання наведені далі в роботі.

#### 5.3.1 Результати навчання згорткової нейронної мережі

Отримані за результатами навчання згорткової нейронної мережі метрики:

– акуратність (accuracy): 0.9833;

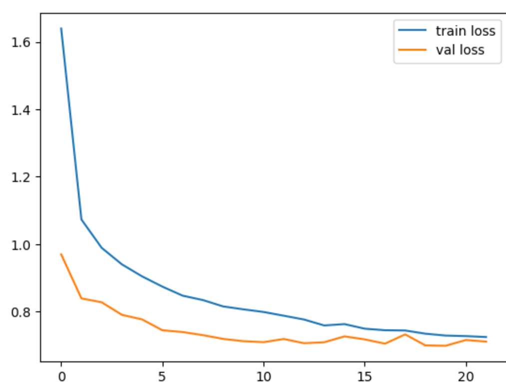
– top-5-accuracy: 0.9996;

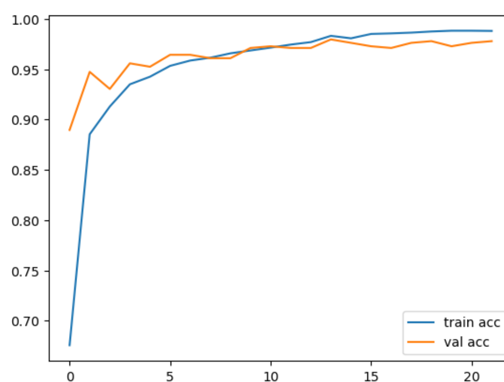
– втрати (loss): 0.7039;

– кількість епох: 22.

Як бачимо з наведених вище даних, використана модель продемонструвала непогану ефективність. Так, на тестовій виборці даних акуратність (точність класифікації) визначення становила 98,33%. При цьому зразки п’яти найбільш поширених видів шкідливого програмного забезпечення визначалися з точністю 99,96%.

В процесі тренування моделі спостерігалися зміни цільових метрик функції втрат (loss) та акуратності (accuracy), відображені на рис. 5.3.1.1.




а б

Рисунок 5.3.1.1 – Результати тренування моделі CNN в залежності від кількості епох: а – графік втрат; б – графік акуратності

Так, по осі X зазначена кількість епох навчання моделі, а по осі Y, відповідно, - зміни цільових метрик (loss, accuracy). Збіжність по втратах спостерігалась тільки після 20 епох. Після 22 епох ми бачимо, що досягли збіжності функції втрат на навчальній та валідаційній вибірках даних. Таким чином, за результатами навчання була отримана модель, що є гнучкою і адекватною при роботі з даними, які не були задіяні як частина навчального набору даних. Це підтверджується значеннями акуратності на рис. 5.3.1.1 (частина б) – розбіжність між значеннями для навчального та валідаційного наборів даних є візуально незначною.

В процесі прогнозування була сформована наступна матриця помилок (рис. 5.3.1.2).

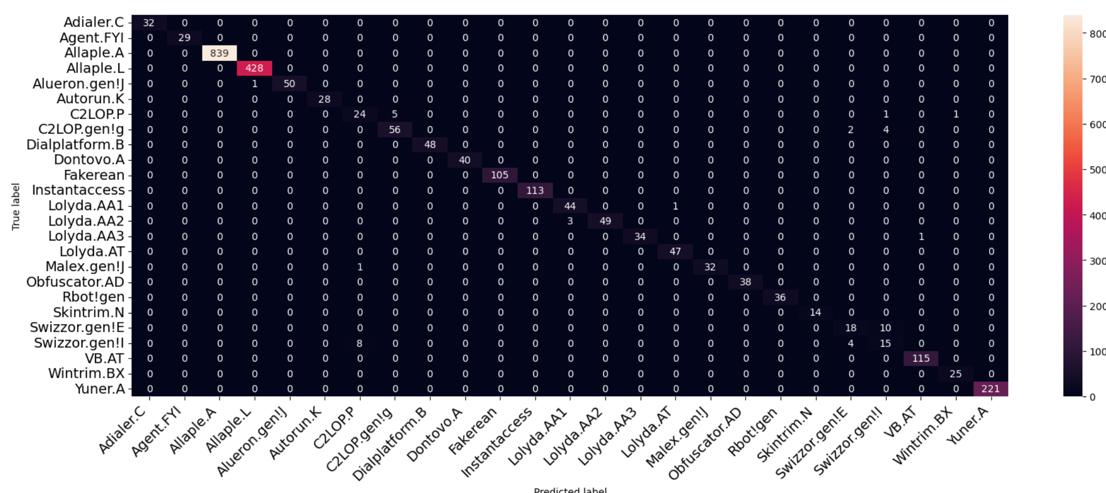



Рисунок 5.3.1.2 – Результати тренування моделі CNN – матриця помилок (confusion matrix)

З матриці помилок ми бачимо, що більшість типів шкідливого програмного класифікується коректно (наприклад, найкращі результати спостерігаються при визначенні класів Allaple.A, Allaple.L). В той же час, спостерігається, що модель досить часто помилково визначає класи шкідливого програмного забезпечення Swizzor.gen!E та Swizzor.gen!I.

Як бачимо з табл. 5.3.1.1 найгірше розпізнаються класи шкідливого програмного забезпечення Swizzor.gen!I (50%), Swizzor.gen!E (75%) та C2LOP.P (72,7%). При цьому точність класифікації по всім іншим класам є доволі високою (більше 90%).

Таблиця 5.3.1.1 – Метрики по класам ШПЗ для CNN

| № | class_label | precision | recall | f1-score | support |
|---|---|---|---|---|---|
| 0 | Adialer.C | 1 | 1 | 1 | 32 |
| 1 | Agent.FYI | 1 | 1 | 1 | 29 |
| 2 | Allaple.A | 1 | 1 | 1 | 839 |
| 3 | Allaple.L | 0.998 | 1 | 0.999 | 428 |
| 4 | Alueron.gen!J | 1 | 0.98 | 0.99 | 51 |
| 5 | Autorun.K | 1 | 1 | 1 | 28 |
| 6 | C2LOP.P | 0.727 | 0.774 | 0.75 | 31 |
| 7 | C2LOP.gen!g | 0.918 | 0.903 | 0.911 | 62 |
| 8 | Dialplatform.B | 1 | 1 | 1 | 48 |
| 9 | Dontovo.A | 1 | 1 | 1 | 40 |
| 10 | Fakerean | 1 | 1 | 1 | 105 |
| 11 | Instantaccess | 1 | 1 | 1 | 113 |
| 12 | Lolyda.AA1 | 0.936 | 0.978 | 0.957 | 45 |
| 13 | Lolyda.AA2 | 1 | 0.942 | 0.97 | 52 |
| 14 | Lolyda.AA3 | 1 | 0.971 | 0.986 | 35 |
| 15 | Lolyda.AT | 0.979 | 1 | 0.989 | 47 |
| 16 | Malex.gen!J | 1 | 0.97 | 0.985 | 33 |
| 17 | Obfuscator.AD | 1 | 1 | 1 | 38 |

Продовження таблиці 5.3.1.1

| 18 | Rbot!gen | 1 | 1 | 1 | 36 |
|---|---|---|---|---|---|
| 19 | Skintrim.N | 1 | 1 | 1 | 14 |
| 20 | Swizzor.gen!E | 0.75 | 0.643 | 0.692 | 28 |
| 21 | Swizzor.gen!I | 0.5 | 0.556 | 0.526 | 27 |
| 22 | VB.AT | 0.991 | 1 | 0.996 | 115 |
| 23 | Wintrim.BX | 0.962 | 1 | 0.98 | 25 |
| 24 | Yuner.A | 1 | 1 | 1 | 221 |

Загальна кількість представників по всіх класах (support) становила 2522.

Метрики з табл. 5.3.1.2, що отримані з використанням бібліотеки Sklearn, підтверджують дуже гарні результати (accuracy>0.98) за результатами використання моделі згорткової нейронної мережі.

Таблиця 5.3.1.2 – Агреговані метрики CNN для набору даних

| № | class_label | precision | recall | f1-score |
|---|---|---|---|---|
| 1 | accuracy | 0.983347 | 0.983347 | 0.983347 |
| 2 | macro avg | 0.950449 | 0.948697 | 0.949224 |
| 3 | weighted avg | 0.983807 | 0.983347 | 0.983464 |

Таким чином, CNN продемонструвала високу ефективність при розпізнаванні класів шкідливого програмного забезпечення з використанням набору даних Malimg.

### 5.3.2 Результати навчання Swin трансформер (V1)

Отримані за результатами навчання нейронної мережі Swin трансформер (V1) метрики:

– акуратність (accuracy): 0.9611;

– top-5-accuracy: 0.9972;

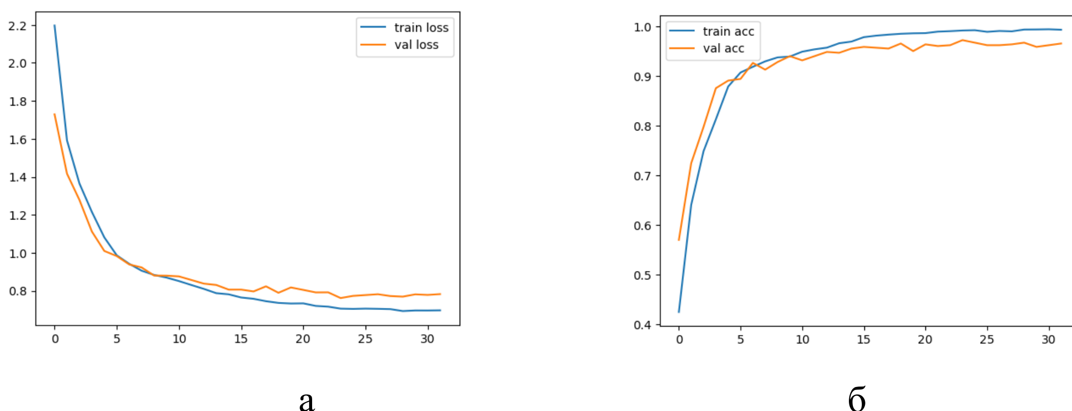
– втрати (loss): 0.7732;

– кількість епох: 32.

Як бачимо з наведених вище даних, використана модель продемонструвала непогану ефективність (але незначно гіршу, ніж CNN). Так, на тестовій виборці

даних акуратність (точність класифікації) визначення становила 96,11 (CNN - 98,33%). При цьому, зразки п'яти найбільш поширених видів шкідливого програмного забезпечення визначалися з точністю 99,72% (CNN - 99,96%).

В процесі тренування моделі спостерігалися зміни цільових метрик функції втрат (loss) та акуратності (accuracy), відображені на рис. 5.3.2.1.




а б

Рисунок 5.3.2.1 – Результати тренування моделі Swin трансформер v.1 в залежності від кількості епох: а – графік втрат; б – графік акуратності

Як бачимо на рис. 5.3.2.1 якість моделі Swin трансформера v.1 (збіжність починає розходитись) дещо гірша за модель згорткової нейронної мережі. Разом з цим, графік акуратності, також свідчить про достатньо непогану якість моделі Swin трансформера V1.

В процесі прогнозування була сформована наступна матриця помилок (рис. 5.3.2.2).

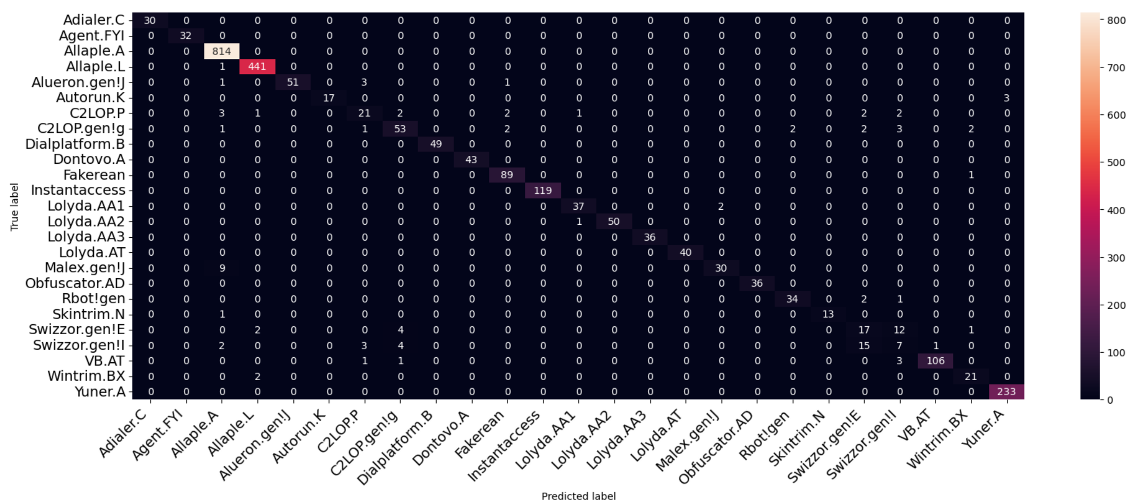



Рисунок 5.3.2.2 – Результати тренування моделі Swin трансформер v.1 - матриця помилок (confusion matrix)

З матриці помилок ми бачимо, що більшість типів шкідливого програмного класифікується коректно (наприклад, гарні результати спостерігаються при визначенні класів Allaple.A, Allaple.L). В той же час, спостерігається, що модель досить часто помилково визначає класи шкідливого програмного забезпечення Swizzor.gen!E та Swizzor.gen!I (так само як і у випадку з CNN).

Як бачимо з табл. 5.3.2.1 модель Swin трансформеру v.1 має дещо гірші результати, ніж CNN. Так, дуже погано розпізнаються класи шкідливого програмного забезпечення Swizzor.gen!I (25%), Swizzor.gen!E (44,7%) та C2LOP.P (72,4%). При цьому точність прогнозування по всім іншим класам є відносно високою (більше 84%).

Таблиця 5.3.2.1 – Метрики по класам для Swin трансформер v.1

| № | class_label | precision | recall | f1-score | support |
|---|---|---|---|---|---|
| 0 | Adialer.C | 1 | 1 | 1 | 30 |
| 1 | Agent.FYI | 1 | 1 | 1 | 32 |
| 2 | Allaple.A | 0.978 | 1 | 0.989 | 814 |
| 3 | Allaple.L | 0.989 | 0.998 | 0.993 | 442 |
| 4 | Alueron.gen!J | 1 | 0.911 | 0.953 | 56 |
| 5 | Autorun.K | 1 | 0.85 | 0.919 | 20 |
| 6 | C2LOP.P | 0.724 | 0.618 | 0.667 | 34 |
| 7 | C2LOP.gen!g | 0.828 | 0.803 | 0.815 | 66 |
| 8 | Dialplatform.B | 1 | 1 | 1 | 49 |
| 9 | Dontovo.A | 1 | 1 | 1 | 43 |
| 10 | Fakerean | 0.947 | 0.989 | 0.967 | 90 |
| 11 | Instantaccess | 1 | 1 | 1 | 119 |
| 12 | Lolyda.AA1 | 0.949 | 0.949 | 0.949 | 39 |
| 13 | Lolyda.AA2 | 1 | 0.98 | 0.99 | 51 |
| 14 | Lolyda.AA3 | 1 | 1 | 1 | 36 |
| 15 | Lolyda.AT | 1 | 1 | 1 | 40 |
| 16 | Malex.gen!J | 0.938 | 0.769 | 0.845 | 39 |

Продовження таблиці 5.3.2.1

| 17 | Obfuscator.AD | 1 | 1 | 1 | 36 |
|---|---|---|---|---|---|
| 18 | Rbot!gen | 0.944 | 0.919 | 0.932 | 37 |
| 19 | Skintrim.N | 1 | 0.929 | 0.963 | 14 |
| 20 | Swizzor.gen!E | 0.447 | 0.472 | 0.459 | 36 |
| 21 | Swizzor.gen!I | 0.25 | 0.219 | 0.233 | 32 |
| 22 | VB.AT | 0.991 | 0.955 | 0.972 | 111 |
| 23 | Wintrim.BX | 0.84 | 0.913 | 0.875 | 23 |
| 24 | Yuner.A | 0.987 | 1 | 0.994 | 233 |

Загальна кількість представників по всіх класах (support) становила 2522.

Метрики з табл. 5.3.2.2, що отримані з використанням бібліотеки Sklearn, підтверджують гарні результати (accuracy > 0.95) за результатами використання моделі Swin трансформеру v.1.

Таблиця 5.3.2.2 – Агреговані для набору даних метрики Swin трансформеру v.1

| № | class_label | precision | recall | f1-score |
|---|---|---|---|---|
| 1 | accuracy | 0.959159 | 0.959159 | 0.959159 |
| 2 | macro avg | 0.912488 | 0.890913 | 0.900647 |
| 3 | weighted avg | 0.957913 | 0.959159 | 0.958131 |

Таким чином, модель Swin трансформеру v.1 продемонструвала достатню ефективність при розпізнаванні класів шкідливого програмного забезпечення з використанням набору даних Malimg, але гіршу в порівнянні з CNN.

### 5.3.3 Результати навчання Swin трансформер (V2)

Отримані за результатами навчання нейронної мережі Swin трансформер (V2) метрики:

– акуратність (accuracy): 0.9754;

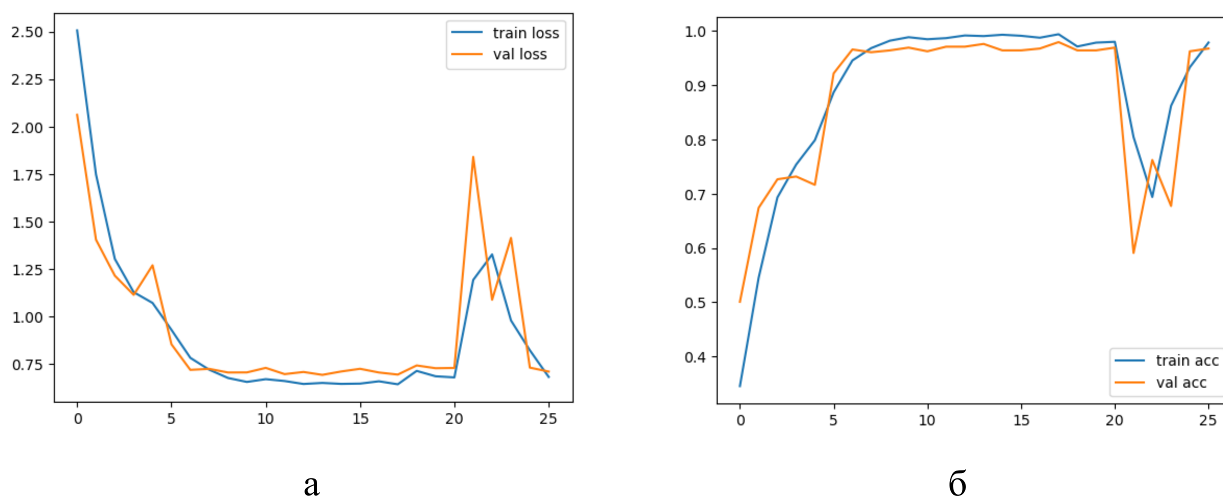
– top-5-accuracy: 0.9968;

– втрати (loss): 0.6920;

– кількість епох: 26.

Як бачимо з наведених вище даних, використана модель продемонструвала дуже непогану ефективність (кращу за Swin трансформер v.1, але незначно гіршу, ніж CNN). Так, на тестовій виборці даних акуратність (точність класифікації) визначення становила 97,54% (Swin1 - 96,11%, CNN - 98,33%). При цьому, зразки п'яти найбільш поширених видів шкідливого програмного забезпечення визначалися з точністю 99,68% (Swin1 - 99,72%, CNN - 99,96%).

В процесі тренування моделі спостерігалися зміни цільових метрик функції втрат (loss) та акуратності (accuracy), відображені на рис. 5.3.3.1.




а б

Рисунок 5.3.3.1 – Результати тренування моделі Swin трансформер v.2 в залежності від кількості епох: а – графік втрат; б – графік акуратності

Як можна побачити на рис. 5.3.3.1, у моделі Swin трансформер v.2 результати збіжності значно кращі, ніж у Swin трансформер v.1.

В процесі прогнозування була сформована наступна матриця помилок (рис. 5.3.3.2).

З матриці помилок ми бачимо, що більшість типів шкідливого програмного класифікується коректно (наприклад, гарні результати також спостерігаються при визначенні класів Allaple.A, Allaple.L). В той же час, спостерігається, що модель досить часто помилково визначає класи шкідливого програмного забезпечення Swizzor.gen!E та Swizzor.gen!I (так само як і у випадку з Swin трансформер v.2 та CNN).

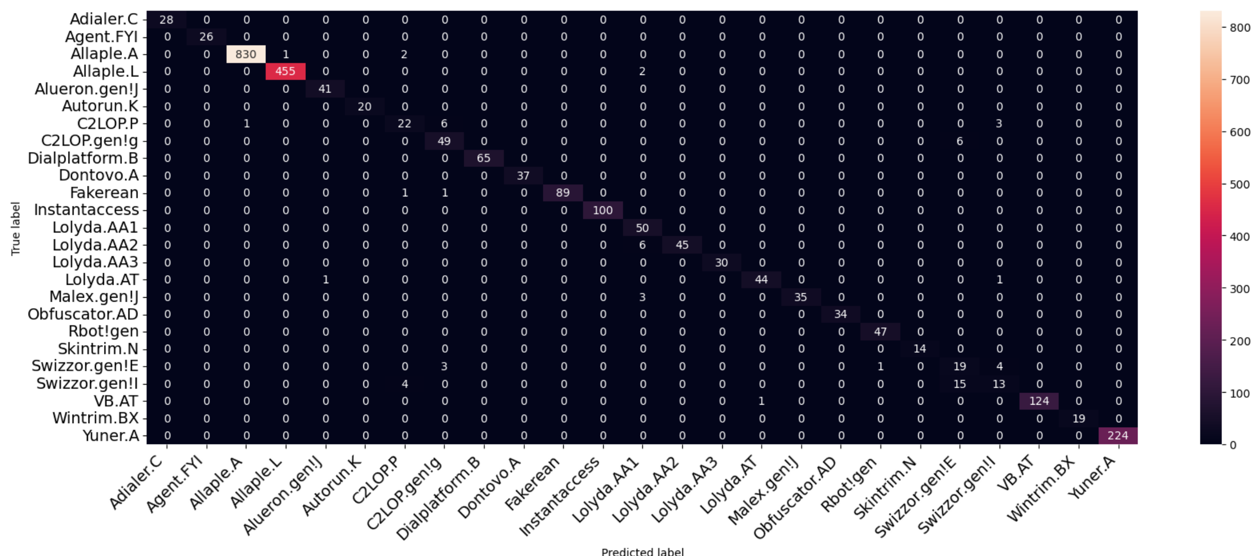



Рисунок 5.3.3.2 – Результати тренування моделі Swin трансформер v.2 - матриця помилок (confusion matrix)

Таблиця 5.3.3.1 – Метрики по класам для Swin трансформер v.2

| № | class_label | precision | recall | f1-score | support |
|---|---|---|---|---|---|
| 0 | Adialer.C | 1 | 1 | 1 | 28 |
| 1 | Agent.FYI | 1 | 1 | 1 | 26 |
| 2 | Allaple.A | 0.999 | 0.996 | 0.998 | 833 |
| 3 | Allaple.L | 0.998 | 0.996 | 0.997 | 457 |
| 4 | Alueron.gen!J | 0.976 | 1 | 0.988 | 41 |
| 5 | Autorun.K | 1 | 1 | 1 | 20 |
| 6 | C2LOP.P | 0.759 | 0.688 | 0.721 | 32 |
| 7 | C2LOP.gen!g | 0.831 | 0.891 | 0.86 | 55 |
| 8 | Dialplatform.B | 1 | 1 | 1 | 65 |
| 9 | Dontovo.A | 1 | 1 | 1 | 37 |
| 10 | Fakerean | 1 | 0.978 | 0.989 | 91 |
| 11 | Instantaccess | 1 | 1 | 1 | 100 |
| 12 | Lolyda.AA1 | 0.82 | 1 | 0.901 | 50 |
| 13 | Lolyda.AA2 | 1 | 0.882 | 0.938 | 51 |
| 14 | Lolyda.AA3 | 1 | 1 | 1 | 30 |
| 15 | Lolyda.AT | 0.978 | 0.957 | 0.967 | 46 |

Продовження таблиці 5.3.3.1

| 16 | Malex.gen!J | 1 | 0.921 | 0.959 | 38 |
|---|---|---|---|---|---|
| 17 | Obfuscator.AD | 1 | 1 | 1 | 34 |
| 18 | Rbot!gen | 0.979 | 1 | 0.989 | 47 |
| 19 | Skintrim.N | 1 | 1 | 1 | 14 |
| 20 | Swizzor.gen!E | 0.475 | 0.704 | 0.567 | 27 |
| 21 | Swizzor.gen!I | 0.619 | 0.406 | 0.491 | 32 |
| 22 | VB.AT | 1 | 0.992 | 0.996 | 125 |
| 23 | Wintrim.BX | 1 | 1 | 1 | 19 |
| 24 | Yuner.A | 1 | 1 | 1 | 224 |

Загальна кількість представників по всіх класах (support) становила 2522.

Як бачимо з табл. 5.3.3.1 модель Swin трансформеру v.2 має кращі результати, ніж модель Swin трансформеру v.2. Так, все ще погано розпізнаються класи шкідливого програмного забезпечення Swizzor.gen!I (61.9% - значно вища, ніж у у Swin 2), Swizzor.gen!E (47,5%) та C2LOP.P (75,9%). При цьому точність прогнозування по всім іншим класам є відносно високою (більше 82%).

Метрики з табл. 5.3.3.2, що отримані з використанням бібліотеки Sklearn, підтверджують дуже гарні результати (accuracy > 0.97) за результатами використання моделі Swin трансформеру v.2.

Таблиця 5.3.3.2 – Агреговані для набору даних метрики Swin трансформеру v.2

| № | class_label | precision | recall | f1-score |
|---|---|---|---|---|
| 1 | accuracy | 0.975416 | 0.975416 | 0.975416 |
| 2 | macro avg | 0.937303 | 0.936413 | 0.934385 |
| 3 | weighted avg | 0.977236 | 0.975416 | 0.975475 |

Таким чином, модель Swin трансформеру v.2 продемонструвала високу ефективність при розпізнаванні класів шкідливого програмного забезпечення з використанням набору даних Malimg, але все рівно гіршу в порівнянні з CNN.

### 5.3.4 Результати навчання CoAtNet

CoAtNet належить до гібридних моделей, побудованих на основі двох ключових ідей [54]:

– згортання в глибину і самоувага можуть бути природно об'єднані за допомогою простої відносної уваги;

– вертикальне розташування шарів згортання і шарів уваги принциповим чином напрочуд ефективно покращує узагальнення, пропускну спроможність та ефективність.

Отримані за результатами навчання гібридн6ої нейронної мережі CoAtNet метрики:

– акуратність (accuracy): 0.9877;

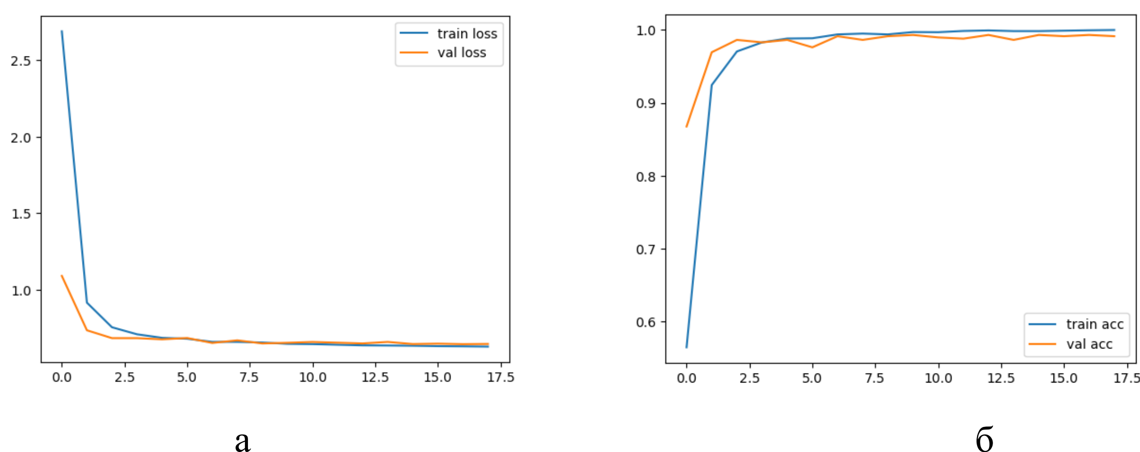
– top-5-accuracy: 0.9972;

– втрати (loss): 0.6732;

– кількість епох: 18.

Як бачимо з наведених вище даних, використана модель продемонструвала найкращу ефективність (кращу, ніж CNN, Swin трансформер v.1 та v.2). Так, на тестовій виборці даних акуратність (точність) визначення становила 98,77 (CNN - 98,33%). При цьому, зразки п’яти найбільш поширених видів шкідливого програмного забезпечення визначалися з точністю 99,72% (CNN - 99,96%).

В процесі тренування моделі спостерігалися зміни цільових метрик функції втрат (loss) та акуратності (accuracy), відображені на рис. 5.3.4.1.




а б

Рисунок 5.3.4.1 – Результати тренування моделі CoAtNet в залежності від кількості епох: а – графік втрат; б – графік акуратності

Як відображено на рис. 5.3.4.1, модель CoAtNet має найкращу збіжність протягом майже всього періоду навчання. Це робить її лідером у порівнянні з іншими трьома моделями.

В процесі прогнозування була сформована наступна матриця помилок (рис. 5.3.4.2)

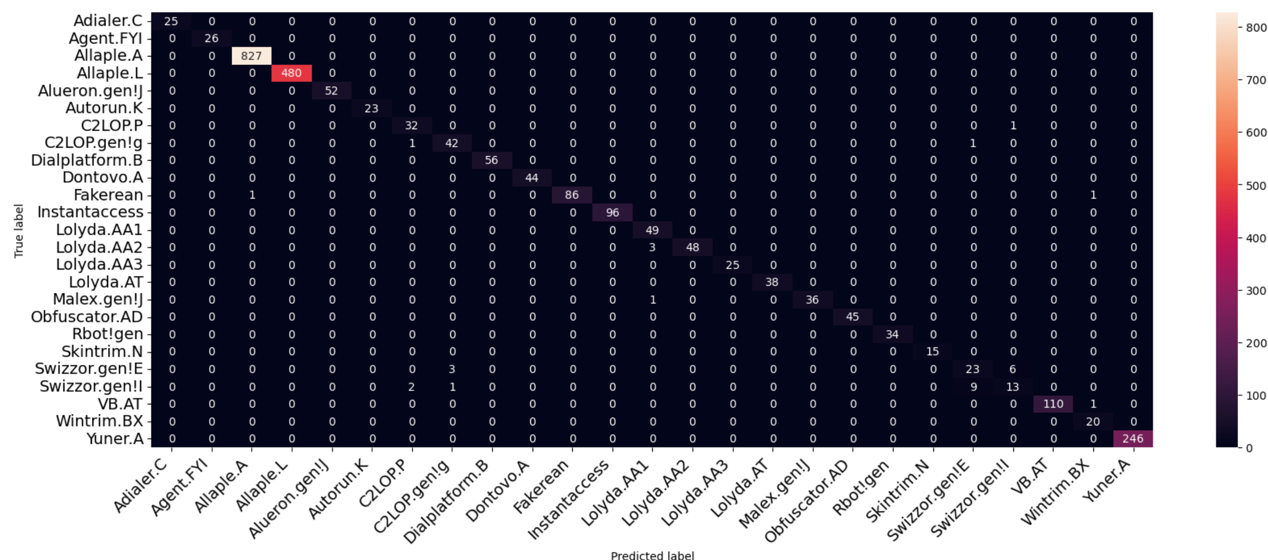



Рисунок 5.3.4.2 – Результати тренування моделі CoAtNet - матриця помилок (confusion matrix)

З матриці помилок ми бачимо, що більшість типів шкідливого програмного класифікується коректно. В той же час, спостерігається, що модель все що відносно часто помилково визначає класи шкідливого програмного забезпечення Swizzor.gen!E та Swizzor.gen!I (так само, як і у випадках із трьома іншими моделями).

Як бачимо з табл. 5.3.4.1 модель CoAtNet має дуже добрі результати (в порівнянні із трьома попередніми моделями). Значно краще розпізнаються класи шкідливого програмного забезпечення Swizzor.gen!I (65%), Swizzor.gen!E (69,7%) та C2LOP.P (91.4%). При цьому, точність прогнозування по всім іншим класам є дуже високою (більше 90%).

Таблиця 5.3.4.1 – Метрики по класам для CoAtNet

| № | class_label | precision | recall | f1-score | support |
|---|---|---|---|---|---|
| 0 | Adialer.C | 1 | 1 | 1 | 25 |
| 1 | Agent.FYI | 1 | 1 | 1 | 26 |
| 2 | Allaple.A | 0.999 | 1 | 0.999 | 827 |
| 3 | Allaple.L | 1 | 1 | 1 | 480 |
| 4 | Alueron.gen!J | 1 | 1 | 1 | 52 |
| 5 | Autorun.K | 1 | 1 | 1 | 23 |
| 6 | C2LOP.P | 0.914 | 0.97 | 0.941 | 33 |
| 7 | C2LOP.gen!g | 0.913 | 0.955 | 0.933 | 44 |
| 8 | Dialplatform.B | 1 | 1 | 1 | 56 |
| 9 | Dontovo.A | 1 | 1 | 1 | 44 |
| 10 | Fakerean | 1 | 0.977 | 0.989 | 88 |
| 11 | Instantaccess | 1 | 1 | 1 | 96 |
| 12 | Lolyda.AA1 | 0.925 | 1 | 0.961 | 49 |
| 13 | Lolyda.AA2 | 1 | 0.941 | 0.97 | 51 |
| 14 | Lolyda.AA3 | 1 | 1 | 1 | 25 |
| 15 | Lolyda.AT | 1 | 1 | 1 | 38 |
| 16 | Malex.gen!J | 1 | 0.973 | 0.986 | 37 |
| 17 | Obfuscator.A | 1 | 1 | 1 | 45 |
| 18 | Rbot!gen | 1 | 1 | 1 | 34 |
| 19 | Skintrim.N | 1 | 1 | 1 | 15 |
| 20 | Swizzor.gen!E | 0.697 | 0.719 | 0.708 | 32 |
| 21 | Swizzor.gen!I | 0.65 | 0.52 | 0.578 | 25 |
| 22 | VB.AT | 1 | 0.991 | 0.995 | 111 |
| 23 | Wintrim.BX | 0.909 | 1 | 0.952 | 20 |
| 24 | Yuner.A | 1 | 1 | 1 | 246 |

Загальна кількість представників по всіх класах (support) становила 2522.

Метрики з табл. 5.3.2.2, що отримані з використанням бібліотеки Sklearn, підтверджують найкращі результати (accuracy > 0.98) за результатами використання моделі CoAtNet.

Таблиця 5.3.4.2 – Агреговані метрики CoAtNet для набору даних

| № | class_label | precision | recall | f1-score |
|---|---|---|---|---|
| 1 | accuracy | 0.987708 | 0.987708 | 0.987708 |
| 2 | macro avg | 0.960268 | 0.961816 | 0.960501 |
| 3 | weighted avg | 0.987464 | 0.987708 | 0.987421 |

Таким чином, модель гібридної нейронної мережа CoAtNet продемонструвала в цілому найкращу ефективність при розпізнаванні класів шкідливого програмного забезпечення з використанням набору даних Malimg в порівнянні з трьома попередніми моделями (CNN, Swin v.1, Swin v.2) .

### 5.3.5 Порівняння результатів

За результатами роботи моделей з розпізнавання шкідливого програмного забезпечення по класам, можна емпірично побачити, що по всім чотирьом моделям два класи (Swizzor.gen!E, а також, Swizzor.gen!I) мають найгіршу акуратність розпізнавання.

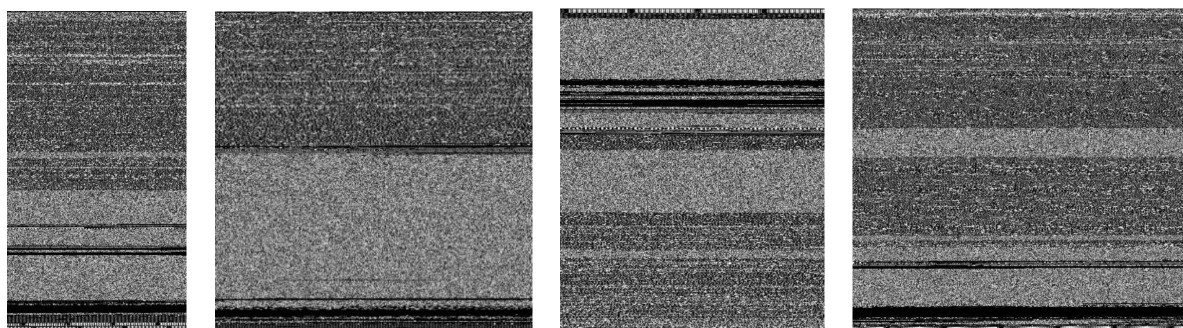


Рисунок 5.3.5.1 – Зразки зображень класу Swizzor.gen!E

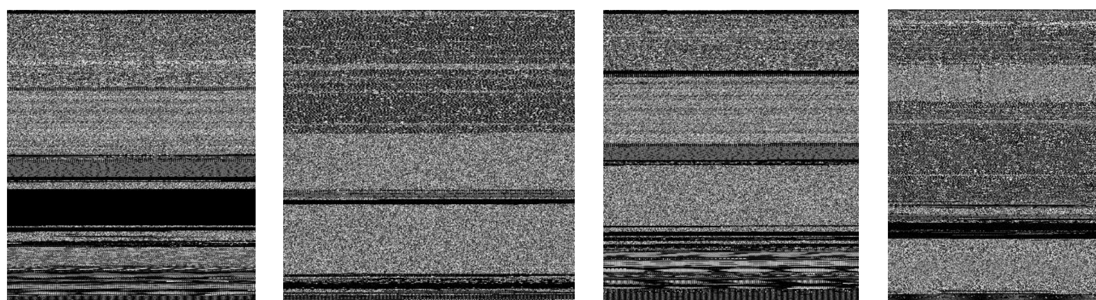


Рисунок 5.3.5.2 – Зразки зображень класу Swizzor.gen!I

Як бачимо, зразки зображень по класам шкідливого програмного забезпечення Swizzor.gen!E та Swizzor.gen!I візуально подібні. Це, ймовірно, вплинуло на значний рівень помилок при їхній класифікації моделями згорткової нейронної мережі, а також, Swin трансформерів першої та другої версій.

Разом з цим, нижче приведені зразки зображень класу Adialer.C (по якому акуратність як результат по всім чотирьом моделям дорівнює одиниці).

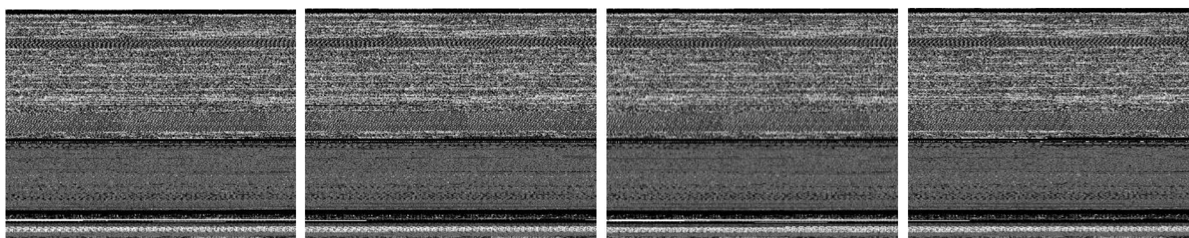


Рисунок 5.3.5.3 – Зразки зображень класу Adialer.C

Зразки класу Adialer.C складно сплутати зі зразками інших класів шкідливого програмного забезпечення з обраного набору даних Malimg.

Виходячи з візуального порівняння зразків зображень вище зазначених класів, можна зробити висновок, що найгірший показник акуратності є по класах, представники якого мають значні візуальні відмінності. При цьому класи, по яких зображення дуже схожі візуально, мають найкращі (максимальні) показники розпізнавання.

Однак, в результаті використання моделі гібридної нейронної мережі CoAtNet було отримано дуже цікавий результат розпізнавання зразків найбільш складного класу ШПЗ - Swizzor.gen!I. По ньому модель CoAtNet показала найкращий результат.

Таблиця 5.3.5.1 – Метрики по деяких класах ШПЗ та типах моделей глибинного навчання

| Клас ШПЗ / Нейронна мережа | CNN | Swin 1 | Swin 2 | CoAtNet |
|---|---|---|---|---|
| C2LOP.P | 0.727 | 0.724 | 0.759 | 0.914 |
| C2LOP.gen!g | 0.918 | 0.828 | 0.831 | 0.913 |
| Swizzor.gen!E | 0.75 | 0.447 | 0.475 | 0.697 |
| Swizzor.gen!I | 0.5 | 0.25 | 0.619 | 0.65 |

Додатково до раніше наведеного, з таблиці 5.3.5.2 можна побачити, що гібридна штучна нейронна мережа CoAtNet по більшості показників має найкращі результати виконання завдання з розпізнавання зображень шкідливого програмного забезпечення на наборі даних Malimg. Це корелює з висновками авторів моделі CoAtNet.

Таблиця 5.3.5.2 – Метрики по типах моделей глибинного навчання

| Метрика / Нейронна мережа | CNN | Swin 1 | Swin 2 | CoAtNet |
|---|---|---|---|---|
| Акуратність (accuracy): | 0.9833 | 0.9611 | 0.9754 | 0.9877 |
| Втрати (loss): | 0.7039 | 0.7732 | 0.6920 | 0.6732 |
| Топ-5 акуратність (top-5-accuracy): | 0.9996 | 0.9972 | 0.9968 | 0.9972 |
| Кількість епох (epochs): | 22 | 32 | 26 | 18 |

Тим не менш, всі чотири моделі нейронних мереж (CNN, Swin трансформер v.1 та v.2, CoAtNet) показують доволі високий рівень акуратності розпізнавання на наборі даних Malimg – більший 96%. Це робить зазначені моделі придатними для їх подальшого використання в реальних сценаріях розпізнавання шкідливого програмного забезпечення.

# ВИСНОВКИ

Майже рік тому в Україні почалася війна. Для того, щоб завдати найбільшої шкоди нашій країні, зараз використовується різна зброя. Особливу роль в цьому відіграють кібератаки.

Проблема ідентифікації та розпізнавання шкідливих програм є дуже складним завданням, яке не має ідеального рішення.

Дана робота спрямована на дослідження розпізнавання шкідливого програмного забезпечення з використанням архітектури нейронних мереж трансформерів.

В рамках даної роботи вирішено ряд завдань, а саме:

– зроблено огляд наукової літератури з конвертації бінарних файлів шкідливого програмного забезпечення в зображення, а також, наукової літератури з розвитку технологій нейронних мереж (згорткових, трансформерів зору, гібридних);

– обрано набір даних Malimg, як найбільш релевантний для цього дослідження;

– проведено навчання моделей глибинного навчання з архітектурами згорткової нейронної мережі, Swin трансформерів (першої та другої версій), гібридної нейронної мережі CoAtNet для розпізнавання шкідливого програмного забезпечення;

– проведено аналіз отриманих результатів.

За результатами проведеного аналізу, найкращу ефективність показала гібридна штучна нейронна мережа CoAtNet.

Отримані результати мають наукову новизну та практичне значення, відповідно:

1. Наукова новизна – вперше була досліджена ефективність використання нейромережевої архітектури трансформерів, а саме, Swin трансформерів першої та другої версій, а також гібридної нейромережі CoAtNet (що використовує механізми згортки та самоуваги) у порівнянні зі згортковою нейронною мережею

при вирішенні задач із розпізнавання зображень шкідливого програмного забезпечення у відтінках сірого, на прикладі набору даних Malimg.

2. Практичне значення отриманих результатів полягає в визначенні моделі гібридної нейромережі CoAtNet як найбільш ефективної для точних класифікаційних рішень з розпізнавання зображень шкідливого програмного забезпечення в порівнянні з моделями CNN та Swin трансформерів.

Подальші дослідження будуть спрямовані на використання інших гібридних архітектур з реалізацією функціоналу уваги або самоуваги для розпізнавання зображень шкідливого програмного забезпечення. Також, перспективною в умовах війни може бути подальша робота з аналізом зображень з високою роздільною здатністю в інших сферах (наприклад, в дистанційному зондуванні (remote sensing) [62]) із новітніми архітектурами траснсформерів.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Frolov, D., Radziewicz, W., Saienko, V., Kuchuk, N., Mozhaiev, M., Gnusov, Y., & Onishchenko, Y. (2021). Theoretical And Technological Aspects Of Intelligent Systems: Problems Of Artificial Intelligence. International Journal of Computer Science and Network Security, 21(5), 35-38. DOI 10.22937/IJCSNS.2021.21.5.6.
2. Global Cybersecurity Market Could Exceed $320 Billion in Revenues by 2027, July 29, 2020 [Електронний ресурс] – Режим доступу до ресурсу: https://www.bloomberg.com/press-releases/2020-07-29/global-cybersecurity-market-could-exceed-320-billion-in-revenues-by-2027.
3. Alanazi SA, Kamruzzaman MM, Alruwaili M, Alshammari N, Alqahtani SA, Karime A. Meas-uring and preventing COVID-19 using the SIR model and machine learning in smart health care. Journal of healthcare engineering. 2020 October. https://www.hindawi.com/journals/jhe/2020/8857346/.
4. Kang Z, Catal C, Tekinerdogan B. Machine learning applications in production lines: A sys-tematic literature review. Computers & Industrial Engineering. 2020 November 1;149:106773 [Електронний ресурс] – Режим доступу до ресурсу: https://www.sciencedirect.com/science/article/pii/S036083522030485X.
5. Turkson RE, Baagyere EY, Wenya GE. A machine learning approach for predicting bank credit worthiness. In 2016 Third International Conference on Artificial Intelligence and Pattern Recognition (AIPR) 2016 September (pp. 1-7). IEEE [Електронний ресурс] – Режим доступу до ресурсу: https://ieeex-plore.ieee.org/document/7585216.
6. Roy R, George KT. Detecting insurance claims fraud using machine learning techniques. In 2017 International Conference on Circuit, Power and Computing Technologies (ICCPCT) 2017 April (pp. 1-6). IEEE [Електронний ресурс] – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/8074258.

7. Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B.S. (2011). Malware images: visualization and automatic classification. Visualization for Computer Security. [Електронний ресурс] – Режим доступу до ресурсу: https://www.semanticscholar.org/paper/Malware-images%3A-visualization-and-automatic-Nataraj-Karthikeyan/87ff7ef37bd35d97ea0316398e762cd642df64a6.
8. European Union Agency For Cybersecurity. ENISA Threat Landscape 2022 [Електронний ресурс] – Режим доступу до ресурсу: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022
9. Шкідливий програмний засіб. Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Шкідливий_програмний_засіб.
10. Сигнатура атаки. Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Сигнатура_атаки.
11. OPWNAI: Cybercriminals Starting to Use ChatGPT, January 6, 2023 [Електронний ресурс] – Режим доступу до ресурсу: https://research.checkpoint.com/2023/opwnai-cybercriminals-starting-to-use-chatgpt.
12. Intel HEX. Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Intel_HEX.
13. Shalaginov, A.; Banin, S.; Dehghantanha, A.; and Franke, K.2018. Machine Learning Aided Static Malware Analysis: A Survey and Tutorial. Cyber Threat Intelligence. Advances in Information Security 70:7–45.
14. Oliva A, Torralba A. Modeling the shape of the scene: A holistic representation of the spatial envelope. International journal of computer vision. 2001 May;42(3):145-75 [Електронний ресурс] – Режим доступу до ресурсу: https://link.springer.com/article/10.1023/A:1011139631724.
15. Gibert D, Mateu C, Planes J, Vicens R. Using convolutional neural networks for classification of malware represented as images. Journal of Computer Virology and Hacking Techniques. 2019 March;15(1):15-28 [Електронний

ресурс] – Режим доступу до ресурсу: – Режим доступу до ресурсу: https://link.springer.com/article/10.1007/s11416-018-0323-0.

16. G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks”, science, vol. 313, no. 5786, pp. 504–507, 2006.
17. Q. Zhang, L. T. Yang, Z. Chen, and P. Li, “A survey on deep learning for big data”, Information Fusion, vol. 42, pp. 146–157, 2018, ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2017.10.006 [Електронний ресурс] – Режим доступу до ресурсу: – Режим доступу до ресурсу: http://www.sciencedirect.com/science/article/pii/S1566253517305328.
18. Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning”, nature, vol. 521, no. 7553, p. 436, 2015.
19. F. Rosenblatt, ‘The perceptron: A probabilistic model for information storage and organization in the brain.,’ Psychological review, vol. 65 6, pp. 386– 408, 1958.
20. W. S. McCulloch and W. Pitts, ‘A logical calculus of the ideas immanent in nervous activity,’ The bulletin of mathematical biophysics, vol. 5, pp. 115–133, 1943. DOI: 10.1007/BF02478259 [Електронний ресурс] – Режим доступу до ресурсу: – Режим доступу до ресурсу: https://link.springer.com/article/10.1007/BF02478259.
21. L. Vu-Quoc, Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals, Distributed on wikimedia commons under the CC BY-SA 3.0 lisence, 2018 [Електронний ресурс] – Режим доступу до ресурсу: https://commons.wikimedia.org/wiki/File:Neuron3.svg.
22. Chrislb, Diagram of an artificial neuron, Distributed on wikimedia commons under the CC BY-SA 3.0 lisence, 2005 [Електронний ресурс] – Режим доступу до ресурсу: https://commons.wikimedia.org/wiki/File:ArtificialNeuronModel_english.png.
23. K. Andrej, Neural networks part 1 [Електронний ресурс] – Режим доступу до ресурсу: http://cs231n.github.io/neural-networks- 1/.

24. Soma (biology). Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Soma_(biology).
25. Offnfopt, Illustration of an artificial feed-forward neural network, Distributed on wikimedia commons under the CC BY-SA 3.0 lisence, 2015 [Електронний ресурс] – Режим доступу до ресурсу: https://commons.wikimedia.org/wiki/File:Multi-Layer_Neural_Network-Vector-Blank.svg.
26. M. A. Nielsen, Neural networks and Deep Learning. Determination Press, 2015, ch. 2.
27. N. Brunel, V. Hakim, and M. J. Richardson, “Single neuron dynamics and computation”, Current opinion in neurobiology, vol. 25, pp. 149–155, 2014.
28. K. Andrej, Convolutional networks [Електронний ресурс] – Режим доступу до ресурсу: http://cs231n.github.io/.
29. Yegnanarayana B. Artificial neural networks. PHI Learning Pvt. Ltd.; 2009 January [Електронний ресурс] – Режим доступу до ресурсу: http://cdn.iiit.ac.in/cdn/speech.iiit.ac.in/svlpubs/book/Yegna1999.pdf.
30. Nwankpa C, Ijomah W, Gachagan A, Marshall S. Activation functions: Comparison of trends in practice and research for deep learning. 2018 November. arXiv:1811.03378 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/1811.03378.
31. Перехресна ентропія. Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Перехресна_ентропія.
32. Dauphin YN, De Vries H, Bengio Y. Equilibrated adaptive learning rates for non-convex optimization. 2015 April. arXiv:1502.04390 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/1502.04390.
33. Sweta Shaw. RMSprop : A Better Way to Optimize Your Model. 2019 [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@shwetaka1988/rmsprop-a-better-way-to-optimize-your-model-bc4eaca33090.

34. Kingma DP, Ba J. Adam: A method for stochastic optimization. 2014 December. arXiv:1412.6980 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/1412.6980.
35. CIFAR-10. CIFAR-10 Dataset [Електронний ресурс] – Режим доступу до ресурсу: https://www.cs.toronto.edu/~kriz/cifar.html.
36. Neural Network and Deep Learning - From Theory to Realization: Building L-Layer Neural Network from Zero (2019) [Електронний ресурс] – Режим доступу до ресурсу: https://programmer.help/blogs/building-l-layer-neural-network-from-zero.html.
37. Multi-Class Neural Networks: Softmax. 2020 March [Електронний ресурс] – Режим доступу до ресурсу: https://develop-ers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax.
38. Classification on imbalanced data [Online]. 2021 November [Електронний ресурс] – Режим доступу до ресурсу: https://www.tensor-flow.org/tutorials/structured_data/imbalanced_data.
39. Christopher Riggio. What's the deal with Accuracy, Precision, Recall and F1?. 2019 November [Електронний ресурс] – Режим доступу до ресурсу: https://towardsdatascience.com/whats-the-deal-with-accuracy-precision-recall-and-f1-f5d8b4db1021.
40. Grandini M, Bagli E, Visani G. Metrics for multi-class classification: an overview. 2020 August. arXiv:2008.05756 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/2008.05756.
41. K. He, X. Zhang, S. Ren and J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015. DOI: 10.48550/ARXIV.1502.01852 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/1502.01852.
42. A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit and N. Houlsby, An image is worth 16x16 words: Transformers for image

recognition at scale, 2020. DOI: 10.48550/ARXIV.2010.11929 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/2010.11929.
43. Convolutional neural network [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Convolu-tional_neural_network.
44. Akinyelu, A.A.; Zaccagna, F.; Grist, J.T.; Castelli, M.; Rundo, L. Brain Tumor Diagnosis Using Machine Learning, Convolutional Neural Networks, Capsule Neural Networks and Vision Transformers, Applied to MRI: A Survey. J. Imaging 2022, 8, 205 [Електронний ресурс] – Режим доступу до ресурсу: https://doi.org/10.3390/jimaging8080205.
45. Kaparthy, A. 2016. Cnn convolution matrix - cs231n [Електронний ресурс] – Режим доступу до ресурсу: http://cs231n.github.io/convolutional-networks/.
46. J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net", arXiv preprint arXiv:1412.6806, 2014.
47. Z. Yuan, Y. Lu, Z.Wang, and Y. Xue, "Droid-sec: Deep learning in android malware detection", in ACM SIGCOMM Computer Communication Review, ACM, vol. 44, 2014, pp. 371–372.
48. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L.Kaiser and I. Polosukhin, Attention is all you need, 2017. DOI: 10.48550/ARXIV.1706.03762 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/1706.03762.
49. Y. Liu, Y. Zhang, Y.Wang, F. Hou, J. Yuan, J. Tian, Y. Zhang, Z. Shi, J. Fan and Z. He, A survey of visual transformers, 2021. arXiv: 2111.06091 [cs.CV].
50. D. Hendrycks and K. Gimpel, Gaussian error linear units (gelus), 2016. DOI: 10.48550/ARXIV.1606.08415 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/1606.08415.
51. Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin and B. Guo, Swin transformer: Hierarchical vision transformer using shifted windows, 2021. DOI: 10.48550/ARXIV.2103.14030 [Електронний ресурс] – Режим доступу до ресурсу: https://arxiv.org/abs/2103.14030.

52. Loy, J., A Comprehensive Guide to Microsoft's Swin Transformer. In-depth Explanation and Animations, 20.05.2022 [електронний ресурс], https://towardsdatascience.com/a-comprehensive-guide-to-swin-transformer-64965f89d14c.
53. Z. Liu et al., "Swin Transformer V2: Scaling Up Capacity and Resolution," 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, LA, USA, 2022, pp. 11999-12009, doi: 10.1109/CVPR52688.2022.01170.
54. Dai, Z., Liu, H., Le, Q.V., & Tan, M. (2021). CoAtNet: Marrying Convolution and Attention for All Data Sizes. ArXiv, abs/2106.04803.
55. Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 7132–7141, 2018.
56. Kun Yuan, Shaopeng Guo, Ziwei Liu, Aojun Zhou, Fengwei Yu, and Wei Wu. Incorporating convolution designs into visual transformers. arXiv preprint arXiv:2103.11816, 2021.
57. Wenhai Wang, Enze Xie, Xiang Li, Deng-Ping Fan, Kaitao Song, Ding Liang, Tong Lu, Ping Luo, and Ling Shao. Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. arXiv preprint arXiv:2102.12122, 2021.
58. Ashish Vaswani, Prajit Ramachandran, Aravind Srinivas, Niki Parmar, Blake Hechtman, and Jonathon Shlens. Scaling local self-attention for parameter efficient visual backbones. arXiv preprint arXiv:2103.12731, 2021.
59. Набір даних образів шкідливого програмного забезпечення «malimg_dataset9010» [Електронний ресурс] – Режим доступу до ресурсу: https://www.kaggle.com/datasets/keerthicheepurupalli/malimg-dataset9010.
60. Kechit Goyal. Top 10 Deep Learning Frameworks in 2021 You Can't Ignore. 2021 January [Електронний ресурс] – Режим доступу до ресурсу: https://www.upgrad.com/blog/top-deep-learning-frameworks/.

61. Scikit-learn. Scikit-learn: Machine Learning in Python [Електронний ресурс] – Режим доступу до ресурсу: https://scikit-learn.org/stable/.
62. Aleissaee, Abdulaziz Amer, Amandeep Kumar, Rao Muhammad Anwer, Salman Khan, Hisham Cholakkal, and Gui-Song Xia. "Transformers in remote sensing: A survey." arXiv preprint arXiv:2209.01206, 2022.

# Додаток А. Програмний код основних програмних модулів, побудованих на архітектурі згорткової нейронної мережі

```
import sys
import os
from math import log
import numpy as np
import pandas as pd
import scipy as sp
from PIL import Image
import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow_addons as tfa
from tensorflow import keras

import tensorflow as tf
from tensorflow.keras import models, layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten , BatchNormalization
from tensorflow.keras.layers import Conv2D, MaxPooling2D , AveragePooling2D,GlobalAveragePooling2D
from tensorflow.keras.callbacks import ReduceLROnPlateau,EarlyStopping
```

## Data Preprocessing and Basic EDA

```
# prepare the data

train_root_path = "./malimg_dataset/train"
val_root_path = "./malimg_dataset/validation"

from tensorflow.keras.preprocessing.image import ImageDataGenerator
batches = ImageDataGenerator().flow_from_directory(directory=train_root_path, target_size=(150,150), batch_size=10000)
```

```
Found 8404 images belonging to 25 classes.
```

```
batches.class_indices
```

```
imgs, labels = next(batches)
imgs.shape
```

```
(8404, 150, 150, 3)
```

```
labels.shape
```

```
(8404, 25)
```

```
# plots images with labels within jupyter notebook
def plots(ims, figsize=(20,30), rows=10, interp=False, titles=None):
    if type(ims[0]) is np.ndarray:
        ims = np.array(ims).astype(np.uint8)
        if (ims.shape[-1] != 3):
            ims = ims.transpose((0,2,3,1))
    f = plt.figure(figsize=figsize)
    cols = 10 # len(ims)//rows if len(ims) % 2 == 0 else len(ims)//rows + 1
    for i in range(0,50):
        sp = f.add_subplot(rows, cols, i+1)
        sp.axis('Off')
        if titles is not None:
            sp.set_title(list(batches.class_indices.keys())[np.argmax(titles[i])], fontsize=16)
        plt.imshow(ims[i], interpolation=None if interp else 'none')
```

```
plots(imgs, titles = labels)
```

```
classes = batches.class_indices.keys()
perc = (sum(labels)/labels.shape[0])*100

plt.xticks(rotation='vertical')
plt.bar(classes,perc)
```

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(imgs/255.,labels, test_size=0.3)
```

```
x_train.shape
```

```
(5882, 150, 150, 3)
```

```
y_test.shape
```

```
(2522, 25)
```

## Train CNN model

```
import tensorflow_addons as tfa

batch_size = 32 #128

learning_rate = 1e-3

num_epochs = 100
validation_split = 0.1
weight_decay = 0.0001
label_smoothing = 0.1

def malware_model(width , height):
    Malware_model = Sequential()
    Malware_model.add(Conv2D(30, kernel_size=(3, 3),
                     activation='relu',
                     #input_shape=(batch_size, width, height, 3)))
                    # https://stackoverflow.com/questions/47665391/keras-valueerror-input-0-is-incompatible-with-layer-conv2
                     input_shape=(width, height, 3)))

    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(Conv2D(15, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(Dropout(0.25))
    Malware_model.add(Flatten())
    Malware_model.add(Dense(128, activation='relu'))
    Malware_model.add(Dropout(0.5))
    Malware_model.add(Dense(50, activation='relu'))

    Malware_model.add(Dense(25, activation='softmax'))
    # Malware_model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False), optimizer = 'adam', metri

    Malware_model.compile(
        loss=keras.losses.CategoricalCrossentropy(label_smoothing=label_smoothing),
        optimizer=tfa.optimizers.AdamW(
            learning_rate=learning_rate, weight_decay=weight_decay
        ),
        metrics=[
            keras.metrics.CategoricalAccuracy(name="accuracy"),
            keras.metrics.TopKCategoricalAccuracy(5, name="top-5-accuracy"),
        ],
    )
    return Malware_model
```

```
lr_reduction = ReduceLROnPlateau(monitor='val_accuracy',patience=4, verbose=1,  factor=0.4, min_lr=0.0001)

early_stop = EarlyStopping(monitor='val_accuracy', min_delta=0.00001, patience=8, mode='auto', restore_best_weights=True)
```

```
model=malware_model(150 , 150)
model_fit = model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=num_epochs,
    validation_split=validation_split,
    verbose =1,
    callbacks=[early_stop,lr_reduction]
)
```

```
model.evaluate(x_test,  y_test)
```

```
79/79 [==============================] - 5s 59ms/step - loss: 0.7039 - accuracy: 0.9833 - top-5-accuracy: 0.9996
[0.7038777470588684, 0.9833465218544006, 0.9996035099029541]
```

```
# plot the loss
plt.plot(model_fit.history['loss'], label='train loss')
plt.plot(model_fit.history['val_loss'], label='val loss')
plt.legend()
plt.show()
plt.savefig('LossVal_loss.jpg',format='jpg')

plt.close()
# plot the accuracy
plt.plot(model_fit.history['accuracy'], label='train acc')
plt.plot(model_fit.history['val_accuracy'], label='val acc')
plt.legend()
plt.show()
plt.savefig('AccVal_acc.jpg',format="jpg")

plt.close()
```

## Make predictions¶

```
pred_x = model.predict(x_test, verbose=0)
y_pred=np.argmax(pred_x,axis=1)
y_pred
```

```
array([ 2,  9,  4, ...,  2, 24, 10], dtype=int64)
```

```
y_test2 = np.argmax(y_test, axis=1)
y_test2
```

```
array([ 2,  9,  4, ...,  2, 24, 10], dtype=int64)
```

```
from sklearn import metrics
c_matrix = metrics.confusion_matrix(y_test2, y_pred)
```

```
import seaborn as sns
def confusion_matrix(confusion_matrix, class_names, figsize = (10,7), fontsize=14):
    df_cm = pd.DataFrame(
        confusion_matrix, index=class_names, columns=class_names,
    )
    fig = plt.figure(figsize=figsize)
    try:
        heatmap = sns.heatmap(df_cm, annot=True, fmt="d")
    except ValueError:
        raise ValueError("Confusion matrix values must be integers.")
    heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsize=fontsize)
    heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=45, ha='right', fontsize=fontsize)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
from sklearn import metrics
# Print the precision and recall, among other metrics
report =  metrics.classification_report(y_test2, y_pred, digits=3, output_dict=True)

df = pd.DataFrame(report).transpose().reset_index()
df = df.rename(columns={"index": "class_label"})
```

Let's review the part of the classification report related to each individual class

```
clf_rep = metrics.precision_recall_fscore_support(y_test2, y_pred)
out_dict = {
             "precision" : clf_rep[0].round(3)
            ,"recall" : clf_rep[1].round(3)
            ,"f1-score" : clf_rep[2].round(3)
            ,"support" : clf_rep[3]
            }
out_df = pd.DataFrame(out_dict).reset_index().rename(columns={"index": "class_label"})
class_label_values = dict(zip(range(0,len(batches.class_indices)), batches.class_indices))
out_df['class_label'] = out_df['class_label'].map(class_label_values)
out_df
```

```
# display aggregated values - selecting rows based on condition
options = ['accuracy', 'macro avg',  'weighted avg']
agg_df = df[df['class_label'].isin(options)]
agg_df
```

# Додаток Б. Програмний код основних програмних модулів, побудованих на архітектурі Swin трансформеру v.1

```
#setup
import sys
import os
from math import log
import numpy as np
import pandas as pd
import scipy as sp
from PIL import Image
import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow_addons as tfa
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.callbacks import ReduceLROnPlateau,EarlyStopping
```

```
# prepare the data

train_root_path = "./malimg_dataset/train"
val_root_path = "./malimg_dataset/validation"

from tensorflow.keras.preprocessing.image import ImageDataGenerator
batches = ImageDataGenerator().flow_from_directory(directory=train_root_path, target_size=(64,64), batch_size=10000)
```

```
Found 8404 images belonging to 25 classes.
```

```
batches.class_indices
```

```
imgs, labels = next(batches)
imgs.shape
```

```
(8404, 64, 64, 3)
```

```
labels.shape
```

```
(8404, 25)
```

```
# plots images with labels within jupyter notebook
def plots(ims, figsize=(20,30), rows=10, interp=False, titles=None):
    if type(ims[0]) is np.ndarray:
        ims = np.array(ims).astype(np.uint8)
        if (ims.shape[-1] != 3):
            ims = ims.transpose((0,2,3,1))
    f = plt.figure(figsize=figsize)
    cols = 10 # len(ims)//rows if len(ims) % 2 == 0 else len(ims)//rows + 1
    for i in range(0,50):
        sp = f.add_subplot(rows, cols, i+1)
        sp.axis('Off')
        if titles is not None:
            sp.set_title(list(batches.class_indices.keys())[np.argmax(titles[i])], fontsize=16)
        plt.imshow(ims[i], interpolation=None if interp else 'none')
```

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(imgs/255.,labels, test_size=0.3)
```

```
x_train.shape
```

```
(5882, 64, 64, 3)
```

```
x_test.shape
```

```
(2522, 64, 64, 3)
```

```
y_train.shape
```

```
(5882, 25)
```

```
y_test.shape
```

```
(2522, 25)
```

```
num_classes = len(classes)
# we do not need to apply one-hot encoding to the labels as in https://keras.io/examples/vision/swin_transformers/
# since the dataset data is already prepared for the multi-class classification
#y_train = keras.utils.to_categorical(y_train, num_classes)
#y_test = keras.utils.to_categorical(y_test, num_classes)
print(f"x_train shape: {x_train.shape} - y_train shape: {y_train.shape}")
print(f"x_test shape: {x_test.shape} - y_test shape: {y_test.shape}")
```

```
x_train shape: (5882, 64, 64, 3) - y_train shape: (5882, 25)
x_test shape: (2522, 64, 64, 3) - y_test shape: (2522, 25)
```

## Configure the image input shape

As specified in https://www.kaggle.com/code/tiletisaitejareddy/malware-classification-ism

```
input_shape=(64,64,3)  # (64,64,3)
```

## Configure the hyperparameters

A key parameter to pick is the patch_size, the size of the input patches. In order to use each pixel as an individual input, you can set patch_size to (1, 1). Below, we take inspiration from the original paper settings for training on ImageNet-1K, keeping most of the original settings for this solution.

**Note:** inspired by https://keras.io/examples/vision/swin_transformers/

```
patch_size = (2, 2)  # 2-by-2 sized patches
dropout_rate = 0.03  # Dropout rate
num_heads = 8  # Attention heads
embed_dim = 64  # Embedding dimension
num_mlp = 256  # MLP layer size
qkv_bias = True  # Convert embedded patches to query, key, and values with a learnable additive value
window_size = 2  # Size of attention window
shift_size = 1  # Size of shifting window
image_dimension = 64  # Initial image size - dictated by our dataset specifics

num_patch_x = input_shape[0] // patch_size[0]
num_patch_y = input_shape[1] // patch_size[1]

learning_rate = 1e-3
batch_size = 128
num_epochs = 40 ## TBD - to increase?
validation_split = 0.1
weight_decay = 0.0001
label_smoothing = 0.1
```

## Helper functions

We create two helper functions to help us get a sequence of patches from the image, merge patches, and apply dropout.

```
def window_partition(x, window_size):
    _, height, width, channels = x.shape
    patch_num_y = height // window_size
    patch_num_x = width // window_size
    x = tf.reshape(
        x, shape=(-1, patch_num_y, window_size, patch_num_x, window_size, channels)
    )
    x = tf.transpose(x, (0, 1, 3, 2, 4, 5))
    windows = tf.reshape(x, shape=(-1, window_size, window_size, channels))
    return windows
```

```
def window_reverse(windows, window_size, height, width, channels):
    patch_num_y = height // window_size
    patch_num_x = width // window_size
    x = tf.reshape(
        windows,
        shape=(-1, patch_num_y, patch_num_x, window_size, window_size, channels),
    )
    x = tf.transpose(x, perm=(0, 1, 3, 2, 4, 5))
    x = tf.reshape(x, shape=(-1, height, width, channels))
    return x
```

```
class DropPath(layers.Layer):
    def __init__(self, drop_prob=None, **kwargs):
        super().__init__(**kwargs)
        self.drop_prob = drop_prob

        #self.add_weight(name='drop_path')

    def call(self, x):
        input_shape = tf.shape(x)
        batch_size = input_shape[0]
        rank = x.shape.rank
        shape = (batch_size,) + (1,) * (rank - 1)
        random_tensor = (1 - self.drop_prob) + tf.random.uniform(shape, dtype=x.dtype)
        path_mask = tf.floor(random_tensor)
        output = tf.math.divide(x, 1 - self.drop_prob) * path_mask
        return output
```

## Window based multi-head self-attention

Usually Transformers perform global self-attention, where the relationships between a token and all other tokens are computed. The global computation leads to quadratic complexity with respect to the number of tokens. Here, as the original paper(https://arxiv.org/abs/2103.14030) suggests, we compute self-attention within local windows, in a non-overlapping manner. Global self-attention leads to quadratic computational complexity in the number of patches, whereas window-based self-attention leads to linear complexity and is easily scalable.

```
class WindowAttention(layers.Layer):
    def __init__(
        self, dim, window_size, num_heads, qkv_bias=True, dropout_rate=0.0, **kwargs
    ):
        super().__init__(**kwargs)
        self.dim = dim
        self.window_size = window_size
        self.num_heads = num_heads
        self.scale = (dim // num_heads) ** -0.5
        self.qkv = layers.Dense(dim * 3, use_bias=qkv_bias)
        self.dropout = layers.Dropout(dropout_rate)
        self.proj = layers.Dense(dim)

        #self.add_weight(name='window_attention')
```

```python
def build(self, input_shape):
    num_window_elements = (2 * self.window_size[0] - 1) * (
        2 * self.window_size[1] - 1
    )
    self.relative_position_bias_table = self.add_weight(
        shape=(num_window_elements, self.num_heads),
        initializer=tf.initializers.Zeros(),
        trainable=True,
        name='window_attention',
    )
    coords_h = np.arange(self.window_size[0])
    coords_w = np.arange(self.window_size[1])
    coords_matrix = np.meshgrid(coords_h, coords_w, indexing="ij")
    coords = np.stack(coords_matrix)
    coords_flatten = coords.reshape(2, -1)
    relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :]
    relative_coords = relative_coords.transpose([1, 2, 0])
    relative_coords[:, :, 0] += self.window_size[0] - 1
    relative_coords[:, :, 1] += self.window_size[1] - 1
    relative_coords[:, :, 0] *= 2 * self.window_size[1] - 1
    relative_position_index = relative_coords.sum(-1)

    self.relative_position_index = tf.Variable(
        initial_value=tf.convert_to_tensor(relative_position_index), trainable=False
    )
```

```python
def call(self, x, mask=None):
    _, size, channels = x.shape
    head_dim = channels // self.num_heads
    x_qkv = self.qkv(x)
    x_qkv = tf.reshape(x_qkv, shape=(-1, size, 3, self.num_heads, head_dim))
    x_qkv = tf.transpose(x_qkv, perm=(2, 0, 3, 1, 4))
    q, k, v = x_qkv[0], x_qkv[1], x_qkv[2]
    q = q * self.scale
    k = tf.transpose(k, perm=(0, 1, 3, 2))
    attn = q @ k

    num_window_elements = self.window_size[0] * self.window_size[1]
    relative_position_index_flat = tf.reshape(
        self.relative_position_index, shape=(-1,)
    )
    relative_position_bias = tf.gather(
        self.relative_position_bias_table, relative_position_index_flat
    )
    relative_position_bias = tf.reshape(
        relative_position_bias, shape=(num_window_elements, num_window_elements, -1)
    )
    relative_position_bias = tf.transpose(relative_position_bias, perm=(2, 0, 1))
    attn = attn + tf.expand_dims(relative_position_bias, axis=0)

    if mask is not None:
        nW = mask.get_shape()[0]
        mask_float = tf.cast(
            tf.expand_dims(tf.expand_dims(mask, axis=1), axis=0), tf.float32
        )
        attn = (
            tf.reshape(attn, shape=(-1, nW, self.num_heads, size, size))
            + mask_float
        )
        attn = tf.reshape(attn, shape=(-1, self.num_heads, size, size))
        attn = keras.activations.softmax(attn, axis=-1)
    else:
        attn = keras.activations.softmax(attn, axis=-1)
    attn = self.dropout(attn)

    x_qkv = attn @ v
    x_qkv = tf.transpose(x_qkv, perm=(0, 2, 1, 3))
    x_qkv = tf.reshape(x_qkv, shape=(-1, size, channels))
    x_qkv = self.proj(x_qkv)
    x_qkv = self.dropout(x_qkv)
    return x_qkv
```

## The complete Swin Transformer model

Finally, we put together the complete Swin Transformer by replacing the standard multi-head attention (MHA) with shifted windows attention. As suggested in the original paper, we create a model comprising of a shifted window-based MHA layer, followed by a 2-layer MLP with GELU nonlinearity in between, applying LayerNormalization before each MSA layer and each MLP, and a residual connection after each of these layers.

Notice that we only create a simple MLP with 2 Dense and 2 Dropout layers. Often you will see models using ResNet-50 as the MLP which is quite standard in the literature. However in this paper the authors use a 2-layer MLP with GELU nonlinearity in between.

```
class SwinTransformer(layers.Layer):
    def __init__(
        self,
        dim,
        num_patch,
        num_heads,
        window_size=7,
        shift_size=0,
        num_mlp=1024,
        qkv_bias=True,
        dropout_rate=0.0,
        **kwargs,
    ):
        super().__init__(**kwargs)

        self.dim = dim  # number of input dimensions
        self.num_patch = num_patch  # number of embedded patches
        self.num_heads = num_heads  # number of attention heads
        self.window_size = window_size  # size of window
        self.shift_size = shift_size  # size of window shift
        self.num_mlp = num_mlp  # number of MLP nodes

        self.norm1 = layers.LayerNormalization(epsilon=1e-5)
        self.attn = WindowAttention(
            dim,
            window_size=(self.window_size, self.window_size),
            num_heads=num_heads,
            qkv_bias=qkv_bias,
            dropout_rate=dropout_rate,
        )
        self.drop_path = DropPath(dropout_rate)
        self.norm2 = layers.LayerNormalization(epsilon=1e-5)

        self.mlp = keras.Sequential(
            [
                layers.Dense(num_mlp),
                layers.Activation(keras.activations.gelu),
                layers.Dropout(dropout_rate),
                layers.Dense(dim),
                layers.Dropout(dropout_rate),
            ]
        )

        if min(self.num_patch) < self.window_size:
            self.shift_size = 0
            self.window_size = min(self.num_patch)
```

```
    def build(self, input_shape):
        if self.shift_size == 0:
            self.attn_mask = None
        else:
            height, width = self.num_patch
            h_slices = (
                slice(0, -self.window_size),
                slice(-self.window_size, -self.shift_size),
                slice(-self.shift_size, None),
            )
            w_slices = (
                slice(0, -self.window_size),
                slice(-self.window_size, -self.shift_size),
                slice(-self.shift_size, None),
            )
            mask_array = np.zeros((1, height, width, 1))
            count = 0
            for h in h_slices:
                for w in w_slices:
                    mask_array[:, h, w, :] = count
                    count += 1
            mask_array = tf.convert_to_tensor(mask_array)

            # mask array to windows
            mask_windows = window_partition(mask_array, self.window_size)
            mask_windows = tf.reshape(
                mask_windows, shape=[-1, self.window_size * self.window_size]
            )
            attn_mask = tf.expand_dims(mask_windows, axis=1) - tf.expand_dims(
                mask_windows, axis=2
            )
            attn_mask = tf.where(attn_mask != 0, -100.0, attn_mask)
            attn_mask = tf.where(attn_mask == 0, 0.0, attn_mask)
            self.attn_mask = tf.Variable(initial_value=attn_mask, trainable=False)
```

```
    def call(self, x):
        height, width = self.num_patch
        _, num_patches_before, channels = x.shape
        x_skip = x
        x = self.norm1(x)
        x = tf.reshape(x, shape=(-1, height, width, channels))
        if self.shift_size > 0:
            shifted_x = tf.roll(
                x, shift=[-self.shift_size, -self.shift_size], axis=[1, 2]
            )
        else:
            shifted_x = x

        x_windows = window_partition(shifted_x, self.window_size)
        x_windows = tf.reshape(
            x_windows, shape=(-1, self.window_size * self.window_size, channels)
        )
        attn_windows = self.attn(x_windows, mask=self.attn_mask)

        attn_windows = tf.reshape(
            attn_windows, shape=(-1, self.window_size, self.window_size, channels)
        )
        shifted_x = window_reverse(
            attn_windows, self.window_size, height, width, channels
        )
        if self.shift_size > 0:
            x = tf.roll(
                shifted_x, shift=[self.shift_size, self.shift_size], axis=[1, 2]
            )
        else:
            x = shifted_x

        x = tf.reshape(x, shape=(-1, height * width, channels))
        x = self.drop_path(x)
        x = x_skip + x
        x_skip = x
        x = self.norm2(x)
        x = self.mlp(x)
        x = self.drop_path(x)
        x = x_skip + x
        return x
```

# Model training and evaluation

## Extract and embed patches

We first create 3 layers to help us extract, embed and merge patches from the images on top of which we will later use the Swin Transformer class we built.

```
class PatchExtract(layers.Layer):
    def __init__(self, patch_size, **kwargs):
        super().__init__(**kwargs)
        self.patch_size_x = patch_size[0]
        self.patch_size_y = patch_size[0]

        #self.add_weight(name='patch_extract')

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=(1, self.patch_size_x, self.patch_size_y, 1),
            strides=(1, self.patch_size_x, self.patch_size_y, 1),
            rates=(1, 1, 1, 1),
            padding="VALID",
        )
        patch_dim = patches.shape[-1]
        patch_num = patches.shape[1]
        return tf.reshape(patches, (batch_size, patch_num * patch_num, patch_dim))
```

```
class PatchEmbedding(layers.Layer):
    def __init__(self, num_patch, embed_dim, **kwargs):
        super().__init__(**kwargs)
        self.num_patch = num_patch
        self.proj = layers.Dense(embed_dim)
        self.pos_embed = layers.Embedding(input_dim=num_patch, output_dim=embed_dim)

        #self.add_weight(name='patch_embedding')

    def call(self, patch):
        pos = tf.range(start=0, limit=self.num_patch, delta=1)
        return self.proj(patch) + self.pos_embed(pos)
```

```
class PatchMerging(tf.keras.layers.Layer):
    def __init__(self, num_patch, embed_dim):
        super().__init__()
        self.num_patch = num_patch
        self.embed_dim = embed_dim
        self.linear_trans = layers.Dense(2 * embed_dim, use_bias=False)

        #self.add_weight(name='patch_merging')

    def call(self, x):
        height, width = self.num_patch
        _, _, C = x.get_shape().as_list()
        x = tf.reshape(x, shape=(-1, height, width, C))
        x0 = x[:, 0::2, 0::2, :]
        x1 = x[:, 1::2, 0::2, :]
        x2 = x[:, 0::2, 1::2, :]
        x3 = x[:, 1::2, 1::2, :]
        x = tf.concat((x0, x1, x2, x3), axis=-1)
        x = tf.reshape(x, shape=(-1, (height // 2) * (width // 2), 4 * C))
        return self.linear_trans(x)
```

## Build the model

We put together the Swin Transformer model.

```
input = layers.Input(input_shape)
x = layers.RandomCrop(image_dimension, image_dimension)(input)
x = layers.RandomFlip("horizontal")(x)
x = PatchExtract(patch_size)(x)
x = PatchEmbedding(num_patch_x * num_patch_y, embed_dim)(x)
x = SwinTransformer(
    dim=embed_dim,
    num_patch=(num_patch_x, num_patch_y),
    num_heads=num_heads,
    window_size=window_size,
    shift_size=0,
    num_mlp=num_mlp,
    qkv_bias=qkv_bias,
    dropout_rate=dropout_rate,
)(x)
x = SwinTransformer(
    dim=embed_dim,
    num_patch=(num_patch_x, num_patch_y),
    num_heads=num_heads,
    window_size=window_size,
    shift_size=shift_size,
    num_mlp=num_mlp,
    qkv_bias=qkv_bias,
    dropout_rate=dropout_rate,
)(x)
x = PatchMerging((num_patch_x, num_patch_y), embed_dim=embed_dim)(x)
x = layers.GlobalAveragePooling1D()(x)
output = layers.Dense(num_classes, activation="softmax")(x)
```

## Train the model

```
model = keras.Model(input, output)
model.compile(
    loss=keras.losses.CategoricalCrossentropy(label_smoothing=label_smoothing),
    optimizer=tfa.optimizers.AdamW(
        learning_rate=learning_rate, weight_decay=weight_decay
    ),
    metrics=[
        keras.metrics.CategoricalAccuracy(name="accuracy"),
        keras.metrics.TopKCategoricalAccuracy(5, name="top-5-accuracy"),
    ],
)

model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 64, 64, 3)]       0

 random_crop (RandomCrop)    (None, 64, 64, 3)         0

 random_flip (RandomFlip)    (None, 64, 64, 3)         0

 patch_extract (PatchExtract  (None, 1024, 12)         0
 )

 patch_embedding (PatchEmbed  (None, 1024, 64)         66368
 ding)

 swin_transformer (SwinTrans  (None, 1024, 64)         50072
 former)

 swin_transformer_1 (SwinTra  (None, 1024, 64)         54168
 nsformer)

 patch_merging (PatchMerging  (None, 256, 128)         32768
 )

 global_average_pooling1d (G  (None, 128)              0
 lobalAveragePooling1D)

 dense_10 (Dense)            (None, 25)                3225

=================================================================
Total params: 206,601
Trainable params: 202,473
Non-trainable params: 4,128
_________________________________________________________________
```

Start the model training

```
model.evaluate(x_test,  y_test)
```

```
79/79 [==============================] - 26s 324ms/step - loss: 0.7732 - accuracy: 0.9611 - top-5-accuracy: 0.9972
[0.7731767296791077, 0.9611419439315796, 0.9972244501113892]
```

## Make Predictions

```
pred_x = model.predict(x_test, verbose=0)
y_pred=np.argmax(pred_x,axis=1)
y_pred
```

```
array([ 2,  2,  2, ...,  2, 24,  2], dtype=int64)
```

```
y_test2 = np.argmax(y_test, axis=1)
y_test2
```

```
array([ 2,  2,  2, ...,  2, 24,  2], dtype=int64)
```

```
from sklearn import metrics
c_matrix = metrics.confusion_matrix(y_test2, y_pred)
```

```
import seaborn as sns
def confusion_matrix(confusion_matrix, class_names, figsize = (10,7), fontsize=14):
    df_cm = pd.DataFrame(
        confusion_matrix, index=class_names, columns=class_names,
    )
    fig = plt.figure(figsize=figsize)
    try:
        heatmap = sns.heatmap(df_cm, annot=True, fmt="d")
    except ValueError:
        raise ValueError("Confusion matrix values must be integers.")
    heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsize=fontsize)
    heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=45, ha='right', fontsize=fontsize)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
from sklearn import metrics
# Print the precision and recall, among other metrics
report =  metrics.classification_report(y_test2, y_pred, digits=3, output_dict=True)

df = pd.DataFrame(report).transpose().reset_index()
df = df.rename(columns={"index": "class_label"})
```

Let's review the part of the classification report related to each individual class

```
clf_rep = metrics.precision_recall_fscore_support(y_test2, y_pred)
out_dict = {
             "precision" : clf_rep[0].round(3)
            ,"recall" : clf_rep[1].round(3)
            ,"f1-score" : clf_rep[2].round(3)
            ,"support" : clf_rep[3]
            }
out_df = pd.DataFrame(out_dict).reset_index().rename(columns={"index": "class_label"})
class_label_values = dict(zip(range(0,len(batches.class_indices)), batches.class_indices))
out_df['class_label'] = out_df['class_label'].map(class_label_values)
out_df
```

```
# display aggregated values - selecting rows based on condition
options = ['accuracy', 'macro avg',  'weighted avg']
agg_df = df[df['class_label'].isin(options)]
agg_df
```

## Додаток В. Програмний код основних програмних модулів, побудованих на архітектурі Swin трансформеру v.2

```
#setup
import sys
import os
from math import log
import numpy as np
import pandas as pd
import scipy as sp
from PIL import Image
import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.callbacks import ReduceLROnPlateau,EarlyStopping
```

### Data Preprocessing and Basic EDA

```
# prepare the data

train_root_path = "./malimg_dataset/train"
val_root_path = "./malimg_dataset/validation"

from tensorflow.keras.preprocessing.image import ImageDataGenerator
batches = ImageDataGenerator().flow_from_directory(directory=train_root_path, target_size=(64,64), batch_size=10000)
```

```
Found 8404 images belonging to 25 classes.
```

```
batches.class_indices
```

```
imgs, labels = next(batches)
imgs.shape
```

```
(8404, 64, 64, 3)
```

```
labels.shape
```

```
(8404, 25)
```

```
# plots images with labels within jupyter notebook
def plots(ims, figsize=(20,30), rows=10, interp=False, titles=None):
    if type(ims[0]) is np.ndarray:
        ims = np.array(ims).astype(np.uint8)
        if (ims.shape[-1] != 3):
            ims = ims.transpose((0,2,3,1))
    f = plt.figure(figsize=figsize)
    cols = 10 # len(ims)//rows if len(ims) % 2 == 0 else len(ims)//rows + 1
    for i in range(0,50):
        sp = f.add_subplot(rows, cols, i+1)
        sp.axis('Off')
        if titles is not None:
            sp.set_title(list(batches.class_indices.keys())[np.argmax(titles[i])], fontsize=16)
        plt.imshow(ims[i], interpolation=None if interp else 'none')
```

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(imgs/255.,labels, test_size=0.3)
```

```
x_train.shape
```

```
(5882, 64, 64, 3)
```

```
y_test.shape
```

```
(2522, 25)
```

```
num_classes = len(classes)
# we do not need to apply one-hot encoding to the labels as in https://keras.io/examples/vision/swin_transformers/
# since the dataset data is already prepared for the multi-class classification
#y_train = keras.utils.to_categorical(y_train, num_classes)
#y_test = keras.utils.to_categorical(y_test, num_classes)
print(f"x_train shape: {x_train.shape} - y_train shape: {y_train.shape}")
print(f"x_test shape: {x_test.shape} - y_test shape: {y_test.shape}")
```

```
x_train shape: (5882, 64, 64, 3) - y_train shape: (5882, 25)
x_test shape: (2522, 64, 64, 3) - y_test shape: (2522, 25)
```

### Swin Transformer V2 Model

```
from keras_cv_attention_models import swin_transformer_v2
# requires
input_shape=(64,64,3)  # (64,64,3)
num_epochs = 40
batch_size = 128

learning_rate = 1e-3

num_epochs = 40 ## TBD - to increase?
validation_split = 0.1
weight_decay = 0.0001
label_smoothing = 0.1

model = swin_transformer_v2.SwinTransformerV2Tiny_window8(
    input_shape=input_shape,
    num_classes=num_classes,
    classifier_activation="softmax",
    pretrained="imagenet")
```

```
import tensorflow_addons as tfa
model.compile(
    loss=keras.losses.CategoricalCrossentropy(label_smoothing=label_smoothing),
    optimizer=tfa.optimizers.AdamW(
        learning_rate=learning_rate, weight_decay=weight_decay
    ),
    metrics=[
        keras.metrics.CategoricalAccuracy(name="accuracy"),
        keras.metrics.TopKCategoricalAccuracy(5, name="top-5-accuracy"),
    ],
)

# model.summary()
```

```
lr_reduction = ReduceLROnPlateau(monitor='val_accuracy',patience=4, verbose=1,  factor=0.4, min_lr=0.0001)

early_stop = EarlyStopping(monitor='val_accuracy', min_delta=0.00001, patience=8, mode='auto', restore_best_weights=True)

model_fit = model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=num_epochs,
    validation_split=validation_split,
    verbose =1,
    callbacks=[early_stop,lr_reduction]
)
```

```
model.evaluate(x_test,  y_test)
```

```
79/79 [==============================] - 29s 367ms/step - loss: 0.6920 - accuracy: 0.9754 - top-5-accuracy: 0.9968
[0.6919856667518616, 0.975416362285614, 0.9968279004096985]
```

## Make predictions

```
pred_x = model.predict(x_test, verbose=0)
y_pred=np.argmax(pred_x,axis=1)
y_pred
```

```
array([ 3, 20,  3, ..., 11,  2,  3], dtype=int64)
```

```
y_test2 = np.argmax(y_test, axis=1)
y_test2
```

```
array([ 3, 20,  3, ..., 11,  2,  3], dtype=int64)
```

```
from sklearn import metrics
c_matrix = metrics.confusion_matrix(y_test2, y_pred)
```

```
import seaborn as sns
def confusion_matrix(confusion_matrix, class_names, figsize = (10,7), fontsize=14):
    df_cm = pd.DataFrame(
        confusion_matrix, index=class_names, columns=class_names,
    )
    fig = plt.figure(figsize=figsize)
    try:
        heatmap = sns.heatmap(df_cm, annot=True, fmt="d")
    except ValueError:
        raise ValueError("Confusion matrix values must be integers.")
    heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsize=fontsize)
    heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=45, ha='right', fontsize=fontsize)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
class_names= batches.class_indices.keys()
confusion_matrix(c_matrix, class_names, figsize = (20,7), fontsize=14)
```

```
from sklearn import metrics
# Print the precision and recall, among other metrics
report =  metrics.classification_report(y_test2, y_pred, digits=3, output_dict=True)

df = pd.DataFrame(report).transpose().reset_index()
df = df.rename(columns={"index": "class_label"})
```

Let's review the part of the classification report related to each individual class

```
clf_rep = metrics.precision_recall_fscore_support(y_test2, y_pred)
out_dict = {
             "precision" : clf_rep[0].round(3)
            ,"recall" : clf_rep[1].round(3)
            ,"f1-score" : clf_rep[2].round(3)
            ,"support" : clf_rep[3]
            }
out_df = pd.DataFrame(out_dict).reset_index().rename(columns={"index": "class_label"})
class_label_values = dict(zip(range(0,len(batches.class_indices)), batches.class_indices))
out_df['class_label'] = out_df['class_label'].map(class_label_values)
out_df
```

```
# display aggregated values - selecting rows based on condition
options = ['accuracy', 'macro avg',  'weighted avg']
agg_df = df[df['class_label'].isin(options)]
agg_df
```

## Додаток Г. Програмний код основних програмних модулів, побудованих на архітектурі згорткової мережі CoAtNet

```
#setup
import sys
import os
from math import log
import numpy as np
import pandas as pd
import scipy as sp
from PIL import Image
import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.callbacks import ReduceLROnPlateau,EarlyStopping
```

### Data Preprocessing and Basic EDA

```
# prepare the data

train_root_path = "./malimg_dataset/train"
val_root_path = "./malimg_dataset/validation"

from tensorflow.keras.preprocessing.image import ImageDataGenerator
batches = ImageDataGenerator().flow_from_directory(directory=train_root_path, target_size=(64,64), batch_size=10000)
```

```
Found 8404 images belonging to 25 classes.
```

```
batches.class_indices
```

```
imgs, labels = next(batches)
imgs.shape
```

```
(8404, 64, 64, 3)
```

```
labels.shape
```

```
(8404, 25)
```

```
# plots images with labels within jupyter notebook
def plots(ims, figsize=(20,30), rows=10, interp=False, titles=None):
    if type(ims[0]) is np.ndarray:
        ims = np.array(ims).astype(np.uint8)
        if (ims.shape[-1] != 3):
            ims = ims.transpose((0,2,3,1))
    f = plt.figure(figsize=figsize)
    cols = 10 # len(ims)//rows if len(ims) % 2 == 0 else len(ims)//rows + 1
    for i in range(0,50):
        sp = f.add_subplot(rows, cols, i+1)
        sp.axis('Off')
        if titles is not None:
            sp.set_title(list(batches.class_indices.keys())[np.argmax(titles[i])], fontsize=16)
        plt.imshow(ims[i], interpolation=None if interp else 'none')
```

```
plots(imgs, titles = labels)
```

```
classes = batches.class_indices.keys()
perc = (sum(labels)/labels.shape[0])*100

plt.xticks(rotation='vertical')
plt.bar(classes,perc)
```

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(imgs/255.,labels, test_size=0.3)
```

```
x_train.shape
```

```
(5882, 64, 64, 3)
```

```
y_test.shape
```

```
(2522, 25)
```

```
num_classes = len(classes)
```

```
print(f"x_train shape: {x_train.shape} - y_train shape: {y_train.shape}")
print(f"x_test shape: {x_test.shape} - y_test shape: {y_test.shape}")
```

```
x_train shape: (5882, 64, 64, 3) - y_train shape: (5882, 25)
x_test shape: (2522, 64, 64, 3) - y_test shape: (2522, 25)
```

## CoatNet Model

```
from keras_cv_attention_models import coatnet
# requires
input_shape=(64,64,3)  # (64,64,3)
num_epochs = 40

batch_size = 128 # 128

learning_rate = 1e-3

num_epochs = 40
validation_split = 0.1
weight_decay = 0.0001
label_smoothing = 0.1

model = coatnet.CoAtNet0(
    input_shape=input_shape,
    num_classes=num_classes,
    drop_connect_rate=0.2,
    classifier_activation="softmax")
```

```
import tensorflow_addons as tfa
model.compile(
    loss=keras.losses.CategoricalCrossentropy(label_smoothing=label_smoothing),
    optimizer=tfa.optimizers.AdamW(
        learning_rate=learning_rate, weight_decay=weight_decay
    ),
    metrics=[
        keras.metrics.CategoricalAccuracy(name="accuracy"),
        keras.metrics.TopKCategoricalAccuracy(5, name="top-5-accuracy"),
    ],
)

# model.summary()
```

```
lr_reduction = ReduceLROnPlateau(monitor='val_accuracy',patience=4, verbose=1,  factor=0.4, min_lr=0.0001)

early_stop = EarlyStopping(monitor='val_accuracy', min_delta=0.00001, patience=8, mode='auto', restore_best_weights=True)

model_fit = model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=num_epochs,
    validation_split=validation_split,
    verbose =1,
    callbacks=[early_stop,lr_reduction]
)
```

```
model.evaluate(x_test,  y_test)
```

```
79/79 [==============================] - 27s 341ms/step - loss: 0.6732 - accuracy: 0.9877 - top-5-accuracy: 0.9972
[0.6731753349304199, 0.9877081513404846, 0.9972244501113892]
```

## Make predictions

```
pred_x = model.predict(x_test, verbose=0)
y_pred=np.argmax(pred_x,axis=1)
y_pred
```

```
array([ 2,  2,  2, ..., 11, 24,  6], dtype=int64)
```

```
y_test2 = np.argmax(y_test, axis=1)
y_test2
```

```
array([ 2,  2,  2, ..., 11, 24,  6], dtype=int64)
```

```
from sklearn import metrics
c_matrix = metrics.confusion_matrix(y_test2, y_pred)
```

```
import seaborn as sns
def confusion_matrix(confusion_matrix, class_names, figsize = (10,7), fontsize=14):
    df_cm = pd.DataFrame(
        confusion_matrix, index=class_names, columns=class_names,
    )
    fig = plt.figure(figsize=figsize)
    try:
        heatmap = sns.heatmap(df_cm, annot=True, fmt="d")
    except ValueError:
        raise ValueError("Confusion matrix values must be integers.")
    heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsize=fontsize)
    heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=45, ha='right', fontsize=fontsize)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
from sklearn import metrics
# Print the precision and recall, among other metrics
report =  metrics.classification_report(y_test2, y_pred, digits=3, output_dict=True)

df = pd.DataFrame(report).transpose().reset_index()
df = df.rename(columns={"index": "class_label"})
```

Let's review the part of the classification report related to each individual class

```
clf_rep = metrics.precision_recall_fscore_support(y_test2, y_pred)
out_dict = {
             "precision" : clf_rep[0].round(3)
            ,"recall" : clf_rep[1].round(3)
            ,"f1-score" : clf_rep[2].round(3)
            ,"support" : clf_rep[3]
            }
out_df = pd.DataFrame(out_dict).reset_index().rename(columns={"index": "class_label"})
class_label_values = dict(zip(range(0,len(batches.class_indices)), batches.class_indices))
out_df['class_label'] = out_df['class_label'].map(class_label_values)
out_df
```

```
# display aggregated values - selecting rows based on condition
options = ['accuracy', 'macro avg',  'weighted avg']
agg_df = df[df['class_label'].isin(options)]
agg_df
```