МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ЮКАЛЬЧУК АНДРІЙ ІВАНОВИЧ

Допускається до захисту:
В. о. завідувача кафедри
прикладної математики та
кібербезпеки, доктор
філософії з математики,
_____ Луценко А.В.
«__»_____20__ р.

РОЗРОБКА АРХІТЕКТУРИ НЕЙРОННОЇ МЕРЕЖІ ДЛЯ ЗАПОБІГАННЯ ТА ВИЯВЛЕННЯ РІЗНИХ ТИПІВ КІБЕРАТАК НА МЕРЕЖЕВІ РЕСУРСИ В ПЕРІОД ВІЙСЬКОВОГО СТАНУ

Спеціальність 105 Прикладна фізика та наноматеріали

Кваліфікаційна (магістерська) робота

Науковий керівник:
Л. В. Загоруйко
канд. техн. наук, доцент,
доцент кафедри прикладної
математики та кібербезпеки,
_____
(підпис)

Оцінка:___/___/_____
(бали за шкалою ЄКТС/за національною шкалою)
Голова ЕК:_____
(підпис)

Вінниця - 2025

АНОТАЦІЯ

Юкальчук А. І. Розробка архітектури нейронної мережі для запобігання та виявлення різних типів кібератак на мережеві ресурси в період військового стану. Спеціальність 105 «Прикладна фізика та наноматеріали» освітня програма «Технології інтернету речей». Донецький національний університет імені Василя Стуса, Вінниця, 2025.

У кваліфікаційній (магістерській) роботі досліджується різні типи нейронних мереж для попередження і виявлення кібератак на підприємства критичної інфраструктури, а також аналіз їх ефективності та програмна реалізація запропонованих архітектур мовою програмування *Python*.

Ключові слова: нейронна мережа, комп'ютерна атака, DOS, R2L, KDD99, U2R, навчання нейромережі.

94 с., 20 рис., 1 дод., 51 джерел.

ABSTRACT

Yukalchuk A. I. Development of neural network architecture for the prevention and detection of various types of cyberattacks on network resources during martial law. Specialty 105 "Applied Physics and Nanomaterials" educational program "Internet of Things Technologies." Vasyl Stus Donetsk National University, Vinnytsia, 2025.

This qualification (master's) thesis explores various types of neural networks for preventing and detecting cyberattacks on critical infrastructure enterprises, as well as analyzing their effectiveness and software implementation of the proposed architectures in the Python programming language.

Keywords: neural network, computer attack, DOS, R2L, KDD99, U2R, neural network training.

94 p., 20 figures, 1 appendix, 51 sources.

ЗМІСТ

ВСТУП

Сучасні комп’ютерні мережі є критичною складовою функціонування державних установ, підприємств і інфраструктури — особливо в умовах воєнного стану, коли від надійності інформаційних сервісів залежить безпека й стійкість життєво важливих процесів. Наслідки успішних кібератак можуть бути катастрофічними: компрометація конфіденційних даних, порушення працездатності сервісів, блокування доступу або повна зупинка ключових систем, що в умовах бойових дій загрожує не лише економічним втратам, а й безпеці населення. У зв’язку з цим зростає потреба в ефективних інтелектуальних системах запобігання та виявлення вторгнень, здатних працювати в реальному часі та адаптуватися до нових типів атак; водночас їх практична корисність залежить від наявності якісних даних для навчання й адекватного тестування в умовах, наближених до реальних.

У рамках цієї роботи обґрунтовано підхід до розробки архітектури нейронної мережі для запобігання та виявлення різних типів кібератак на мережеві ресурси в період воєнного стану. Особлива увага приділена проблемі виявлення рідкісних і критичних категорій атак, а також питанням побудови тестового середовища та оцінки ефективності алгоритмів на репрезентативних наборах даних. Для обґрунтування вибору моделей і методів використано аналіз сучасних досліджень у галузі і експериментальну перевірку на стандартному датасеті KDD99 із урахуванням підбору архітектур, механізмів балансування класів та процедур валідації.

Актуальність. В умовах постійного зростання складності й інтенсивності кібератак, особливо під час воєнних дій, критично важливими є питання оцінки й підвищення ефективності систем захисту значущих об’єктів критичної інфраструктури; необхідні адаптивні рішення, здатні виявляти як масові, так і рідкісні атаки.

Мета дослідження. розробка архітектури нейронної мережі для ефективного запобігання та виявлення різних типів кібератак на мережеві

ресурси в умовах воєнного стану, із забезпеченням балансу між точністю класифікації та здатністю виявляти рідкісні типи вторгнень.

Для досягнення поставленої мети потрібно виконати наступні завдання:

1) Провести аналіз сучасних досліджень та методів використання технологій глибокого навчання (Deep Learning) для задач кібербезпеки та виявлення вторгнень.
2) Здійснити класифікацію та порівняльний аналіз архітектур нейронних мереж (MLP, CNN, SOM, мережі Хопфілда), визначити їхні переваги та недоліки в контексті роботи з мережевим трафіком.
3) Обґрунтувати вибір та розробити програмну реалізацію декількох архітектур нейронних мереж, зокрема гібридних моделей, для детекції атак.
4) Провести експериментальні дослідження на тестових наборах даних (KDD99), оцінити ефективність запропонованих моделей за метриками точності (accuracy) та повноти (recall) для різних класів атак та визначити найбільш ефективну архітектуру.

Практична значимість. Результати роботи можуть бути використані при проектуванні й тестуванні IDS для об’єктів критичної інфраструктури, зокрема для вибору архітектури, налаштування механізмів обробки дисбалансу класів і побудови тестових макетів.

Об’єкт дослідження. Алгоритми машинного навчання та нейронні архітектури, застосовувані для задач детекції та класифікації мережевого трафіку.

Предмет дослідження. Методики підготовки даних, особливості архітектурних рішень та критерії оцінки ефективності нейромереж у виявленні різних типів кібератак у складних умовах експлуатації мереж.

Перелік умовних скорочень. IDS (Intrusion Detection System) – система виявлення вторгнень (СВВ). HIDS (Host-based IDS) – хост-орієнтована система виявлення вторгнень. NIDS (Network IDS) – мережева система виявлення вторгнень. DoS (Denial of Service) – атака типу «відмова в обслуговуванні».

DDoS (Distributed Denial of Service) – розподілена атака типу «відмова в обслуговуванні». R2L (Remote to Local) – віддалена атака для отримання локального доступу. U2R (User to Root) – атака підвищення привілеїв користувача до рівня адміністратора. КС – комп'ютерні системи. ШНМ – штучні нейронні мережі. MLP (Multi-Layer Perceptron) – багатошаровий перцептрон. CNN (Convolutional Neural Network) – згорткова нейронна мережа (ЗНМ). RNN (Recurrent Neural Network) – рекурентна нейронна мережа. LSTM (Long Short-Term Memory) – довга короткочасна пам'ять. GRU (Gated Recurrent Unit) – керований рекурентний блок. SOM (Self-Organizing Map) – самоорганізована карта (карта Кохонена). РБНМ (RBFN, Radial Basis Function Network) – радіально-базисна нейронна мережа. DNN (Deep Neural Network) – глибока нейронна мережа. GNN (Graph Neural Network) – графова нейронна мережа. AE (Autoencoder) – автокодер. ART (Adaptive Resonance Theory) – теорія адаптивного резонансу. BMU (Best Matching Unit) – нейрон-переможець (у картах Кохонена). IoT (Internet of Things) – інтернет речей.

Апробація. За матеріалами магістерської роботи опубліковано:

1) Юкальчук А. І. Структура нейронної мережі для попередження та виявлення кібератак типу User to Root та Remote to Local на підприємства критичної інфраструктури. Вісник Студентського наукового товариства ДонНУ імені Василя Стуса. 2025. Вип. 17, Т. 1. С. 238–242. URL: https://jvestnik-sss.donnu.edu.ua/article/view/17340
2) Юкальчук А. І., Загоруйко Л. В. Аналіз застосування нейронних мереж для запобігання та виявлення різних типів кібератак на мережеві ресурси в період військового стану. Вісник Студентського наукового товариства ДонНУ імені Василя Стуса. 2025. Вип. 17, Т. 2. С.
3) Юкальчук А. І., Загоруйко Л. В. Порівняння ефективності застосування різних архітектур нейронних мереж для запобігання та виявлення кібератак на мережеві ресурси в період військового стану. Матеріали IV Міжнародної науково-практичної конференції (м. Вінниця, 05 листопада 2025 р.). Вінниця : ДонНУ імені Василя Стуса, 2025. С.

4) Підсумкова науково-практична конференція міжнародного конкурсу студентських наукових робіт зі штучного інтелекту на базі КПІ ім. Ігоря Сікорського.

РОЗДІЛ 1. АНАЛІЗ СУЧАСНИХ ДОСЛІДЖЕНЬ

1.1 Аналіз сучасних досліджень використання нейронних мереж для попередження та виявлення кібератак

В літературі існує ряд робіт, пов'язаних з виявлення вторгнень на основі глибокого навчання. Останніми роками багато досліджень у сфері кібербезпеки зосереджуються на виявленні вторгнень за допомогою ШІ, а для покращення здатності виявляти кіберзагрози були запропоновані різні методи на основі ШІ та машинного навчання. Хоча ці дослідження досягли значних результатів, використовуючи методи ШІ та машинного навчання, вони все ще обмежуються конкретними тестовими наборами даних, такими як NSLKDD та KDD99. Інші дослідження, однак, використовували події безпеки та журнали, зібрані з реального світу. Ці дослідження ближчі до нашого дослідження з точки зору вирішення вищезгаданих проблем.

Hleha та ін [1]. досліджували оптимізацію методів Explainable Artificial Intelligence (XAI) для створення нейронних систем виявлення вторгнень з мінімальною затримкою. У своїй роботі автори провели системний огляд та порівняльний аналіз різних архітектур глибокого навчання (CNN, LSTM, GRU, автокодери, гібридні CNN-LSTM моделі) та техніки XAI (SHAP, LIME, Integrated Gradients, DeepLIFT, Grad-CAM, Anchors). Експериментальна перевірка на наборах даних CICIDS2017, NSL-KDD та UNSW-NB15 показала, що гібридні моделі CNN-LSTM з механізмами уваги (ELAI framework) досягли точності виявлення більше 98% з часом висновку менше 10 мс, та поліпшили виявлення атак нульового дня до 91,6%.

Chao та Xie [2] представили DeepNetGuard, інноваційний алгоритм глибокого навчання для ефективного виявлення потенційних загроз безпеки у великомасштабному трафіку мережі. Модель використовує багатовекторну стратегію вилучення ознак, яка поєднує базові, статистичні та поведінкові характеристики, та інтегрує технології автокодера та генеративної змаганнєвої

мережі для виявлення як відомих, так і невідомих загроз. DeepNetGuard продемонстрував переважну продуктивність порівняно з традиційними системами виявлення та іншими моделями глибокого навчання, досягнувши високих показників точності, повноти, точності та F1 у середовищах внутрішніх мереж та центрів обробки даних.

Vincent та Ibidun [3] покращили системи виявлення вторгнень шляхом інтеграції Explainable Artificial Intelligence (XAI) для вирішення проблеми непрозорості та підвищення інтерпретабельності та надійності при збереженні високої предиктивної продуктивності. Використовуючи набір даних UNSW-NB15, вони розробили та оцінили декілька моделей машинного навчання, включаючи дерева рішень, багатошарові перцептрони, XGBoost, випадкові ліси, CatBoost, логістичну регресію та наївний байєсів класифікатор. Моделі XGBoost та CatBoost досягли найвищої точності 87% з помилковою позитивною та негативною ставками 0,07 та 0,12, відповідно, при одночасному забезпеченні дій інтерпретабельності через техніку XAI.

Maureen Okafor [4] дослідила застосування техніки глибокого навчання, включаючи згорткові та рекурентні нейронні мережі, у виявленні загроз у реальному часі та реагуванні на них. Гібридні архітектури CNN-RNN продемонстрували переважну продуктивність порівняно з традиційними підходами машинного навчання завдяки їхній здатності використовувати просторові та часові ознаки для детектування складних кіберзагроз з високою точністю, точністю та повнотою на різних наборах даних.

Jyothi та ін. [5] запропонували нову оптимізовану модель нейронної мережі для виявлення кібератак з використанням покращеного алгоритму оптимізації китів (EWOA). Модель EWOA-ANN була розроблена для вирішення проблем атак підстановки облікових даних, виявлення відмов і прогнозування. Експериментальні результати показали високу ефективність моделі у виявленні кібератак в режимі реального часу.

Ojo та ін. [6] провели дослідження з оцінки варіантів глибокого навчання для виявлення кібератак та багатокласової класифікації в IoT-мережах.

Дослідження використовує новий датасет CICIoT2023 з 47 ознаками та 33 типами атак. Автори порівняли варіанти моделей глибокого навчання (DNN, CNN та RNN) та продемонстрували ефективність запропонованого підходу у точному прогнозуванні кібератак.

Ahmad та ін. [7] представили адаптивні нейронні мережі для кібербезпеки на основі еволюційного підходу для покращення виявлення та класифікації атак. Автори запропонували новий Multi-Layer Perceptron (MLP) тренер, що використовує методи еволюційних обчислень для динамічної оптимізації ваг і зсувів мережі. Модель була протестована на п'яти визнаних датасетах: NSL-KDD, CICIDS2017, UNSW-NB15, Bot-IoT та CSE-CIC-IDS2018, показавши перевагу над 10 сучасними алгоритмами оптимізації.

1.2 Аналіз використання спеціалізованих архітектур та методів захисту від змагальних атак в сучасних дослідженнях

Окремий напрямок досліджень стосується захисту самих нейромереж та роботи в специфічних умовах.

Barr [8] розробив надійну CNN проти adversarial атак для застосування на ресурсно-обмежених IoT-пристроях. Дослідження використовує інноваційну техніку APE-GAN для повторного навчання CNN, значно покращуючи її стійкість проти просунутих adversarial методів, таких як Deepfool та L-BFGS, на датасеті MNIST.

Alzaidy та Binsalleeh [9] запропонували методи adversarial атак з механізмами захисту на CNN та RNN для класифікації шкідливого ПЗ. Автори провели white-box атаки JSMA та C&W на Windows PE файли та оцінили два механізми захисту: дистиляцію та adversarial навчання, досягаючи загального рівня неправильної класифікації 73,5% після застосування padding.

Dalal та ін. [10] запропонували Extremely Boosted Neural Network для більш точного прогнозування багатоетапних кібератак у хмарному середовищі. Модель досягла точності 99,72% на Multi-Step Cyber-Attack Dataset (MSCAD),

значно перевершуючи Quest Model (94,09%), Bayesian Network (97,29%) та стандартну Neural Network (99,09%).

Wang та ін. [11] запропонували DDoS-MSCT - метод виявлення DDoS-атак на основі багатомасштабної згортки та трансформера. Архітектура вводить блок DDoS-MSCT, що складається з модуля локального вилучення ознак (LFEM) та модуля глобального вилучення ознак (GFEM), досягаючи точності 99% на датасеті CIC-DDoS2019 з 100% precision, recall та F1-score.

Neto та ін. [12] розробили CICIoT2023 - датасет для large-scale атак в IoT середовищі в реальному часі. Датасет використовує екстенсивну топологію з 105 реальних IoT пристроїв та включає 33 атаки, поділені на 7 класів (DDoS, DoS, Recon, Web-based, Brute Force, Spoofing та Mirai), забезпечуючи реалістичне середовище для тестування ML та DL алгоритмів.

Saini та співавтори провели всебічний огляд дуальності адверсаріального навчання в мережевому виявленні вторгнень, досліджуючи атаки та контрзаходи. Їх дослідження показало, що графові нейронні мережі (GNN) демонструють особливу перспективність для виявлення мережевих вторгнень завдяки їх здатності обробляти складні залежності та взаємозв'язки в мережевих даних. Автори підкреслили важливість розвитку стійких механізмів захисту для протидії потенційним порушенням мережевої безпеки та приватності, викликаним адверсаріальними атаками [13].

Zhong та співавтори провели всебічне дослідження застосування графових нейронних мереж для систем виявлення вторгнень. Їх систематична таксономія класифікує існуючі та майбутні дослідження методів виявлення вторгнень на основі GNN, підкреслюючи переваги цих архітектур у захопленні складних взаємозв'язків у структурованих графом даних [14].

Chhetri та Namin дослідили застосування трансформерних моделей для прогнозування наслідків кібератак. Їх робота демонструє ефективність архітектури BERT та ієрархічних мереж уваги (HAN) для аналізу текстових описів кібератак та прогнозування їх потенційного впливу, що є важливим кроком у напрямку автоматизації оцінки загроз [15].

Sharma запропонував реалізацію моделі, яка використовує аналіз мережевого трафіку (NTA) та дані лічильників продуктивності апаратного забезпечення (HPC) для точного позначення zero-day атак. Дослідження спрямоване на створення моделі на основі автокодера, яка об'єднує апаратні та мережеві характеристики для ефективної класифікації, використовуючи стандартні набори даних кібербезпеки CICIDS2017, NSL-KDD та D.A.V.I.D.E HPC [16].

Rasikha та співавтори розробили ансамблевий метод глибокого навчання для виявлення кібератак, підкресливши, що методи глибокого навчання є надзвичайно бажаними для виявлення кібератак, оскільки вони не потребують інженерії ознак або припущень про розподіл даних. Їх підхід продемонстрував високу ефективність у виявленні різноманітних типів атак в IoT-мережах [17].

Ахмед Ахмім та ін. [18] запропонували нову ієрархічну IDS, яка базується на моделях на основі дерева рішень та правил. Вони також використовували CICIDS2017 як набір даних для оцінки продуктивності своєї моделі. Запропонована ними модель поєднує в собі дерево скороченого обрізання помилок (REP Tree) та алгоритм JRip на першому етапі. На цьому етапі вхідні ознаки набору даних використовуються як вхідні дані для класифікації трафіку як атаки або доброякісний трафік. Потім класифікатор Forest PA використовує результати роботи двох класифікаторів на першому етапі в поєднанні з вхідними ознаками початкового набору даних для отримання остаточного результату класифікації. Їх модель досягла пристойної продуктивності майже для всіх класів руху в CICIDS2017. Вони також надали порівняння ефективності запропонованої ними моделі з 11 відомими класифікаторами, щоб підтвердити її класифікаційну здатність. Серед 12 моделей класифікаторів їхня модель показала найкращі результати класифікації для семи класів атак і найнижчий рівень хибних спрацьовувань для безпечного трафіку. Ця модель є конкурентоспроможною завдяки своїй високій загальній ефективності класифікації на CICIDS2017. Тому в розділі результатів цієї статті

запропонована модель IDS порівнюється з їхньою новою ієрархічною IDS, щоб оцінити ефективність запропонованої моделі.

Tang та ін. [19] пропонують модель DNN для виявлення аномалій на основі потоку. Їх перша спроба застосувати DNN для мережевої безпеки призвела до створення відносно простої DNN, яка складається з одного вхідного шару, трьох прихованих шарів і одного вихідного шару. Деякі експерименти були проведені на наборі даних NSL-KDD, де було доведено, що запропонована модель DNN здатна виявляти атаки «нульового дня» і поводиться краще, ніж інші методи машинного навчання.

Щоб розширити можливості DNN, Li та ін. [20] пропонують нову мережеву структуру під назвою HashTran-DNN для класифікації шкідливого програмного забезпечення для Android. Їхня найбільша інновація полягає у перетворенні вхідних зразків за допомогою хеш-функцій для збереження характеристик локальності. Після перетворення вхідних даних HashTran-DNN використовує AE для виконання завдання згладжування, щоб класифікатор DNN міг отримати інформацію про локалізацію в потенційному просторі для кращої продуктивності. Проаналізувавши результати експерименту, можна помітити, що HashTran-DNN може ефективно захищатися від чотирьох спеціальних тестових атак, тоді як стандартний DNN не може виявити всі ці атаки.

Для мережевого адміністратора нагальним завданням є запобігання вторгненню зловмисних мережевих хакерів та підтримання мережевої системи та комп'ютера в безпечному та нормальному робочому стані. Peng та ін. [21] пропонують метод виявлення мережевих вторгнень на основі глибокого навчання, який використовує глибоку нейронну мережу для вилучення особливостей даних моніторингу мережі, а нейронна мережа BP використовується для класифікації типів вторгнень. Метод оцінено на наборі даних KDDCup 99. Результати показують, що метод досягає точності 95,45%, і він має значне покращення порівняно з традиційним методом машинного навчання.

Kolosnjaji та ін. [22] намагалися побудувати нейронну мережу зі згортковим та рекурсивним мережевими шарами, яка отримує класифікаційні ознаки для моделювання системи виявлення шкідливого програмного забезпечення. За допомогою запропонованого ними методу вони отримують ієрархічну архітектуру вилучення ознак, яка поєднує в собі переваги операції згортки від згорткового шару та моделювання послідовності від рекурсивного мережевого шару. Згодом Kolosnjaji та ін. [23] розвинули цей метод, включивши до нього ознаки, отримані з заголовків портативних виконуваних файлів, що дозволило досягти досить високої точності та швидкості відкликання у випадках злиття даних.

1.3 Висновки до розділу 1

Сучасні дослідження підтверджують високу ефективність методів глибокого навчання для виявлення та прогнозування кібератак — від CNN, LSTM і автокодерів до гібридних і ієрархічних архітектур — з хорошими показниками на еталонних наборах (KDD99, NSL-KDD, CICIDS, UNSW-NB15 тощо). Водночас огляд виявляє типові обмеження: багато підходів оптимізовані під конкретні бенчмарки, мають проблеми з узагальненням на реальні логи, чутливістю до дисбалансу класів та інтерпретованістю рішень. Значну роль відіграють комбіновані рішення (гібриди, стекові підходи, відбір ознак), які часто підвищують стійкість і точність моделей. Отже, попри помітний прогрес, для практичного впровадження необхідні більш репрезентативні дані, стійкі методи обробки дисбалансу та дослідження масштабованості й пояснюваності моделей.

РОЗДІЛ 2. ПОНЯТТЯ СИСТЕМ ВИЯВЛЕННЯ ВТОРГНЕНЬТА ОСНОВНІ АРХІТЕКТУРИ НЕЙРОННИХ МЕРЕЖ

2.1 Поняття системи виявлення вторгнень

Вивчення подій, що відбувалися в межах комп'ютерної системи або мережі, з метою виявлення потенційних загроз і спроб несанкціонованого доступу, називається виявленням вторгнень. Цей процес дозволяє своєчасно визначити інциденти, що можуть становити небезпеку для інформаційної безпеки. Найчастіше такі загрози спричиняють зловмисники, які намагаються обійти захисні механізми системи, отримавши розширені привілеї задля досягнення власних цілей [24].

Система виявлення вторгнень (IntrusionDetectionSystem, IDS) є технічним рішенням — програмним або апаратним, — яке дозволяє фіксувати підозрілу активність, аналізувати її, запобігати розвитку загроз і повідомляти відповідальних осіб про події в режимі реального часу. IDS не замінює традиційні засоби безпеки, такі як брандмауери, але ефективно їх доповнює, забезпечуючи глибший рівень контролю та реагування.

Під час своєї роботи така система постійно спостерігає за поведінкою користувачів і мережевими процесами, фіксує відхилення від нормального функціонування, здатна розпізнавати характерні ознаки атак і вести детальну реєстрацію всіх інцидентів. Отримані дані аналізуються, і на їхній основі формується звіт, що дозволяє адміністраторам оперативно виявляти вразливості, коригувати налаштування безпеки та запобігати повторним загрозам. Крім цього, IDS виконує функції контролю за дотриманням політик користування системою, перевіряє цілісність важливих файлів і автоматично інформує системних адміністраторів про будь-які підозрілі дії, використовуючи різні канали сповіщення — від електронної пошти до спеціалізованих інтерфейсів.

2.1.1 Технології систем виявлення вторгнень

Інтуїтивно зрозуміло, що вторгнення в інформаційну систему - це дії, які порушують політику безпеки системи, а виявлення вторгнень - це процес, який використовується для виявлення цих дій. Виявлення вторгнень вивчається вже близько 20 років. Воно ґрунтується на переконанні, що поведінка зловмисника буде помітно відрізнятися від поведінки законного користувача і що багато несанкціонованих дій можна буде виявити. Системи виявлення вторгнень поділяються на три категорії. Нижче перераховані різні типи методів виявлення вторгнень.

Системи виявлення на основі сигнатур

Системи виявлення на основі сигнатур (також звані системами на основі зловживань) дуже ефективні проти відомих атак, але вони залежать від регулярного оновлення шаблонів атак; вони не здатні виявити раніше невідомі загрози або нові варіанти [25-29]. Однією з головних проблем IDS на основі підписів є те, що кожен підпис вимагає запису в базі даних, а повна база даних може містити сотні або навіть тисячі записів. Кожен пакет повинен порівнюватися з усіма записами в базі даних, що вимагає багато ресурсів, сповільнює пропускну здатність і робить IDS вразливою до DoS-атак. Деякі інструменти обходу IDS використовують цю вразливість, переповнюючи системи IDS на основі сигнатур надмірною кількістю пакетів, що призводить до тайм-ауту, втрати пакетів і, можливо, до пропуску атак. Крім того, цей тип IDS все ще вразливий до невідомих атак, оскільки для виявлення вторгнень він покладається виключно на наявні в базі даних підписи.

Системи виявлення на основі аномалій

Системи виявлення аномалій класифікують мережеву активність як нормальну або аномальну на основі правил або евристик, а не фіксованих сигнатур, і їх реалізація вимагає попереднього моделювання нормальної поведінки мережі [26,27]. На відміну від систем, заснованих на зловживаннях, системи, засновані на аномаліях, можуть виявляти раніше невідомі загрози, але

вони, як правило, дають більше помилкових спрацьовувань. Оскільки сигнатура нової атаки не відома, поки вона не буде виявлена і проаналізована, робити висновки на основі невеликої кількості пакетів складно. Системи на основі аномалій виявляють аномальну поведінку і генерують сигнали тривоги, коли вони спостерігають відхилення від нормальних шаблонів трафіку або поведінки додатків. Типові аномалії, які можуть бути зафіксовані, включають неправильне використання мережевих протоколів, наприклад, перекриття IP-фрагментів або запуск стандартного протоколу на нестандартному порту, нехарактерні шаблони трафіку, такі як непропорційно велика кількість UDP-пакетів у порівнянні з TCP-пакетами, підозрілі шаблони в корисному навантаженні додатків [27].

Основними проблемами систем виявлення аномалій є визначення нормальної поведінки мережі, встановлення відповідних порогових значень для запуску тривог і запобігання хибним спрацьовуванням. Користувачі-люди за своєю природою непередбачувані, і якщо нормальна модель не визначена ретельно, система буде генерувати багато помилкових тривог і страждати від погіршення продуктивності [27].

Системи виявлення на основі специфікацій

Системи виявлення на основі специфікацій контролюють процеси, порівнюючи фактичні потоки даних із заздалегідь визначеними програмними специфікаціями. У разі відхилення від цих специфікацій видається попередження. Щоб система залишалася ефективною проти відомих і невідомих атак, її необхідно підтримувати та оновлювати щоразу, коли змінюються програми спостереження. Підходи на основі специфікацій зазвичай генерують менше хибних спрацьовувань порівняно з системами виявлення аномалій [30].

2.1.2 Класифікація систем виявлення вторгнень

Розглядаючи джерело даних, що використовуються для виявлення вторгнень, IDS можна також класифікувати за типом системи, яку вони захищають: IDS на базі хоста (HIDS), IDS на базі мережі (NIDS) та гібридні IDS, що поєднують в собі обидва підходи [29]. Системи виявлення вторгнень в основному поділяються на три типи.

Хост-орієнтовані IDS (HIDS)

Хост-орієнтовані IDS встановлюються на окремих пристроях, таких як сервери або робочі станції, де дані аналізуються локально. Вони можуть використовувати методи виявлення як аномалій, так і зловживань [29]. Агенти, встановлені на контрольованих хостах, збирають дані з різних джерел, пишуть журнали і запускають тривоги на основі заздалегідь визначених політик. HIDS обмежуються моніторингом хостів, на яких встановлені агенти, і не можуть контролювати всю мережу. Вони зазвичай використовуються для захисту критично важливих серверів.

Недоліками HIDS є складність аналізу спроб вторгнення на декількох машинах, проблеми з підтримкою систем у мережах з різними операційними системами та конфігураціями, а також можливість того, що зловмисники можуть відключити HIDS після компрометації хоста.

Прикладами компонентів моніторингу HIDS є системи, які відстежують вхідні з'єднання з портами TCP або UDP (наприклад, PortSentry), системи, які перевіряють корисне навантаження пакетів на наявність шкідливого вмісту, і системи, які відстежують активність входу в систему для виявлення незвичайних шаблонів доступу. Такі продукти, як Snort, DragonSquire, EmeraldeXpert-BSM, NFR HID та IntruderAlert виконують моніторинг на основі хостів.

Мережеві IDS (NIDS)

Мережеві СЗІ зазвичай складаються з сенсорного пристрою з мережевою інтерфейсною картою, що працює в режимі проміскуїтету, і окремого

інтерфейсу управління. Розгорнуті в стратегічних точках мережевої інфраструктури, NIDS відстежують весь трафік в сегменті мережі, виявляючи відомі атаки шляхом зіставлення шаблонів і сканування на предмет аномальної активності. Також відомі як сніфери пакетів, NIDS перехоплюють і аналізують всі пакети, що проходять через сегмент, що контролюється, забезпечуючи захист всіх машин в цьому сегменті. Вони також можуть бути встановлені на активних мережевих пристроях, таких як маршрутизатори.

Деякі NIDS зосереджені на статистичному аналізі мережевого навантаження для виявлення атак типу flood, створюючи статистику трафіку без глибокої перевірки пакетів (наприклад, NovellAnalyzer, Microsoft NetworkMonitor). Типові комерційні продукти NIDS включають CiscoSecure IDS (раніше NetRanger), Hogwash, Dragon та E-Trust IDS.

Гібридні IDS

Гібридні IDS інтегрують дані управління та оповіщення як з хост, так і з мережевих пристроїв виявлення, поєднуючи сильні сторони кожного підходу для підвищення гнучкості та стійкості до атак [29]. Хоча мережеві IDS простіше розгортати та підтримувати, вони залежать від відомих сигнатур і можуть пропустити нові експлойти. IDS на основі хостів надають детальну інформацію про активність хостів, але вимагають кваліфікованого адміністрування. Гібридне рішення використовує обидва варіанти для покращення загальної здатності виявлення.

2.2 Класифікація архітектур штучних нейронних мереж для виявлення кібератак

Обробка параметрів мережевого трафіку з метою виявлення кібератак на інформаційно-комунікаційні системи, їх локалізація та аналіз динаміки змін в автоматичному режимі є надзвичайно важливим елементом будь-якої системи виявлення атак. Нейронні мережі широко і ефективно використовуються для вирішення завдань виявлення атак на інформаційні системи. Це область

досліджень, що інтенсивно розвивається. Постійно з'являються нові задачі, пов'язані з розробкою нових мережевих структур, метою яких є підвищення стійкості комп'ютерних систем (КС) до атак при повній автоматизації процедури їх виявлення [31].

Протягом останніх років однією з найактуальніших проблем в галузі інформаційної безпеки є підвищення ефективності методів виявлення атак на інформаційні ресурси комп'ютерних систем. При цьому важливим напрямком підвищення ефективності є «інтелектуалізація» методів виявлення атак за рахунок використання теорії штучних нейронних мереж (ШНМ). Перспективність такого підходу підтверджується деякими успішними застосуваннями ШНМ в інструментах виявлення атак (продукти Cisco) та великою кількістю відповідних теоретичних і практичних досліджень. Водночас, велика кількість хибних спрацьовувань, тривалий термін і нестабільність навчання, недостатня адаптація до багатьох особливостей сучасних КС, що пов'язано, в першу чергу, з методологічними недоліками, суттєво обмежують їх практичну цінність. Тому виникла необхідність вирішення проблеми, зумовленої перспективністю використання існуючих нейроподібних засобів виявлення атак, з одного боку, та недосконалістю методів розробки і застосування таких засобів, з іншого.

В цілому можна відзначити, що в останнє десятиліття спостерігається стрімкий розвиток ШНМ-технологій у сфері кібербезпеки. Різноманітними є не тільки можливості використання ШНМ для виявлення кібератак, але й їхня архітектура. Класифікаційний аналіз структур таких ШНМ для виявлення кібератак представлено на рис. 2.1.

Слід зазначити, що структуру будь-якого ШНМ, який використовується в багатьох практичних галузях і, зокрема, у виявленні атак кібербезпеки, можна розглядати як орієнтований граф з сильно зв'язаними ребрами, вершинами якого є штучні нейрони. За архітектурою зв'язків ШНМ можна згрупувати у два класи (рис. 2.1): мережі прямого поширення, в яких графи не мають петель, та рекурентні мережі, або мережі зі зворотним зв'язком

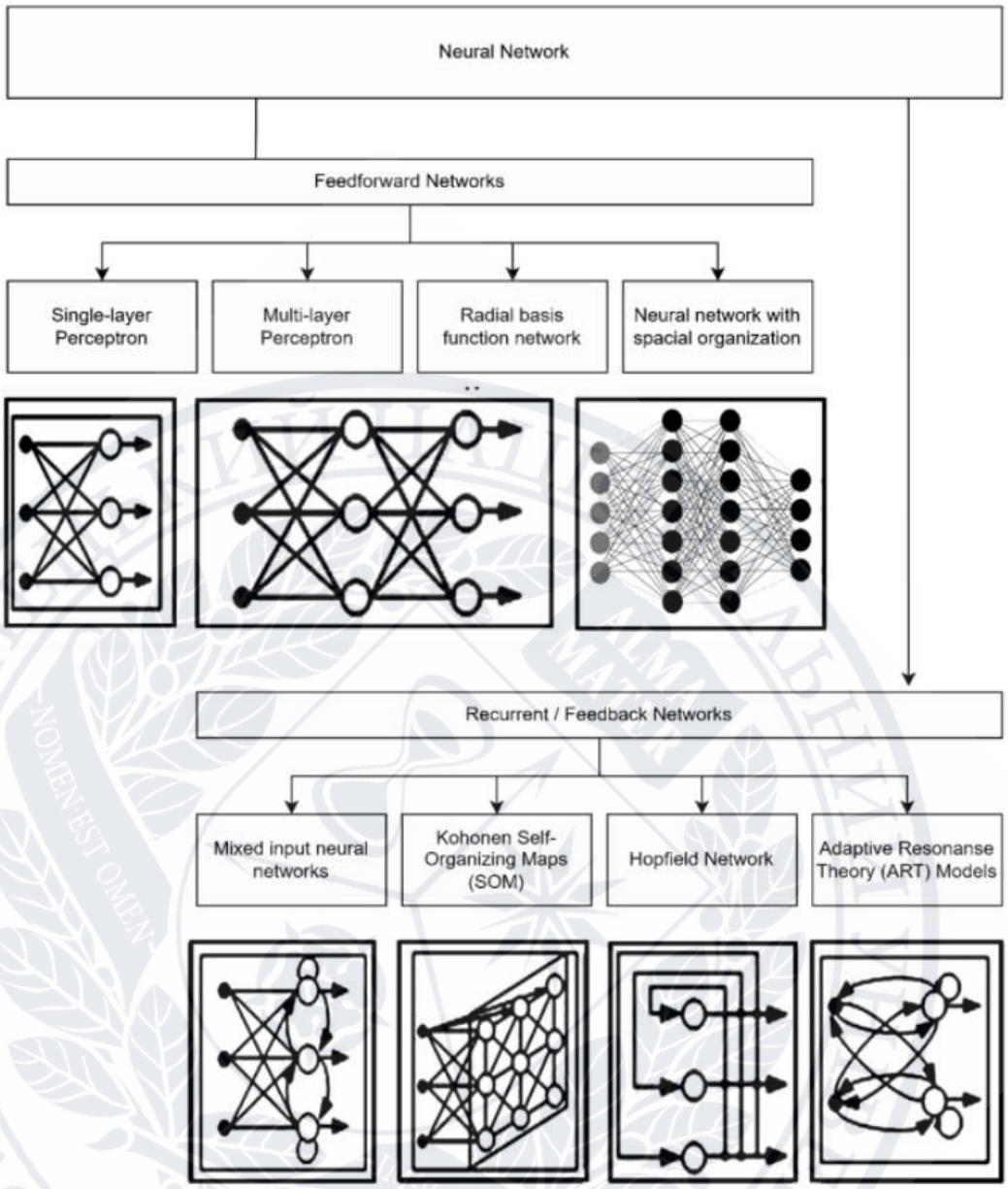

Рисунок 2.1- Класифікація архітектур штучних нейронних мереж для виявлення кібератак[31]

2.3 Нейронні мережі прямого поширення

Нейронні мережі прямого поширення сигналу — це клас штучних нейронних мереж, у яких нейрони згруповано в послідовні прошарки, а зв’язки між ними налаштовано таким чином, що інформація рухається в єдиному напрямку: від вхідного прошарку до вихідного. Вхідний прошарок приймає зовнішні дані (сигнал), перетворює їх у внутрішні представлення й передає

результуючі значення на входи нейронів наступного прошарку. У цих мережах відсутні зворотні або бічні зв’язки між нейронами всередині одного прошарку — кожен нейрон взаємодіє тільки з нейронами сусіднього попереднього або наступного прошарку [32,33].

Сховані прошарки — ті, що розташовані між вхідним та вихідним — виконують послідовне перетворення сигналу, поступово виокремлюючи важливі ознаки вхідних даних та створюючи дедалі абстрактніші представлення. Кожен нейрон схованого прошарку отримує на свій вхід лише сукупність результатів роботи всіх нейронів попереднього прошарку, а потім, застосувавши власну активаційну функцію (наприклад, сигмоїдну, ReLU або гіперболічний тангенс), передає своє значення далі. Цей поступовий «витиск» характеристик робить мережі прямого поширення надзвичайно ефективними для завдань класифікації, регресії, розпізнавання образів та прогнозування, адже вони здатні навчитися складним нелінійним залежностям між входом і виходом.

Останній, вихідний прошарок формує фінальний результат обчислень: значення з виходів його нейронів утворюють загальний вихідний сигнал, який може відповідати категорії об’єкта (у задачах класифікації), безперервному числовому прогнозу (у задачах регресії) або задати ймовірнісне розподілення варіантів рішення (при застосуванні софтмакс-активації). Нейронні мережі прямого поширення прості в архітектурі й навчанні — для налаштування їхніх ваг та зсувів зазвичай використовують алгоритм зворотного розповсюдження похибки (backpropagation) у поєднанні з методом градієнтного спуску. Завдяки цьому вони знайшли широку застосованість у штучному інтелекті — від обробки зображень і мовлення до фінансового моделювання та медичного діагнозу

2.3.1 Одношаровий перцептрон

Одношаровий персептрон — це найпростіша архітектура штучної нейронної мережі, що складається всього з одного формального нейрона. Незважаючи на свою простоту, він виконує роль базового «цеглинки» для більш складних моделей та демонструє основні принципи роботи нейронних мереж[33,34,35].

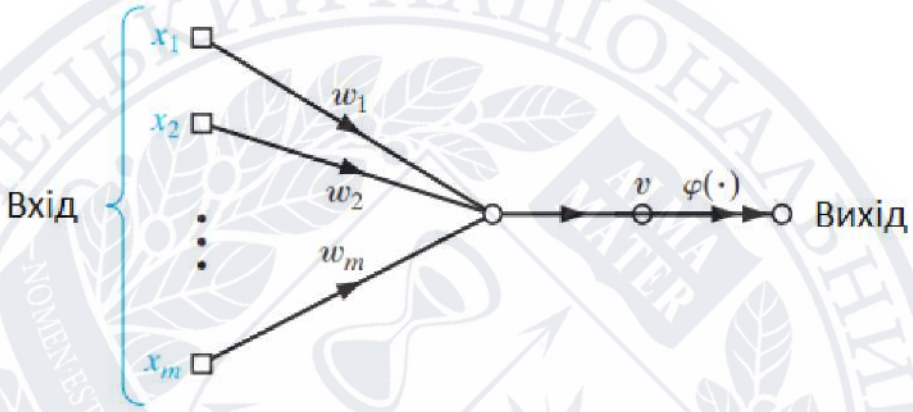

Рисунок 2.2 - Одношарова нейронна мережа[34]

У своїй роботі персептрон приймає набір вхідних сигналів, кожен з яких модифікується відповідною «вагою», а потім комбінація цих сигналів пропускається через активаційну функцію. Залежно від конфігурації, це може бути порогова функція (для дискретного персептрона) або гладка функція (наприклад, сигмоїдна). Результатом роботи одного нейрона виходить просте рішення типу «так/ні» або значення в інтервалі від нуля до одиниці.

Навчання однорівневогоперсептрона відбувається в ітеративному режимі: спочатку випадковим чином задаються початкові ваги, далі мережа послідовно обробляє навчальні зразки з відомою відповіддю, порівнює свій результат із бажаним і adjusts (на основі величини помилки) параметри так, щоб з часом підвищити точність класифікації. Цей процес триває доти, поки мережа не навчиться правильно розпізнавати усі приклади або не досягне встановленого критерію зупинки.

Існують два основні варіанти одношарового персептрона: дискретний, що видає чіткі класифікаційні рішення, та «дійсний» (continuous), який генерує неперервні вихідні значення і може краще працювати з нечіткими чи деталізованими даними. Для випадків, коли навчальні дані не є лінійно роздільними, застосовують модифіковані алгоритми на кшталт правила Уідроу–Хоффа: вони дозволяють поступово зменшувати крок корекції й мінімізувати середньоквадратичну помилку, що розширює застосовність персептрона навіть в умовах «складних» вибірок.

Незважаючи на обмежені можливості одиночного нейрона, коли такі елементи об’єднуються в мережу, вони здатні вирішувати завдання набагато вищої складності. Одношаровий персептрон демонструє фундаментальні принципи роботи штучних нейронних мереж та слугує відправною точкою для вивчення більш глибоких архітектур.

2.3.2 Багатошаровий перцептрон

Багатошаровий персептрон (MLP) — це архітектура штучних нейронних мереж, яка поєднує декілька формальних нейронів у систему з трьома і більше прошарками. Кожен MLP складається щонайменше з вхідного прошарку, одного або декількох прихованих та вихідного прошарків. Вхідний прошарок приймає зовнішні дані та передає їх далі, приховані прошарки обробляють інформацію, виокремлюючи дедалі складніші ознаки, а вихідний прошарок формує остаточний результат [37-39].

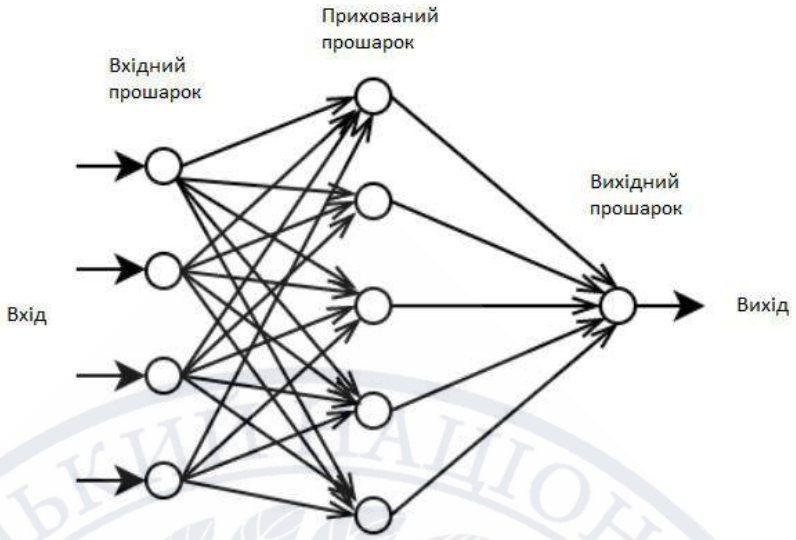

Рисунок 2.3 – Структура багатошарового перцептрона[37]

Кожен нейрон у прошарку отримує на вхід зважену суму сигналів від усіх нейронів попереднього прошарку й пропускає її через свою активаційну функцію. Найпоширеніші активації для прихованих нейронів — це нелінійні функції ReLU, гіперболічний тангенс або сигмоїд, що дають змогу мережі вловлювати складні, нелінійні залежності в даних. Для вихідного прошарку вибір активації залежить від типу завдання: в задачах класифікації часто застосовують softmax, а для регресії — лінійну функцію.

Навчання MLP здійснюється за допомогою алгоритму зворотного поширення помилки (backpropagation) у поєднанні з оптимізатором (наприклад, стохастичним градієнтним спуском або його модифікаціями — Adam, RMSprop тощо). Під час кожної ітерації мережа порівнює свій передбачений вихід з бажаним, обчислює градієнти помилки по відношенню до кожного вагового коефіцієнта та коригує їх у напрямку зменшення загальної похибки. Регуляризаційні методи (dropout, L2-норма, рання зупинка) допомагають запобігти перенавчанню й підвищити узагальнювальну здатність моделі.

Застосування багатошарових персептронів надзвичайно різноманітне: від розпізнавання рукописного тексту та класифікації зображень до побудови рекомендаторних систем і прогнозування часових рядів. MLP є базовим компонентом більш складних глибоких нейронних мереж, де він може

виступати як «щільний» (fullyconnected) шар у поєднанні з іншими архітектурними блоками (згортковими шарами, рекурентними модулями тощо).

2.3.3 Радіально-базисні нейромережі

Радіально-базисні нейромережі (РБНМ – RadialBasisFunctionNet, RBFN) були запропоновані для апроксимації функцій багатьох змінних. У цій архітектурі перший (скритий) прошарок складається з нейронів, які реагують локально на вхідний вектор, оцінюючи відстань до свого центру та пропускаючи її через радіально-базисну (зазвичай гаусову) функцію. Кожен такий нейрон “спрацьовує” лише тоді, коли вхідні дані знаходяться поблизу його центру, що робить РБНМ надзвичайно чутливими до локальних властивостей простору ознак. Вихід цього прошарку — набір активностей, що демонструє, наскільки кожен центр відповідає поточному прикладу [34,40].

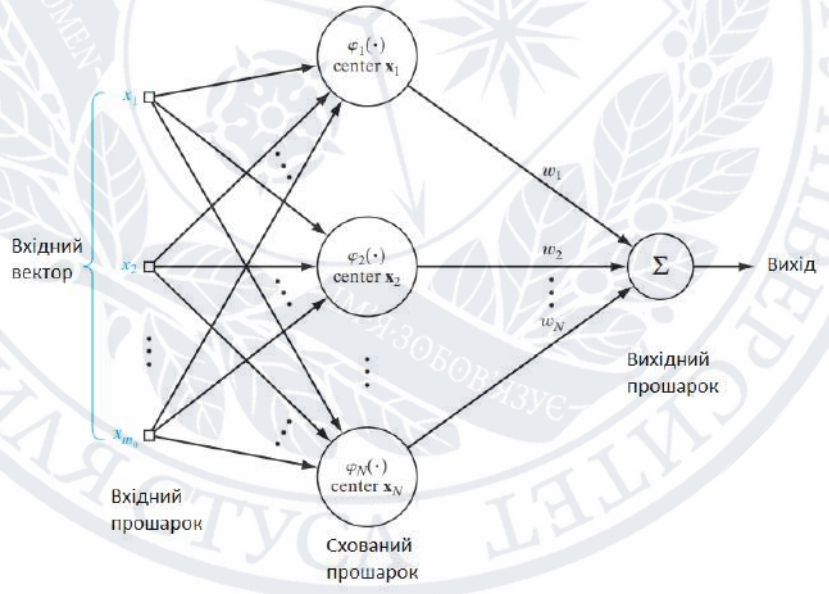

Рисунок 2.4 – Структура радіально-базисної мережі [34]

Другий (вихідний) прошарок формує зважену лінійну комбінацію цих локальних реакцій, перетворюючи їх на остаточний результат — класифікацію або чисельну апроксимацію. Завдяки лінійності другої стадії навчання для неї достатньо вирішити звичайну задачу лінійної регресії, тоді як перший крок — розташування центрів — виконується за допомогою методів кластеризації або

алгоритмів типу k-means. Така двофазова схема (спочатку «вивчають» локальні базиси, потім — глобальні ваги) робить навчання РБНМ швидким і стійким.
РБНМ є універсальним апроксиматором: за достатньої кількості центрів і правильно обраної ширини базисних функцій можна наблизити практично будь-яку гладку функцію з довільною точністю. Водночас локальна природа обробки дозволяє добре працювати з шумними даними та адаптувати модель до виявлення аномалій у вибірці.

У практичних застосуваннях РБНМ успішно використовують для регресії, класифікації, прогнозування часових рядів і рішення задач оптимізації. Незважаючи на відносну простоту, вони часто стають ефективною альтернативою класичним багатошаровим персептронам, особливо коли потрібно швидке й стабільне навчання при обмежених обчислювальних ресурсах.

2.3.4 Згорткова нейронна мережа

Згорткова нейронна мережа (CNN) — це спеціалізована архітектура штучних нейронних мереж, призначена насамперед для автоматичного виявлення локальних ознак у даних високої вимірності. У внутрішній структурі CNN традиційно виділяють чергування згорткових і підвибіркових прошарків, за якими йдуть щільно зв’язані прошарки, що генерують остаточний вихід. Під час згортки кожний нейрон обчислює ковзну згортку вхідного сигналу з набором невеликих ядер–фільтрів, навчання яких дозволяє мережі виявляти характерні патерни: краї, текстури, контури чи інші локальні особливості. Результатом роботи згорткового прошарку стає картка ознак, у якій кожна точка відображає силу виявленої ознаки в певній ділянці вхідного простору [41-43].

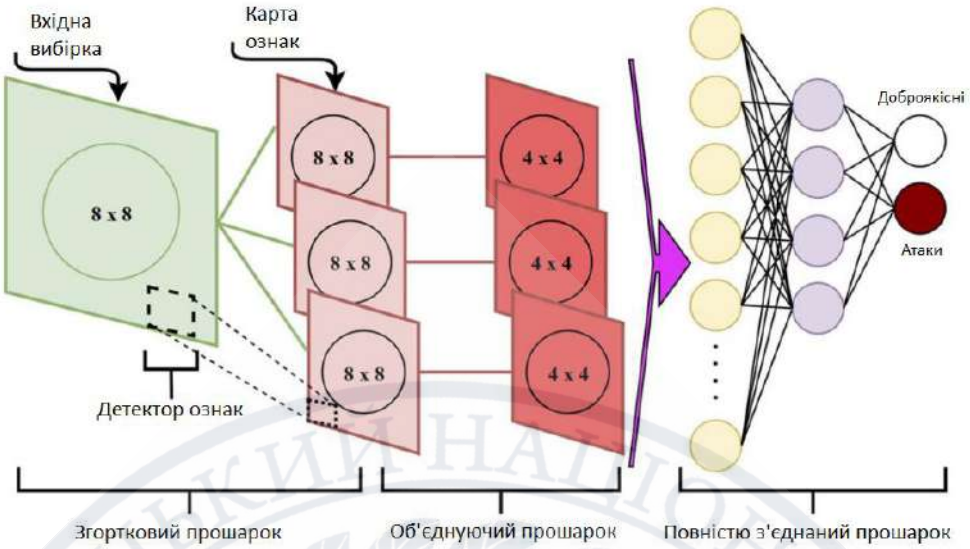

Рисунок 2.5 – Структура згорткової нейронної мережі[41]

Після згортки йде етап підвибірки (пулінг), в якому розмір карток ознак зменшується шляхом агрегування сусідніх значень (наприклад, методом максимального або середнього пулінгу). Це не лише знижує обчислювальні витрати та об’єм пам’яті, а й робить представлення більш інваріантним до незначних зсувів, шуму чи масштабних змін об’єктів. У результаті мережа зосереджується на найбільш релевантних характеристиках, зберігаючи при цьому глобальну структуру сигналу.

У кінці архітектури йдуть щільно зв’язані прошарки, які збирають усі виділені в попередніх етапах ознаки й трансформують їх у фінальний вектор результату — ймовірності класів або безперервні прогнози. Навчання здійснюється під наглядом за допомогою алгоритму зворотного поширення помилки та стохастичних оптимізаторів (наприклад, Adam або RMSprop). Під час кожної ітерації мережа обчислює похибку прогнозу, передає її назад крізь усі прошарки і коригує параметри фільтрів і з’єднань, щоб поступово зменшити відхилення між передбачуваним і бажаним результатом.

Завдяки своїй здатності автоматично вчитися виявляти багаторівневі ознаки, CNN домінують у задачах комп’ютерного зору: розпізнавання зображень, детекція об’єктів, сегментація. Однак їх застосовують і поза межами обробки зображень — для аналізу тексту, коли згортки проходять по векторним

уявленням слів, у відеоаналітиці для витягування просторово-часових патернів, а також для класифікації мережевого трафіку, де багатовимірні вектори ознак трактуються як “зображення” певного розміру. Водночас глибокі CNN можуть вимагати значних обчислювальних ресурсів і страждати від “згасання градієнта” у дуже глибоких конфігураціях, тому їх часто комбінують із регуляризаційними прийомами та іншими архітектурними блоками (рекурентними прошарками чи трансформерами) для покращення стійкості та здатності моделі захоплювати довгі залежності в даних.

2.4 Рекурентні мережі, або мережі зі зворотним зв'язком

Нейромережі зі зворотними зв’язками — це особливий клас штучних нейромереж, у яких інформація проходить не тільки в одному напрямку «вхід→вихід», а й повертається назад, утворюючи циклічні зв’язки. Окрім класичних прямих зв’язків, що з’єднують нейрони послідовних прошарків, такі мережі містять внутрішні петлі: вихід кожного нейрона може надходити назад на його ж власний вхід або ж на входи нейронів попереднього прошарку. Завдяки цьому вони здатні зберігати попередній стан своєї роботи і акумулювати інформацію про попередні кроки обчислень [34, 44].

Циклічні зв’язки надають таким мережам властивість «пам’яті»: в кожен момент часу вихід реагує не лише на актуальний вхід, а й на контекст, що сформований під час попередніх обчислень. Це робить їх надзвичайно корисними для завдань обробки послідовностей — наприклад, для моделювання часових рядів, розпізнавання мовлення чи аналізу тексту, де важливо враховувати порядок і тривалість подій.

У широкому значенні сюди можна включити як рекурентні мережі (RNN), що мають прості циклічні зворотні зв’язки, так і складніші архітектури — з додатковими шарами пам’яті (LSTM, GRU) або мережі Гопфілда, де всі нейрони зв’язані між собою. Завдяки цьому такі моделі виявляють більшу стійкість до помилок і кращу здатність навчатися на нестабільних чи неповних

даних, зберігаючи при цьому достатню гнучкість для адаптації до різних типів завдань.

2.4.1 Нейронна мережа Кохонена SOM

Карта Кохонена, або карта, що самоорганізується (SOM — Self-OrganizingMap), є особливим типом нейронної мережі, призначеної для кластеризації та візуалізації високовимірних даних. Її ключова особливість полягає в тому, що вона виконує навчання без учителя, тобто без попереднього надання мережі правильних відповідей — замість цього SOM сама структурує вхідні дані, розміщуючи подібні за характеристиками вектори в близькій топологічній області вихідного шару [35,45,46].

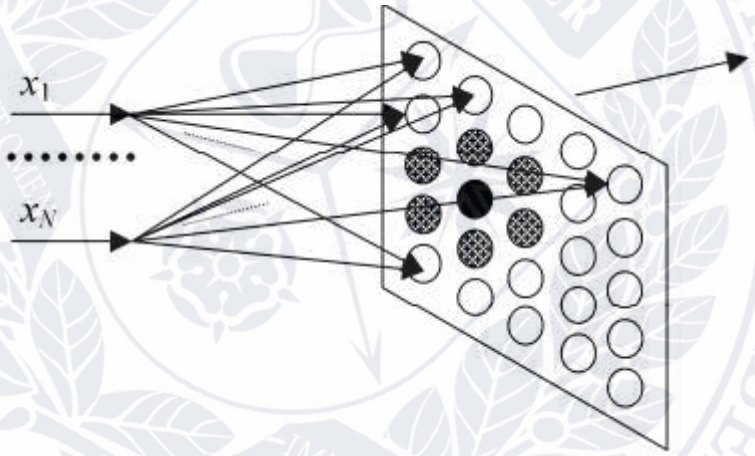

Рисунок 2.6 – Схема нейромережі SOM [35]

Архітектура мережі складається з одношарового масиву нейронів, кожен із яких має вектор ваг, що відповідає розмірності вхідного простору. Коли на вхід мережі подається новий вектор, мережа обчислює відстань між ним і ваговими векторами кожного нейрона — зазвичай використовується евклідова метрика. Нейрон із найменшою відстанню (так званий нейрон-переможець) і його найближчі сусіди в топологічному полі активуються, тобто їхні ваги оновлюються в напрямку наближення до вхідного вектора. З часом розмір цього сусідського поля зменшується, що дозволяє мережі поступово фокусуватися на локальних структурах даних.

У результаті тривалого навчання карта Кохонена формує організовану топологічну структуру, де кожен нейрон “відповідає” певному типу вхідного патерну, а сусідні нейрони реагують на подібні патерни. Це дає змогу використовувати SOM для зведення складних даних до простого, візуально інтерпретованого представлення, наприклад, на двовимірній мапі. Важливо, що при цьому вхідний простір не втрачає топологічної подібності: дані, близькі за змістом, залишаються близькими й на мапі.

Хоча карта Кохонена класифікує дані, її класифікація є умовною та ґрунтується на топологічному зближенні, а не на наявності заздалегідь визначених класів. Для надання фактичного змісту результатам кластеризації можна зіставити кожному нейрону той клас, до якого належить більшість прикладів, які він представляє, тим самим надаючи отриманим кластерам осмислену інтерпретацію. Такий підхід особливо ефективний у задачах, де дані не мають чіткої апріорної структури, як-от сегментація ринку, біоінформатика або аналіз даних сенсорних систем.

2.4.2 Нейронна мережа Хопфілда

Нейронна мережа Хопфілда — це один із класичних прикладів рекурентної нейронної мережі, яка використовується переважно для вирішення задач асоціативної пам’яті. Така мережа складається з фіксованої кількості нейронів, кожен з яких одночасно виступає і входом, і виходом системи. Всі нейрони взаємозв’язані між собою, тобто мережа має повнозв’язну структуру, де кожен нейрон впливає на кожен інший. При цьому зв’язки симетричні — сила зв’язку між двома нейронами однакова в обидві сторони, а самоналаштовані зворотні зв’язки (від нейрона до самого себе) відсутні [35,47].

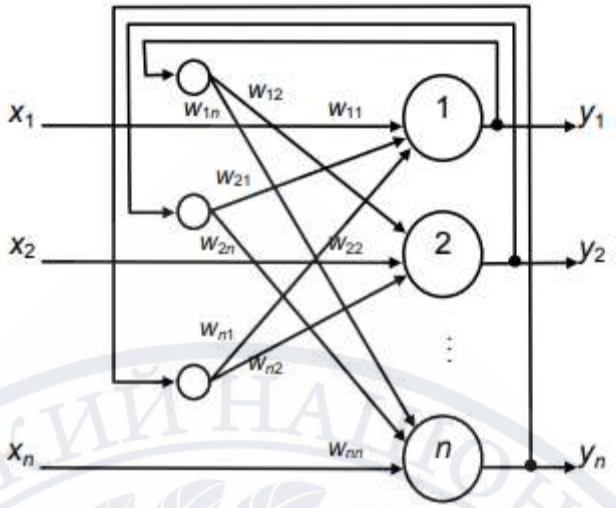

Рисунок 2.6 – Структурна схема нейронної мережі Хопфілда [35]

Стан мережі задається вектором, де кожен елемент — це стан окремого нейрона, який може набувати двох значень, наприклад 1 або -1 (або, в інших реалізаціях, 1 і 0). Зміна стану нейрона відбувається відповідно до зваженої суми сигналів від інших нейронів, скоригованої зовнішнім впливом. Якщо ця сума перевищує певний поріг, нейрон активується (змінює стан), інакше — залишається в попередньому стані.

Мережа може оновлювати свої стани як синхронно (всі нейрони одночасно), так і асинхронно (по черзі або випадково один за одним). У синхронному випадку кожен крок роботи мережі полягає в матричному множенні вектора станів на матрицю ваг з додаванням зовнішнього впливу, після чого до кожного елементу результату застосовується порогова активаційна функція. В асинхронному режимі лише один нейрон змінює стан на основі оновлених даних від інших, що додає гнучкості та стабільності в процесі переходу до остаточного стану.

Ключовою особливістю мережі Хопфілда є її здатність зберігати у своїй структурі певні вектори-пам’яті, до яких вона прагне повернутись у процесі роботи. Це означає, що при подачі на вхід частково спотвореного вектора, мережа "відновлює" його до найближчого збереженого зразка. Така поведінка дозволяє використовувати модель у задачах розпізнавання образів, оптимізації та автоматичного відновлення інформації. Важливо, що мережа може

демонструвати як стійку поведінку (досягнення стабільного стану), так і циклічні коливання, якщо структура збережених даних суперечлива або надмірно складна для наявної кількості нейронів.

2.4.3 ART моделі

Модель адаптивної резонансної теорії (ART) — це клас нейронних мереж із двошаровою структурою (поле подання ознак F1 і поле категорій F2) та механізмом «пильності» (vigilance), який регулює резонанс між шарами. ART навчається без учителя, динамічно створюючи нові категорії або коригуючи існуючі на основі тесту відповідності вхідного сигналу до збережених представлень [48,49].

Елементарні моделі адаптивної резонансної теорії (ART) побудовані за схемою з двома взаємозв’язаними шарами та спеціальним механізмом ухвалення рішень. Перший шар, відомий як поле подання ознак (F1), у режимі прямого проходу передає вхідний вектор безпосередньо в шар категорій (F2) через «довготривалу пам’ять» знизу-вгору, а у зворотному режимі виконує роль компаратора, порівнюючи очікування з поля F2 (за допомогою довготривалої пам’яті згори-вниз) з реальним входом та передаючи результат у орієнтуючу підсистему. Шар категорій F2 відповідає за вибір найбільш відображаючої заданий приклад категорії: його нейрони конкурують у режимі «переможець бере все», і той, що показує найвищу активність за ознакою близькості до вхідного вектора, вважається кандидатом на резонанс.

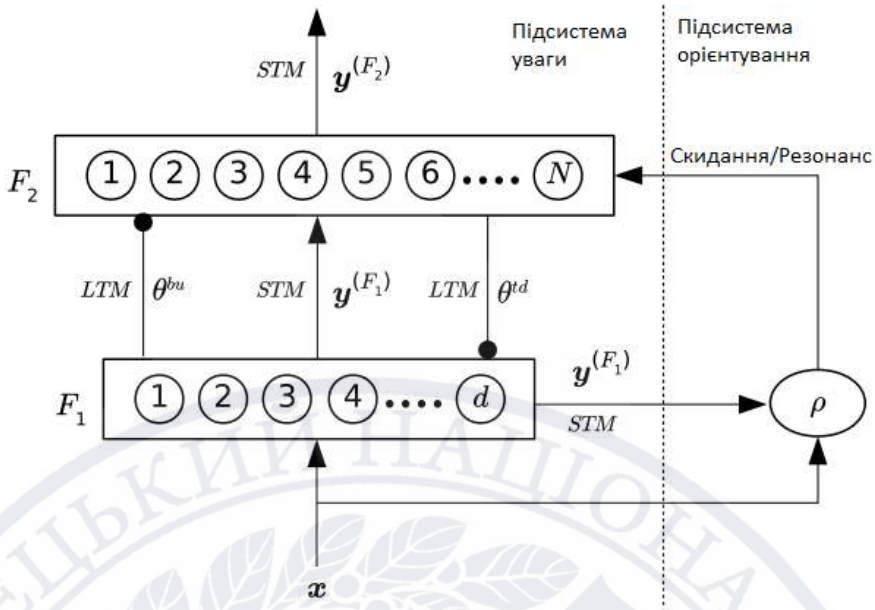

Рисунок 2.7 – Елементарна модель адаптивної резонансної теорії (ART) [48]

Орієнтуюча підсистема регулює так званий «тест пильності» (vigilance), визначаючи, чи достатньо обрана категорія точно відображає поточний вхід. Цей тест заснований на функції відповідності (match): якщо ступінь подібності перевищує поріг пильності, система входить у стан резонансу і здійснює адаптацію вагових параметрів поточної категорії, навчаючи мережу реагувати на подібні зразки в майбутньому. Якщо ж критерій не виконано, поточна категорія тимчасово відключається, а конкуренція відновлюється серед інших нейронів F2. У разі відсутності жодного узгодження мережа створює новий нейрон-категорію, здатний представити цей тип вхідного сигналу.

Завдяки такій динаміці ART моделі об’єднують у собі здатність самостійно структурувати дані без нагляду, підтримувати навчання в режимі реального часу та уникати необхідності задавати наперед кількість кластерів. Параметр пильності визначає деталізацію кластеризації: низькі значення сприяють узагальненню та злиттю подібних груп, тоді як високі дозволяють створювати вузько сфокусовані категорії. У результаті elementary ART демонструє ефективність і гнучкість у задачах кластеризації, розпізнавання шаблонів та адаптивної пам’яті.

2.5 Висновки до розділу 2

Цей розділ окреслює базові поняття IDS та основні архітектури нейронних мереж, що застосовуються для їх реалізації: IDS — це системи для моніторингу подій і виявлення загроз, які поділяються за методом роботи (на основі сигнатур, аномалій, специфікацій) та за місцем розгортання (HIDS, NIDS, гібридні); кожний підхід має свої сильні й слабкі сторони (сигнатурні рішення точні для відомих атак, але не виявляють нові, аномалійні — чутливі до невідомих загроз, проте дають більше помилкових спрацьовувань). Далі розглянуто класифікацію архітектур ШНМ: мережі прямого поширення (MLP, RBFN) як ефективні для класифікації та швидкого навчання, згорткові мережі (CNN) — для виділення локальних ознак і багатовимірних представлень, рекурентні моделі (RNN, LSTM/GRU) — для роботи з послідовностями, а також спеціальні структури: SOM для кластеризації та візуалізації, мережі Хопфілда як асоціативна пам’ять і ART-моделі для адаптивної онлайн-кластеризації. Наведено їхні принципи роботи та типові застосування в IDS, а також вказано на обмеження — потребу в обчислювальних ресурсах, нестабільність навчання, складність налаштування порогів і ризик хибних спрацьовувань — що обґрунтовує популярність гібридних і ансамблевих рішень для підвищення стійкості й практичної придатності систем виявлення.

РОЗДІЛ 3. ПОБУДОВА ТА ПОРІВНЯННЯ ЕФЕКТИВНОСТІ ОСНОВНИХ ВИДІВ НЕЙРОННИХ МЕРЕЖ У ВИЯВЛЕННІ КІБЕРАТАК НА ПІДПРИЄМСТВАХ КРИТИЧНОЇ ІНФРАСРУКТУРИ В УМОВАХ ВОЄННОГО СТАНУ

3.1 Опис використовуваної бази даних

Навчальний набір даних KDD складається з 10% вихідного набору даних, тобто приблизно 494 020 векторів одиночних з'єднань, кожен з яких містить 41 ознаку і позначений одним конкретним типом атаки, тобто або нормальним, або атакою. Кожен вектор позначений як нормальний або як атака, з точно одним конкретним типом атаки. Відхилення від «нормальної поведінки», все, що не є «нормальним», вважається атакою. [50] Атаки, позначені як нормальні, - це записи з нормальною поведінкою. Навчальний набір даних містить 19,69% нормальних і 80,31% атакуючих з'єднань. KDD CUP 99 був найбільш широко використаний в атаках на мережу. Змодельована атака підпадає під одну з наступних чотирьох категорій [51]:

1. Атака на відмову в обслуговуванні (DOS): У цій категорії зловмисник робить деякі обчислювальні ресурси або ресурси пам'яті занадто зайнятими або занадто заповненими для обробки законних запитів, або відмовляє законним користувачам у доступі до машини. До DOS належать атаки «neptune», «back», «smurf», «pod», «land» та «teardrop».

2. Атаки на користувачів до кореневої системи (U2R): У цій категорії зловмисник починає з доступу до звичайного облікового запису користувача в системі і може використати певну вразливість для отримання root-доступу до системи. U2R містить атаки: 'buffer_overflow', 'loadmodule', 'rootkit' та 'perl'.

3. Віддалена локальна атака (R2L): У цій категорії зловмисник надсилає пакети на комп'ютер через мережу, але не має облікового запису на цьому комп'ютері, і використовує певну вразливість, щоб отримати локальний доступ

як користувач цього комп'ютера. До R2L належать атаки 'warezclient', 'multihop', 'ftp_write', 'imap', 'guess_passwd', 'warezmaster', 'spy' та 'phf'.

4. Атака зондування (PROBE): У цій категорії зловмисник намагається зібрати інформацію про мережу комп'ютерів з явною метою обійти її захист. До PROBE належать атаки: «portsweep», «satan», «nmap» та «ipsweep».

Основними завданнями, які виконує виявлення мережевих вторгнень, є розпізнавання рідкісних типів атак, таких як U2R і R2L, підвищення точності виявлення підозрілої активності та підвищення ефективності моделей виявлення вторгнень у реальному часі. Навчальний набір даних складався з 494 019 записів, серед яких 97 277 (19,69%) були «нормальними», 391 458 (79,24%) DOS, 4 107 (0,83%) Probe, 1 126 (0,23%) R2L і 52 (0,01%) U2R-атаки. Кожен запис має 41 атрибут, що описує різні особливості, і мітку, що відносить його до типу «атака» або «нормальний».

3.2 Вибір та побудова архітектур нейронних мереж

У процесі дослідження для моделювання та подальшого аналізу ефективності у виявленні кібератак було відібрано кілька різних підходів до побудови нейронних мереж. Серед них розглядалися класичні багатошарові персептрони, що традиційно застосовуються для задач класифікації; згорткові нейронні мережі, які добре працюють з послідовними та структурованими даними; самоорганізовані карти Кохонена, що належать до несупервізійних методів та дозволяють виконувати кластеризацію і візуалізацію складних даних; асоціативна мережа Хопфілда, яка імітує механізми пам’яті та здатна відновлювати зразки за частковими вхідними даними; а також гібридні архітектури, що поєднують властивості кількох підходів з метою досягнення більш збалансованої продуктивності у виявленні як масових, так і рідкісних типів атак. Такий вибір моделей дозволив комплексно оцінити можливості різних парадигм нейронних мереж у контексті задач кіберзахисту.

Багатошаровий персептрон (MLP)

Перша модель є класичним багатошаровим персептроном для мультикласової класифікації. На вхід подається вектор ознак, що описує мережевий трафік (числові й закодовані категорійні атрибути), і кінцевий результат — це приналежність прикладу до одного з п’яти класів: чотири типи атак і нормальна поведінка. Архітектурно мережа складається з послідовності трьох прихованих шарів, кожен з яких здійснює афінне перетворення вектора (лінійна композиція ваг і зсувів) і застосовує до результату елементну нелінійність. Формально дія одного шару записується як лінійне перетворення

$$z = Wx + b, \quad (3.1)$$

де x — вхідний вектор шару, W — матриця ваг, b — вектор зміщень; потім до z застосовується нелінійність, у вигляді функції Rectified Linear Unit: $ReLU(z) = max(0, z)$. Така комбінація (лінійна операція + ReLU) повторюється триразово, що дає моделі можливість побудувати ієрархію ознак: перші шари відсилають до сукупностей простих поєднань початкових атрибутів, середній шар — до складніших абстракцій, а останній прихований шар формує компактний представник, який служить входом для фінального відображення у простір класів.

Вихідна частина мережі перетворює представлення останнього прихованого шару в п’ятикомпонентний вектор, кожна координата якого інтерпретується як міра належності до відповідного класу. На цю п’ятикомпонентну величину накладається сигмоїдна дія для кожної координати:

$$\sigma(s) = \frac{1}{1+e^{-s}}, \quad (3.2)$$

що обмежує значення між нулем і одиницею. Для прийняття остаточного класового рішення застосовується операція вибору індексу з максимальною вихідною координатою, тобто $\hat{c} = \arg max_j \hat{y}_j$. Ця комбінація дозволяє моделі видавати конкурентні оцінки для всіх п’яти напрямів одночасно і потім обирати найімовірніший клас для конкретного зразка.

що обмежує значення між нулем і одиницею. Для прийняття остаточного класового рішення застосовується операція вибору індексу з максимальною вихідною координатою, тобто $\hat{c} = \arg max_j \hat{y}_j$ Ця комбінація дозволяє моделі видавати конкурентні оцінки для всіх п’яти напрямів одночасно і потім обирати найімовірніший клас для конкретного зразка.

Підготовка міток здійснюється у формат one-hot: кожен приклад має бінарний вектор довжини п’ять із одиницею на позиції істинного класу. Такий формат узгоджується з вихідним п’ятикомпонентним вектором моделі і дозволяє величинам відображати імовірнісні або «інтенсивні» оцінки належності. При навчанні мінімізується середньоквадратична помилка між прогнозом мережі й one-hot вектором; вона виражається як середнє значення квадратів відхилень компонентів.

Щодо режиму навчання: модель проходить через багато епох обробки набору навчальних прикладів; перед кожною епохою порядок прикладів перемішується, а навчання йде батчами фіксованого розміру. Після кожної епохи проводиться незалежна оцінка величини функції втрат на виділеній тестовій підмножині, і стан мережі з найменшим значенням цієї метрики зберігається як найкращий — таким чином у фіналі використовують параметри, які показали найкращі зовнішні результати на тесті. Цей механізм дозволяє зменшити ймовірність вибору перегруженої моделі з поганою узагальнюваністю.

Оцінювання моделі в кінці експерименту здійснюється двома головними способами: обчисленням загальної частки правильних передбачень по всьому набору даних і розкладанням цієї оцінки по класах для аналізу сильних і слабких напрямків. У підсумку модель дає для кожного вхідного зразка числовий вектор «схильностей» до кожного з п’яти класів; потім з цього вектора формується остаточне класове рішення через вибір максимальної координати.

Self-Organizing Map (SOM)

Ця модель — саморганізуючася карта Кохонена (SOM), яка перетворює багатовимірні вектори ознак мережевого трафіку у впорядковану двовимірну топологію нейронів. Кожен нейрон карти має власний вектор ваг тієї самої розмірності, що й вхідні вектори, і карта служить просторовим індексом для кластерів подібних зразків. На практиці це означає, що SOM не навчається безпосередньо розпізнаванню міток, а формує непряме, проєктивне відображення простору ознак у двовимірну решітку таким чином, щоб схожі вектори займали сусідні нейрони, а відмінні — віддалені.

Початковий етап роботи мережі — побудова топології і ініціалізація вагових векторів нейронів. У процесі навчання для кожного випадкового вхідного вектора спочатку знаходиться найбільш близький нейрон — Best Matching Unit (BMU). Формально це записується як вибір індексу c, що мінімізує евклідову відстань між вхідним вектором x і ваговими векторами w_i:

$$c = \arg\min_i \left||x - w_i|\right| . \quad (3.3)$$

Після визначення BMU здійснюється оновлення ваг не лише самого переможного нейрона, а й його сусідів у топологічному просторі карти. Оновлення вагового вектора нейрона i у момент часу t записується у вигляді:

$$w_i(t+1) = w_i(t) + \alpha(t) h_{c,i}(t)\big(x(t) - w_i(t)\big), \quad (3.4)$$

де $\alpha(t)$ — крок навчання, що зменшується з часом, а $h_{c,i}(t)$ — функція сусідства (наприклад, гауссівська), яка задає інтенсивність впливу BMU на сусідні нейрони і теж з часом звужується. Ця локальна адаптація забезпечує, що карта з часом набуває гладкої топології: вагові вектори сусідніх нейронів стають подібними, а відстані між групами нейронів відображають локальні переходи у початковому просторі ознак.

У результаті навчання SOM створює двовимірний простір, де кожний нейрон відповідає своєму «центру» кластеру ознак. Для візуальної інтерпретації якості навчання зазвичай використовують U-матрицю — карту міжнейронних відстаней, яка демонструє кордони між кластерами у вигляді областей з підвищеними відстанями. На такій матриці глибокі «канавки» або вузли з

великими значеннями відстаней вказують на роздільні групи, тоді як рівномірні малі відстані свідчать про гомогенну область карти.

Оцінювання якості роботи в контексті виявлення кібератак ведеться двома рівнями: по-перше, робиться кількісна оцінка точності на тестовій підмножині — частка випадків, коли домінантна мітка BMU збігається з фактичною; по-друге, проводиться детальний аналіз по класах, здебільшого у вигляді recall (відношення правильно класифікованих прикладів класу до усієї кількості прикладів цього класу). Така деталізація критична для сценаріїв кібербезпеки, адже в ній важливі не тільки загальні відсотки, а й здатність моделі виявляти рідкісні, але критичні типи атак.

Мережа Хопфілда

Третя модель — класична мережа Хопфілда, застосована у вигляді набору асоціативних пам'ятей по класах: для кожного класу (normal, dos, r2l, u2r, probe) будується власна вагова матриця, що кодує набір біполярних шаблонів. Вхідні вектори перед подачею в мережу перетворюються в двійкове біполярне представлення $\{-1, +1\}$ (в цьому випадку — через побінаризацію по медіані кожного стовпця), після чого кожен клас інтерпретується як множина патернів у просторі таких біполярних векторів. Навчання кожної окремої мережі реалізується за класичною правилом Хебба: вагова матриця класу формується як середнє зовнішніх добутків шаблонів цього класу (при цьому на діагональ ставиться нуль, щоб уникнути самозв'язку). Формально, якщо $P = \{p^{(1)}, \dots, p^{(M)}\}$ — множина біполярних патернів одного класу (кожен $p^{(k)} \in \{-1,1\}^N$), то вагова матриця цього класу обчислюється як:

$$W = \frac{1}{M}\sum_{k=1}^{M} p^{(k)}(p^{(k)})^{\top}, \text{ з умовою } W_{ii} = 0 \forall i. \quad (3.5)$$

Ця матриця зберігає кореляції між бітами патернів класу і служить основою для асоціативного відновлення.

Процедура відновлення (recall) для довільного вхідного вектора починається з ініціалізації стану $s(0)$ рівним бінаризованому вхідному вектору. Далі проводиться кілька кроків ітераційного оновлення, у яких кожен нейрон

змінює свій стан за правилом уздовжнюваного локального поля. При асинхронному оновленні одиничний нейрон i оновлюється за формулою:

$$s_i(t+1) = sign\left(\sum_j W_{ij} S_j(t)\right), \quad (3.6)$$

де $sign(u) = +1$ якщо $u \geq 0, i - 1$ інакше. Оновлення проводяться в випадковому порядку індексів і повторюються задану кількість кроків — це дозволяє системі еволюціонувати до одного з притягувачів (атракторів) динаміки, якими у випадку симетричної матриці ваг є стабільні стани мережі. Енергійна функція, яку зменшує подібна динаміка, записується як:

$$E(s) = -\frac{1}{2} s^\top W s, \quad (3.7)$$

і спад цієї величини гарантує збіжність послідовності станів мережі до локального мінімуму енергії.

У задачі класифікації, коли для кожного класу є своя матриця $W^{(c)}$, для заданого вхідного вектора запускають процес відновлення на кожному класі окремо і вимірюють, наскільки відновлений стан $s^{(c)}$ відрізняється від початкового вектора. Критерієм близькості в даній реалізації є Хеммінгова відстань між початковим і відновленим біполярним вектором, яку можна записати через скалярний добуток:

$$d_H(x, s) = \sum_{i=1}^{N} 1\{x_i \neq s_i\} = \frac{1}{2}(N - x^\top s). \quad (3.8)$$

Клас прогнозується як той, для якого відновлений стан має мінімальну Хеммінгову відстань від початкового вектора. Якщо деякі вузли мережі не навчені (не мають прикладів) і BMU–логіка не застосовується, використовують запасну стратегію — повернення найчастішого класу у тренувальному наборі.

Особливість підходу — кожен клас реалізує власну асоціативну пам’ять, тобто модель зберігає кореляційні рисунки, притаманні конкретному типу поведінки мережі. Це перетворює задачу класифікації на порівняння між тим, як «добре» кожна кластер-пам’ять може відновити вхідний патерн. З практичної точки зору для побудови таких матриць застосовується підбір підмножини тренувальних прикладів (наприклад, обмеження кількості «normal» прикладів до певної величини й залучення всіх доступних атак), після чого

кожна матриця нормується поділом на кількість вкладених патернів, а діагональні елементи зануляються.

Оцінка ефективності у рамках експерименту проводиться стандартними метриками класифікації: для атак (класи 1..4) будуються матриця плутанини й обчислюється recall по кожному класу як відношення істинно позитивних відкладень до загальної кількості прикладів цього класу; додатково підраховується загальна точність по всьому датасету і точність, обмежена тільки набором атак. Така методика дозволяє кількісно визначити, наскільки кожна класова Hopfield-матриця має притягувачі, що відповідають реальним зразкам атак або нормального трафіку.

У підсумку реалізація Хопфілда в цій системі виступає як набір клас-специфічних асоціативних сховищ: для кожного нового вхідного прикладу проводиться процес асоціативного відновлення на кожній пам’яті, обчислюються відстані між відновленими й початковим станом, і обирається клас з найменшою невідповідністю.

Згорткова нейронна мережа (CNN)

Згорткова нейронна мережа, призначена для перетворення вектора ознак мережевого трафіку у набір числових оцінок належності до п’яти класів (normal, dos, r2l, u2r, probe). Вхідні приклади представлені як одномірні послідовності ознак: кожен зразок сприймається як сигнал з одним каналом і фіксованою довжиною, що робить можливим застосування операцій згортки по вимірі ознак. Перетворення категорійних полів у числовий формат і представлення міток у вигляді one-hot векторів формують підґрунтя для роботи мережі з табличними даними.

Виділена екстракційна частина складається з двох послідовних блоків згорток. Кожен блок здійснює локальний фільтр по сусідніх ознаках (малий розмір ядра), за яким слідує нелінійна трансформація та операція зменшення розмірності (пулінг). Формально операція згортки 1D над сигналом $x[t]$ з ядром $k[\tau]$ записується як:

$$(y * x)[t] = \sum_{\tau} k[\tau]\, x[t-\tau], \quad (3.9)$$

і дає змогу виявляти локальні взаємозв’язки між сусідніми ознаками. Нелінійність у блоках реалізована через елементну Rectified Linear Unit, яка пропускає позитивні компоненти і обнуляє негативні, що дозволяє мережі будувати складніші ознаки з простих локальних шаблонів. Після кожної згортки застосовується субдискретизація (max-pooling) з фіксованим співвідношенням, яка зменшує довжину сигнала вдвічі; двічі застосований пулінг призводить до скорочення просторового розміру в чотири рази від початкової кількості ознак, і тим самим переводить локальні дескриптори у компактний набір ознак.

Після блоку згорток отримане багатоканальне представлення згладжується в один вектор через операцію флеттену, після чого вступає в дію повнозв’язна «голова». Ця частина складається з щонайменше одного лінійного перетворення, елементної нелінійності, ймовірнісного зменшення взаємозв’язності через дропаут, і фінального лінійного відображення в простір п’яти вихідних величин. Лінійне перетворення в загальному вигляді записується як $z = Wx + b$, де W — матриця ваг, b — вектор зсувів; таке перетворення служить для комбінування локальних ознак у глобальні представлення, релевантні для розпізнавання типів атак.

Фінальний шар видає логіти — необроблені скалярні значення на п’ять виходів. Для побудови імовірнісних інтерпретацій поверх логітів застосовують сигмоїдну трансформацію поелементно; під час навчання використовується бінарна крос-ентропійна функція втрат із вбудованою стабілізацією аргументів (BCEWithLogits).

Процес навчання організований пакетоутворено: набір тренувальних прикладів ділиться на батчі певного розміру, у кожному батчі обчислюються передбачення й значення функції втрат, далі виконуються кроки зворотного розповсюдження та оновлення ваг оптимізатором, що реалізує адаптивну стратегію оновлень. На кожній епосі виконуються як етапи навчання на тренувальній підмножині, так і перевірка величини функції втрат на окремому тестовому наборі; стан мережі з найменшим значенням валідаційної метрики

зберігається і відновлюється після завершення циклу навчання. Таке чергування навчання та перевірки формує процедуру селекції найкращих параметрів у межах прогону експерименту.

Оцінювання результатів здійснюється через перетворення логітів у ймовірності, застосування порога 0.5 до кожної координати (тобто поелементне порівняння з 0.5 після сигмоїди) і підрахунок кількості істинно позитивних передбачень відносно фактичної кількості прикладів кожного класу. Для кожного з п’яти класів обчислюється показник recall у відсотках як відношення істинно позитивних передбачень до загальної кількості прикладів даного класу у датасеті, що дає уявлення про здатність моделі виявляти саме ті типи поведінки, які відповідають відповідним атакам або нормальному трафіку.

Гібрид (Hopfield + Transformer)

Ця мережа — гібридна архітектура, яка поєднує трансформероподібний енкодер для табличних ознак із модулем прототипів у стилі Хопфілда, і націлена на мультикласову класифікацію п’яти категорій мережевого трафіку. Ідея полягає в тому, щоб поєднати два різних погляди на дані: диференційоване, навчальне представлення властивостей окремих ознак через увагу й багаторівневі перетворення, та символьну/асоціативну оцінку подібності в просторі біполярних прототипів, отриманих зі зразків тренувального набору. У тексті нижче я описую кожен компонент і їхню взаємодію в логічному порядку, з мінімальною кількістю ключових формул.

Спочатку дані проходять стандартну підготовку: категорійні поля переводяться у числові коди, числові ознаки підлягають стандартизації (зведення до нульового середнього й одиничної дисперсії на основі тренувальної підмножини). Стандартизація уніфікує масштаби ознак і робить навчання нейронів стабільнішим та порівнянним між ознаками.

Перший великий блок — трансформероподібний табличний енкодер. Кожна скалярна ознака проєктується в векторне вбудування фіксованої розмірності і отримує невелике навчане вбудовування індексу ознаки (feature id embedding), що дозволяє мережі розрізняти ролі різних полів. Отриманий

послідовний набір вбудувань обробляється кількома шарами самоуваги й позиційно-інформованого перетворення (Transformer encoder). Після проходження через шари уваги по ознаках робиться агрегація (пулінг — середнє по ознаках), і отримане векторне уявлення служить узагальненим «зведеним описом» прикладу. Універсальний математичний вигляд одного лінійно-проєкційного шару енкодера — афінне перетворення, а механіка уваги формує ваговані суми між вбудуваннями, що дозволяє енкодеру враховувати міжполіці взаємодії в контексті всього вхідного вектора.

Паралельно до навчуваного енкодера будується модуль прототипів у стилі Хопфілда. Прототипи формуються на основі тренувальної множини шляхом біполяризації ознак по стовпцевих медіанах: для кожної ознаки значення, що вище медіани, кодується як +1, нижче — як −1. Для кожного класу утворюється прототип як середнє біполярних векторів цього класу, і цей прототип нормується у вектор одиничної довжини. На фазі інференсу вхідний приклад теж біполяризується аналогічно, а між цим біполярним вектором і кожним нормованим прототипом обчислюється міра подібності у вигляді косинусної схожості:

$$sim(x, p) = \frac{x^{\top} p}{||x|| ||p||}. \quad (3.10)$$

Оскільки прототипи нормовані, обчислення зводиться до скалярного добутку, який дає одну числову оцінку подібності до кожного з C класів; вектор таких оцінок служить дискримінативним «хопфілдовим» описом прикладу.

Кінцева класифікаційна частина будується як конкатенація двох джерел інформації: з одного боку — pooled-вектор від трансформерного табличного енкодера, що представляє диференційовані взаємозв’язки між ознаками, з іншого боку — вектор косинусних подібностей до класових прототипів, що відображає наскільки вхід близький до типової репрезентації кожного класу в біполярному просторі. Після об’єднання цієї подвійної репрезентації застосовується повнозв’язна підмережа з нелінійністю й дропаутом, яка виводить фінальні логіти по класах. Такий дизайн дозволяє моделі одночасно

користуватися тонкою, навченою структурною інформацією та емпіричними асоціативними ознаками, витягнутими безпосередньо зі згрупованих прототипів.

Для оптимізації використовується мультикласова крос-ентропійна втрата з вагами класів, що компенсують дисбаланс у тренувальному наборі. Математично зважена крос-ентропія для одного прикладу записується як:

$$L = -\sum_{j=1}^{C} w_j y_j \, log \frac{e^{s_j}}{\sum_k e^{s_k}}, \quad (3.11)$$

де s_j — логіт для класу j, y_j — істинна мітка у one-hot форматі, а w_j — ваговий множник для класу j. Ваги підбираються обернено пропорційно частотам класів і масштабуються для стабільності, що змушує оптимізатор звертати більше уваги на рідкісні або пріоритетні класи під час навчання.

Навчальний цикл організовано пакетоутворено та з семплером, що зважує приклади відповідно до класових ваг — це дає ефект надвибірки пріоритетних класів у батчах без явної змінності розміру даних. У процесі тренування для кожного батча подаються стандартизовані ознаки та попередньо обчислені векторні значення подібностей до прототипів (щоб уникнути постійної перетворювальної накладної роботи під час батчевого навчання). За епохами обчислюється метрика валідності (наприклад, per-class recall та загальна точність) на валідаційній підмножині, і зберігається стан моделі, що дає найкращий цільовий показник (в описаному сценарії часто це recall для критичного класу, наприклад normal або іншого пріоритетного класу).

Оцінювання моделі відбувається на двох рівнях: по-перше, класичними підрахунками пер-класних метрик (recall, precision, F1 для кожного з п’яти класів) і загальною точністю; по-друге, аналізом внеску двох підсистем у прийняття рішень — тобто розглядом випадків, де модель спирається на трансформерну частину, і випадків, де на прототипну схожість. Практично це дозволяє інтерпретувати рішення: високі значення компоненти подібності вказують на явну відповідність вхідного патерну якомусь класовому прототипу,

а високі значення енкодера — на складні внутрішні взаємозв’язки між ознаками.

Таким чином архітектура поєднує навчений, контекстно-залежний опис прикладу з емпірично витягнутими класовими шаблонами, що робить її гнучкою в умовах як структурованих взаємозв’язків ознак, так і наявності стабільних, повторюваних шаблонів атак — все це з відтворюваною процедурою підготовки даних, навчання з ваговою крос-ентропією і валідованою оцінкою метрик по класах.

Гібрид (CNN + MLP)

Остання модель — гібридна архітектура, що поєднує локальну обробку структурних залежностей через одномірні згортки і глобальне узагальнення через повнозв’язну обробку, і призначена для мультикласової класифікації мережевого трафіку на п’ять категорій. Вхідний приклад сприймається як одномірний сигнал із одним каналом (послідовність фіч), причому перед самою подачею в мережу числові ознаки стандартизуються — кожна ознака зводиться до нульового середнього й одиничного стандартного відхилення на основі тренувальної підмножини; це робить градієнтні оновлення більш стабільними й дозволяє коректно поєднувати різноманітні фічі в одному масштабі.

Екстракційна частина побудована як послідовність двох блоків локального фільтрування. Кожний блок виконує згортку по вимірі ознак з невеликим ядром, потім нормалізує проміжне представлення по каналах (batch normalization) і застосовує елементну нелінійність, після чого слідує операція субдискретизації (max-pooling), яка зменшує довжину послідовності. Математично одиничну одномірну згортку можна представити як дискретний корельований оператор:

$$y[t] = \sum_{\tau} k[\tau] x[t - \tau], \quad (3.12)$$

де x — вхідна послідовність, k — ядро згортки, а y — вихідний канал. Така операція дозволяє виявляти локальні шаблони та кореляції між сусідніми ознаками (наприклад, суміжні пакети або суміжні статистики сесій), а шар

нормалізації знижує внутрішній ковзаючий розкид активацій, що прискорює та стабілізує навчання.

Результат блоку згорток редукується до векторного представлення (шляхом «зглажування» багатоканального тензора у вектор) і далі подається на невелику повнозв'язну голову, яка складається з лінійного перетворення, нелінійності, стохастичного відсікання (dropout) та фінального лінійного шару, що породжує логіти по п'яти класах. Повнозв'язне перетворення узагальнює локально вивчені дескриптори у глобальні предиктори класу: лінійний блок описується стандартною афінною формулою $z = Wx + b$, де ваги W та зсуви bbb навчаються під задачу мінімізації функції втрат.

Як функція втрат застосовується мультикласова крос-ентропія, у якій ваги класів вводяться для компенсації дисбалансу в тренувальному наборі. Формально для одного прикладу з логітами s_j і істинною міткою y (впакованою у one-hot) зважена крос-ентропія має вигляд:

$$L = -\sum_{j=1}^{C} w_j y_j \, log \frac{e^{s_j}}{\sum_k e^{s_k}}, \quad (3.13)$$

де w_j — ваговий множник для класу j. Введення ненульових w_j дозволяє підсилити вклад рідкісних або пріоритетних класів у сумарну похибку і, таким чином, змінити спрямованість навчання. У розглянутій реалізації вага для класу «normal» підсилюється множником (параметром), а потім ваги нормалізуються так, щоб абсолютні значення були стабільні; крім того, для формування батчів застосовується вибірка зваженим семплером, де кожному прикладу присвоюється вага, що залежить від класу прикладу, — це фактично здійснює надвибірку прикладів певних класів в кожній епосі та підвищує ймовірність їх потрапляння до градієнтного кроку.

Оптимізація ваг здійснюється сучасним адаптивним методом з регуляризацією ваг (адаптивний оптимізатор з вагою розпаду), що поєднує швидку збіжність і контроль переобучення через L2-штраф на параметри. Під час навчання зберігається політика вибору «найкращої» моделі на підставі специфічної метрики: критерієм раннього відбору або збереження найкращого

стану виступає recall для класу normal — тобто модель зберігають тоді, коли вона найкраще виявляє нормальний трафік у валідаційній підмножині. Такий фокус підкреслює прагнення системи мінімізувати помилкові спрацьовування або втрати чутливості щодо нормальної поведінки, що може бути актуальним у певних прикладних сценаріях.

Оцінювання здійснюється детально: для кожного класу обчислюється recall (чутливість) як відношення істинно позитивних передбачень до загальної кількості позитивних випадків цього класу; загальна точність також підраховується як частка правильних передбачень серед усіх прикладів. Технічно прогноз для прикладу формується шляхом вибору індексу з максимальною логітною координатою (операція $\arg max$), тобто модель повертає клас з найбільшою передбачуваною «схильністю».

Регуляризаційні механізми в архітектурі включають batch normalization (що вирівнює статистики по пакету), dropout у повнозв’язній частині (що випадково вимикає частину нейронів під час тренування, знижуючи кореляцію між ознаками і запобігаючи перенавчанню) та L2-регуляризацію через weight decay в оптимізаторі. Крім того, балансування даних реалізується на двох рівнях: через перетворення класових ваг у функції втрат і через семплер, що забезпечує надвибірку пріоритетних класів у формуванні батчів. Це поєднання впливає як на статистику градієнтів, так і на емпіричну частоту появи прикладів у підсумкових оновленнях параметрів.

З погляду інтерпретації, згорткова частина працює як локальний детектор шаблонів у векторі ознак (виявляє локальні взаємозв’язки між суміжними атрибутами), а повнозв’язна голова трансформує ці локальні ознаки у глобальні дискримінативні представлення, придатні для відокремлення класів у просторі ознак. Завдяки стандартизації вхідних даних та нормалізації в глибині мережі, навчання більш стабільне, а введення класових та семплерних ваг дозволяє спрямувати навчання на бажані компроміси між recall різних класів.

3.3 Результати експериментальних досліджень

Після вибору та побудови нейронних мереж були визначені їх результати та оцінена ефективність

Self-Organizing Map (нейронна мережа SOM)

SOM показала дуже сильну здатність відокремлювати масивні класи: «dos» і «normal» мають винятково високі recalls (99.76% і 99.50% відповідно). Також SOM добре виділяє «probe» (88.26%). Проте для рідкісних і тонких класів результат слабкий: «r2l» — 57.64%, «u2r» — 0.00%. Це типовий профіль: непогане групування великих кластерів і видно відмінні візуалізаційні кордони (U-matrix), але слабка чутливість до дуже рідкісних шаблонів, якщо вони не представлені достатньо в топології карти або після прив'язки BMU→лейбл.

```
Точність по класах (у %):
 dos: 390510/391458 = 99.76%
 normal: 96792/97278 = 99.50%
 probe: 3625/4107 = 88.26%
 r2l: 649/1126 = 57.64%
 u2r: 0/52 = 0.00%
```

Рисунок - 3.1 Результат перевірки ефективності Self-Organizing Map

Гібрид (нейронна мережа CNN + MLP)

Ця модель демонструє найкращі загальні результати в наборі: усі класи мають дуже високі recalls — всі близькі до 99% і більше (u2r = 100.00%, r2l = 99.56%, probe ≈ 99.88%, dos ≈ 99.34%, normal ≈ 99.22%), overall accuracy 99.32%. Це вказує на те, що архітектура ефективно вловлює як масивні, так і рідкісні патерни. Такий профіль свідчить про дуже хорошу збалансованість навчання і високу дискримінаційну здатність мережі.

```
Final recalls on full dataset:
    dos: 388889/391458 -> 99.34%
    r2l: 1121/1126 -> 99.56%
    u2r: 52/52 -> 100.00%
  probe: 4102/4107 -> 99.88%
 normal: 96515/97278 -> 99.22%

Overall accuracy: 99.32%
```

Рисунок - 3.2 Результат перевірки ефективності Гібрида (CNN + MLP)

MLP (нейронна мережа багатошаровий персептрон)

MLP дає високу загальну точність (99.20%) і дуже добрий результат по «dos» і «normal» (≈99+%), але абсолютно провалює класи r2l та u2r (обидва 0.00%) і має лише 95.08% для probe. Це означає, що модель навчилась «переводити» майже всі приклади в кілька домінуючих класів (особливо dos/normal), і втрачає рідкісні класи — ймовірно через дисбаланс і недостатньо складну архітектуру/функцію втрат для підсилення рідкісних класів.

```
Точність по класах (у %):
  dos: 99.36%
  r2l: 0.00%
  u2r: 0.00%
  probe: 95.08%
  normal: 99.89%
```

Рисунок - 3.3 Результат перевірки ефективності багатошарового персептрона

Гібрид (нейронна мережа Hopfield + Transformer)

Гібридна модель дає змішаний результат: дуже хороші показники для атакних класів (dos 98.66%, r2l 99.47%, u2r 98.08%, probe 99.71%), але значно гірший результат по «normal» — 88.16%, через що overall падає до 96.60%. Тобто модель добре розрізняє атаки, але часто відносить нормальний трафік до атак (зниження recall(normal)). Це може вказувати на те, що комбінований вплив прототипної (біполярної) складової і трансформерного енкодера зміщує межу прийняття рішень у бік більш «агресивного» виявлення атак, підвищивши recall атак, але за рахунок великого числа помилкових спрацьовувань на нормальні приклади.

```
Final recalls on full dataset:
   dos: 386216/391458 -> 98.66%
   r2l: 1120/1126 -> 99.47%
   u2r: 51/52 -> 98.08%
 probe: 4095/4107 -> 99.71%
normal: 85761/97278 -> 88.16%

Overall accuracy: 96.60%
```

Рисунок - 3.4 Результат перевірки ефективності Гібрида (Hopfield + Transformer)

Hopfield (нейронна мережа Хопфілда)

Hopfield-підхід дає середні/нижчі результати: загальна точність ~72.79%, по атаках recall в межах 63–92% (найкраще u2r ≈ 92.5%, найгірше — dos/probe ≈ 64–72%). Архітектура добре зберігає деякі кореляційні шаблони (через зовнішні добутки), але в цілому не дозволяє досягти конкурентних показників у задачі мультикласової класифікації великого, дисбалансного реального датасету: висока вартість обчислень для кожного зразка при одночасному відносно низькому узагальненні.

```
Оцінка для класів атак (по кожному класу):
    dos: 281054/391451 -> 71.80% правильно (recall)
    r2l: 715/1118 -> 63.95% правильно (recall)
    u2r: 37/40 -> 92.50% правильно (recall)
  probe: 2622/4094 -> 64.04% правильно (recall)

Загальна точність (всі класи): 72.79%
Точність тільки по атакам: 71.69%
```

Рисунок - 3.5 Результат перевірки ефективності мережі Хопфілда

CNN (згорткова нейронна мережа)

Чиста CNN дуже добре розпізнає «normal» (99.90%) і «dos» (99.29%) та має хороші результати для probe (93.86%), але майже не виявляє r2l і u2r (r2l ≈ 0.98%, u2r = 0.00%). Отже, як і MLP, CNN фокусується на найбільш «помітних» шаблонах і слабко розпізнає дуже рідкісні атаки без додаткових механізмів балансування або архітектурних підсилень.

```
Точність по класах (у %):
 normal: 97176/97278 = 99.90%
 dos: 388680/391458 = 99.29%
 r2l: 11/1126 = 0.98%
 u2r: 0/52 = 0.00%
 probe: 3855/4107 = 93.86%
```

Рисунок - 3.6 Результат перевірки ефективності згорткової мережі

Результати дослідження по атаках

Dos — майже всі моделі демонструють дуже високий recall: SOM 99.76%, CNN+MLP 99.34%, MLP ≈99.50% (оцінка), Hopf+Trans 98.66%, CNN 99.29%; суттєво відстає Hopfield (≈64%). Підсумок: dos-патерн детектується більшістю архітектур, крім Hopfield.

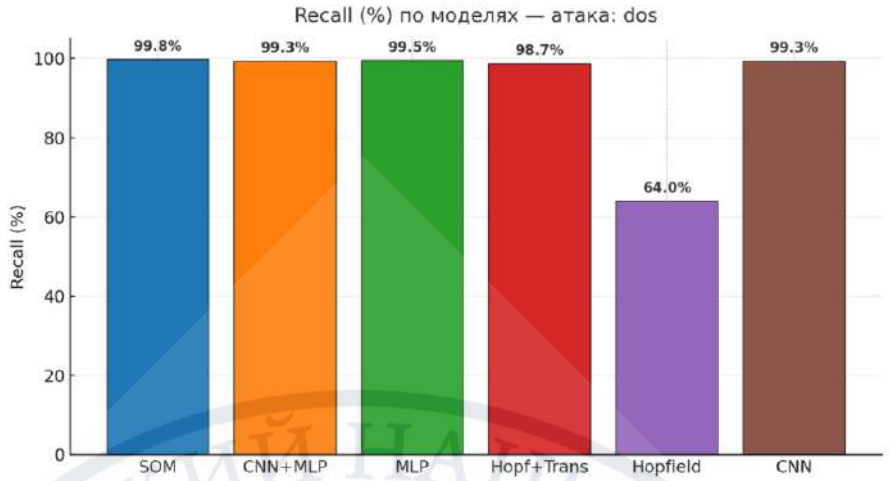

Рисунок - 3.7 Графік ефективності роботи моделей нейронних мереж по виявленню атак dos

R2L — великий розрив між моделями: CNN+MLP 99.56% і Hopf+Trans 99.47% — дуже високі; Hopfield ≈63%; SOM 57.64%; MLP 0.00% і CNN ≈0.98% — майже не виявляють. Підсумок: r2l дає сильно нерівномірні результати між підходами.

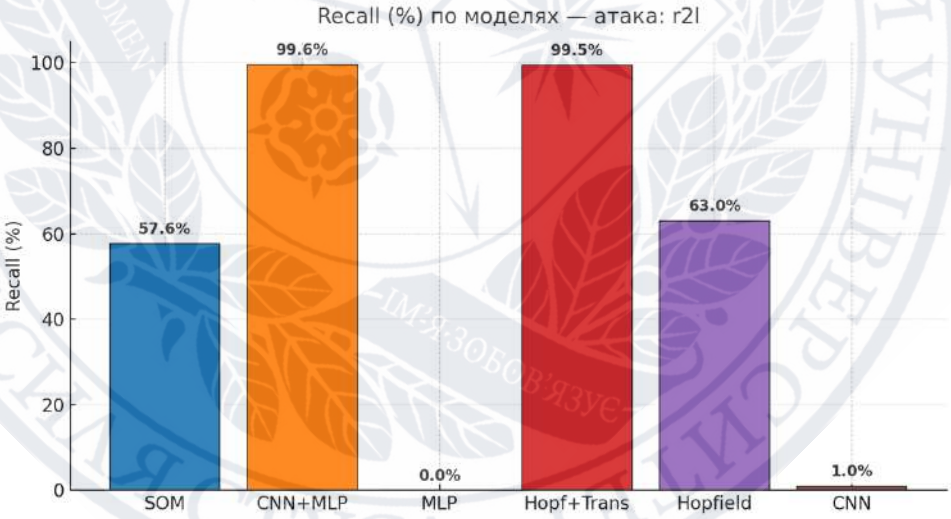

Рисунок - 3.8 Графік ефективності роботи моделей нейронних мереж по виявленню атак r2l

U2R — показники дуже розділені: CNN+MLP 100.00%, Hopf+Trans 98.08%, Hopfield ≈92.50%; натомість SOM 0.00%, MLP 0.00% і CNN 0.00% — практично не виявляють. Підсумок: u2r виявляється лише у деяких спеціалізованих/гібридних архітектурах.

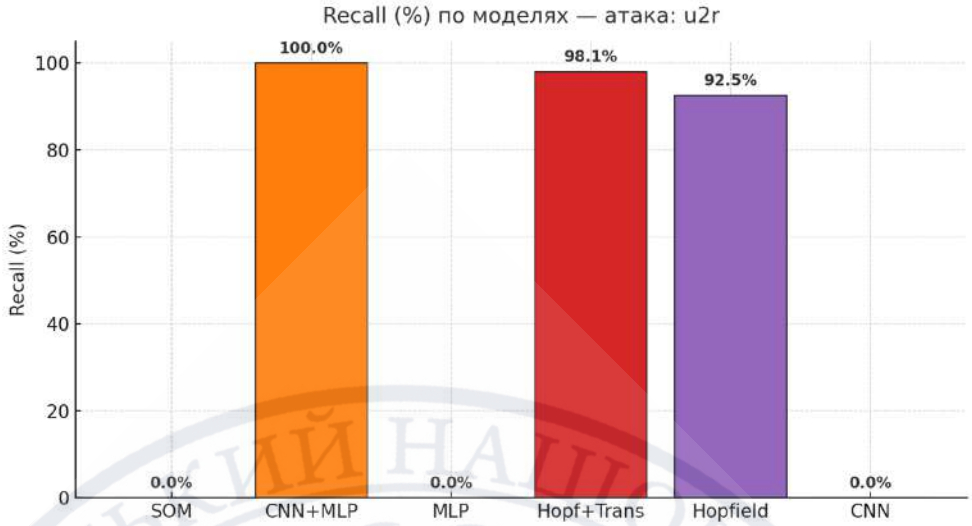

Рисунок - 3.9 Графік ефективності роботи моделей нейронних мереж по виявленню атак u2r

Probe — розкид результатів: CNN+MLP 99.88% і Hopf+Trans 99.71% — найкраще; MLP 95.08%, CNN 93.86%, SOM 88.26%, Hopfield ≈72%. Підсумок: для probe кращі показники у гібридних підходів.

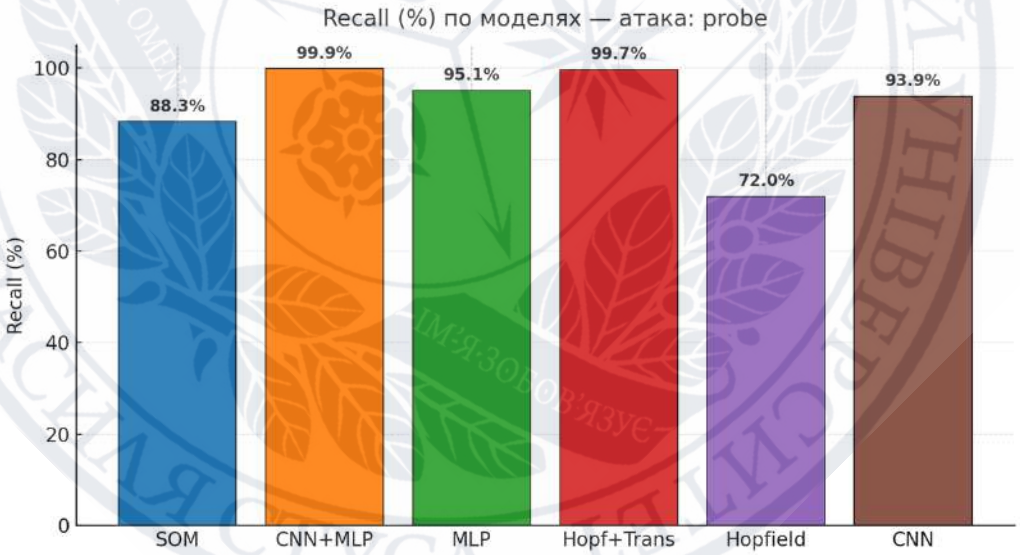

Рисунок - 3.10 Графік ефективності роботи моделей нейронних мереж по виявленню атак probe

Normal — високі показники для більшості моделей: CNN 99.90%, SOM 99.50%, MLP ≈99.40%, CNN+MLP 99.22%; нижчі у Hopf+Trans (88.16%) і Hopfield (≈70%). Підсумок: нормальний трафік добре відокремлюється більшістю моделей, з окремими винятками.

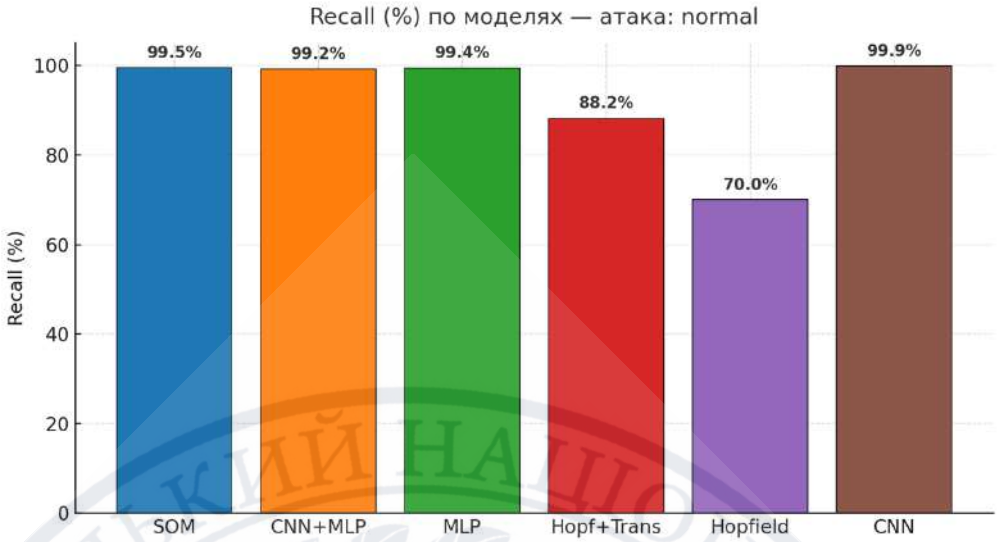

Рисунок - 3.11 Графік ефективності роботи моделей нейронних мереж по виявленню нормального трафіку

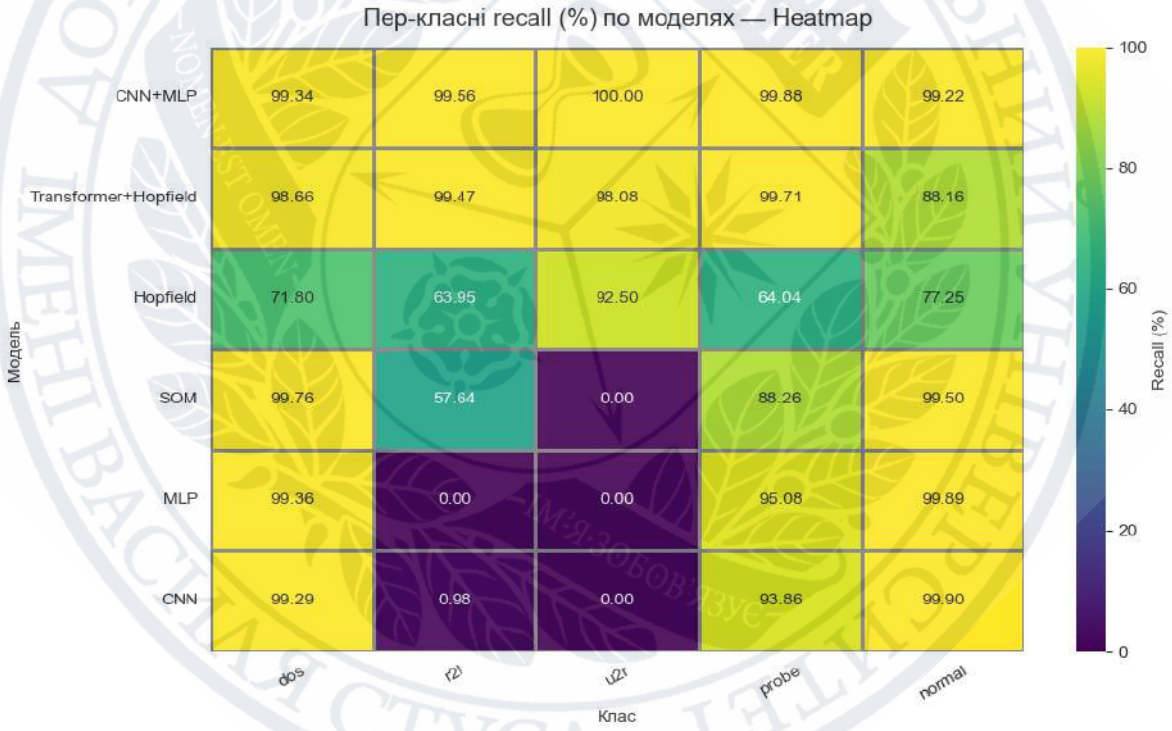

Рисунок - 3.12 Теплова карта порівняльного аналізу повноти виявлення різних типів кібератак досліджуваними нейронними мережами

За поданими результатами найзбалансованішою та найефективнішою моделлю є гібрид CNN+MLP, яка демонструє майже стовідсоткові показники у всіх п’яти класах (≈99%+), тобто найкраще поєднує виявлення масових і рідкісних атак. Модель Transformer+Hopfield також дуже сильна в розпізнаванні атак (високі recalls для dos, r2l, u2r, probe), проте за рахунок цього

суттєво страждає виявлення нормального трафіку (normal ≈88%), що робить її «агресивним» детектором атак з підвищеною кількістю хибних тривог. Чисті архітектури (MLP і CNN) демонструють добрі результати для домінуючих класів (dos, normal), але виявилися неробочими для рідкісних атак (r2l, u2r близько 0%), тобто недостатньо чутливі до небагаточисельних шаблонів. SOM показала відмінне групування для dos і normal і пристойний probe, але слабко пропрацьовує дуже рідкісні u2r і частково r2l. Hopfield-підхід дав помірні результати в цілому (середній рівень точності), проте не конкурує з найкращими сучасними архітектурами за загальною ефективністю. У підсумку, для універсального застосування найкращий вибір — гібрид CNN+MLP; якщо ж пріоритет — «не пропустити» атак будь-якою ціною (навіть ціною фп), варто розглянути Transformer+Hopfield.

3.4 Висновки до розділу 3

Проведене експериментальне порівняння на вибірці KDD (сильний дисбаланс класів) показало, що різні архітектури виконують різні ролі: гібрид CNN+MLP виявився найзбалансованішим і найефективнішим загалом (≈99%+ по більшості класів), Transformer+Hopfield демонструє високу чутливість до атак, але знижену точність для нормального трафіку (normal ≈88%), тоді як чисті MLP і CNN добре працюють для домінуючих класів (dos, normal) але практично не виявляють рідкісні R2L/U2R; SOM ефективно групує великі кластери (dos, normal, probe) але слабшає на рідкісних шаблонах, а підхід на базі Hopfield показав помірні результати й обмежену узагальнювальну здатність при великому дисбалансі. Загалом, результати вказують на перевагу гібридних рішень для універсальної детекції та на необхідність спеціальних механізмів (ваги класів, семплери, прототипні компоненти) для підвищення чутливості до рідкісних атак.

ВИСНОВКИ

У підсумку виконане дослідження продемонструвало, що застосування та порівняння різних архітектур штучних нейронних мереж на класичному наборі даних KDD дає чіткі практичні висновки: гібридні підходи (зокрема CNN+MLP) забезпечують найвищу загальну ефективність і збалансованість між виявленням масових і рідкісних атак, тоді як чисті архітектури (MLP, CNN) добре працюють для домінуючих класів, але суттєво втрачають чутливість до мало представлених категорій (R2L, U2R). Моделі із прототипною або асоціативною складовою (Hopfield, Transformer+Hopfield) демонструють високу здатність «ловити» окремі типи атак, проте іноді це досягається ціною зростання хибних спрацьовувань на нормальний трафік. Самоорганізовані карти (SOM) виявилися корисними як інструмент кластеризації та візуалізації — вони чудово структурують великі кластери (dos, normal), але мало допомагають із дуже рідкісними шаблонами.

Результати підкреслюють важливість двох практичних напрямів: по-перше, використання гібридних і ансамблевих архітектур, які поєднують локальне витягування ознак і глобальні/асоціативні механізми; по-друге, інтеграція заходів для боротьби з дисбалансом (вагові втрати, зважена семплінг-стратегія, синтетичні дані) — саме ці механізми значно підвищують чутливість до рідкісних атак. Оцінювання слід проводити багаторівнево (per-class recall, precision, F1, матриці плутанини та загальна точність), оскільки загальна метрика може приховувати критичні слабкі місця моделі в задачі кібербезпеки.

Дослідження підтверджує, що для практичних IDS найефективнішими є гібридні архітектури з вбудованими механізмами обробки дисбалансу і можливістю інтерпретації рішень; подальша робота повинна бути спрямована на валідацію таких рішень на реальних даних, оптимізацію для реального часу та розробку процедур адаптації моделей під змінні умови загроз.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Hleha, K., & Hol, V. (2025). XAI Optimization for Low-Latency Neural-Based Intrusion Detection Systems in Network Environments. Bulletin of V.N. Karazin Kharkiv National University, Series «Mathematical Modeling. Information Technology. Automated Control Systems», 66, 19-36. https://doi.org/10.26565/2304-6201-2025-66-02.

2. Chao, Jinjin, and Tian Xie. "Deep Learning-Based Network Security Threat Detection and Defense." International Journal of Advanced Computer Science & Applications 15.11 (2024).

3. Mohale, Vincent Zibi, and Ibidun Christiana Obagbuwa. "Evaluating machine learning-based intrusion detection systems with explainable AI: enhancing transparency and interpretability." Frontiers in Computer Science 7 (2025): 1520741.

4. Okafor, Maureen Oluchukwuamaka. "Deep learning in cybersecurity: Enhancing threat detection and response." World Journal of Advanced Research and Reviews 24.3 (2024): 1116-1132.

5. Jyothi, K.K., Borra, S.R., Srilakshmi, K. et al. A novel optimized neural network model for cyber attack detection using enhanced whale optimization algorithm. Sci Rep 14, 5590 (2024). https://doi.org/10.1038/s41598-024-55098-2

6. Abbas, Sidra, et al. "Evaluating deep learning variants for cyber-attacks detection and multi-class classification in IoT networks." PeerJ Computer Science 10 (2024): e1793.

7. Al Hwaitat, A.K.; Fakhouri, H.N. Adaptive Cybersecurity Neural Networks: An Evolutionary Approach for Enhanced Attack Detection and Classification. Appl. Sci. 2024, 14, 9142. https://doi.org/10.3390/app14199142

8. M. Barr, “A Robust Neural Network against Adversarial Attacks”, Eng. Technol. Appl. Sci. Res., vol. 15, no. 2, pp. 20609–20615, Apr. 2025.

9. Alzaidy, S.; Binsalleeh, H. Adversarial Attacks with Defense Mechanisms on Convolutional Neural Networks and Recurrent Neural Networks for Malware Classification. Appl. Sci. 2024, 14, 1673. https://doi.org/10.3390/app14041673

10. Dalal, S., Manoharan, P., Lilhore, U.K. et al. Extremely boosted neural network for more accurate multi-stage Cyber attack prediction in cloud computing environment. J Cloud Comp 12, 14 (2023). https://doi.org/10.1186/s13677-022-00356-9

11. Wang, Bangli, et al. "DDoS-MSCT: A DDoS Attack Detection Method Based on Multiscale Convolution and Transformer." IET Information Security 2024.1 (2024): 1056705.

12. Neto, E.C.P.; Dadkhah, S.; Ferreira, R.; Zohourian, A.; Lu, R.; Ghorbani, A.A. CICIoT2023: A Real-Time Dataset and Benchmark for Large-Scale Attacks in IoT Environment. Sensors 2023, 23, 5941. https://doi.org/10.3390/s23135941

13. Saini, Shalini, Anitha Chennamaneni, and Babatunde Sawyerr. "A Review of the Duality of Adversarial Learning in Network Intrusion: Attacks and Countermeasures." arXiv preprint arXiv:2412.13880 (2024).

14. Zhong, Meihui, et al. "A survey on graph neural networks for intrusion detection systems: Methods, trends and challenges." Computers & Security 141 (2024): 103821.

15. Chhetri, Bipin, and Akbar Siami Namin. "The Application of Transformer-Based Models for Predicting Consequences of Cyber Attacks." 2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2025.

16. Sharma, Abhijeet, et al. "Neural Network Based Zero-Day-Attack Detection: A Machine Learning Approach to Enhancing Cybersecurity." Available at SSRN 4848658 (2024).

17. Rasikha, V., and P. Marikkannu. "An ensemble deep learning-based cyber attack detection system using optimization strategy." Knowledge-Based Systems 301 (2024): 112211.

18. A. Ahmim et al., “A novel hierarchical intrusion detection system based on decision tree and rules-based models,” in Proc. IEEE 15th Int. Conf. Distribution Comput. Sensor Syst., 2019, pp. 228–233.

19. T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, “Deep learning approach for network intrusion detection in software defined networking,” in Proceedings of 2016 International Conference on Wireless Networks and Mobile Communications (WINCOM), IEEE, Reims, France, pp. 258–263, October 2016.

20. D. Li, R. Baral, T. Li, H. Wang, Q. Li, and S. Xu, “Hashtrandnn: a framework for enhancing robustness of deep neural networks against adversarial malware samples,” 2018, http:// arxiv.org/abs/ 1809.06498.

21. W. Peng, X. Kong, G. Peng, X. Li, and Z. Wang, “Network intrusion detection based on deep learning,” in Proceedings of 2019 International Conference on Communications, Information System and Computer Engineering (CISCE), pp. 431– 435, Haikou, China, July 2019.

22. B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, “Deep learning for classification of malware system call sequences,” in Proceedings of Australasian Joint Conference on Artificial Intelligence, pp. 137–149, Springer, Hobart, Australia, December 2016.

23. B. Kolosnjaji, G. Eraisha, G. Webster, A. Zarras, and C. Eckert, “Empowering convolutional networks for malware classification and analysis,” in Proceedings of 2017 International Joint Conference on Neural Networks (IJCNN), pp. 3838–3845, San Diego, CA, USA, June 2017.

24. Uppal, Hussain Ahmad Madni, MemoonaJaved, and M. Arshad. "An overview of intrusion detection system (IDS) along with its commonly used techniques and classifications." International Journal of Computer Science and Telecommunications 5.2 (2014): 20-24.

25. Anderson D, Lunt TF, Javitz H, Tamaru A, Valdes A. Detecting unusual program behaviour using the statistical component of the next-generation intrusion detection expert system (NIDES). Menlo Park, CA, USA: Computer Science Laboratory, SRI International; 1995. SRIO-CSL-95-06.

26. Ghosh, A.K., Wanken, J., & Charron, F. Detecting anomalous and unknown intrusions against programs. In K. Keus (Ed), Proceedings of the 14th annual

computer security applications conference , 1998, (pp. 259--267). IEEE Computer Society, Los Alamitos,CA.

27. G. Macia Fernandez and E. Vazquez, “Anomaly-based Network intrusion detection: Techniques, systems and Challenges”, Computers & Security, Vol. 28, No. 1-2, pp. 18-28, February-March 2009.

28. Harley Kozushko, “Intrusion Detection: Host-Based and Network-Based Intrusion Detection Systems”, on September 11, 2003.

29. Paul Innella Tetrad, “The Evolution of Intrusion Detection Systems”, Digital Integrity,LLC on November 16, 2001.

30. Sekar R., Gupta A., Frullo J., Shanbhag T., Tiwari A., Yang H., et al. Specification-based anomaly detection: a new approach for detecting network intrusions. In: Proceedings of the Ninth ACM Conference on Computer and Communications Security; 2002. p. 265–74.

31. Zahoruiko, Lubov, et al. "Mathematical model and structure of a neural network for detection of cyber attacks on information and communication systems." Informatyka, Automatyka, Pomiary w GospodarceiOchronieŚrodowiska 14.3 (2024): 49-55.

32. Nøkland, Arild. "Direct feedback alignment provides learning in deep neural networks." Advances in neural information processing systems 29 (2016).

33. Subbotin, Sergey Alexandrovich, and Sergey AlexandrovichSubbotin. “Neural Networks: Theory and Practice.” (2020).

34. Haykin, Simon. Neural networks and learning machines, 3/E. Pearson Education India, 2009.

35. Savchenko, Alina Stanislavivna, and Alexey AlekseevichSinelnikov. “Methods and Systems of Artificial Intelligence (Textbook).” (2017).

36. Singh, Jaswinder, and Rajdeep Banerjee. "A study on single and multi-layer perceptron neural network." 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC). IEEE, 2019.

37. Bebeshko, Bohdan, et al. "Use of Neural Networks for Predicting Cyberattacks." CPITS I. 2021.

38. Ali, Osman, and Paul Cotae. "Towards DoS/DDoS attack detection using artificial neural networks." 2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON). IEEE, 2018.

39. Adams, Samuel Olorunfemi, EdnahAzikwe, and Mohammed Anono Zubair. "Artificial Neural Network Analysis of Some Selected KDD CUP99 Dataset for Intrusion Detection." Acta Informatica Malaysia (AIM) (2022).

40. Akbilgic, O., Bozdogan, H. & Balaban, M.E. A novel Hybrid RBF Neural Networks model as a forecaster. Stat Comput 24, 365–375 (2014). https://doi.org/10.1007/s11222-013-9375-7

41. Ho, Samson, et al. "A novel intrusion detection model for detecting known and innovative cyberattacks using convolutional neural network." IEEE Open Journal of the Computer Society 2 (2021): 14-25.

42. Bhuyan, Md Khokan, et al. "Convolutional Neural Networks Based Detection System for Cyber-Attacks in Industrial Control Systems." Journal of Computer Science and Technology Studies 6.3 (2024): 86-96.

43. Lee, Jonghoon, et al. "Cyber threat detection based on artificial neural networks using event profiles." Ieee Access 7 (2019): 165607-165626.

44. Li, Chaopeng, Jinlin Wang, and Xiaozhou Ye. "Using a recurrent neural network and restricted Boltzmann machines for malicious traffic detection." NeuroQuantology 16.5 (2018).

45. Kayacik, H. Gunes, A. Nur Zincir-Heywood, and Malcolm I. Heywood. "A hierarchical SOM-based intrusion detection system." Engineering applications of artificial intelligence 20.4 (2007): 439-451.

46. Kumar, Parasuraman, et al. "Analysis of intrusion detection in cyber attacks using DEEP learning neural networks." Peer-to-Peer Networking and Applications 14.4 (2021): 2565-2584.

47. Joya, Gonzalo, M. A. Atencia, and Francisco Sandoval. "Hopfield neural networks for optimization: study of the different dynamics." Neurocomputing 43.1-4 (2002): 219-237.

48. da Silva, Leonardo Enzo Brito, Islam Elnabarawy, and Donald C. Wunsch II. "A survey of adaptive resonance theory neural network models for engineering applications." Neural Networks 120 (2019): 167-203.

49. Carpenter, Gail A., and Stephen Grossberg. Adaptive resonance theory. Boston University Center for Adaptive Systems and Department of Cognitive and Neural Systems, 2009.

50. J. F. Nieves, “Data Clustering for Anomaly Detection in Network Intrusion Detection”, Research Alliance in Math and Science. http://info.ornl.gov/sites/rams09/j_nieves_rodrigues/ Documents/report.pdf, (2009) August 14.

51. M. Tavallaee, E. Bagheri, W. Lu and A. Ghorbani, “A Detailed Analysis of the KDD’99 CUP Data Set”, The 2nd IEEE Symposium on Computational Intelligence Conference for Security and Defense Applications (CISDA), (2009)

ДОДАТОК А

```
# Self-Organizing Map
import os
from collections import Counter, defauldict
import numpy as np
import pandas as pd
from minisom import MiniSom
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
import matplotlib.pyplot as plt
# Конфігурація
DATA_PATH = r'R:/KDD99/kddcup.data_10percent_corrected'
RND = 42
TEST_FRAC = 0.25
SOM_DIM = 18
SOM_ITERS = 10000
SOM_SIGMA = 1.0
SOM_LR = 0.5
np.random.seed(RND)
# Імена стовпців
COLS = [
'duration','protocol_type','service','flag','src_bytes','dst_bytes','land','wrong_fragment','urgent','hot','n
um_failed_logins','logged_in','num_compromised','root_shell',
'su_attempted','num_root','num_file_creations','num_shells','num_access_files','num_outbound_cmd
s','is_host_login','is_guest_login','count','srv_count','serror_rate',
'srv_serror_rate','rerror_rate','srv_rerror_rate','same_srv_rate','diff_srv_rate','srv_diff_host_rate','dst_
host_count','dst_host_srv_count','dst_host_same_srv_rate','dst_host_diff_srv_rate',
'dst_host_same_src_port_rate','dst_host_srv_diff_host_rate','dst_host_serror_rate','dst_host_srv_serr
or_rate','dst_host_rerror_rate','dst_host_srv_rerror_rate','Attack_type'
]
# Маппінг атак
DOS = {'back.','land.','neptune.','pod.','smurf.','teardrop.'}
R2L = {'ftp_write.','guess_passwd.','imap.','multihop.','phf.','spy.','warezclient.','warezmaster.'}
U2R = {'buffer_overflow.','loadmodule.','perl.','rootkit.'}
```

```
PROBE = {'ipsweep.','nmap.','portsweep.','satan.'}
def collapse_attack_label(lbl: str) -> str:
    # Зведення оригинальних атак у пять категорій
    if lbl == 'normal.':
        return 'normal'
    if lbl in DOS:
        return 'dos'
    if lbl in R2l:
        return 'r2l'
    if lbl in U2R:
        return 'u2r'
    if lbl in PROBE:
        return 'PROBE'
    return 'other'
# Зчитування та підготовка
def load_and_prepare(path):
    if not os.path.exists(path):
        raise FileNotFoundError(f"Файл не знайдено: {path}")
    df = pd.read_csv(path, header=None, names=COLS)
    df['group'] = df['Attack_type'].map(collapse_attack_label)
    df = df[df['group'].isin({'normal','dos','r2l','u2r','probe'})].reset_index(drop=True)
    groups = df['group'].copy()
    features_df = df.drop(columns=['Attack_type','group'])
    for c in features_df.select_dtypes(include=['object']).columns:
        le = LabelEncoder()
        features_df[c] = le.fit_transform(features_df[c].astype(str))
    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(features_df.values.astype(np.float32))
    return X_scaled, groups.values, scaler, features_df.columns.tolist()
X, y_raw, scale_obj, feature_names = load_and_prepare(DATA_PATH)
print(f"Підготовлено зразків: {X.shape[0]}, фіч: {X.shape[1]}")
# Перетворимо мітки у числові індекси
lbl_encoder = LabelEncoder()
y_idx = lbl_encoder.fit_transform(y_raw)
# Розбиття train/test
```

```
X_tr, X_te, y_tr_idx, y_te_idx = train_test_split(
    X, y_idx, test_size=TEST_FRAC, random_state=RND, stratify=y_idx
)
# Налаштування та навчання SOM
som = MiniSom(x=SOM_DIM, y=SOM_DIM, input_len=X_tr.shape[1],
              sigma=SOM_SIGMA, learning_rate=SOM_LR,
              neighborhood_function='gaussian', random_seed=RND)
som.random_weights_init(X_tr)
print("Починаю навчання SOM...")
som.train_random(data=X_tr, num_iteration=SOM_ITERS)
print("Навчання завершено")
# Побудова U-matrix
um = som.distance_map()
plt.figure(figsize=(8, 6))
plt.imshow(um.T, origin='lower', cmap='virdis')
plt.colorbar(label='U-distance')
plt.title('U-Matrix (SOM)')
plt.xlabel('X')
plt.ylabel('L')
plt.tight_layout()
plt.show()
# Привязка вузлів до класів
def build_bm_label_map(som_model, data_vectors, data_labels):
    node_to_labels = defaultdict(list)
    for vec, lab in zip(data_vectors, data_labels):
        winner = som_model.winner(vec)
        node_to_labels[winner].append(int(lab))
    node_to_dominant = {}
    for node, labs in node_to_labels.items():
        most_common = Counter(labs).most_common(1)[0][0]
        node_to_dominant[node] = most_common
    return node_to_dominant
bmu_to_label = build_bmu_label_map(som, X_tr, y_tr_idx)
# Функція прогнозу для одиночного вектора
def predict_with_som(som_model, bmu_map, vector, fallback_label):
```

```
    winner = som_model.winner(vector)
    return bmu_map.get(winner, fallback_label)
fallback = Counter(y_tr_idx).most_common(1)[0][0]
# Оцінка передбачення на тесті
y_pred_te = np.array([predict_with_som(som, bmu_to_label, v, fallback) for v in X_te])
acc_test = accuracy_score(y_te_idx, y_pred_te)
print(f"\n Точність на тесті: {acc_test:.4f}")
# Прогін по всьому датасету і підрахунок recall по класах
y_pred_all = np.array([predict_with_som(som, bmu_to_label, v, fallback) for v in X])
print("\nТочність по класах (у %):")
for class_index, class_name in enumerate(lbl_encoder.classes_):
    total = np.sum(y_idx == class_index)
    correct = np.sum((y_idx == class_index) & (y_pred_all == class_index))
    recall_pct = 100.0 * correct / total if total > 0 else 0.0
    print(f" {class_name}: {correct}/{total} = {recall_pct:.2f}%")


# Hopfield для KDD'99
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, accuracy_score
DATA_PATH = r'R:\KDD99\kddcup.data_10_percent_corrected'
# Зчитування і предобробка
def prepare_kdd99(path):
    cols = [
        'duration','protocol_type','service','flag','src_bytes','dst_bytes','land','wrong_fragment','urgent',
        'hot','num_failed_logins','logged_in','num_compromised','root_shell','su_attempted','num_root',
'num_file_creations','num_shells','num_access_files','num_outbound_cmds','is_host_login','is_guest
_login', 'count','srv_count','serror_rate','srv_serror_rate','rerror_rate','srv_rerror_rate','same_srv_rate',
'diff_srv_rate','srv_diff_host_rate','dst_host_count','dst_host_srv_count','dst_host_same_srv_rate',
'dst_host_diff_srv_rate','dst_host_same_src_port_rate','dst_host_srv_diff_host_rate','dst_host_serror
_rate',  'dst_host_srv_serror_rate','dst_host_rerror_rate','dst_host_srv_rerror_rate','attack_label'
    ]
    df = pd.read_csv(path, header=None, names=cols)
    # Групуємо типи атак у класи
```

```
    mapping = {'normal.': 0}
    dos_attacks = ['back.','land.','neptune.','pod.','smurf.','teardrop.']
    r2l_attacks =
['ftp_write.','guess_passwd.','imap.','multihop.','phf.','spy.','warezclient.','warezmaster.']
    u2r_attacks = ['buffer_overflow.','loadmodule.','perl.','rootkit.']
    probe_attacks = ['ipsweep.','nmap.','portsweep.','satan.']
    for a in dos_attacks:    mapping[a] = 1
    for a in r2l_attacks:    mapping[a] = 2
    for a in u2r_attacks:    mapping[a] = 3
    for a in probe_attacks:  mapping[a] = 4
    df['y'] = df['attack_label'].map(mapping)
    df.drop(columns=['attack_label'], inplace=True)
    # Категоріальні -> числові
    cat_cols = df.select_dtypes(include=['object']).columns.tolist()
    for c in cat_cols:
        le = LabelEncoder()
        df[c] = le.fit_transform(df[c].astype(str))
    X = df.drop(columns=['y']).values.astype(float)
    y = df['y'].values.astype(int)
    return X, y
# Побудова Hopfield-матриць
def build_class_hopfields(X_train, y_train):
    classes = np.unique(y_train)
    class_weights = {}
    for cls in classes:
        P = X_train[y_train == cls]
        if P.shape[0] == 0:
            continue
        W = np.zeros((P.shape[1], P.shape[1]), dtype=float)
        for p in P:
            W += np.outer(p, p)
        W /= max(1, P.shape[0])
        np.fill_diagonal(W, 0.0)
        class_weights[int(cls)] = W
    return class_weights
```

```
# запуск мережі та прогноз
def recall_and_classify(class_nets, X, steps=5):
    n_samples, n_feats = X.shape
    preds = np.zeros(n_samples, dtype=int)
    classes = sorted(class_nets.keys())
    for i in range(n_samples):
        x0 = X[i].copy()
        unique_vals = np.unique(x0)
        if not set(unique_vals).issubset({-1, 1}):
            x0 = np.where(x0 >= 0, 1, -1)
        dist_per_class = {}
        for cls in classes:
            W = class_nets[cls]
            s = x0.copy()
            for _ in range(steps):
                idxs = np.random.permutation(n_feats)
                for idx in idxs:
                    net = W[idx, :].dot(s)
                    s[idx] = 1 if net >= 0 else -1
            dist = np.sum(s != x0)
            dist_per_class[cls] = dist
        preds[i] = min(dist_per_class, key=dist_per_class.get)
    return preds
# бінаризація
def bipolar_binarize_by_median(X):
    med = np.median(X, axis=0)
    Xb = np.where(X > med, 1, -1)
    return Xb
# Головний виконувальний блок
if __name__ == '__main__':
    print("Завантаження та предобробка даних...")
    X_raw, y = prepare_kdd99(DATA_PATH)
    print(f"Розмір повного набору: {X_raw.shape}, класи: {np.unique(y)}")
    X_bin = bipolar_binarize_by_median(X_raw)
    print("Бінаризація завершена (значення -1/+1).")
```

```
    normal_indices = np.where(y == 0)[0][:10000]
    attack_indices = np.where(y != 0)[0]
    train_indices = np.concatenate([normal_indices, attack_indices])
    X_train = X_bin[train_indices]
    y_train = y[train_indices]
    print(f"Тренувальних прикладів: {X_train.shape[0]} (normal: {normal_indices.size}, attacks: {attack_indices.size})")
    nets = build_class_hopfields(X_train, y_train)
    print(f"Побудовано вагові матриці для класів: {sorted(nets.keys())}")
    # Прогноз для всього набору
    print("Прогноз для всього набору (може зайняти час)...")
    y_pred = recall_and_classify(nets, X_bin, steps=4)
    attack_classes = [1, 2, 3, 4]
    labels_names = {1: 'dos', 2: 'r2l', 3: 'u2r', 4: 'probe'}
    cm = confusion_matrix(y, y_pred, labels=attack_classes)
    print("\nОцінка для класів атак (по кожному класу):")
    for i, cls in enumerate(attack_classes):
        tp = cm[i, i]
        total = cm[i].sum()
        perc = (tp / total * 100) if total > 0 else 0.0
        print(f"  {labels_names[cls]:>6}: {tp}/{total} -> {perc:.2f}% правильно (recall)")
    # Загальна точність по всьому датасету (включаючи нормальні)
    acc_all = accuracy_score(y, y_pred) * 100
    mask_attacks = y != 0
    acc_attacks = accuracy_score(y[mask_attacks], y_pred[mask_attacks]) * 100 if mask_attacks.any() else 0.0
    print(f"\nЗагальна точність (всі класи): {acc_all:.2f}%")
    print(f"Точність тільки по атакам: {acc_attacks:.2f}%")


# mlp
import os
import copy
import numpy as np
import pandas as pd
from collections import Counter
```

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import torch
import torch.nn as nn
import torch.optim as optim
# Налаштування
FILE_PATH = r'R:\KDD99\kddcup.data_10_percent_corrected'
RND = 4
TRAIN_SHARE = 0.75
BATCH = 300
EPOCHS = 200
LR = 0.00456789
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# Імена стовпців
COLS = [
 'duration','protocol_type','service','flag','src_bytes','dst_bytes','land','wrong_fragment','urgent','hot',
'num_failed_logins','logged_in','num_compromised','root_shell','su_attempted','num_root','num_file
_creations','num_shells','num_access_files','num_outbound_cmds','is_host_login','is_guest_login','c
ount','srv_count','serror_rate','srv_serror_rate','rerror_rate','srv_rerror_rate','same_srv_rate','diff_srv_
rate','srv_diff_host_rate',
'dst_host_count','dst_host_srv_count','dst_host_same_srv_rate','dst_host_diff_srv_rate','dst_host_sa
me_src_port_rate','dst_host_srv_diff_host_rate','dst_host_serror_rate','dst_host_srv_serror_rate','dst
_host_rerror_rate', 'dst_host_srv_rerror_rate','Attack_type'
]
# Маппінг атак
DOS_LIST = {'back.', 'land.', 'neptune.', 'pod.', 'smurf.', 'teardrop.'}
R2L_LIST = {'ftp_write.', 'guess_passwd.', 'imap.', 'multihop.', 'phf.', 'spy.', 'warezclient.',
'warezmaster.'}
U2R_LIST = {'buffer_overflow.', 'loadmodule.', 'perl.', 'rootkit.'}
PROBE_LIST = {'ipsweep.', 'nmap.', 'portsweep.', 'satan.'}
def map_to_group(x):
    if x in DOS_LIST:
        return 'dos'
    if x in R2L_LIST:
```

```
        return 'r2l'
    if x in U2R_LIST:
        return 'u2r'
    if x in PROBE_LIST:
        return 'probe'
    if x == 'normal.':
        return 'normal'
    return 'other'
# Завантаження і підготовка датасету
def load_and_prepare(path):
    if not os.path.exists(path):
        raise FileNotFoundError(f"Файл не знайдено: {path}")
    df = pd.read_csv(path, header=None, names=COLS)
    df['group'] = df['Attack_type'].map(map_to_group)
    df = df[df['group'].isin({'dos','r2l','u2r','probe','normal'})].reset_index(drop=True)
    for c in df.columns:
        if df[c].dtype == object and c not in ['Attack_type','group']:
            le = LabelEncoder()
            df[c] = le.fit_transform(df[c])
    X = df.drop(columns=['Attack_type','group']).values.astype(np.float32)
    label_order = ['dos','r2l','u2r','probe','normal']
    y_indices = df['group'].map({k:i for i,k in enumerate(label_order)}).values
    y_onehot = np.zeros((len(y_indices), 5), dtype=np.float32)
    y_onehot[np.arange(len(y_indices)), y_indices] = 1.0
    return X, y_onehot, y_indices, label_order
X_all, Y_onehot_all, Y_idx_all, LABELS = load_and_prepare(FILE_PATH)
n_samples, n_features = X_all.shape
print(f"Фіч: {n_features}, зразків: {n_samples}")
# Розбиття на train/test
train_X, test_X, train_y_onehot, test_y_onehot = train_test_split(
    X_all, Y_onehot_all, train_size=TRAIN_SHARE, random_state=RND, stratify=Y_idx_all,
shuffle=True
)
# конвертація у тензори
Xtr = torch.tensor(train_X, dtype=torch.float32).to(DEVICE)
```

```
Ytr = torch.tensor(train_y_onehot, dtype=torch.float32).to(DEVICE)
Xte = torch.tensor(test_X, dtype=torch.float32).to(DEVICE)
Yte = torch.tensor(test_y_onehot, dtype=torch.float32).to(DEVICE)
# Структуємо модель
class MLPNet(nn.Module):
    def __init__(self, inp_dim, h1=55, h2=28, h3=12, out_dim=5):
        super().__init__()
        self.fc1 = nn.Linear(inp_dim, h1)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(h1, h2)
        self.act2 = nn.ReLU()
        self.fc3 = nn.Linear(h2, h3)
        self.act3 = nn.ReLU()
        self.out = nn.Linear(h3, out_dim)
        self.final = nn.Sigmoid()
    def forward(self, x):
        x = self.act1(self.fc1(x))
        x = self.act2(self.fc2(x))
        x = self.act3(self.fc3(x))
        x = self.final(self.out(x))
        return x
model = MLPNet(n_features).to(DEVICE)
# Оптимізатор і loss
optimizer = optim.Adam(model.parameters(), lr=LR)
loss_fn = nn.MSELoss()
# Підготовка індексів батчів
num_train = Xtr.shape[0]
batch_starts = list(range(0, num_train, BATCH))
best_mse = float('inf')
best_state = None
history_mse = []
# тренування
for epoch in range(EPOCHS):
    model.train()
    epoch_loss = 0.0
```

```
    perm = torch.randperm(num_train, device=DEVICE)
    Xtr_shuffled = Xtr[perm]
    Ytr_shuffled = Ytr[perm]
    for s in batch_starts:
        xb = Xtr_shuffled[s:s+BATCH]
        yb = Ytr_shuffled[s:s+BATCH]
        preds = model(xb)
        loss = loss_fn(preds, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item() * xb.size(0)
    # оцінка на тесті кожну епоху
    model.eval()
    with torch.no_grad():
        y_pred_test = model(Xte)
        mse = loss_fn(y_pred_test, Yte).item()
    history_mse.append(mse)
    if mse < best_mse:
        best_mse = mse
        best_state = copy.deepcopy(model.state_dict())
    if epoch % 20 == 0 or epoch == EPOCHS-1:
        print(f"Епоха {epoch+1}/{EPOCHS} - train_loss≈{epoch_loss/num_train:.5f},
test_mse={mse:.6f}, best_mse={best_mse:.6f}")
# повертаємо найкращу модель
if best_state is not None:
    model.load_state_dict(best_state)
# перетворення виходу в класи і підрахунок відсотків
model.eval()
with torch.no_grad():
    all_preds = model(torch.tensor(X_all, dtype=torch.float32).to(DEVICE)).cpu().numpy()  # (N,5)
pred_indices = np.argmax(all_preds, axis=1)
true_indices = np.argmax(Y_onehot_all, axis=1)
acc = (pred_indices == true_indices).mean() * 100.0
print(f"\nЗагальна точність на всьому датасеті: {acc:.2f}%")
```

```
label_map = {0:'dos',1:'r2l',2:'u2r',3:'probe',4:'normal'}
pred_counts = Counter(label_map[i] for i in pred_indices)
true_counts = Counter(label_map[i] for i in true_indices)
# докладніша статистика
per_class_acc = {}
for cls_idx, cls_name in label_map.items():
    mask = (true_indices == cls_idx)
    if mask.sum() == 0:
        per_class_acc[cls_name] = None
    else:
        per_class_acc[cls_name] = (pred_indices[mask] == cls_idx).mean() * 100.0
print("\nТочність по класах (у %):")
for k in ['dos','r2l','u2r','probe','normal']:
    v = per_class_acc[k]
    if v is None:
        print(f"  {k}: - (немає зразків)")
    else:
        print(f"  {k}: {v:.2f}%")

# MLP+CNN
import os
import copy
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler
# Конфігурація
DATA_FILE = r'R:/KDD99/kddcup.data_10_percent_corrected'
SEED = 42
BATCH = 256
EPOCHS = 40
```

```
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
NORMAL_BOOST = 3.0
np.random.seed(SEED)
torch.manual_seed(SEED)
COLS = [
    'duration','protocol_type','service','flag','src_bytes','dst_bytes','land','wrong_fragment','urgent',
    'hot','num_failed_logins','logged_in','num_compromised','root_shell','su_attempted','num_root',
'num_file_creations','num_shells','num_access_files','num_outbound_cmds','is_host_login','is_guest
_login','count','srv_count','serror_rate','srv_serror_rate','rerror_rate','srv_rerror_rate','same_srv_rate',
    'diff_srv_rate','srv_diff_host_rate','dst_host_count','dst_host_srv_count','dst_host_same_srv_rate',
'dst_host_diff_srv_rate','dst_host_same_src_port_rate','dst_host_srv_diff_host_rate','dst_host_serror
_rate','dst_host_srv_serror_rate','dst_host_rerror_rate','dst_host_srv_rerror_rate','Attack_type'
]
# Завантаження
def load_and_onehot(path):
    df = pd.read_csv(path, header=None, names=COLS)
    dos = {'back.','land.','neptune.','pod.','smurf.','teardrop.'}
    r2l = {'ftp_write.','guess_passwd.','imap.','multihop.','phf.','spy.','warezclient.','warezmaster.'}
    u2r = {'buffer_overflow.','loadmodule.','perl.','rootkit.'}
    probe = {'ipsweep.','nmap.','portsweep.','satan.'}
    idx_map = {}
    idx_map.update({k:0 for k in dos})
    idx_map.update({k:1 for k in r2l})
    idx_map.update({k:2 for k in u2r})
    idx_map.update({k:3 for k in probe})
    idx_map['normal.'] = 4
    indices = df['Attack_type'].map(lambda v: idx_map.get(v, 4)).astype(int).values
    onehots = np.eye(5, dtype=np.float32)[indices]
    df = df.drop(columns=['Attack_type'])
    # Категоріальні кодуємо через pandas
    for c in ['protocol_type','service','flag']:
        df[c] = pd.Categorical(df[c]).codes
    X = df.values.astype(np.float32)
    return X, onehots, indices
# Датасет
```

```
class KDDDataset(Dataset):
    def __init__(self, X, y_idx):
        self.X = torch.from_numpy(X).float()
        self.y = torch.from_numpy(np.array(y_idx)).long()
    def __len__(self):
        return len(self.X)
    def __getitem__(self, i):
        return self.X[i], self.y[i]
# Невелика CNN-модель
class BetterNet(nn.Module):
    def __init__(self, in_features=41, n_classes=5):
        super().__init__()
        self.conv1 = nn.Conv1d(1, 16, kernel_size=5, padding=2)
        self.bn1 = nn.BatchNorm1d(16)
        self.conv2 = nn.Conv1d(16, 32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(32)
        self.pool = nn.MaxPool1d(2)
        conv_out = (in_features // 2) * 32
        self.fc1 = nn.Linear(conv_out, 128)
        self.drop = nn.Dropout(0.3)
        self.fc2 = nn.Linear(128, n_classes)
    def forward(self, x):
        x = x.unsqueeze(1)
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.drop(x)
        return self.fc2(x)
# Метрика
def compute_recalls(model, X_np, y_idx_np, device=DEVICE):
    model.eval()
    with torch.no_grad():
        X_t = torch.from_numpy(X_np).float().to(device)
        logits = model(X_t)
```

```
        preds = logits.argmax(dim=1).cpu().numpy()
    true = np.array(y_idx_np)
    recalls = {}
    for c, name in enumerate(['dos','r2l','u2r','probe','normal']):
        tp = int(((preds == c) & (true == c)).sum())
        tot = int((true == c).sum())
        recalls[name] = (tp, tot, (tp / tot * 100) if tot > 0 else 0.0)
    overall = (preds == true).mean() * 100
    return recalls, overall
# Тренувальний цикл
def train_loop_focus_normal(model, opt, loss_fn, train_loader, val_loader, X_val_full,
y_val_idx_full, epochs=EPOCHS, device=DEVICE):
    best_normal_recall = -1.0
    best_state = None
    for ep in range(1, epochs+1):
        model.train()
        running = 0.0
        for xb, yb in train_loader:
            xb = xb.to(device); yb = yb.to(device)
            logits = model(xb)
            loss = loss_fn(logits, yb)
            opt.zero_grad(); loss.backward(); opt.step()
            running += loss.item() * xb.size(0)
        train_loss = running / len(train_loader.dataset)
        # Валідація
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for xb, yb in val_loader:
                xb, yb = xb.to(device), yb.to(device)
                val_loss += loss_fn(model(xb), yb).item() * xb.size(0)
        val_loss /= len(val_loader.dataset)
        recalls, overall = compute_recalls(model, X_val_full, y_val_idx_full, device=device)
        normal_tp, normal_tot, normal_pct = recalls['normal']
        # зберігаємо модель з найкращим recall
```

```
        if normal_pct > best_normal_recall:
            best_normal_recall = normal_pct
            best_state = copy.deepcopy(model.state_dict())
        print(f"[{ep}/{epochs}] train_loss={train_loss:.4f} val_loss={val_loss:.4f} normal_recall={normal_pct:.2f}% overall={overall:.2f}%")
        print("   per-class recalls:", {k: f"{v[0]}/{v[1]} ({v[2]:.2f}%)" for k,v in recalls.items()})
    return best_state, best_normal_recall
def main():
    if not os.path.exists(DATA_FILE):
        raise FileNotFoundError(f"Файл не знайдено: {DATA_FILE}")
    X, y_onehot, y_idx = load_and_onehot(DATA_FILE)
    print("Loaded X,y shapes:", X.shape, y_onehot.shape)
    X_tr, X_val, idx_tr, idx_val = train_test_split(X, y_idx, train_size=0.75, random_state=SEED, stratify=y_idx)
    # Стандартизація
    scaler = StandardScaler().fit(X_tr)
    X_tr_s = scaler.transform(X_tr)
    X_val_s = scaler.transform(X_val)
    X_all_s = scaler.transform(X)
    # Підготовка датасетів
    train_ds = KDDDataset(X_tr_s, idx_tr)
    val_ds = KDDDataset(X_val_s, idx_val)
    unique, counts = np.unique(idx_tr, return_counts=True)
    class_counts = dict(zip(unique, counts))
    print("Train class counts:", class_counts)
    n_classes = 5
    base_weights = np.zeros(n_classes, dtype=np.float32)
    total = len(idx_tr)
    for c in range(n_classes):
        cnt = class_counts.get(c, 0)
        base_weights[c] = (total / (cnt + 1e-9)) if cnt>0 else 0.0
    base_weights[4] *= NORMAL_BOOST
    base_weights = base_weights / base_weights.sum() * n_classes
    class_weights_t = torch.from_numpy(base_weights).float().to(DEVICE)
```

```
    print("Class weights (used in CrossEntropyLoss):", dict(zip(range(n_classes), base_weights.tolist())))
    sample_weights = np.array([base_weights[lbl] for lbl in idx_tr], dtype=np.float32)
    sampler = WeightedRandomSampler(weights=sample_weights, num_samples=len(sample_weights), replacement=True)
    train_loader = DataLoader(train_ds, batch_size=BATCH, sampler=sampler, drop_last=False)
    val_loader = DataLoader(val_ds, batch_size=BATCH, shuffle=False)
    model = BetterNet(in_features=X.shape[1], n_classes=n_classes).to(DEVICE)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-5)
    loss_fn = nn.CrossEntropyLoss(weight=class_weights_t)  # мультикласові логіти
    best_state, best_normal = train_loop_focus_normal(model, optimizer, loss_fn, train_loader, val_loader,
                                                      X_val_s, idx_val, epochs=EPOCHS, device=DEVICE)
    if best_state is not None:
        model.load_state_dict(best_state)
    print(f"\nLoaded best model by normal recall: {best_normal:.2f}% on validation")
    # Оцінка на всьому датасеті
    recalls_all, overall_all = compute_recalls(model, X_all_s, y_idx, device=DEVICE)
    print("\nFinal recalls on full dataset:")
    for k,v in recalls_all.items():
        print(f" {k:>6}: {v[0]}/{v[1]} -> {v[2]:.2f}%")
    print(f"\nOverall accuracy: {overall_all:.2f}%")

if __name__ == '__main__':
    main()

# Гібрид (Hopfield + Transformer)
import os
import copy
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler
# Конфігурація
DATA_FILE = r'R:/KDD99/kddcup.data_10_percent_corrected'
SEED = 42
BATCH = 256
EPOCHS = 30
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
np.random.seed(SEED)
torch.manual_seed(SEED)
COLS = [
    'duration','protocol_type','service','flag','src_bytes','dst_bytes','land','wrong_fragment','urgent',
    'hot','num_failed_logins','logged_in','num_compromised','root_shell','su_attempted','num_root',
'num_file_creations','num_shells','num_access_files','num_outbound_cmds','is_host_login','is_guest
_login','count','srv_count','serror_rate','srv_serror_rate','rerror_rate','srv_rerror_rate','same_srv_rate',
    'diff_srv_rate','srv_diff_host_rate','dst_host_count','dst_host_srv_count','dst_host_same_srv_rate',
'dst_host_diff_srv_rate','dst_host_same_src_port_rate','dst_host_srv_diff_host_rate','dst_host_serror
_rate', 'dst_host_srv_serror_rate','dst_host_rerror_rate','dst_host_srv_rerror_rate','Attack_type'
]
# Утиліти даних
def load_and_prepare(path):
    df = pd.read_csv(path, header=None, names=COLS)
    dos = {'back.','land.','neptune.','pod.','smurf.','teardrop.'}
    r2l = {'ftp_write.','guess_passwd.','imap.','multihop.','phf.','spy.','warezclient.','warezmaster.'}
    u2r = {'buffer_overflow.','loadmodule.','perl.','rootkit.'}
    probe = {'ipsweep.','nmap.','portsweep.','satan.'}
    idx_map = {}
    idx_map.update({k:0 for k in dos})
    idx_map.update({k:1 for k in r2l})
    idx_map.update({k:2 for k in u2r})
    idx_map.update({k:3 for k in probe})
    idx_map['normal.'] = 4
    y_idx = df['Attack_type'].map(lambda v: idx_map.get(v, 4)).astype(int).values
    y_onehot = np.eye(5, dtype=np.float32)[y_idx]
    df = df.drop(columns=['Attack_type'])
```

```
    # кодуэмо категоріально за допомогою кодів pandas
    for c in ['protocol_type','service','flag']:
        df[c] = pd.Categorical(df[c]).codes
    X = df.values.astype(np.float32)
    return X, y_onehot, y_idx
# Датасет
class TabDataset(Dataset):
    def __init__(self, X, y_idx):
        self.X = torch.from_numpy(X).float()
        self.y = torch.from_numpy(np.array(y_idx)).long()
    def __len__(self):
        return len(self.X)
    def __getitem__(self, i):
        return self.X[i], self.y[i]
# Прототип модуля, подібного до Хопфілда
class HopfieldPrototypes:
    def __init__(self, n_classes=5):
        self.n_classes = n_classes
        self.prototypes = None
        self.medians = None
    def fit(self, X_train, y_train_idx):
        med = np.median(X_train, axis=0)
        self.medians = med
        Xb = np.where(X_train > med, 1.0, -1.0)  # bipolar
        protos = np.zeros((self.n_classes, Xb.shape[1]), dtype=np.float32)
        for c in range(self.n_classes):
            sel = Xb[np.where(y_train_idx == c)]
            if sel.shape[0] == 0:
                protos[c, :] = 0.0
            else:
                p = sel.mean(axis=0)
                norm = np.linalg.norm(p) + 1e-9
                protos[c, :] = (p / norm).astype(np.float32)
        self.prototypes = protos
    def transform(self, X):
```

```
        if self.prototypes is None or self.medians is None:
            raise RuntimeError("HopfieldPrototypes not fitted")
        Xb = np.where(X > self.medians, 1.0, -1.0).astype(np.float32)
        sims = Xb.dot(self.prototypes.T)
        return sims
# Табличний кодер трансформаторного типу
class TabTransformerEncoder(nn.Module):
    def __init__(self, n_features, d_model=64, n_heads=4, n_layers=2, dropout=0.1):
        super().__init__()
        self.n_features = n_features
        self.d_model = d_model
        self.feature_proj = nn.Linear(1, d_model)
        self.feature_id_embed = nn.Embedding(n_features, d_model)
        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=n_heads,
dropout=dropout, dim_feedforward=128, activation='relu')
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=n_layers)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x):
        B, F = x.size()
        x_unsq = x.unsqueeze(-1)
        proj = self.feature_proj(x_unsq)
        ids = torch.arange(F, device=x.device).unsqueeze(0).expand(B, -1)
        id_emb = self.feature_id_embed(ids)
        seq = proj + id_emb
        seq = seq.permute(1, 0, 2)
        out = self.transformer(seq)
        out = out.permute(1, 0, 2)
        pooled = out.mean(dim=1)
        pooled = self.dropout(pooled)
        return pooled
# Повністю гібридна модель
class HybridTransformerHopfield(nn.Module):
    def __init__(self, n_features, n_classes=5, d_model=64, hopfield_dim=None):
        super().__init__()
        self.encoder = TabTransformerEncoder(n_features=n_features, d_model=d_model)
```

```
        self.n_classes = n_classes
        total_dim = d_model + n_classes
        self.fc1 = nn.Linear(total_dim, 128)
        self.drop = nn.Dropout(0.3)
        self.fc2 = nn.Linear(128, n_classes)
    def forward(self, x, hop_sims):
        enc = self.encoder(x)
        cat = torch.cat([enc, hop_sims], dim=1)
        h = F.relu(self.fc1(cat))
        h = self.drop(h)
        logits = self.fc2(h)
        return logits
# Помічник з навчання / оцінювання
def compute_recalls(model, hop_module, scaler, X_np, y_idx_np, device=DEVICE):
    model.eval()
    with torch.no_grad():
        Xs = scaler.transform(X_np)
        X_t = torch.from_numpy(Xs).float().to(device)
        hop_sims = torch.from_numpy(hop_module.transform(X_np)).float().to(device)
        logits = model(X_t, hop_sims)
        preds = logits.argmax(dim=1).cpu().numpy()
    true = np.array(y_idx_np)
    classes = ['dos','r2l','u2r','probe','normal']
    recalls = {}
    for i,name in enumerate(classes):
        tp = int(((preds == i) & (true == i)).sum())
        tot = int((true == i).sum())
        recalls[name] = (tp, tot, (tp/tot*100) if tot>0 else 0.0)
    overall = (preds == true).mean() * 100
    return recalls, overall
def train_loop(model, hop_module, scaler, train_loader, val_X, val_y_idx, optimizer, loss_fn,
epochs=EPOCHS, device=DEVICE):
    best_state = None
    best_metric = -1.0
    for ep in range(1, epochs+1):
```

```
        model.train()
        running = 0.0
        for xb, yb in train_loader:
            xb = xb.to(device)
            yb = yb.to(device)
            raise RuntimeError
    return best_state
# Тренувальна програма з симуляторами стрибків для кожного зразка
def run_training(X, y_onehot, y_idx):
    X_tr, X_val, idx_tr, idx_val = train_test_split(X, y_idx, train_size=0.75, random_state=SEED,
stratify=y_idx)
    scaler = StandardScaler().fit(X_tr)
    X_tr_s = scaler.transform(X_tr)
    X_val_s = scaler.transform(X_val)
    hop = HopfieldPrototypes(n_classes=5)
    hop.fit(X_tr, idx_tr)
    hop_sims_tr = hop.transform(X_tr)
    hop_sims_val = hop.transform(X_val)
    class HybridDataset(torch.utils.data.Dataset):
        def __init__(self, X_scaled, hop_sims, y_idx):
            self.Xs = torch.from_numpy(X_scaled).float()
            self.hop = torch.from_numpy(hop_sims).float()
            self.y = torch.from_numpy(np.array(y_idx)).long()
        def __len__(self):
            return len(self.Xs)
        def __getitem__(self, i):
            return self.Xs[i], self.hop[i], self.y[i]
    train_ds = HybridDataset(X_tr_s, hop_sims_tr, idx_tr)
    val_ds   = HybridDataset(X_val_s, hop_sims_val, idx_val)
    # ваги класів для CrossEntropy
    unique, counts = np.unique(idx_tr, return_counts=True)
    class_counts = dict(zip(unique, counts))
    n_classes = 5
    total = len(idx_tr)
    class_weights = np.zeros(n_classes, dtype=np.float32)
```

```
    for c in range(n_classes):
        cnt = class_counts.get(c, 0)
        class_weights[c] = (total / (cnt + 1e-9)) if cnt>0 else 0.0
    # нормалізувати масштаб
    class_weights = class_weights / class_weights.sum() * n_classes
    class_weights_t = torch.from_numpy(class_weights).float().to(DEVICE)
    sample_weights = np.array([class_weights[lbl] for lbl in idx_tr], dtype=np.float32)
    sampler = WeightedRandomSampler(weights=sample_weights,
num_samples=len(sample_weights), replacement=True)
    def collate_fn(batch):
        Xb = torch.stack([b[0] for b in batch], dim=0)
        Hopb = torch.stack([b[1] for b in batch], dim=0)
        yb = torch.stack([b[2] for b in batch], dim=0)
        return Xb, Hopb, yb
    train_loader = DataLoader(train_ds, batch_size=BATCH, sampler=sampler,
collate_fn=collate_fn)
    val_loader   = DataLoader(val_ds, batch_size=BATCH, shuffle=False, collate_fn=collate_fn)
    model = HybridTransformerHopfield(n_features=X.shape[1], n_classes=n_classes,
d_model=64).to(DEVICE)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-5)
    loss_fn = nn.CrossEntropyLoss(weight=class_weights_t)
    best_state = None
    best_normal_recall = -1.0
    for ep in range(1, EPOCHS+1):
        model.train()
        running_loss = 0.0
        for Xb, Hopb, yb in train_loader:
            Xb = Xb.to(DEVICE)
            Hopb = Hopb.to(DEVICE)
            yb = yb.to(DEVICE)
            logits = model(Xb, Hopb)
            loss = loss_fn(logits, yb)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
            running_loss += loss.item() * Xb.size(0)
        train_loss = running_loss / len(train_loader.dataset)
        # метрики перевірки з використанням попередньо обчисленого hop_sims_val та скалера
        recalls, overall = compute_recalls(model, hop, scaler, X_val, idx_val, device=DEVICE)
        normal_recall = recalls['normal'][2]
        if normal_recall > best_normal_recall:
            best_normal_recall = normal_recall
            best_state = copy.deepcopy(model.state_dict())
        print(f"[Epoch {ep}/{EPOCHS}] train_loss={train_loss:.4f}
val_normal_recall={normal_recall:.2f}% overall_val={overall:.2f}%")
        print("   per-class (val):", {k: f"{v[0]}/{v[1]} ({v[2]:.2f}%)" for k,v in recalls.items()})
    # Завантаження накращого
    if best_state is not None:
        model.load_state_dict(best_state)
    # остаточна оцінка повного набору даних
    recalls_full, overall_full = compute_recalls(model, hop, scaler, X, y_idx, device=DEVICE)
    print("\nFinal recalls on full dataset:")
    for k,v in recalls_full.items():
        print(f" {k:>6}: {v[0]}/{v[1]} -> {v[2]:.2f}%")
    print(f"\nOverall accuracy: {overall_full:.2f}%")
    return model, hop, scaler
if __name__ == '__main__':
    if not os.path.exists(DATA_FILE):
        raise FileNotFoundError(f"Файл не знайдено: {DATA_FILE}")
    X, y_onehot, y_idx = load_and_prepare(DATA_FILE)
    model, hop_mod, scaler_obj = run_training(X, y_onehot, y_idx)


# CNN
import os
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
```

```
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
# Налаштування
DATA_FILE = r'R:/KDD99/kddcup.data_10_percent_corrected'
RND_SEED = 4
TRAIN_RATIO = 0.75
BATCH_SZ = 256
EPOCHS = 50
LR = 1e-3
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# встановимо сіда для відтворюваності
np.random.seed(RND_SEED)
torch.manual_seed(RND_SEED)
COLS = [
 'duration','protocol_type','service','flag','src_bytes','dst_bytes','land','wrong_fragment','urgent','hot',
'num_failed_logins','logged_in','num_compromised','root_shell','su_attempted','num_root','num_file
_creations','num_shells','num_access_files','num_outbound_cmds','is_host_login','is_guest_login','c
ount','srv_count','serror_rate','srv_serror_rate','rerror_rate','srv_rerror_rate','same_srv_rate','diff_srv_
rate','srv_diff_host_rate',
'dst_host_count','dst_host_srv_count','dst_host_same_srv_rate','dst_host_diff_srv_rate','dst_host_sa
me_src_port_rate',
 'dst_host_srv_diff_host_rate','dst_host_serror_rate','dst_host_srv_serror_rate','dst_host_rerror_rate',
 'dst_host_srv_rerror_rate','Attack_type'
]
# Маппінг атак
DOS_SET   = {'back.','land.','neptune.','pod.','smurf.','teardrop.'}
R2L_SET                                                                    =
{'ftp_write.','guess_passwd.','imap.','multihop.','phf.','spy.','warezclient.','warezmaster.'}
U2R_SET   = {'buffer_overflow.','loadmodule.','perl.','rootkit.'}
PROBE_SET = {'ipsweep.','nmap.','portsweep.','satan.'}
CLASS_ORDER = ['normal','dos','r2l','u2r','probe']
ONE_HOT = {
    'normal.': [1,0,0,0,0],
    **{a: [0,1,0,0,0] for a in DOS_SET},
    **{a: [0,0,1,0,0] for a in R2L_SET},
```

```
    **{a: [0,0,0,1,0] for a in U2R_SET},
    **{a: [0,0,0,0,1] for a in PROBE_SET}
}
def map_attack_to_onehot(lbl: str):
    # fallback до normal якщо зустрінеться невідомий лейбл
    return ONE_HOT.get(lbl, [1,0,0,0,0])
# Зчитування даних
if not os.path.exists(DATA_FILE):
    raise FileNotFoundError(f"Файл не знайдено: {DATA_FILE}")
raw = pd.read_csv(DATA_FILE, header=None, names=COLS)
# Додаємо колонку з one-hot вектором для кожного рядка
raw['onehot'] = raw['Attack_type'].map(map_attack_to_onehot)
# Відокремлюємо фічі
df_features = raw.drop(columns=['Attack_type','onehot']).copy()
for col in df_features.select_dtypes(include=['object']).columns:
    df_features[col] = LabelEncoder().fit_transform(df_features[col].astype(str))
# перетворюємо у числові масиви
X_all = df_features.values.astype(np.float32)
y_all = np.vstack(raw['onehot'].values).astype(np.float32)
print(f"Підготовлено: фічей={X_all.shape[1]}, зразків={X_all.shape[0]}")
# Розбиття
stratify_idx = np.argmax(y_all, axis=1)
X_train, X_test, y_train, y_test = train_test_split(
    X_all, y_all, train_size=TRAIN_RATIO, random_state=RND_SEED, stratify=stratify_idx,
shuffle=True
)
X_train = X_train.reshape(-1, 1, X_all.shape[1])
X_test  = X_test.reshape(-1, 1, X_all.shape[1])
X_full  = X_all.reshape(-1, 1, X_all.shape[1])
train_ds = TensorDataset(torch.from_numpy(X_train).float(), torch.from_numpy(y_train).float())
test_ds  = TensorDataset(torch.from_numpy(X_test).float(),  torch.from_numpy(y_test).float())
full_ds  = TensorDataset(torch.from_numpy(X_full).float(),  torch.from_numpy(y_all).float())
train_loader = DataLoader(train_ds, batch_size=BATCH_SZ, shuffle=True)
test_loader  = DataLoader(test_ds,  batch_size=BATCH_SZ)
full_loader  = DataLoader(full_ds,  batch_size=512)
```

```
# Модель
class ConvKDD(nn.Module):
    def __init__(self, in_ch=1, feat_len=41, out_dim=5):
        super().__init__()
        self.extract = nn.Sequential(
            nn.Conv1d(in_ch, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool1d(2),
            nn.Conv1d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool1d(2)
        )
        flattened = 64 * (feat_len // 4)
        self.head = nn.Sequential(
            nn.Flatten(),
            nn.Linear(flattened, 128),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(128, out_dim)
        )
    def forward(self, x):
        x = self.extract(x)
        x = self.head(x)
        return x
model = ConvKDD(in_ch=1, feat_len=X_full.shape[2], out_dim=5).to(DEVICE)
# Втрата та оптимізатор
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=LR)
# Навчання
best_dev_loss = float('inf')
best_state = None
for epoch in range(1, EPOCHS + 1):
    model.train()
    running_loss = 0.0
    for xb, yb in train_loader:
```

```
        xb = xb.to(DEVICE)
        yb = yb.to(DEVICE)
        optimizer.zero_grad()
        logits = model(xb)
        loss = criterion(logits, yb)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * xb.size(0)
    train_loss = running_loss / len(train_ds)
    # валідація на тесті
    model.eval()
    dev_loss = 0.0
    with torch.no_grad():
        for xb, yb in test_loader:
            xb = xb.to(DEVICE)
            yb = yb.to(DEVICE)
            out = model(xb)
            dev_loss += criterion(out, yb).item() * xb.size(0)
    dev_loss = dev_loss / len(test_ds)
    if dev_loss < best_dev_loss:
        best_dev_loss = dev_loss
        best_state = {k: v.cpu().clone() for k, v in model.state_dict().items()}
    print(f"Епоха {epoch}/{EPOCHS} — Train: {train_loss:.4f}, Test: {dev_loss:.4f}")
# Відновлюємо найкращу модель
if best_state is not None:
    model.load_state_dict(best_state)
# Оцінка на всьому датасеті
model.eval()
accum_preds = []
accum_true = []
with torch.no_grad():
    for xb, yb in full_loader:
        xb = xb.to(DEVICE)
        logits = model(xb)
        probs = torch.sigmoid(logits).cpu().numpy()
```

```
        preds = (probs >= 0.5).astype(int)
        accum_preds.append(preds)
        accum_true.append(yb.numpy().astype(int))
y_pred_all = np.vstack(accum_preds)
y_true_all = np.vstack(accum_true)
class_names = ['normal','dos','r2l','u2r','probe']
print("\nТочність по класах (у %):")
for idx, cname in enumerate(class_names):
    true_positive = np.logical_and(y_true_all[:, idx] == 1, y_pred_all[:, idx] == 1).sum()
    total_actual = (y_true_all[:, idx] == 1).sum()
    recall_pct = 100.0 * true_positive / total_actual if total_actual > 0 else 0.0
    print(f" {cname}: {true_positive}/{total_actual} = {recall_pct:.2f}%")
```