

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ
ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ
ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова
праця на правах рукопису

КОРЕНКО ДМИТРО ВОЛОДИМИРОВИЧ

УДК 004.72

ДИСЕРТАЦІЯ

МЕТОД ТА ЗАСІБ КОНСТРУЮВАННЯ ТРАФІКУ В ПРОГРАМНО-
КОНФІГУРОВАНИХ МЕРЕЖАХ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ

123 Комп'ютерна інженерія

12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Науковий керівник: Кулаков Юрій Олексійович, д.т.н., професор

Київ – 2025

АНОТАЦІЯ

Коренко Д. В. Метод та засіб конструювання трафіку в програмно-конфігурованих мережах на основі штучного інтелекту. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії з галузі знань 12 Інформаційні технології за спеціальністю 123 Комп'ютерна інженерія. – Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ, 2025.

Дисертаційна робота присвячена розробці комплексного методу конструювання трафіку в програмно-конфігурованих мережах на основі методів штучного інтелекту, що дозволяє підвищити продуктивність та ефективність конструювання трафіку у програмно-конфігурованих мережах (SDN) та покращити якість обслуговування (QoS) за рахунок використання методів штучного інтелекту для побудови маршрутів передачі трафіку та його балансування.

Було отримано ряд нових наукових результатів, зокрема, запропоновано та обґрунтовано удосконалену архітектуру системи конструювання трафіку в програмно-конфігурованих мережах на основі методів штучного інтелекту, застосування якої дозволяє спростити процес балансування трафіку в мережі та яка, на відміну від існуючих методів, забезпечує можливість використання різних показників для балансування навантаження в залежності від типу мережі та вимог до неї.

Отримав подальший розвиток метод обрахунку ознак для вибору шляху в програмно-конфігурованих мережах з урахуванням особливостей даного типу мереж та вимог, що виносяться до них.

Розроблено модель нейронної мережі для її використання у задачах конструювання трафіку, а саме балансування навантаження, у програмно-конфігурованих мережах. Розроблену нейронну мережу було

треновано на наборі даних IP Network Traffic Flows Labeled with 75 Apps.

Оптимізовано модель нейронної мережі для її використання у задачах балансування навантаження. Експериментальним методом було визначено оптимальну конфігурацію та налаштування нейронної мережі для запобігання перенавчення та отримання високої точності прогнозування оптимального шляху.

Проведено аналіз результатів застосування запропонованого методу конструювання трафіку в програмно-конфігурованих мережах на основі методів штучного інтелекту, що включає в себе оптимізацію роботи нейронної мережі для виконання задач динамічного балансування трафіку в мереж, підвищення ефективності балансування навантаження в мережі в залежності від обраних ознак шляху та запобігання неконтрольованої зміни навантаження шляху в мережі (джитеру) при передачі потоку даних. За результатами застосування розробленого методу було досягнуто кращу продуктивність мережі на 14%.

Розроблений метод оптимізації нейронної мережі є складовою комплексного методу конструювання трафіку в SDN мережах і дозволяє оптимізувати модель нейронної мережі до використання в програмно-конфігурованих мережах в залежності від висунутих вимог.

Ключові слова: прикладне програмне забезпечення, архітектура програмного забезпечення, SDN мережі, конструювання трафіку, балансування навантаження, оптимізація, комп'ютерні системи, штучний інтелект, нейронні мережі, машинне навчання.

ABSTRACTS

Korenko D.V. Method and mean of traffic engineering in software-defined networks based on artificial intelligence.

Dissertation for the degree of Doctor of Philosophy in the field of knowledge 12 Information Technology, speciality 123 Computer Engineering at the National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, 2025.

The thesis is dedicated to the development of a comprehensive method for traffic engineering in software-defined networks based on artificial intelligence, which allows to increase the productivity and efficiency of traffic engineering in software-defined networks (SDN) and to improve the quality of service (QoS) by using artificial intelligence to build and balance traffic routes.

A number of new scientific results were obtained, in particular, an improved architecture of the traffic design system in software-configurable networks based on artificial intelligence was proposed and substantiated, which allows simplifying the process of traffic balancing in the network and, unlike existing methods, provides the possibility of using different criteria for load balancing depending on the type of network and its requirements.

The method of calculating path selection features in software-configurable networks has been further developed, taking into account the characteristics of and requirements for this type of network.

A neural network model was developed for use in traffic design tasks, namely load balancing, in software configurable networks. The developed neural network was trained on the dataset IP Network Traffic Flows Labeled with 75 Apps.

The neural network model was optimised for use in load balancing tasks. The optimal configuration and tuning of the neural network was

experimentally determined to avoid overtraining and to achieve high accuracy in predicting the optimal path.

An analysis of the results of applying the proposed method of traffic design in software-configurable networks based on artificial intelligence . The method includes optimising the operation of a neural network to perform the tasks of dynamic traffic balancing in networks, increasing the efficiency of load balancing in the network depending on the selected path characteristics, and preventing uncontrolled changes in the load of the path in the network (jitter) during the transmission of the data stream, is carried out. The application of the developed method resulted in an improvement in network performance of 14%.

The developed method of neural network optimisation constitutes a component of a comprehensive method of traffic design in SDN networks, whereby the neural network model can be optimised for use in software-configurable networks in accordance with the requisite specifications.

Keywords: application software, software architecture, SDN networks, traffic engineering, load balancing, optimisation, computer systems, artificial intelligence, neural networks, machine learning.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці в яких опубліковано основні наукові результати дисертації:

1. Kulakov, Y., & Korenko, D. (2021). Modified Method of Traffic Engineering in DCN with a Ramified Topology. International Journal of Advanced Computer Science and Applications, 12(12). ISSN: 2156-5570 ISSN: 2158-107X DOI: [10.14569/ijacsa.2021.0121258](https://doi.org/10.14569/ijacsa.2021.0121258)
2. Loutskii, H., Volokyta, A., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & **Korenko, D.** (2021). Topology synthesis method based on excess de bruijn and dragonfly. In Advances in Computer Science for Engineering and Education IV (pp. 315-325). Springer International Publishing. DOI: [10.1007/978-3-030-80472-5_27](https://doi.org/10.1007/978-3-030-80472-5_27)
3. Volokyta A., Loutskii H., Rehida P., Honcharenko O., **Korenko D.**, Rusinov V., Ivanishchev B., Kaplunov A. Convolutionary neural networks regarding problem of monitoring data balancing in de bruijn topology. Bulgarian Journal for Engineering Design, 2021, Mechanical Engineering Faculty, Technical University-Sofia. ISSN 1313-7530
4. Volokyta, A., Loutskii, H., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & **Korenko, D.** (2022). Extended DragonDeBruijn topology synthesis method. International Journal of Computer Network and Information Security, 9(6), 23. DOI: [10.5815/ijcnis.2022.06.03](https://doi.org/10.5815/ijcnis.2022.06.03)
5. **Korenko, D.**, Cherevatenko, O., Rusinov, V., & Kulakov, Y. (2022). Creation of the method of multipath routing using known paths in software-defined networks. Technology audit and production reserves, 4(2/66), 19-24. DOI: [10.15587/2706-5448.2022.262787](https://doi.org/10.15587/2706-5448.2022.262787)
6. Kulakov, Y. O., & **Korenko, D. V.** Methods of applying artificial intelligence in software-defined networks. Problems of Informatization and Control, 1(73), 2023, 23-27. DOI: [10.18372/2073-4751.73.17640](https://doi.org/10.18372/2073-4751.73.17640)
7. Artem Volokyta, Alla Kogan, Oleksii Cherevatenko, **Dmytro**

Korenko, Dmytro Oboznyi, Yurii Kulakov, "Traffic Engineering with Specified Quality of Service Parameters in Software-defined Networks", International Journal of Computer Network and Information Security(IJCNIS), Vol.16, No.5, pp.1-13, 2024. DOI: [10.5815/ijcnis.2024.05.01](https://doi.org/10.5815/ijcnis.2024.05.01)

8. Кулаков Ю. О., Коренко Д. В. Метод балансування навантаження в мережах SDN з використанням штучного інтелекту. Проблеми інформатизації та управління, 2(78), 2024, ст. 31-39. ISSN 2073-4751 DOI: <https://doi.org/10.18372/2073-4751.78.18959>

Зміст

АНОТАЦІЯ	2
ABSTRACTS	4
СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА	6
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	10
ВСТУП.....	11
РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ КОНСТРУЮВАННЯ ТРАФІКУ У ПРОГРАМНО-КОНФІГУРОВАНИХ МЕРЕЖАХ	17
1.1. Аналіз вимог до програмного забезпечення ШІ для конструювання трафіку.....	17
1.2. Порівняльний аналіз методів конструювання трафіку в SDN мережах	33
1.3. Огляд існуючих систем штучного інтелекту у SDN мережах.....	36
Висновки до розділу 1	45
РОЗДІЛ 2. МЕТОД КОНСТРУЮВАННЯ ТРАФІКУ У ПРОГРАМНО- КОНФІГУРОВАНИХ МЕРЕЖАХ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ .	46
2.1. Архітектура системи конструювання трафіку.....	46
2.2. Обрахування показників вибору шляху	57
2.3. Балансування навантаження на основі штучної нейронної мережі.....	63
Висновки до розділу 2	71
РОЗДІЛ 3. ЗАСОБИ КОНСТРУЮВАННЯ ТРАФІКУ У ПРОГРАМНО- КОНФІГУРОВАНИХ МЕРЕЖАХ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ .	72
3.1. Вибір засобів розробки системи балансування навантаження.....	72
3.2. Формування та попередня обробка навчального набору даних.....	74
3.3. Побудова та навчання моделі нейронної мережі	85
3.4. Реалізація користувацького інтерфейсу	89
3.5. Модифікація SDN контролера ONOS	104
Висновки до розділу 3	117
РОЗДІЛ 4. ТЕСТУВАННЯ ТА АНАЛІЗ МЕТОДУ КОНСТРУЮВАННЯ ТРАФІКУ У ПРОГРАМНО-КОНФІГУРОВАНИХ МЕРЕЖАХ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ	118
4.1. Розробка тестового середовища.....	118

4.2. Аналіз результатів тестування	121
Висновки до розділу 4.....	142
ВИСНОВКИ.....	144
ЛІТЕРАТУРА.....	146
ДОДАТОК А	154
Частина програмного коду	154
ДОДАТОК Б.....	180
Список публікацій здобувача	180

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

SDN – Software-Defined Networking

AI – Artificial Intelligence

ML – Machine Learning

NN – Neural Network

LBBNN – Load Balancing Based on Neural Network

GUI – Graphical User Interface

ONOS - Open Network Operating System

ВСТУП

Актуальність теми. Для практичних завдань реалізації інформаційного обміну в системах реального часу, таких як телемедицина, управління рухомими об'єктами, контроль динамічних процесів у складних розподілених структурах подвійного призначення, критично важливо забезпечити стійку та адаптивну роботу мережі. В умовах високої щільності потоку інформаційних повідомлень, обмеженого часу на обробку та прийняття рішень, а також у випадках збоїв у електроживленні чи пікових навантажень на мережеві ресурси, традиційні методи маршрутизації часто виявляються недостатньо ефективними.

Особливої уваги потребують ситуації, коли мережа зазнає динамічних змін у топології, що ускладнює забезпечення безперервного обміну інформацією. Неповна комбінаторика доступних маршрутів через відмови вузлів або каналів зв'язку призводить до необхідності швидкого знаходження оптимального шляху передачі трафіку з урахуванням поточних умов. Класичні алгоритми маршрутизації, такі як OSPF, RIP або сучасні рішення на основі програмно-конфігурованих мереж (SDN), мають обмеження щодо швидкості адаптації до змін і ефективного розподілу ресурсів у реальному часі.

Використання технології SDN забезпечує більш ефективне використання мережевих ресурсів та знижує витрати на керування мережею. Впровадження програмного керування сприяє швидкому та простому зміні мережевих конфігурацій, дистанційному адмініструванню мережевого обладнання та спостереженню за станом мережі.

Крім того, програмно-конфігуровані мережі забезпечують підвищену безпеку, оскільки вони дозволяють блокувати доступ до певних ресурсів і програм у разі потенційних загроз і встановлювати

правила доступу та авторизації для користувачів мережі.

В свою чергу, застосування штучного інтелекту (ШІ) у сфері інформаційних технологій є ключовим аспектом побудови комп'ютерних мереж. Це дозволяє швидше й ефективніше вирішувати складні проблеми, мінімізувати кількість помилок і підвищити рівень роботи.

З урахуванням цих особливостей та потреб постає актуальна науково-практична проблемна задача розроблення штучного інтелекту для конструювання трафіку, а саме динамічного балансування навантаження в комп'ютерних мережах.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційна робота входить в план наукової роботи кафедри обчислювальної техніки КПІ ім. Ігоря Сікорського і виконана в рамках наступних пошукових досліджень (ініціативних тематик): «Високопродуктивні комп'ютерні системи та мережі: теорія, методи і засоби апаратної та програмної реалізації» (факультет інформатики та обчислювальної техніки – керівник: доц. А. М. Волокита), № договору: Д/р №0121U108261, дата реєстрації: 11.02.2021.

Мета і завдання дослідження. Метою дисертаційної роботи є підвищення ефективності процедури динамічного конструювання трафіку в сучасних комп'ютерних мережах за рахунок використання методів штучного інтелекту, що забезпечує підвищення якості та швидкості передачі даних в програмно-конфігурованих мережах. Це досягається за рахунок зниження середнього часу передачі пакетів, мінімізації втрат даних, підвищення пропускної здатності каналів зв'язку, оптимізації балансування навантаження та покращення точності прогнозування маршруту передачі трафіку.

Для досягнення вказаної мети потрібно вирішити такі завдання:

Дослідити сучасні методи конструювання трафіку в SDN мережах, а саме балансування навантаження.

Дослідити методи використання штучного інтелекту в задачах конструювання трафіку та підвищення якості обслуговування.

Розробити архітектуру системи керування мережевим трафіком з урахуванням особливостей використання штучного інтелекту.

Обрати та обрахувати ознаки шляху для подальшого їх використання у процесі балансування навантаження в SDN мережі.

Розробити модель нейронної мережі з урахуванням особливостей її використання у контексті програмно-конфігурованих мереж та вимог, висунутих до цієї мережі, яка буде не лише адаптивною до змін в мережевому трафіку, але й здатною планувати потоки на основі інтегрального навантаження на кожному шляху.

Об’єкт дослідження – процес конструювання трафіку в програмно-конфігурованих мережах.

Предмет дослідження – методи та засоби конструювання трафіку в програмно-конфігурованих мережах.

Методи дослідження. Методичною основою дослідження є системне опрацювання та аналіз теоретичного матеріалу, присвяченого підвищенню продуктивності та ефективності конструювання трафіку у SDN мережах, підвищення якості обслуговування (QoS) за рахунок використання методів штучного інтелекту, адаптації нейронних мереж до задач балансування навантаження. В процесі даного дослідження були використані методи балансування навантаження, що базуються на використанні ознак шляху, для побудови маршрутів.

Наукова новизна отриманих результатів.

Запропоновано та обґрунтовано модифікований метод конструювання трафіку в SDN мережах на основі методів штучного інтелекту, який враховує особливості SDN-архітектури та вимоги до неї. На відміну від існуючих підходів, метод базується на динамічному аналізі параметрів мережевого трафіку та прогнозуванні його, що дозволяє адаптивно балансувати навантаження та інтегрувати механізми

глибокого навчання для врахування комплексних метрик, які впливають на продуктивність мережі. Особливістю запропонованого методу є використання інтегрального показника вибору оптимального маршруту, що сприяє рівномірному розподілу трафіку та ефективному використанню мережевих ресурсів. На основі запропонованого методу було розроблено засіб конструювання трафіку в SDN мережах.

Запропоновано та обґрунтовано удосконалену архітектуру системи конструювання трафіку в програмно-конфігурованих мережах на основі методів штучного інтелекту, яка, на відміну від існуючих методів, забезпечує можливість використання інтегрального показника для балансування навантаження в залежності від типу трафіка.

Отримав подальший розвиток спосіб обрахунку показників для вибору шляху в програмно-конфігурованих мережах з урахуванням особливостей мереж та вимог, висунутих до них, що надає можливість використовувати комплексний метод конструювання трафіку, який включає у себе метод обрахунку показників шляху та балансування навантаження. Це, на відміну від існуючих підходів, дозволяє адаптувати модель нейронної мережі до вимог, які висунуті до мережі, та враховувати глобальний стан мережі для конструювання трафіку.

Практичне значення отриманих результатів. Одержані результати дозволяють застосовувати штучний інтелект для конструювання трафіку у програмно-конфігурованих мережах.

Розроблений метод дозволяє використовувати комплексний підхід та враховувати глобальний стан мережі у задачі балансування навантаження, що забезпечує зниження часу передачі пакетів на 14%, зменшення втрат пакетів до 3% та підвищення пропускної здатності на 17,1% порівняно з традиційними алгоритмами.

Експериментальним способом було визначено оптимальну конфігурацію та налаштування нейронної мережі, що дозволяє уникнути перенавчання та досягти точності прогнозування маршруту на

рівні 96,4%.

Розроблена модель має високу ефективність у задачі прогнозування оптимального шляху передачі пакетів. Вона може бути використана як основа для подальших досліджень та вдосконалення мережевих алгоритмів маршрутизації, а також для розробки нових методів оптимізації передачі даних у сучасних комп'ютерних мережах.

Особистий внесок здобувача. Дисертація є результатом самостійних наукових досліджень, в яких вкладено авторський підхід у метод конструювання трафіку у програмно-конфігурованих мережах на основі методів штучного інтелекту. Наукові положення та основні результати, які містяться в дисертації, отримані здобувачем самостійно у процесі науково-дослідницької роботи. В роботах, опублікованих у співавторстві, дисертанту належать:

[1] – розроблений комплексний метод балансування навантаження у програмно-конфігурованих мережах на основі методів штучного інтелекту;

[2] – запропоновано метод обрахунку ознак для вибору шляху в програмно-конфігурованих мережах з урахуванням особливостей мереж та вимог, що виносяться до них

[3] – адаптована модель нейронної мережі під задачі балансування навантаження з урахуванням особливостей програмно-конфігурованих мереж;

[4] – проаналізовано вплив якості набору даних для тренування моделей на точність вибору оптимального шляху;

[5] – проаналізовано вплив розміру порції даних на час обробки даних нейронною мережею та на похибку в розрахунках;

[6] – проаналізовано вплив кількості нейронів у прихованому шарі на час обробки даних нейронною мережею та на похибку в розрахунках;

[7] – аналіз продуктивності та ефективності балансування навантаження нейронною мережею.

Апробація результатів дисертації. Основні результати роботи опубліковано та обговорено на міжнародних та всеукраїнських наукових конференціях, зокрема на: Bulgarian Journal for Engineering Design, 2021, Mechanical Engineering Faculty, Technical University-Sofia. ISSN 1313-7530 (м. Софія, 2021), XV Міжнародна Науково-Практична Конференція, Комп'ютерні Системи Та Мережні Технології (м. Київ, 2024).

Публікації. За результатами дисертаційних досліджень опубліковано 7 наукових статей, 4 з яких входять до наукометричних баз даних з міжнародним індексом цитування Scopus, а 3 – фахові видання категорії Б.

Структура і обсяг роботи. Дисертаційна робота складається зі вступу, чотирьох розділів, загальних висновків, списку використаних джерел із 72 найменувань та додатків. Загальний обсяг дисертації становить 180 сторінок, з яких 131 сторінки основного тексту, та містить 62 рисунки, 12 формул, 5 таблиць.

РОЗДІЛ 1

АНАЛІЗ МЕТОДІВ КОНСТРУЮВАННЯ ТРАФІКУ У ПРОГРАМНО-КОНФІГУРОВАНИХ МЕРЕЖАХ

1.1. Аналіз вимог до програмного забезпечення ШІ для конструювання трафіку

Сучасна комп'ютерна мережа – це розподілена мережа з комутацією пакетів, яка підключена майже до кожного цифрового пристрою та доступна в усьому світі. Традиційна мережа має кілька проблем, як-от залежність від постачальників, складність керування великою мережею, динамічна зміна політики пересилання тощо. Щоб подолати проблеми, властиві існуючій традиційній мережі, була розроблена концепція програмно-конфігурованої мережі [1].

Концепція програмно-конфігурованих мереж базується на ідеї програмованих мереж із централізованим керуванням. Цей підхід пропонує спрощене рішення складних завдань, таких як керування трафіку [2], оптимізація мережі [3] та балансування навантаження [4]. Крім того, сучасні мережеві додатки вимагають більш масштабованої архітектури, яка може надавати надійні послуги на основі певного типу трафіку. Цього можна досягти завдяки використанню технології SDN, яка пропонує комплексне керування станом мережі та полегшує керування на рівні потоку [5]. Ця ідея призвела до кардинальних змін у проектуванні та конфігурації мереж [6-9].

Застосування технології SDN при побудові комп'ютерних мереж допомагає забезпечити більш ефективне використання мережевих ресурсів та знижує витрати на керування мережею.

Впровадження програмно-конфігурованої мережі дає доступ до швидкої та адаптивної зміну мережевих конфігурацій, тим самим

помітно спрощуючи процес створення інфраструктури. Завдяки цій технології адміністратори можуть віддалено керувати мережевими обладнанням, налаштовувати та оптимізувати його роботу без необхідності фізичної присутності на об'єкті. Це значно підвищує ефективність роботи, оскільки дозволяє оперативно реагувати на зміни в мережі або вирішувати технічні проблеми.

Крім того, програмне керування дозволяє здійснювати безперервний моніторинг стану мережі в режимі реального часу. Це дозволяє відстежувати активність мережі, швидко виявляти збої або проблеми та виправляти проблеми до того, як вони матимуть негативний вплив на загальну продуктивність. Впровадження автоматизованих систем збору та аналізу даних дозволяє підтримувати роботу мережі та прогнозувати потенційні проблеми на основі аналізу поточних показників. Це також забезпечує вищий рівень безпеки, оскільки дозволяє оперативно змінювати правила доступу, виявляти аномальну активність і захищати мережу від потенційних загроз.

Парадигма SDN передбачає відокремлення рівня керування від рівня інфраструктури, що дозволяє досягти більш оперативного і динамічного підходу в порівнянні зі звичайною архітектурою мережі. Рівень керування може бути розділений на декілька віртуальних мереж, кожна з яких реалізує окрему політику [10]. Отже, цю парадигму можна розглядати як інструмент, що дозволяє вирішувати різноманітні мережеві проблеми з різних точок зору [6]. Крім того, вона може бути використана для задоволення вимог нових технологій, таких як Інтернет речей (IoT) і 5G [11]. Тим не менш, її прийняття залежить від її здатності забезпечити відповідне рішення проблем, які не можуть бути вирішені традиційними мережевими протоколами та архітектурами [12]. Ряд великих корпорацій, включаючи Microsoft та Google, вже прийняли її для своїх центрів обробки даних [13, 14]. З іншого боку, штучний інтелект надає значний потенціал для розвитку інновацій в SDN.

Архітектура SDN розроблена на основі ідеї поділу між площиною керування та площиною даних (Control Plane і Data Plane). (рис. 1.1).

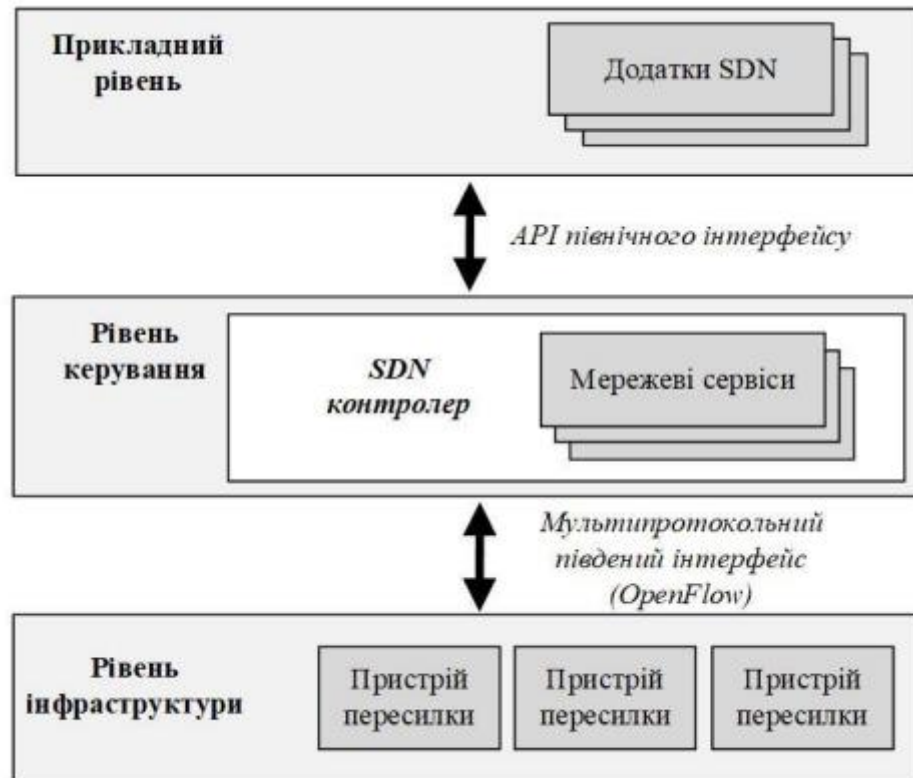


Рис. 1.1 Базова архітектура SDN [15]

У традиційних мережах, керування та обробка пакетів відбуваються на кожному мережевому пристрої окремо. У SDN ці функції розділені, що дозволяє зробити мережу більш гнучкою, масштабованою та простішою у керуванні [15].

- **Рівень додатків (Management Plane):** цей рівень включає різні мережеві служби та програми, які керують поведінкою мережі, як-от балансування навантаження, файерволи тощо. Площина керування конфігурує та контролює мережеві пристрої.
- **Рівень керування (Control Plane):** цей рівень веде себе як інструктор і є мозком програмно-конфігурованої мережі. Рівень керування приймає всі рішення про пересилання пакетів на основі

топології, а також керує різними контролерами.

- Рівень інфраструктури (Data Plane): цей рівень пересилає пакети на основі таблиць маршрутизації. Рівень пересилання даних — це фізичний пристрій, який отримує інструкції з рівня керування та виконує пересилку відповідно ним.

- API північного інтерфейсу: цей інтерфейс в основному використовується для зв'язку між верхнім і середнім рівнем, керуванням і контролем. Інструкції з прикладного рівня передаються контролерам через цей інтерфейс

- Південний інтерфейс: цей інтерфейс в основному використовується для зв'язку між рівнями керування та інфраструктури. Для роз'єднання цих двох рівнів використовується протокол OpenFlow.

Програмно-конфігуровані мережі представляють собою новий підхід до проектування та керування мереж, який відрізняється від традиційної мережевої парадигми. В основі концепції лежить поняття централізованого керування мережею, де ця логіка знаходиться на відокремленому контролері. Це дозволяє використовувати програмне забезпечення для адміністрування мережевих функцій і ресурсів, тим самим полегшуючи конфігурацію, моніторинг і автоматизацію мережевих операцій.

Основним елементом в програмно-конфігурованих мережах є контролер SDN. Цей програмний елемент відокремлений від фізичних мережевих пристроїв, таких як комутатори та маршрутизатори, і відповідає за прийняття рішення щодо конструювання трафіку та налаштування мережевих пристроїв на основі інформації про стан мережі.

Основною функцією контролера SDN є керування трафіком і мережевими ресурсами та забезпечення централізованого контролю над всією мережею. Контролер отримує від комутаторів дані про стан мережі, включаючи інформацію про трафік, стан комутаторів, MAC і IP-

адреси, топологію мережі та інші важливі деталі, а потім передає команди для керування цими пристроями.

Протокол OpenFlow слугує комунікаційним інтерфейсом між контролерами OpenFlow та площинами пересилання OpenFlow [16]. Це перший стандартний протокол зв'язку для середовищ SDN. Його основна перевага полягає в тому, що він дозволяє конфігурувати комутатори різних виробників за допомогою контролерів. Протокол OpenFlow керує двома компонентами комутатора: таблицею потоків і захищеним каналом зв'язку. Останній охоплює ряд функцій, включаючи зашифрований канал, моніторинг трафіку та адміністрування вхідних пакетів, згенерованих різними контролерами [17].

Функціональність і можливості контролера SDN можуть відрізнятися залежно від базової платформи та підходу до реалізації. Він здатний виконувати низку завдань, зокрема:

1. Розподіл трафіку: Контролер SDN визначає оптимальний маршрут для пересилання пакетів, використовуючи статистику мережі. Він аналізує низку показників, включаючи навантаження на мережу, час передачі, пропускну здатність та інші відповідні дані, щоб оптимально розподіляти трафік.

2. Керування політиками безпеки: Контролер встановлює політики шифрування для різних сегментів мережі. Він визначає, які протоколи шифрування (TLS, IPsec тощо) використовуються між певними вузлами. Також слід зазначити управління сертифікатами та ключами шифрування через взаємодію з сервісами керування ключами (Key Management Services, KMS) або сертифікаційними центрами (Certificate Authorities, CA) та інтеграцію пристроями шифрування чи шифрування на рівні додатків або потоків.

3. Динамічне керування мережевими ресурсами: Контролер дозволяє динамічно конфігурувати мережеві ресурси, включаючи пропускну здатність, VLAN (Virtual Local Area Network), QoS та інші

параметри. Це забезпечує ефективне використання мережевих ресурсів відповідно до висунутих вимог.

Контролери SDN здатні забезпечити повний огляд мереж, і для виконання QoS у мережах дуже важливо, щоб вони були розташовані в оптимальному місці. Фактори, які можуть впливати на якість обслуговування через контролери, можна віднести до ряду областей, таких як безпека мережі, керування мережею, ефективність використання ресурсів та керування мережевими послугами. Вплив вищезазначених проблем можна зменшити шляхом забезпечення надійності, масштабованості, узгодженості та балансування навантаження [18–21].

SDN є привабливою областю досліджень у сучасних комп'ютерних мережах. Однак в цій області є проблема забезпечення QoS у відношенні до керування контролерами SDN, яка стосується складності управління трафіком та ресурсами, які залежать від архітектури SDN. Розгортання контролерів доволі важка процедура у масштабних мережах. Надійність та масштабованість є ключовими характеристиками комп'ютерних мереж, які безпосередньо впливають на QoS. Спираючись на це, багато сучасних досліджень спрямовані на пошук відповіді на такі питання, як необхідна кількість контролерів для розгортання SDN інфраструктури, розміщення цих контролерів та забезпечення зв'язку між ними та підключеними пристроями [15].

Існує багато відкритих та комерційних SDN контролерів. Різні особливості платформ контролерів підходять для різних застосувань. В основному, контролери класифікуються на розподілені та централізовані. Централізований контролер реалізує логіку площини керування з одного місця та, як правило, має проблеми з масштабованістю через обмежену потужність контролера. Натомість розподілений контролер не має проблем з масштабованістю та забезпечує високу продуктивність, коли навантаження трафіку велике.

Короткі порівняння різних контролерів щодо різних функцій наведено в таблиці 1.1.

Таблиця 1.1. Порівняння різних типів контролерів

Назва	ONOS	OpenDaylight	FloodLight	RYU	HP VAN SDN Controller
Організація	ON.Lab	Linux Foundation	Big Switch Network	NTT	HP
Мова	Java	Java	Java	Python	Java
Відкритий	Так	Так	Так	Так	Ні
Потоки/с	1 М	106 К	-	-	-
Модульність	Висока	Висока	Середня	Середня	-
Продуктивність	Середня	Середня	Середня	Середня	-

Назва	ONOS	OpenDaylight	FloodLight	RYU	HP VAN SDN Controller
Відмовостійкіс	Так	Ні	Ні	Ні	Ні
Архітектура	Розподілена	Централізована	Централізована, багатопотокова	Централізована, багатопотокова	Розподілена

RYU — це контролер SDN, який визначає пріоритети керування програмованим мережевим API на рівні додатків. Програмне забезпечення було розроблено на Python компанією NTT. RYU розроблено для використання в хмарних середовищах і центрах обробки даних (DCC), пропонуючи гнучкі можливості керування мережею. RYU сумісний з декількома версіями OpenFlow, включаючи 1.0, 1.2 і 1.3, що сприяє ефективному управлінню трафіком. RYU є популярним вибором серед розробників завдяки своїй архітектурі Python, яка полегшує інтеграцію з іншими системами та забезпечує високу масштабованість у великих мережевих середовищах [20, 22].

Floodlight — це SDN-контролер на основі Java, розроблений Веасон, який може працювати як з фізичними, так і з віртуальними комутаторами. Система здатна обробляти до 1,39 мільйона запитів на секунду завдяки використанню багатопотоковості. Цей контролер був розроблений для використання у великих центрах обробки даних, де висока пропускна здатність є ключовою вимогою. Він підтримує власні північні API, які полегшують інтеграцію ряду мережевих рішень.

Водночас Floodlight має певні проблеми з безпекою, оскільки будь-яка зміна даних може потенційно поставити під загрозу його надійність [20, 23].

OpenDaylight – ще один контролер SDN на основі Java, відомий своєю модульною архітектурою та підтримкою відкритих стандартів. Використовуючи OSGi Framework, надає режими CLI та GUI для керування. OpenDaylight підтримує REST API для північних інтерфейсів і забезпечує взаємодію з такими південними API, такими як OpenFlow, OVSDB, PCEP, BGP та SNMP. Контролер використовує RESTCONF і NETCONF для конфігурації, що дозволяє легко масштабувати SDN мережі. Його архітектура базується на рівні адаптації сервісів (Service Adaptation Layer), який розділяє північні та південні API [1, 24].

Контролер SDN ONOS (Open Network Operating Systems) розроблений для підтримки великих мереж WAN і має розподілену архітектуру для надійності та масштабованості. Він здатний обробляти близько одного мільйона потоків за секунду та забезпечує централізоване керування продуктивністю та доступністю мережі. ONOS має можливість глобально контролювати мережу та забезпечує високий рівень продуктивності навіть для великих мереж із високими вимогами до пропускну здатності, що робить його одним із кращих рішень для середовищ SDN із високим навантаженням [20, 25].

Заснований на платформі OpenDaylight, контролер HP VAN SDN підтримує OpenFlow і забезпечує гнучку інтеграцію з іншими мережевими технологіями. Він підтримує такі протоколи, як BGP, SNMP, і включає інтеграцію з OpenStack для побудови ланцюгів мережеских функцій. HP VAN забезпечує високу доступність і багатозадачність завдяки кластеризації. Він також підтримує різні південні API, такі як L3 Agent і L2 Agent [20, 26].

Також слід розглянути такий елемент, як комутатор. Він є ключовим елементом архітектури SDN і виконує роль мережевого

пристрою, який забезпечує пересилання пакетів даних в мережі згідно з рішеннями, прийнятими контролером SDN. Комутатори відрізняються від традиційних мережевих пристроїв тим, що вони мають програмований інтерфейс, що дозволяє контролеру SDN взаємодіяти з ними і визначати правила пересилання трафіку. Як показано на рис. 1.1, кожен комутатор із підтримкою OpenFlow використовує процес прийняття рішень на основі потоку, визначену контролером SDN, який відповідає за підготовку таблиць пересилання комутатора [10]. Типовий комутатор має конвеєр таблиць потоків, які складаються із записів потоку та має три частини: (i) правила відповідності, які використовуються для відповідності вхідних пакетів; (ii) лічильники, які ведуть статистику узгоджених потоків; і (iii) дії або інструкції, які можуть бути налаштовані проактивно або реактивно для виконання після збігу [7, 27]. Елементи пересилання (тобто перемикачі з підтримкою OpenFlow) можуть бути реалізовані як програмним, так і апаратним забезпеченням. Деякі програмні комутатори, такі як Open vSwitch, мають потенціал для надання рішень для центрів обробки даних і віртуальних мереж [28]. З іншого боку, інші API [12, 29] використовуються для певних цілей (наприклад, програми голосового зв'язку через Інтернет-протокол (VOIP) та міждоменної маршрутизації). Крім того, існують численні мови програмування SDN, включаючи Procera [30], NetCore [31] та Frenetic [32], які пропонують API високого рівня, що полегшує розробку різноманітних програм SDN у більш гнучкий та функціональний спосіб.

Основна функція комутатора SDN — приймати пакети даних і пересилати їх відповідно до встановлених правил і інструкцій контролера SDN. Комутатор здатний виконувати наступні завдання:

1. Пересилання пакетів. Це процес, коли комутатор отримує пакети даних і приймає рішення про їх подальше направлення в мережі. Це рішення базується на інструкціях, які надійшли від контролера SDN.

Контролер визначає, куди потрібно направити пакет на основі його заголовка, адреси призначення, типу служби тощо.

2. Виконання політик керування – це процес, де комутатор виконує політики керування, які встановлені контролером SDN. Наприклад, він застосовує правила QoS, які визначають пріоритети трафіку в мережі. Крім того, комутатор застосовує правила маршрутизації, фільтрації або тунелювання, які визначають спосіб обробки пакетів у мережі.

3. Збір і передача інформації. Це процес, де інформація про стан мережі та пакети даних передається від комутаторів до контролера SDN. Це включає дані, що стосуються статистики трафіку, затримки, MAC- та IP-адрес, статусу порту та іншу відповідну інформацію. Контролер використовує цю інформацію для сприяння оптимальному керуванню мережею.

Функціональність і можливості комутаторів SDN відрізняються залежно від конкретної моделі та реалізації. Вони здатні підтримувати різноманітні протоколи, включаючи OpenFlow, що дозволяє їм взаємодіяти з контролером SDN. Крім того, комутатори можна віртуалізувати, що дозволяє створювати віртуальні сегменти мережі та керувати ними за допомогою програмного забезпечення.

В свою чергу, протокол OpenFlow використовується для взаємодії між контролером і комутаторами в SDN мережі. Він встановлює стандартні протоколи та команди, за допомогою яких контролер може здійснювати контроль над комутаторами та налаштовувати їхню поведінку. Комутатор OpenFlow містить одну або кілька таблиць потоків, групову таблицю та API до контролера (рис.1.2).

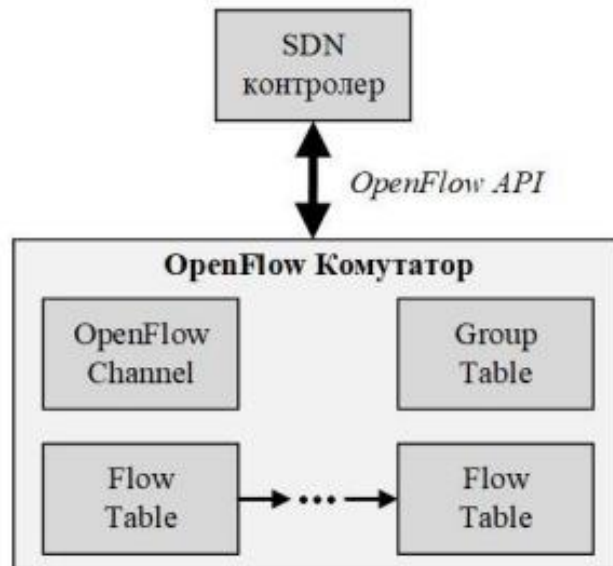


Рис.1.2. Комутатор OpenFlow [10]

OpenFlow вважається найпоширенішим інтерфейсом прикладного програмування у SDN, який постійно розвивається та стандартизується Open Networking Foundation. API OpenFlow забезпечує рівень абстракції, який надає безпечний зв'язок між контролером SDN і елементами пересилання з підтримкою OpenFlow [7]. Пристрої пересилання на основі OpenFlow розроблено таким чином, що вони можуть співіснувати зі звичайними пристроями Ethernet [28]. І навпаки, гібридні комутатори створюють нові можливості завдяки інтеграції портів OpenFlow і інших портів. Як було сказано раніше, контролер може передавати набір керуючих повідомлень, щоб підготувати та оновити таблиці потоків певного комутатора. Типовий комутатор із підтримкою OpenFlow обробляє щойно отримані пакети відповідно до інструкцій, викладених у його таблиці потоків. Рисунок 1.3 ілюструє структуру правила відповідності в рамках OpenFlow. Пропуск таблиці виникає, коли новий пакет не відповідає жодному із записів у таблиці потоку. У цьому випадку комутатор має можливість або відхилити пакет, або переслати його відповідному контролеру через протокол OpenFlow [7].

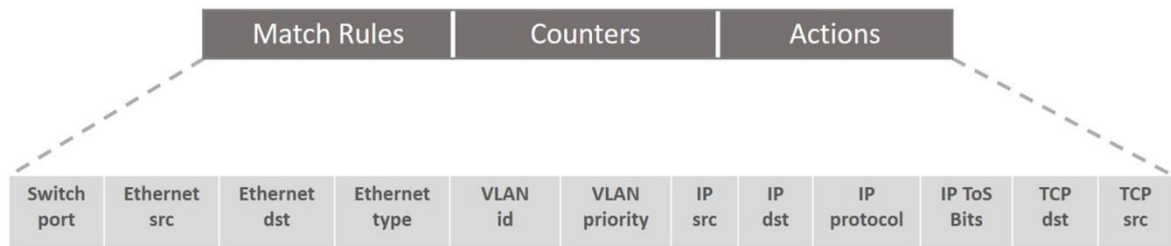


Рис. 1.3 Структура правил відповідності OpenFlow [7]

До основних аспектів протоколу OpenFlow належать:

1. Повідомлення (Messages): Протокол OpenFlow використовує різні типи повідомлень для передачі інформації між контролером і комутатором. Деякі з них включають запити стану, встановлення правил пересилання пакетів, повідомлення про зміни стану мережі, повідомлення про помилки тощо.

2. Потоки (Flows): Протокол OpenFlow базується на концепції потоків, які описують правила пересилання пакетів. Контролер встановлює правила потоків, які визначають, як комутатор повинен обробляти пакети з певними характеристиками, такими як вхідний порт, адреса призначення, тип служби тощо. Комутатор застосовує ці правила для прийняття рішень щодо подальшої обробки пакетів.

3. Таблиці потоків (Flow Tables): Комутатори SDN мають таблиці потоків, в яких зберігаються правила пересилання пакетів. Контролер встановлює правила потоків, вказуючи комутаторові, як обробляти пакети, які відповідають певним показникам. Комутатор зберігає ці правила у своїй таблиці потоків і використовує їх для пересилання пакетів.

4. Таблиці зі статусами (Status Tables): Комутатори також мають таблиці зі статусами, в яких зберігається інформація про стан комутатора, таку як статистика портів, статуси підключень тощо. Контролер запитує ці таблиці для отримання актуальної інформації про стан мережі.

У мережах SDN існує багато методів керування трафіком, які дозволяють контролеру SDN ефективно розподіляти, керувати та оптимізувати трафік. Давай розглянемо деякі з них.

Traffic Routing. Цей метод дозволяє контролеру SDN приймати рішення щодо оптимального маршруту для кожного пакета, заснованого на інформації про стан мережі. Це дозволяє розподіляти трафік по різних шляхах, уникаючи перевантаження та зменшуючи затримки [33].

Bandwidth Management. Цей метод дозволяє контролеру встановлювати політики керування пропускнуою здатністю для різних типів трафіку. Наприклад, він може призначати вищий пріоритет для важливого трафіку [34].

Quality of Service, QoS. Цей метод дозволяє контролеру налаштовувати політики QoS для забезпечення вимог до якості обслуговування різних додатків. Це може включати гарантування мінімальної пропускнуої здатності для певних типів трафіку [35].

Tunneling. Він дає змогу використовувати тунельні протоколи для створення віртуальних приватних мереж або для об'єднання різних мереж у єдину логічну структуру [36].

Load Balancing. Відповідає за балансування навантаження та дозволяє розподіляти трафік між різними маршрутами або серверами з метою рівномірного розподілу навантаження та підвищення продуктивності мережі [37].

Anomaly Detection. За допомогою цього методу можна аналізувати трафік для виявлення незвичних або підозрілих паттернів, що може вказувати на вторгнення або інші проблеми у мережі [38].

Dynamic Rule Adjustment. Метод дозволяє динамічно змінювати правила пересилання пакетів на основі зміни умов у мережі [37].

Access Control. Метод заборони або обмеження доступу дозволяє контролеру встановлювати правила для обмеження або блокування певних типів трафіку або користувачів [39].

Traffic Prioritization. За допомогою пріоритетів трафіку можна встановлювати пріоритети для різних типів трафіку, забезпечуючи більш ефективне використання ресурсів мережі [40].

Використання цих методів окремо або в поєднанні дозволяє налаштувати мережу в залежності від конкретних вимог та цілей керування трафіком.

В свою чергу, застосування штучного інтелекту (ШІ) у сфері інформаційних технологій є одним із найбільш актуальних напрямків розвитку. ШІ дає можливість розв'язувати складні задачі швидше та ефективніше, зменшувати кількість помилок та підвищувати якість роботи.

Для реалізації програмного забезпечення штучного інтелекту для програмно-конфігурованих мереж потрібно розглянути ряд ключових аспектів, які варто враховувати при розробці.

Першочерговим завданням є ретельний **аналіз потреб користувачів**. Це включає вивчення основних вимог та використовуваних сценаріїв, а також ідентифікацію функціональних потреб та можливих проблем, з якими зіштовхуються користувачі у своїй роботі з SDN мережами.

Також потрібно розглянути **функціональні вимоги** до програмного забезпечення штучного інтелекту. Вони включають набір функцій, які система повинна виконувати для задоволення потреб користувачів. Це налаштування політик безпеки, оптимізація мережі, керування трафіком, моніторинг та аналіз мережевих даних, автоматизацію задач керування мережею тощо.

Також програмне забезпечення штучного інтелекту повинно бути **надійним та стабільним** у роботі. Воно повинно забезпечувати безперебійну роботу мережі, виявляти проблеми у разі виникнення, а також мати механізми резервування та відновлення.

Важливо, щоб програмне забезпечення штучного інтелекту мало

високий рівень **безпеки**. Воно повинно забезпечувати захист мережевих ресурсів від несанкціонованого доступу, атак та вразливостей завдяки поєднанню аналізу великих обсягів даних, адаптивності та здатності до самонавчання. Вимоги до безпеки можуть охоплювати ряд заходів, включаючи автентифікацію, авторизацію, шифрування, контроль доступу та інші заходи безпеки даних та інформації. До основних способів реалізації захисту відносять виявлення та запобігання вторгненням (IDS/IPS), аналіз вразливостей та оцінка ризиків, моніторинг та прогнозування загроз, адаптивні механізми захисту за рахунок самонавчання, розподіл ресурсів для захисту від DDoS-атак та інші.

Враховуючи часто розподілену та масштабовану природу програмно-конфігурованих мереж, вкрай важливо, щоб програмне забезпечення ШІ мало здатність ефективно працювати в такому середовищі. Він має бути здатний керувати значною кількістю даних, мережевими пристроями та користувачами.

Крім того, вимоги ШІ включають відкритість і взаємодію. Він повинен використовувати стандартизовані протоколи, такі як OpenFlow, і бути сумісним з продуктами різних виробників комутаторів і контролерів SDN.

Крім того, штучний інтелект повинен бути оснащений механізмами виявлення проблем і логування. Така функція допоможе адміністраторам мережі у виявленні, діагностиці та вирішенні проблем.

Це лише декілька основних вимог до штучного інтелекту для програмно-конфігурованих мереж. Реальні вимоги можуть варіюватися в залежності від конкретних потреб і контексту використання мережі. Важливо провести ретельний аналіз потреб і залучити зацікавлених сторін для визначення конкретних вимог до ШІ.

1.2. Порівняльний аналіз методів конструювання трафіку в SDN мережах

У контексті SDN мереж керування трафіком можна визначити як процес прийняття рішень про маршрутизацію пакетів даних у мережі на основі аналізу різних факторів і мережових умов. У традиційних мережах керування трафіком здійснюється на самому мережевому обладнанні з використанням статичних або динамічних протоколів маршрутизації. На противагу цьому, архітектура SDN, за допомогою якого контролер, використовує централізований процес керування трафіком [33].

Ідеєю керування трафіком є визначення пріоритетів і обмеження пропускної здатності для різних категорій трафіку залежно від їхньої відносної важливості та вимог [34].

До основних аспектів керування трафіку можна віднести наступні пункти:

1. Аналіз стану мережі – це процес, за допомогою якого контролер аналізує робочий стан мережі. Це включе оцінку статистики трафіку, навантаження на різні лінії зв'язку, огляд стану комутаторів, затримки тощо.

2. Прийняття рішень про маршрут – це процес визначення оптимального маршруту для надсилання відповідного пакета на основі результатів аналізу стану мережі. Такі рішення можуть ґрунтуватися на низці різних показників, включаючи мінімальну затримку, найкоротший шлях, резервування смуги пропускання тощо.

3. Класифікація трафіку – це процес розподілу трафіку на окремі класи або категорії, які відповідають специфічним характеристикам програми, вимогам до QoS і рівню пріоритетності трафіку.

4. Встановлення правил пропускної здатності – це процес

розподілу пропускної здатності для кожної категорії мережевого трафіку, що виділяється на рівні контролера.

5. Керування пропускною здатністю передбачає диференціацію класів трафіку відповідно до їхніх рівнів пріоритету. Це означає, що більш важливі дані будуть оброблятися і передаватися в пріоритетному порядку.

6. Оптимізація ресурсів – це процес розподілу пропускної здатності відповідно до вимог різних класів трафіку. Наприклад, у разі значного обсягу критичного трафіку в мережі в певний момент часу, контролер може призначити більшу частку ресурсів для цієї категорії, гарантуючи таким чином безперебійну передачу даних.

7. Динамічна адаптація правил та розподіл пропускної здатності – це процес, який ініціалізує контролер у відповідь на зміни умов і вимог мережі протягом часу.

8. Впровадження керування смугою пропускання сприяє справедливому розподілу ресурсів для різних класів трафіку. Це запобігає монополізації наявних ресурсів однією категорією трафіку, тим самим гарантуючи, що інші категорії не опиняться в несприятливому становищі.

Пріоритезація трафіку являє собою методологію керування мережевим трафіком, за допомогою якої різним типам і класам трафіку присвоюються різні рівні пріоритету. Ця концепція дозволяє забезпечити вищий рівень обслуговування та доступу до мережевих ресурсів для обраного трафіку у порівнянні з менш важливим [41].

До основних аспектів пріоритезації трафіку можна віднести наступні:

1. Визначення пріоритетів трафіку – це процес, який передбачає категоризацію різних типів трафіку на основі їх відносної важливості.

2. Пріоритезація – це процес призначення відповідного рівня пріоритету для кожного класу трафіку. Це може бути досягнуто на рівні

мережевих пристроїв, таких як комутатори або маршрутизатори, де пакети можуть бути позначені певними значеннями пріоритету.

3. Пріоритезація трафіку впливає на керування чергами пакетів у мережевих пристроях, таких як комутатори або маршрутизатори. Пакети з вищим пріоритетом обробляються і передаються швидше, тим самим зменшуючи затримку для критично важливого трафіку.

4. Визначення пріоритетів трафіку за допомогою керування ресурсами дозволяє розподіляти доступні мережеві ресурси відповідно до вимог різних класів трафіку. Основна перевага полягає в тому, що можна виділити більшу пропускну здатність або ресурси для більш важливого трафіку.

5. Система пріоритетів призначена для виявлення випадків, коли мережа зазнає перевантаження через наявність великої кількості низькопріоритетного трафіку. У таких випадках реалізація спеціальних механізмів служить для полегшення навантаження на менш критичні категорії трафіку. Цей механізм називається виявленням перевантажень.

6. Пріоритезація трафіку також є важливим елементом забезпечення QoS, оскільки надання більш високого рівня пріоритету для важливого трафіку дозволяє гарантувати якісну передачу даних.

Балансування навантаження – це метод рівномірного розподілу мережевого трафіку між кількома ресурсами. Ці ресурси можуть включати сервери, комутатори, лінії зв'язку та інші компоненти. Мета балансування навантаження – гарантувати оптимальне використання ресурсів, підвищити продуктивність, гарантувати високу доступність і запобігти перевантаженню окремих компонентів мережі [37].

До основних аспектів балансування навантаження можна віднести наступні:

1. Балансування навантаження – це процес розподілу вхідного трафіку між декількома ресурсами в залежності від їх доступності та поточного навантаження.

2. Системи балансування навантаження призначені для моніторингу та аналізу навантаження на різні ресурси. Це дозволяє виявити перевантажені ресурси або ресурси, які мають вільні потужності.

3. У мережах з декількома шляхами балансування навантаження може передбачати вибір оптимального шляху для кожного пакета з метою забезпечення рівномірного розподілу навантаження.

4. Адаптивність до змін — це механізм балансування навантаження, що дозволяє ефективно реагувати на зміни у навантаженні та доступності ресурсів. У разі недоступності або перевантаження певного ресурсу, трафік повинен бути перенаправлений на альтернативний, доступний ресурс.

5. Балансування навантаження використовує надлишкові ресурси з метою забезпечення високої доступності сервісу навіть у випадку відмови одного з ресурсів.

Балансування навантаження — це механізм, який використовується для оптимізації використання мережевих ресурсів, гарантії стабільної роботи, підвищення продуктивності та забезпечення QoS.

1.3. Огляд існуючих систем штучного інтелекту у SDN мережах

Галузь штучного інтелекту (ШІ) швидко розвивається, охоплюючи різноманітні підгалузі. До них належать, але не обмежуються, представлення знань, міркування, планування, прийняття рішень, оптимізація, машинне навчання (ML) і метаевристичні алгоритми. Тест Тюрінга [42] дає повне визначення інтелекту, де комп'ютер повинен відповісти на деякі запитання, написані людиною-допитувачем.

Відповідно, комп'ютер проходить перевірку, якщо запитувач не може визначити, чи відповідь була написана людиною чи машиною. Щоб пройти тест Тьюринга, комп'ютер повинен мати розширені можливості, такі як обробка природної мови, представлення знань, автоматизоване міркування, ML і комп'ютерне бачення [42]. Дослідження штучного інтелекту почалися в середині 1950-х років, коли літній семінар, організований Мартіном Мінскі та Клодом Шенноном у Дартмутському коледжі, призвів до народження галузі ШІ [43]. Однак перший внесок був зроблений у 1943 році МакКалохом і Піттсом, коли вони запропонували першу модель для штучних нейронних мереж, у якій кожен нейрон має двійковий вихід (-1, +1) із функцією активації знака [43]. Підходи штучного інтелекту зросли, завдяки ключовим внескам, які призвели до появи нових підгалузей, таких як експертні системи, нечітка логіка (FL) та еволюційні обчислення (EC). Подальші зусилля сприяли дослідженням у сфері штучного інтелекту шляхом удосконалення існуючих методів і пропозиції нових гібридних інтелектуальних підходів. ML, метаевристичні алгоритми та системи нечіткої логіки широко використовуються в парадигмі SDN [44, 45].

Оскільки у даній роботі використовується машинне навчання, то розглянемо його більш детально. Методи машинного навчання діляться на чотири групи.

Кероване навчання. Керовані методи навчання забезпечуються попередньо визначеним набором знань. Для ілюстрації, навчальний набір даних, що містить пари введення-виведення, дає змогу системі вивчати функцію, яка відображає заданий вхід на відповідний вихід [42]. Цей підхід вимагає наявності набору даних, який точно представляє систему, що розглядається, який потім можна використовувати для оцінки ефективності вибраного методу [46]. До цієї групи можна віднести наступні методи: Штучні нейронні мережі (ANN) [43, 47], машини підтримки векторів (SVM) [20], дерева рішень (DT) [42, 43],

методи ансамблю [42, 48], кероване глибоке навчання [49-51], рекурентні нейронні мережі (RNN) [51, 52] та згорткові нейронні мережі (CNN) [49, 53].

Некероване навчання. Методи некерованого навчання надаються без попередньо визначених знань (тобто з немаркованими даними) [20]. Тому система в основному зосереджується на пошуку конкретних шаблонів у вхідних даних. Прикладом підходу до некерованого навчання є кластеризація, яка використовується для виявлення корисних кластерів у вхідних даних на основі подібних властивостей, визначених відповідною метрикою відстані, такою як метрика відстані Евкліда, Жаккара та косинус [42, 46]. До цієї групи машинного навчання можна віднести наступні методи: k-means clustering [54-57], самоорганізуючі карти (SOM) [58-60], прихована модель Маркова (HMM) [61], обмежена машина Больцмана (RBM) [51], стековий автоматичний кодувальник (AE) [51, 53], та мережа глибокого переконання (DBN) [51, 53].

Навчання з підкріпленням (RL). RL система навчається на основі набору підкріплень із свого середовища. Наприклад, винагорода чи покарання визначають, чи добре спрацювала система [42]. Кожна взаємодія з навколишнім середовищем повертає інформацію, яку система найкращим чином використовує для вивчення та оновлення своїх знань [62]. Ключовим поняттям RL є властивість Маркова (тобто лише поточний стан впливає на наступний стан) [62]. До цієї групи машинного навчання можна віднести наступні методи: Q-навчання [63] та глибоке RL [62].

Напівкероване навчання. У напівкерованому навчанні система навчається як з маркованих, так і з немаркованих даних, де відсутні мітки, а також маркована частина може містити випадковий шум, утворюючи ситуацію між керованим і некерованим навчанням [42]. Для багатьох реальних програм більш реалістично покладатися на немарковані дані, які не вимагають жодних додаткових витрат або

подальшого експертного процесу маркування [64]. Оскільки він містить деякі невеликі марковані дані, результативність підходів напівкерованого навчання перевершує некероване [64]. У таблиці 1.3 показано порівняння основних типів підходів ML.

Таблиця 1.2. Порівняння підходів ML

ML-підхід	Переваги	Недоліки
Кероване навчання	Використовує марковані дані. Робить точні узагальнення на основі достатнього набору даних.	Потрібен набір даних, який представляє систему. Дані вручну маркуються, що не підходить для багатьох реальних програм.
Некероване навчання	Знаходить приховані шаблони, не покладаючись на марковані дані	Може не дати корисного розуміння прихованих закономірностей і того, що насправді вони означають
Напівкероване навчання	Вчиться як з маркованих, так і з немаркованих даних. Повинні виконуватися певні припущення щодо базового розподілу даних.	Може призвести до погіршення продуктивності, якщо обираю неправильні припущення

ML-підхід	Переваги	Недоліки
Навчання з підкріпленням	Динамічна адаптація та поступове вдосконалення. Агент взаємодіє з невизначеним середовищем, з метою максимізації винагороди агента. Можна використовувати для складних задач, які не мають аналітичного формулювання	Потрібно вказати функцію винагороди, параметризовану політику, стратегію та початкову політику

Підходи штучного інтелекту (AI) та машинного навчання (ML) в даний час відіграють ключову роль в розв'язанні широкого спектру складних проблем, що виникають у сучасних мережах. Серед цих завдань можна виокремити маршрутизацію, класифікацію трафіку, кластеризацію потоків, виявлення вторгнень, балансування навантаження, виявлення збоїв, оптимізацію якості обслуговування (QoS), поліпшення якості користування (QoE), контроль доступу та розподіл ресурсів.

У контексті програмно-конфігурованої мережі (SDN) взаємодія між ШІ та ML стає особливо важливою. Недавні дослідження [35-39, 41] вказують на те, що зростання ролі ШІ в SDN є значущим та відображає напрямок розвитку промисловості та наукового співтовариства. Цей тенденційний рух відзначається впровадженням різноманітних підходів ШІ в архітектуру SDN.

Серед найбільш поширених методів вирішення проблем в мережах можна визначити машинне навчання (ML), мета-евристику та

вибірковий пошук (FS). Ці підходи відзначаються високою ефективністю та гнучкістю у вирішенні різноманітних завдань, таких як адаптація до змін у трафіку, оптимізація роботи маршрутизації, та підвищення загальної продуктивності та надійності мережевого середовища. Такий симбіоз ІІІ та SDN відкриває нові можливості для ефективного керування та оптимізації мережевих систем у сучасних інформаційних технологіях.

Сучасні дослідження багатошляхових мереж активно розглядають питання балансування навантаження. Дві відомі стратегії розподілу навантаження, які широко використовуються в традиційних багатошляхових мережах, - це Equal-Cost MultiPath (ECMP) [4, 65] та Valiant Load Balance (VLB) [5]. Концепція ECMP полягає в рівномірному розподілі потоків даних між комутаторами наступного переходу. У свою чергу, VLB розподіляє трафік між доступними шляхами та вибирає комутатор наступного переходу випадковим чином. Ці стратегії використовують статичні методи і не можуть адаптивно реагувати на зміни в навантаженні на шляху в режимі реального часу.

Однак, незважаючи на їх ефективність у певних умовах, ECMP і VLB сильно обмежені фіксованістю використовуваних методів. Цим стратегіям не вистачає необхідної гнучкості та адаптивності для ефективного врахування та реагування на динамічні зміни в стані мережі. У світлі вищезазначених обмежень існує очевидна потреба в розробці більш гнучких і адаптивних методів балансування навантаження.

Особливо важливим в цьому контексті є розробка нових, більш ефективних методів балансування навантаження, що можуть адаптуватися до змін у топології мережі, забезпечуючи оптимізацію використання ресурсів та підвищення продуктивності багатошляхових мереж. Метою сучасних досліджень має стати створення інтелектуальних методів балансування навантаження, які враховують

динамічний характер мережі та забезпечують оптимальну реакцію на зміни трафіку та стану мережі. Це може сприяти не лише покращенню продуктивності мережі, але й підвищенню стійкості та ефективності в контексті сучасного мережевого середовища.

У ряді наукових досліджень, пов'язаних з оптимізацією балансування мережевого навантаження в архітектурі SDN, як потенційні рішення були запропоновані різноманітні системи балансування [6-7, 10]. У таких системах контролер використовується для аналізу інформації, отриманої від комутаторів OpenFlow, та вносить зміни в таблиці потоків відповідно до стратегії балансування навантаження. Такий підхід дозволяє ефективно планувати шляхи передачі даних у мережі SDN, сприяючи адаптації до змін топології мережі або обсягу трафіку.

Однак вищезазначені стратегії класифікуються як статичні методи балансування навантаження, що обмежує їх здатність адаптивно реагувати на динамічні зміни стану навантаження мережі в режимі реального часу. Важливо відзначити, що ці методи не використовують весь потенціал SDN, оскільки вони не враховують глобальний стан мережі та не взаємодіють з усіма її компонентами.

Одним із важливих шляхів подальших досліджень у цій галузі є розробка більш адаптивних і гнучких стратегій балансування навантаження в архітектурі SDN. Ці стратегії повинні використовувати весь потенціал цієї технології та бути здатними ефективно враховувати та реагувати на динамічні зміни в мережевому середовищі.

Алгоритм динамічного балансування навантаження (DLB), який був представлений у дослідженні [11], використовує стратегію жадібного вибору для визначення наступного каналу з найменшим навантаженням. Хоча цей підхід пропонує балансування навантаження в багатошляховій SDN, він обмежений розглядом лише навантаження кожного наступного стрибка, пропускаючи повний стан мережі в рамках

SDN. Цей підхід може виявитися складним для визначення оптимального шляху передачі в глобальному контексті, тим самим зменшуючи ймовірність досягнення оптимального балансування навантаження.

Стратегія розподілу навантаження, представлена у дослідженні [12], використовує набір змінних, включаючи інформацію про стрибки для кожного шляху, кількість пакетів передачі, номер ключового комутатора та швидкість передачі на кожному порту комутатора. Ці чотири функції інтегровані для обчислення стану навантаження кожного шляху через нечітку комплексну оцінку. Вибір шляху передачі базується на стані навантаження кожного шляху. Тим не менш, використання нечіткої комплексної оцінки вимагає штучного суб'єктивного процесу оцінювання, що ускладнює об'єктивне відображення точного стану навантаження кожного шляху в реальному часі.

У дослідженні [13] було реалізовано евристичний метод балансування навантаження, який використовує оптимізацію мурашиної колонії для визначення шляху передачі. Однак цей підхід використовує простий метод збору інформації про мережевий шлях, оцінюючи його за одним показником. Він є обмеженим та може обтяжувати врахування багатофакторного характеру мережевого середовища та викликати недооцінку реального стану мережі.

Розглядаючи вищезазначені проблеми, можна стверджувати, що більшість існуючих стратегій балансування навантаження ґрунтуються на простих методах з обмеженою ефективністю або функціоналом. Збір інформації про шлях обмежений у своїх можливостях, оскільки не враховує складні фактори та не забезпечує досягнення оптимальних результатів. У цьому контексті запропонований підхід використовує переваги глобального підходу в архітектурі SDN для систематичного збору чотирьох показників: коефіцієнт використання пропускну здатності, коефіцієнт втрат пакетів, часу передачі та кількість хопів на

кожному шляху. Це є основою для використання штучного інтелекту для розрахунку інтегрального стану навантаження кожного шляху.

Запропонований підхід, який використовує штучний інтелект для прогнозування інтегрального стану навантаження, є ефективнішим у порівнянні з традиційними методами. Він використовує глибоке навчання для врахування безлічі аспектів, що стосуються мережевого трафіку. Це забезпечує адаптивне та гнучке балансування навантаження з урахуванням повного діапазону характеристик кожного шляху та динамічних змін у мережевому середовищі. Такий підхід сприяє ефективному використанню ресурсів і підвищує продуктивність передачі даних у SDN мережах.

Висновки до розділу 1

В даному розділі представлено аналіз сучасних методів конструювання трафіку. Огляд існуючої рішень демонструє, що більшість стратегій, які використовуються для балансування навантаження, базуються на спрощених підходах, які за своєю суттю мають певні недоліки. Збір інформації про шлях обмежений у своїй здатності враховувати складні фактори та демонструвати очікувані результати.

Подальше підвищення продуктивності досягається за рахунок інтеграції технології штучного інтелекту з програмно-конфігурованими мережами.

З огляду на вищезазначені результати аналізу, розгортання штучного інтелекту в програмно-конфігурованих мережах вимагає:

- Дослідити методи використання та можливості інтеграції штучного інтелекту у програмно-конфігурованій мережі;
- Провести збір даних про стан мережі, такі як коефіцієнт пропускної здатності, час передачі пакетів, коефіцієнт втрати пакетів та хопи. Використати ці дані для аналізу трафіку та визначення оптимальних маршрутів.
- Розробити модель нейронної мережі для прогнозування стану мережі на основі зібраних даних;
- Виконати подальший аналіз отриманих, використовуючи тестові середовища для перевірки та оптимізації роботи штучного інтелекту в реальних умовах.

РОЗДІЛ 2

МЕТОД КОНСТРУЮВАННЯ ТРАФІКУ У ПРОГРАМНО- КОНФІГУРОВАНИХ МЕРЕЖАХ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ

2.1. Архітектура системи конструювання трафіку

У дисертаційній роботі запропоновано реалізацію системи, призначеної для підвищення ефективності використання ресурсів SDN мережі. У цій системі використовується модуль балансування навантаження на основі штучного інтелекту, який розраховує оптимальні маршрути передачі пакетів у мережі на основі характеристики мережі.

Впровадження подібних рішень може суттєво вплинути на вирішення ключових архітектурних проблем, які можуть виникнути в середовищі SDN. Комутатори в SDN відокремлені від контролерів і комутаторів дампу (dump switches). У великих мережах усі контролери та комутатори активні і це вимагає ретельного підходу до їх розробки та розміщенню. Крім того, проблеми продуктивності у мережі також пов'язані з такими аспектами, як гнучкість, масштабованість, безпека та узгодженість [16, 21, 66].

Іншою важливою проблемою є централізоване керування. Для досягнення збалансованого навантаження трафіку у SDN мережах вводиться декілька контролерів. Зі збільшенням їхньої кількості у мережі концепція централізованого керування має бути переоцінена. Щоб досягти високої якості обслуговування (QoS) в мережі, необхідно розгорнути кілька контролерів, оскільки їх сукупні можливості перевищують можливості одного контролера [66]. Однак централізоване керування таким різноманітним набором функцій контролера може стати складним завданням [67].

Хоча впровадження кількох контролерів полегшує **масштабування SDN мереж**, це може створювати проблеми для QoS через необхідність балансування навантаження між ними [66, 67].

Ще однією складністю є забезпечення **узгодженості між контролерами**. У мережах з кількома контролерами важливо синхронізувати інформацію про стан мережі, щоб уникнути проблеми консенсусу [66]. Через складну реалізацію та затримки, які виникають під час цього процесу, консенсусні підходи не завжди підходять.

Розміщення контролерів — ще одна ключова архітектурна проблема. Використання лише одного контролера в SDN мережах має багато переваг, таких як керування, контроль і моніторинг усієї мережі централізовано з одного вузла. Водночас це може призвести до проблем з надійністю та масштабованістю [66]. У міру зростання мережі ці проблеми погіршують продуктивність. Проблема розміщення контролера відома з 2012 року. Щоб зменшити затримки та підвищити загальну продуктивність мережі, необхідно оптимально розмістити необхідну кількість контролерів на відповідних відстанях у мережі [15]. У великих мережах розгортання контролерів повинно враховувати два основні питання: (1) скільки контролерів потрібно в мережі; (2) де їх розміщувати? Це NP-складні задачі, але вони важливі для розгортання кількох контролерів [15, 66].

Також важливо враховувати протоколи зв'язку між контролерами. У розподіленому середовищі кількість контролерів має прямий вплив на якість обслуговування SDN мережі. Для вирішення цієї проблеми важливо впровадити ефективну систему зв'язку між контролерами. Ефективний зв'язок між контролерами вимагає впровадження стандартного протоколу east-west, який в даний час підтримується протоколом BGP [21, 66].

Хоча використання кількох контролерів дозволяє покращити QoS, виникає проблема **планування роботи кількох контролерів**, щоб

уникнути перевантаження одного з них [66]. Основним викликом залишається швидко та ефективно балансування навантаження.

Це особливо доречно в контексті програмно-конфігурованих мереж, які відрізняються рядом характеристик, включаючи можливість створювати віртуальні мережі та підтримувати декілька таких мереж в одній фізичній інфраструктурі. Крім того, програмно-конфігуровані мережі часто мають самоподібний трафік, а також здатність до гнучкості та масштабованості, забезпечуючи швидку адаптацію до змін у бізнес-процесах. Це забезпечує зростання мережі та інтеграції з хмарними обчисленнями. Також це дає змогу гнучко керувати трафіком між хмарою та локальними ресурсами. Слід вказати, що ще одним важливим пунктом є гарантія якості обслуговування, для підтримки стабільної роботи критично важливих додатків, таких як відеоконференції або системи управління виробничими процесами.

Розглянемо більш детально що являє собою віртуальні мережі та самоподібний трафік.

Віртуальні мережі представляють собою технологічне рішення, яке дозволяє створювати віртуальні ізольовані мережі в одному фізичному мережевому середовищі. Ця концепція часто використовується в комп'ютерних мережах для поділу одного фізичного мережевого ресурсу на кілька логічно відмінних мереж, кожна з яких працює незалежно від інших. Якщо в одному фізичному мережевому середовищі присутні кілька віртуальних мереж, можна виконати динамічне балансування навантаження шляхом розподілу ресурсів між різними віртуальними мережами відповідно до переважаючих вимог і навантаження кожної мережі. У такому випадку, якщо одна віртуальна мережа вимагає більшого розподілу ресурсів або генерує більший обсяг трафіку, система виконає динамічне балансування навантаження та перерозподіл ресурсів, щоб гарантувати оптимальне функціонування всіх віртуальних мереж.

У свою чергу, само подібний трафік (Self-Similar Traffic) – це вид трафіку, який властивий багатьом сучасним SDN мережам, особливо якщо ці мережі передають великий обсяг даних та мають інтенсивний обмін інформацією. Цей вид трафіку характеризується тим, що його статистичні властивості залишаються схожими на різних масштабах часу, тобто трафік виявляє самоподібну або фрактальну структуру. Основні причини самоподібного трафіку в мережах SDN включають активність користувачів, що призводить до нерегулярних періодів активності та бездіяльності; обробка даних, наприклад копіювання файлів; характеристики офісних програм, включаючи електронну пошту, інформаційні системи, чат і спільну роботу над документами; інтенсивність використання ресурсів, таких як доступ до серверів, хмарних служб, відеоконференцій тощо. Ці фактори можуть сприяти коливанням мережевого трафіку. Ефективне управління та аналіз самоподібного трафіку має вирішальне значення для оптимального функціонування SDN мереж. Це включає моніторинг мережевої активності, оптимізацію ресурсів, планування мережевої інфраструктури та застосування таких технологій, як SDN та QoS для підтримки бажаного рівня обслуговування.

У даній роботі показники оцінки стану мережі, включаючи використання пропускну здатності, втрату пакетів, часу передачі та кількість переходів для передачі пакетів, використовується для точного відображення стану завантаження кожного маршруту. Ці параметри є основними детермінантами ефективності маршруту та навантаження, що дозволяє системі адаптивно реагувати на зміни умов мережі та вибирати оптимальні шляхи для передачі даних. Використання цих показників оцінки сприяє покращенню стратегії балансування навантаження, забезпечуючи тим самим надання оптимальних рішень у режимі реального часу.

Отримана стратегія була реалізована за допомогою адаптивного

керування потоком даних, за допомогою якого мережа, використовуючи навчену модель, визначає оптимальний шлях передачі для кожного потоку з метою мінімізації навантаження. Цей підхід гарантує постійну адаптацію стратегії до коливань трафіку, завдяки чому система стає високоефективною у вирішенні потенційних проблем, які можуть виникнути в мережах SDN.

Хоча традиційні маршрутизатори працюють із таблицями маршрутизації, їхні можливості обмежені через недолік інформації. Крім того, обчислення виконуються виключно в контексті наступного переходу без урахування стану та характеристик мережі. На відміну від цього підходу, контролер SDN дає змогу представити комплексне уявлення про мережу з самого початку її роботи.

Активно оновлюючи інформацію про топологію мережі, контролер SDN визначає всі потенційні маршрути між вузлами. Це дає змогу аналізувати та класифікувати стан завантаження кожного глобального шляху, що таким чином сприяє більш глибокому розумінню мережевого середовища. Також це дозволяє об'єктивно оцінювати та керувати навантаженням на рівні кожного окремого шляху, тим самим підвищуючи ефективність і стабільність мережі в цілому.

Архітектура SDN мережі для запропонованої системи балансування навантаження зображена на рисунку 2.1.

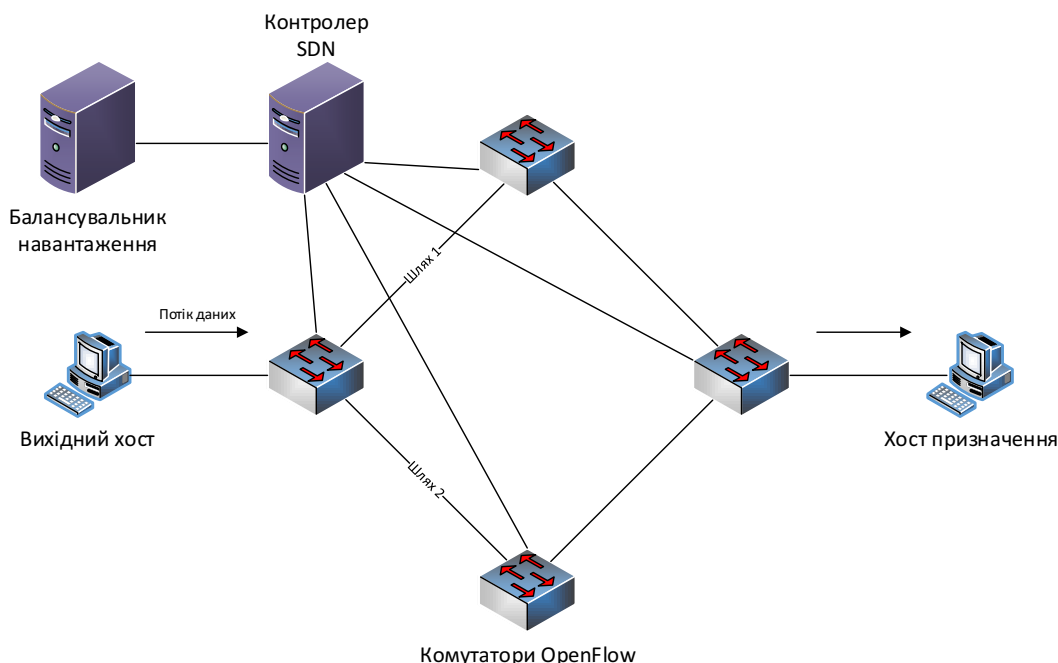


Рис. 2.1. Архітектура SDN мережі

З метою ефективного зменшення навантаження на контролер SDN та оптимізації керування ресурсами мережі, запропонована архітектура впроваджує спеціальний сервер у вигляді балансувальника навантаження (Load Balancer). Він відповідає за балансування навантаження на кожному шляху мережі для ефективного розподілу трафіку та покращення продуктивності.

Щоб визначити найменш перевантажений шлях у реальному часі, Load Balancer використовує стан навантаження, який обчислюється на основі інформації, для різних шляхів, отриманих від контролера SDN. У даному випадку контролер через регулярні проміжки часу передає дані про навантаження на кожному шляху до балансувальнику навантаження. Це дозволяє оперативно реагувати на зміни в стані мережі.

У більшості існуючих систем контролер отримує дані про стан мережі від комутаторів за допомогою протоколів, таких як OpenFlow або інших протоколів керування SDN. На основі отриманих даних контролер приймає рішення щодо керування мережевим трафіком та ресурсами. Контролер генерує поточкові правила (flow rules), які

визначають, яким чином комутатори повинні обробляти пакети у мережі [8, 9]. Стандартна процедура щодо керування мережевим трафіком та ресурсами зображена на рисунку 2.2. У даній роботі пропонується винести логіку генерації правил потоків у балансувальник навантаження, розроблений на основі штучного інтелекту. Це дає можливість робити висновки про стан мережі не тільки щодо окремого сегмента, але й у загальному плані, виходячи з особливостей мереж SDN, що, у свою чергу, сприяє більш ефективному керуванню трафіком.

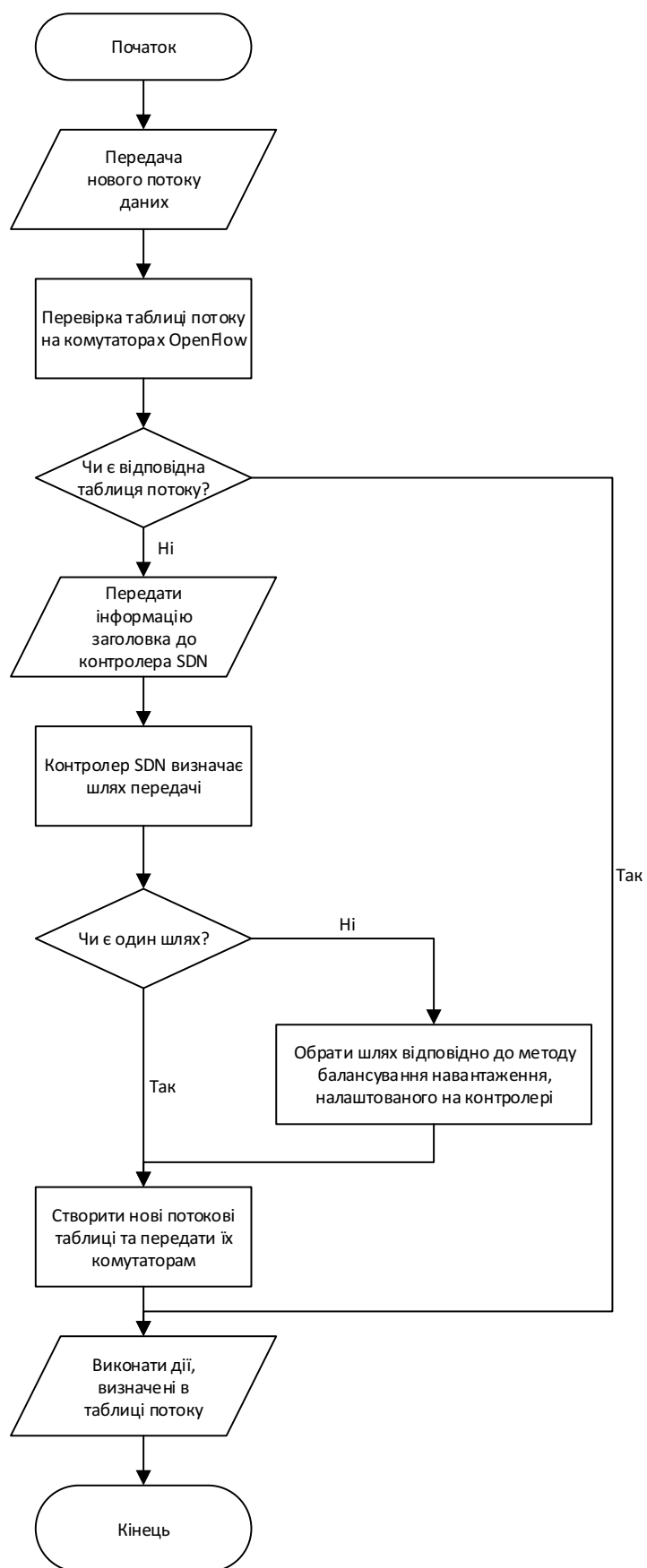


Рис. 2.2 Блок-схема стандартного алгоритму керування мережевим трафіком

При необхідності виконати балансування навантаження, нейронна мережа повертає контролеру інформацію про найменш завантажений шлях, враховуючи стан навантаження кожного шляху. Отримавши цю інформацію, контролер SDN приймає обґрунтовані рішення щодо оптимального шляху передачі даних.

Визначивши оптимальний шлях, контролер створює потокові таблиці для комутаторів OpenFlow, для реалізації плану передачі потоку даних. Цей підхід дозволяє реалізувати інтелектуальне та ефективне керування трафіком у SDN мережі, тим самим сприяючи оптимізації пропускної здатності та забезпеченню надійності передачі даних.

Після аналізу існуючих рішень була розроблена архітектура системи балансування навантаження на основі нейронної мережі Load Balancing Based on Neural Network (LBBNN). Система включає комутатори OpenFlow, SDN контролер і балансер навантаження. Останній відповідає за обчислення навантаження для кожного мережевого шляху та вибір найменш завантаженого для передачі потоку даних.

Розроблено наступну процедуру роботи системи з урахуванням особливостей архітектури програмно-конфігурованих мереж.

Після передачі нового потоку даних до домену SDN, комутатори OpenFlow виконують процес порівняння інформації заголовка пакета з даними, що зберігаються в таблицях потоків. У разі збігу між інформацією заголовка пакета та даними в таблиці потоків, потік даних передається через поле Action у вищезгаданій таблиці. У випадку, якщо відповідна таблиця потоку не ідентифікована, OpenFlow передає інформацію заголовка пакета до SDN контролера для визначення оптимального шляху передачі.

Після визначення шляху передачі даних, контролер SDN генерує нові потокові таблиці і передає їх комутаторам OpenFlow, таким чином

активуючи передачу даних.

У разі наявності декількох шляхів передачі даних, контролер SDN передає інформацію про ці шляхів до балансувальника навантаження (Load Balancer). Load Balancer обчислює навантаження для кожного шляху та передає обраний оптимальний шлях до контролера SDN для подальшої передачі даних.

Контролер SDN отримує обраний шлях від балансувальника та створює потокові таблиці для передачі комутаторам OpenFlow.

На рис 2.3 зображена розроблена процедура керування мережевим трафіком.

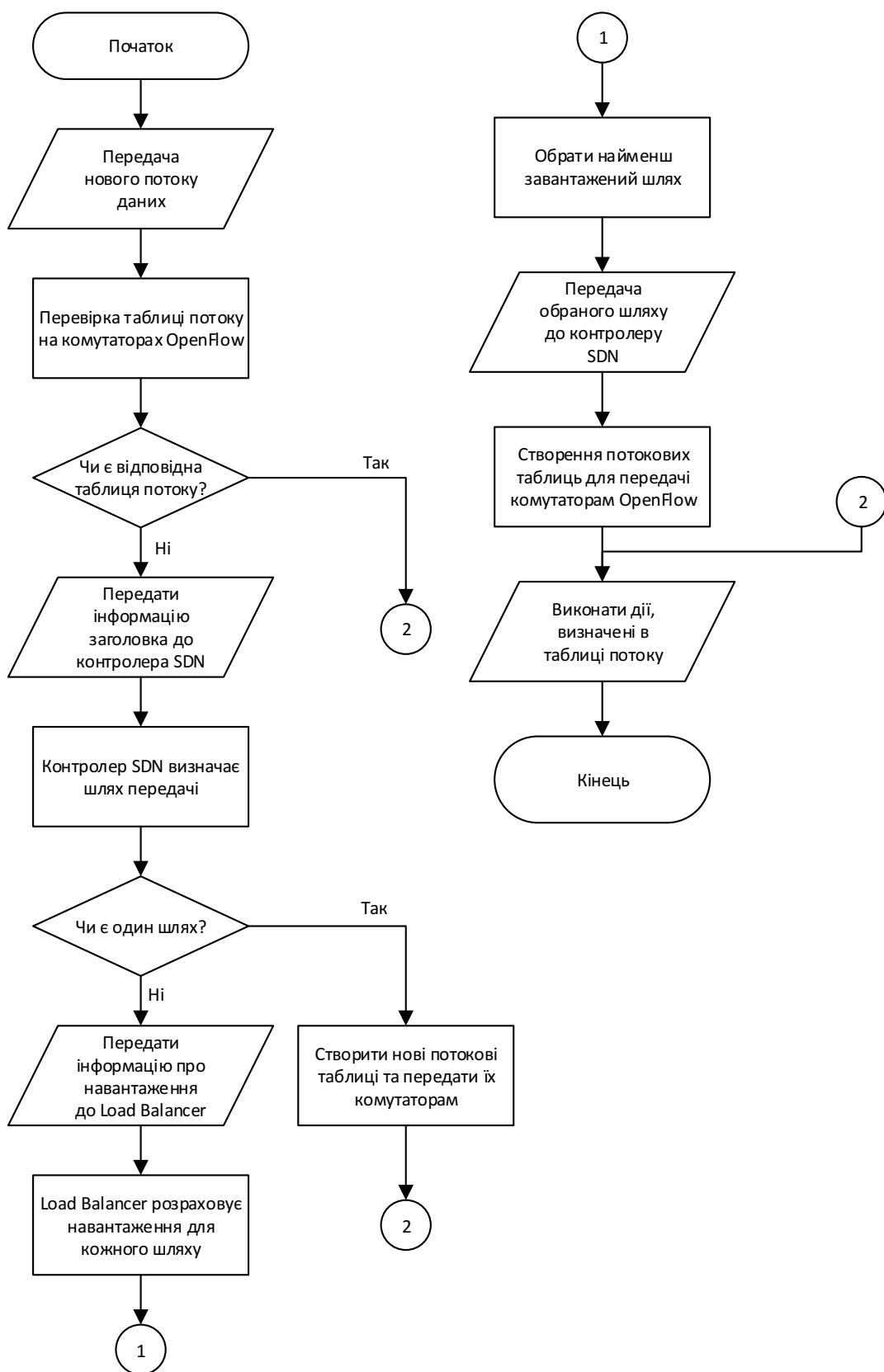


Рис. 2.3 Блок-схема розробленого алгоритму керування мережевим трафіком

Розглянемо функціональність кожного нового блоку більш детально. На відміну від звичайного алгоритму керування мережевим трафіком, у випадках, коли для передачі потоку даних доступно кілька маршрутів, контролер передає нейронній мережі інформацію про всі потенційні маршрути, включаючи метрики навантаження, для подальшого аналізу.

Далі нейронна мережа обробляє отримані дані та обчислює поточне навантаження на кожен з можливих шляхів, використовуючи розроблену модель та алгоритм навчання.

Далі, на основі рішення нейронної мережі обирається шлях з найменшим інтегральним значенням навантаження, який потім вважається оптимальним для передачі потоку даних.

Далі нейронна мережа передає інформацію про обраний шлях контролеру SDN, який, на базі цієї відповіді, оновлює свої дані.

Потім контролер генерує нові записи в таблицях потоків комутаторів OpenFlow вздовж обраного шляху, які визначено нейронною мережею.

Комутатори, отримавши оновлені таблиці потоків, виконують необхідні дії для передачі потоку даних через мережу.

2.2. Обрахування показників вибору шляху

Основною метою стратегії балансування навантаження є визначення найменш завантаженого маршруту з урахуванням вимог до QoS для передачі даних в SDN мережі. Для отримання вичерпних даних про навантаження на кожен маршрут використовується протокол OpenFlow, який дозволяє контролеру SDN отримувати інформацію про навантаження на каналі між двома комутаторами.

У запропонованій концепції системи процес передачі потоку даних ініціюється SDN-контролером, який запитує таблицю потоків і

збирає статистику по кожному порту з комутаторів OpenFlow, розташованих уздовж заданого маршруту передачі. Зібрані дані про навантаження на кожен канал використовуються для розрахунку загального стану навантаження на даному маршруті за допомогою нейронної мережі.

Процес збору та подальшої обробки даних відбувається в режимі реального часу, що дозволяє адаптивно реагувати на зміни мережевого трафіку та забезпечує врахування поточного стану навантаження. Для остаточної передачі даних обирається маршрут з найменшим навантаженням. Використання нейронних мереж в цій стратегії дає змогу враховувати складні взаємозв'язки між елементами мережі, тим самим полегшуючи визначення оптимального маршруту.

У контексті протоколу OpenFlow, контролер SDN може отримувати статистику портів з кожного комутатора OpenFlow за допомогою повідомлень OFPT_STATS_REQUESTS. Зібрана інформація включається в таблиці потоків, які визначаються самим протоколом OpenFlow.

Таблиці потоків є основою для керування потоками даних в комутаторах OpenFlow, причому кожен комутатор може підтримувати одну або більше таких таблиць. Таблиця потоків складається з записів, які можна вважати основними одиницями збігу.

У протоколі OpenFlow запис таблиці потоків складається з шести компонентів, а саме поля відповідності, пріоритети, лічильники, інструкції, тайм-аути та куки (Cookie) [13]. Поля відповідності порівнюються з інформацією заголовка пакета, включаючи IP-адресу джерела, IP-адресу призначення, протокол, порт і, за наявності, метадані. При передачі пакетів на комутатор виконується порівняння між інформацією заголовка пакета і полями відповідності, що зберігаються в таблицях потоків. У випадку, якщо запис у таблиці потоків відповідає пакету, лічильники оновлюються статистикою, що

відноситься до потоку даних, про який йде мова. Сюди входить кількість пакетів, кількість переданих байт, тривалість передачі тощо. Після цього пакет переходить до виконання дій, зазначених в інструкціях. Тайм-аути визначають максимально допустимий час простою для даного елемента таблиці потоків. Для ілюстрації, приклад конфігурації таблиці потоків для комутатора OpenFlow представлено в таблиці 2.1. Коли який-небудь потік даних передається в комутатор OpenFlow та містить інформацію, таку як: `ip_src=10.0.0.1`, `ip_dst=10.0.0.2`, записи таблиці потоків будуть порівнюватися з цією інформацією послідовно. Спочатку відбудеться порівняння з записом таблиці потоків із пріоритетом 1. Оскільки поля відповідності в цьому записі не відповідають інформації заголовку потоку даних, ця інформація буде порівнюватися із наступним записом таблиці потоків. Зрештою, вона буде рівна запису таблиці потоків під номером 3, і тоді комутатор OpenFlow виконає дії, визначені полем інструкцій (Instructions), що пакети будуть передані через `port1`, і оновить лічильники.

Таблиця 2.1. Приклад таблиці потоку в комутаторі OpenFlow

№	Match Fields	Priority	Instructions	Timeouts	Counters
1	<code>ip_src=192.168.1.*</code>	1	Drop	60s	<code>n_packets=30</code> <code>n_bytes=2893</code>
2	<code>mac_dst=AA:BB:CC:DD:EE:FF</code> <code>ip_dst=192.168.3.101</code>	2	Output: <code>port2</code>	60s	<code>n_packets=60</code> <code>n_bytes=3730</code>
3	<code>ip_src=10.0.0.1</code> <code>ip_dst=10.0.0.2</code>	2	Output: <code>port1</code>	60s	<code>n_packets=10</code>

№	Match Fields	Priority	Instructions	Timeouts	Counters
					n_bytes= 169
...
20	mac_dst=AA:BB:CC:D D:EE:FF	10	Output: port3	60s	n_packet s=25 n_bytes= 2450

Далі SDN контролер передає зібрану маршрутну інформацію, яку отримав через протокол OpenFlow, до нейронної мережі для подальшої обробки. Такий підхід приводить архітектуру системи у відповідність до стандартної моделі SDN, тим самим зменшуючи навантаження на SDN контролер.

Вибір відповідних показників залежить від цільового призначення мережі. У цьому дослідженні для аналізу було обрано чотири характеристики шляху: коефіцієнт використання пропускної здатності, коефіцієнт втрат пакетів, час передачі та кількість хопів на кожному шляху. Вони були обрані як вхідні параметри для навчання нейронної мережі.

Коефіцієнт використання пропускної здатності є важливим фактором в оцінці ефективності передачі даних в мережевому середовищі. Він дає уявлення про ступінь навантаження на конкретну лінію зв'язку. Значення, що наближається до одиниці, вказує на високий ступінь використання доступної пропускної здатності, що може свідчити про потенційну перевантаженість шляху.

Для того, щоб розрахувати цей коефіцієнт - K_b , контролер SDN повинен спочатку зібрати кількість переданих байтів, позначену - N_b , на відповідних портах OpenFlow-комутаторів. Віднімаючи значення N_b та N_{b-1} (попередній період), можна отримати обсяг переданих даних за

певний період, який представляє використану пропускну здатність каналу за цей час. Останнім кроком є ділення використаної пропускну здатності на максимальну пропускну здатність (B_{MAX}), що дозволяє отримати остаточне значення коефіцієнта використання пропускну здатності. Його можна розрахувати за допомогою наступного рівняння (2.1):

$$K_B = \frac{N_b - N_{b-1}}{B_{MAX}}, \quad (2.1)$$

, де N_b -обсяги переданих байтів, B_{MAX} -максимальна пропускна здатність.

Якщо конкретний маршрут включає в себе низку зв'язків, позначених як $L_1, L_2, L_3 \dots L_n$, у кожного з яких значення характеристики використання пропускну здатності - $K_{B1}, K_{B2}, K_{B3}, \dots, K_{Bn}$ відповідно, то співвідношення використання пропускну здатності для цього маршруту може бути обчислене за формулою (2.2):

$$K_B = \frac{1}{n} \sum_{i=1}^n K_{Bi}, \quad (2.2)$$

, де K_{Bi} – коефіцієнт пропускну здатності шляху.

Втрати пакетів є невід'ємною характеристикою передачі даних. Це відбувається, коли кількість пакетів, які обробляються комутатором, перевищує його пропускну здатність, що призводить до втрати деяких пакетів. Відповідно, коефіцієнт втрат пакетів (K_{PL}) служить важливим показником для вимірювання навантаження на комутатор і загального стану маршруту. Щоб оцінити цей параметр, контролер SDN збирає інформацію про кількість переданих (P_T) і підтверджених пакетів (P_R) на відповідних портах комутаторів OpenFlow. Відповідно, цей коефіцієнт для конкретного маршруту можна розрахувати за формулою

(2.3):

$$K_{PL} = \frac{P_T - P_R}{P_T} \quad (2.3)$$

, де P_T - кількість переданих пакетів, P_R - кількість підтверджених пакетів.

Час передачі – це характеристика, що представляє час, необхідний вузлу комутатора для передачі даних. Цей показник тісно корелює з продуктивністю комутаторів, розміром пакетів, що передаються, і станом черг передачі. Час передачі можна використовувати як індикатор умов перевантаження та загального навантаження комутатора.

Щоб оцінити час передачі, контролер SDN повинен активно збирати інформацію щодо кількості байтів, переданих за визначений період, і швидкості передачі на відповідних портах комутаторів OpenFlow. Час передачі можна розрахувати на основі вищезазначених даних за формулою (2.4):

$$TD_i = \frac{B}{TS} \quad (2.4)$$

, де B - кількість переданих байтів, TS - швидкість передачі на відповідних портах комутаторів OpenFlow, відповідно $\frac{B}{TS}$ – час передачі.

Якщо один маршрут складається з кількох зв'язків $L_1, L_2, L_3 \dots L_n$ з відповідним часом передачі $TD_1, TD_2, TD_3 \dots TD_n$, то загальний час цього маршруту обраховується як сума затримок на кожному зв'язку (2.5):

$$TD = \frac{1}{n} \sum_{i=1}^n TD_i \quad (2.5)$$

, де TD_i – час передачі на певному шляху.

Кількість переходів у мережі є важливим фактором, який враховується стратегією маршрутизації. Збільшення кількості переходів може призвести до підвищеної ймовірності перевантаження, що може вплинути на продуктивність мережі. Можна припустити, що за тих самих умов передача пакету по маршруту зі зменшеною кількістю переходів призведе до зниження ймовірності втрати пакета та зменшення часу передачі. Зберігаючи топологію мережі в базі даних контролера SDN, кількість переходів між двома комутаторами можна визначити шляхом запиту бази даних за допомогою комутатора джерела та комутатора призначення.

Цей підхід дозволяє враховувати фізичні характеристики мережі, тим самим полегшуючи оптимізацію маршрутизації та сприяючи ефективній передачі даних із мінімальною кількістю переходів. Відповідно, показник кількості переходів має важливе значення для визначення оптимального маршруту з точки зору продуктивності та надійності мережі.

2.3. Балансування навантаження на основі штучної нейронної мережі

Щоб визначити кращий маршрут із найменшим навантаженням, важливо визначити стан завантаження шляху за допомогою аналізу інформації про шлях. Ця інформація визначається як набір даних, який забезпечує повне уявлення про статус шляху в мережі. У даній роботі для цієї мети використовуються такі характеристики: коефіцієнт пропускної здатності, коефіцієнт втрат пакетів, час передачі та кількість хопів. Аналізуючи всі ці аспекти разом, система на основі нейронної мережі приймає рішення щодо вибору маршруту для передачі даних.

Оцінка та прогнозування параметрів трафіку відіграють важливу

роль у підвищенні продуктивності та якості мережеских послуг. Для вирішення цієї проблеми використовується підхід машинного навчання. Оскільки вибір методу або характеристики може вплинути на точність і раціональність оцінки стану навантаження на шляху, важливо вибрати відповідний метод.

Штучна нейронна мережа — це обчислювальна модель, яка представляє нелінійну та самоадаптивну систему обробки інформації, що складається з взаємопов'язаних обчислювальних блоків. Він демонструє нелінійні властивості та властивості самонавчання. У порівнянні з альтернативними методами машинного навчання на основі ймовірностей, такими як логістична регресія, штучна нейронна мережа не обмежена вхідними векторами, що робить її ефективним засобом обробки невизначених даних мережевого трафіку. Формування набору даних, що містить значну кількість даних, дозволяє моделі ефективно працювати зі складними та невизначеними розподілами ймовірностей.

Використання багаторівневого перцептрона (MLP) з метою балансування навантаження в SDN мережі представляє багатообіцяючий напрямок дослідження, враховуючи здатність MLP вирішувати складні проблеми керування трафіком. Природа мережевого трафіку в таких мережах характеризується нелінійністю, завдяки чому зв'язок між різними параметрами, такими як пропускна здатність, час передачі пакетів, втрата пакетів і перевантаженість маршруту, не завжди є лінійною. Багатошаровий перцептрон здатний ефективно обробляти ці нелінійні залежності за допомогою використання нелінійних функцій активації, таких як ReLU або Sigmoid. Це дає змогу багаторівневому перцептрону моделювати складні зв'язки між вхідними характеристиками мережі та визначати оптимальні рішення в контексті конструювання трафіку.

Ключовою перевагою MLP є його властивість універсального апроксиматора. Це означає, що він здатен наближати будь-яку складну

функцію. У контексті балансування навантаження в SDN мережах це дає змогу проводити моделювання різноманітних сценаріїв та приймати рішення на основі широкого спектру параметрів. Завдяки цьому багат шаровий перцептрон адаптується до різноманітних умов у мережі та обирає найбільш ефективний шлях передачі потоку даних, що є важливим пунктом в SDN мережах з високою інтенсивністю трафіку.

Також слід зазначити, що програмно-конфігуровані мережі потребують постійного моніторингу та обробки великого обсягу багатовимірних даних, таких як пропускна здатність каналів, поточне навантаження, затримки та втрати пакетів. Час прийняття рішення про балансування навантаження є багат факторіальною функцією та залежить від багатьох факторів, зокрема кількості вузлів системи, інтенсивність трафіку та складності самої системи. Багат шаровий перцептрон обробляє ці багатовимірні дані та враховує всі взаємозалежності між ними. Це дозволяє приймати зважене рішення відносно балансування навантаження в SDN мережі та дозволяє оптимізувати процес обробки даних для прийняття рішення про маршрутизацію, звільнивши контролер від обробки цих даних. Гнучкість MLP дозволяє йому вчитися на історичних наборах даних з відомими оптимальними маршрутами передачі, тим самим дозволяючи адаптувати свої ваги для більш точного прогнозування оптимальних шляхів у майбутньому.

Крім того, MLP сприяє швидкій та ефективній обробці даних у режимі реального часу. Розпаралелювання процесів дає змогу нейронній мережі швидко обробляти інформацію з різних сегментів мережі та оцінювати альтернативні маршрути передачі даних. Це особливо важливо в програмно-конфігурованих мережах, де трафік динамічно змінюється, що вимагає швидкого прийняття рішень щодо маршрутизації. Варто зазначити, що багат шаровий перцептрон може перенавчатися у відповідь на зміни в структурі трафіку, тим самим

підвищуючи здатність системи ефективно адаптуватися до умов, що змінюються.

Тому використання багат шарового перцептрона в контексті систем балансування навантаження для SDN мереж є перспективним напрямком, враховуючи вищезгадані можливості MLP, а саме здатність враховувати складні нелінійні залежності, гнучкість у навчанні та масштабованість. Це робить багат шаровий перцептрон оптимальним інструментом для вирішення проблем в області інтелектуального керування мережевим трафіком, де ефективність, точність і доцільність прийняття рішень мають першочергове значення для забезпечення безперебійної роботи мережі.

У світлі вищесказаного, багат шаровий перцептрон був обраний як основний компонент нейронної мережі, яка розробляється у даній дисертації (рис 2.4). Це нейронна мережа прямого поширення, в якій сигнал перетворюється у вихідний, проходячи послідовно через декілька шарів. Цей перцептрон складається з декількох шарів – вхідний шар, один або декілька прихованих шарів та вихідний шар.

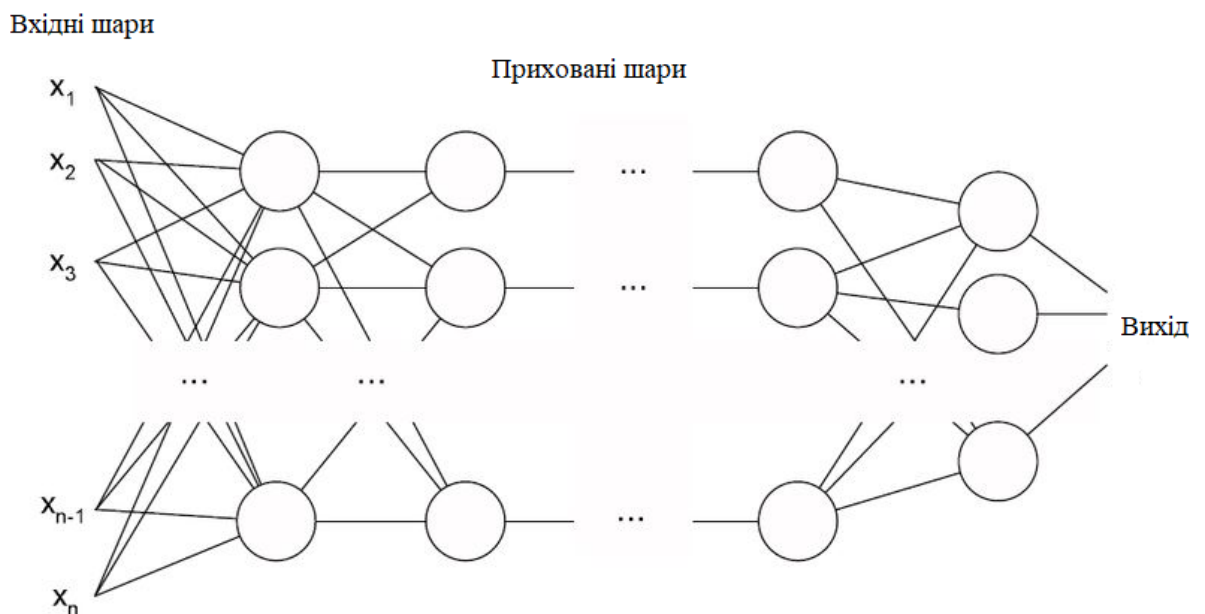


Рис 2.4. Архітектура багат шарового перцептрона

Для навчання мережі використовується метод зворотного поширення помилки. Цей алгоритм навчання багат шарових перцептронів заснований на обрахунку градієнту функції помилки. У процесі навчання ваги нейронів корегуються з урахуванням сигналів, що надійшли з попереднього шару, і зсувів кожного шару, що обчислюється рекурсивно у зворотному напрямку від останнього шару до першого.

У даного виду мережі є декілька входів v_i ($i = 1, \dots, n$), один або декілька виходів та декілька внутрішніх нейронів. Через w_{ik} позначаємо вагу, що знаходиться на ребрі, що з'єднує i -й та k -й нейрони, а через o_i – вихід i -го нейрона. Використовуючи метод найменших квадратів, у випадку, якщо відомо навчальний приклад, функція помилки (2.6) матиме наступний вигляд

$$E(\{w_{ik}\}) = \frac{1}{2} \sum_m (t_m - o_m)^2 \quad (2.6)$$

, де t_m - правильні відповіді мережі, а m – вихідний нейрон.

Для модифікування ваг мережі використовується стохастичний градієнтний спуск. Тобто, після кожного навчального прикладу ваги на ребрах корегуються і пересуваються у багатовимірному просторі ваг. Оскільки потрібно дібратись до мінімуму помилки, то корегування ваг відбувається у сторону, протилежну градієнту. Тобто, опираючись на кожну групу правильних відповідей, до кожної ваги додається w_{ik} .

$$\Delta w_{ik} = -n \frac{\partial E}{\partial w_{ik}}, \quad (2.7)$$

, де $0 < n < 1$ – множник, що задає швидкість «руху».

Структура штучної нейронної мережі, яка складається з багат шарового перцептрону, формується на основі формул 2.8 - 2.11.

Такий підхід дозволяє моделі ефективно враховувати складні взаємозв'язки в даних трафіку та забезпечує високий рівень аналізу та передбачення стану завантаження шляху в мережі.

$$u_k = \sum_{i=1}^n w_{ik} v_i \quad (2.8)$$

, де v_i ($i = 1, \dots, n$) відображає вектори введення, а w_{ik} ($i = 1, \dots, n$) представляє вагу нейрона k .

У рівнянні (2.8) визначається ваговий коефіцієнт, який вказує на ступінь впливу введення на дію нейрона. Зокрема, якщо $w_{ik} > 0$, це вказує на стан збудження, тоді як $w_{ik} < 0$ вказує на стан стримування. Тут n позначає кількість векторів введення, що використовуються для навчання нейронної мережі. Параметр u_k визначається як лінійна комбінація вектора введення та ваги нейрона.

Похідна (2.9) у такому випадку, для нейрону останнього рівня, розраховується наступним чином:

$$\frac{\partial E}{\partial w_{ik}} = \frac{\partial E}{\partial u_k} * \frac{\partial u_k}{\partial w_{ik}} = v_i \frac{\partial E}{\partial u_k} \quad (2.9)$$

На загальну помилку u_k також впливає тільки в рамках вихідного нейрона всієї мережі:

$$\begin{aligned} \frac{\partial E}{\partial u_k} &= \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial u_k} = \left(\frac{\partial}{\partial o_k} \frac{1}{2} \sum_m (t_m - o_m)^2 \right) \left(\frac{\partial y_k}{\partial u_k} \right) = \\ &= \left(\frac{1}{2} \frac{\partial}{\partial o_k} (t_k - o_k)^2 \right) (o_k (1 - o_k)) = -o_k (1 - o_k) (t_k - o_k), \end{aligned} \quad (2.10)$$

, де y_k – це функція активації.

$$y_k = f(u_k + b_k) \quad (2.11)$$

Функція активації $f(x)$ (2.11) визначає реакцію нейрона на вхід, враховуючи ваги та порогове значення. Порогове значення нейрона (зсув) позначається як b_k і використовується для керування входом функції активації. Цей параметр визначає, чи має стан збудження або стримування певний нейрон. Вихід нейрона, позначений як y_k , є результатом функції активації та представляє собою фінальний вихід нейрона. Всі ці параметри і структурні елементи штучної нейронної мережі визначають здатність адаптуватися і вирішувати завдання навчання або прогнозування в мережевому середовищі.

Для інтеграції інформації про шлях використовується тришарова штучна нейронна мережа, яка використовує метод зворотного поширення. Процес навчання нейронної мережі складається з двох основних фаз. Початкова фаза - це фаза прямого навчання. Під час цієї фази вхідні нейрони обробляють вхідні вектори з попередньо встановленими вагами за допомогою функції активації. Далі результат з прихованого шару передається на вихідний шар, де генеруються остаточні результати.

Після завершення фази навчання нейронна мережа переходить до фази налаштування ваги. На цьому етапі невідповідність між кінцевими результатами та очікуваними результатами використовується для уточнення ваг. У випадку, якщо результати, отримані вихідним шаром, є надмірними, помилка використовується для зворотного поширення, таким чином це дозволяє змінювати ваги нейронів у кожному шарі. Цей процес налаштування нейронної мережі триває до тих пір, поки помилка не досягне прийнятного рівня.

У дисертації використано 4 характеристики для балансування трафіку у програмно-конфігурованих мережах як вхідний вектор, що відповідає наявності 4 вхідних нейронів у нейронній мережі. Тим не

менш, для досягнення оптимальних результатів важливо визначити кількість нейронів у прихованому шарі. Для цього була використана формула (2.12):

$$N_h = \sqrt{n_{out} + n_{in}} + a, \quad (2.12)$$

, де N_h - це кількість нейронів у прихованому шарі, n_{in} - кількість нейронів у вхідному шарі, n_{out} - кількість нейронів у вихідному шарі, a - константа від 1 до 10.

Для проведення експерименту встановлено значення 1000 як кількість навчань, а ціль навчання встановлено як 0,001. Щодо кількості нейронів у прихованому шарі будуть вибрані різні значення константи a : $a=1$, $a=3$, $a=5$, $a=7$ та $a=9$. Відповідно, отримані кількості нейронів у прихованому шарі дорівнюють 3, 5, 7, 9 та 11.

Висновки до розділу 2

1. В даному розділі запропоновано метод балансування навантаження LBBNN, який, за рахунок використання системи штучного інтелекту, спрямований на вибір оптимального маршруту з найменшим навантаженням для передачі потоку даних у програмно-конфігурованій мережі з урахуванням їх особливостей.

2. Запропоновано метод розрахунку інтегрального навантаження кожного маршруту на основі чотири ключові характеристики мережі - коефіцієнту використання пропускної здатності, коефіцієнту втрат пакетів, часу передачі та кількості стрибків для передачі пакетів, які дозволяють обрати оптимальні маршрути для передачі нового потоку даних.

3. Запропоновано та обґрунтовано метод використання модифікованої нейронної мережі зворотного поширення для реалізації балансувальника навантаження у програмно-конфігурованій мережі.

Застосування запропонованої схеми сприяє підвищенню ефективності використання ресурсів мережі SDN та забезпечує оптимальне розподілення навантаження, що в результаті сприяє підвищенню продуктивності та якості обслуговування у системах передачі даних.

РОЗДІЛ 3

ЗАСОБИ КОНСТРУЮВАННЯ ТРАФІКУ У ПРОГРАМНО- КОНФІГУРОВАНИХ МЕРЕЖАХ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ

3.1. Вибір засобів розробки системи балансування навантаження

Для розробки системи балансування навантаження трафіку у програмно-конфігурованих мережах основною мовою програмування обрано Python. Це мова високого рівня, яка дотримується суворої динамічної типізації та об'єктно-орієнтованого підходу, що полегшує інтерпретацію та аналіз коду.

Ключовою перевагою цієї мови є її сумісність з великою кількістю бібліотек, що використовуються для реалізації алгоритмів машинного навчання. Це робить її чудовим інструментом для розробки систем, які потребують використання нейронних мереж для аналізу трафіку та балансування навантаження. Було використано наступні бібліотеки та інтерфейси: TensorFlow, Keras та Scikit-learn (Sklean).

TensorFlow - це інструмент для глибокого навчання та розробки нейронних мереж, призначений для створення та навчання складних моделей з безліччю шарів та параметрів. Він пропонує комплексну функціональність, включаючи підтримку зворотного поширення помилки - фундаментального алгоритму для навчання нейронних мереж. TensorFlow також надає гнучкий API, який дозволяє легко створювати і налагоджувати різноманітні моделі нейронних мереж, а також маніпулювати різними типами даних.

Keras - це інтерфейс високого рівня, який використовується для розробки та навчання нейронних мереж. Він пропонує простий і прозорий підхід до створення та оптимізації моделей. Keras полегшує

побудову нейронних мереж з різноманітною архітектурою та їх навчання на власній обчислювальній платформі користувача, включаючи TensorFlow та PyTorch. Він також дозволяє швидко проводити експерименти з різними архітектурами та конфігураціями, тим самим спрощуючи процес розробки нейронних мереж.

Pandas - це програмна бібліотека, що призначена для обробки та аналізу даних. Вона надає інструменти для роботи з різними структурами даних та виконання операцій над ними, включаючи фільтрацію, сортування, групування та агрегацію. Ці інструменти полегшують маніпуляції з числовими таблицями та часовими рядами. Також ця бібліотека надає можливість читати і записувати дані з різних джерел, включаючи файли CSV, бази даних SQL та файли Excel.

Scikit-learn – це бібліотека машинного навчання, яка надає широкий спектр алгоритмів для вирішення різних типів завдань, включаючи класифікацію, регресію, кластеризацію та інші. Бібліотека використовується для розробки моделей класифікації трафіку в SDN мережах. Вона пропонує зручний інтерфейс для роботи з даними та використання алгоритмів машинного навчання.

Mininet - це інструмент, який використовується для створення віртуальних SDN мереж на основі OpenFlow. Він надає доступне середовище для розробки, тестування та експериментів з мережевими протоколами та алгоритмами. Mininet дозволяє створювати віртуальні мережі, що складаються з вузлів, маршрутизаторів, комутаторів та хостів, які можна конфігурувати та керувати за допомогою Python або інших мов програмування.

Mininet дає можливість моделювати складні мережеві топології та оцінювати різні алгоритми маршрутизації та керування мережею в регламентованому середовищі. Цей інструмент широко використовується у сфері освіти та досліджень у галузі мережевих технологій, оскільки він дозволяє швидко створювати та тестувати нові

концепції та ідеї без наявності фізичного обладнання.

ONOS - це програмне забезпечення з відкритим кодом. Воно розроблене спільнотою для створення та керування розподіленими мережами SDN. ONOS забезпечує централізовану систему керування мережею, яка дозволяє ефективно керувати трафіком та ресурсами в мережі. Він надає ряд функцій, включаючи маршрутизацію, комутацію, моніторинг та керування мережевими послугами. Система має розподілену архітектуру, що задовольняє вимоги великих мереж і середовищ з високою інтенсивністю трафіку.

3.2. Формування та попередня обробка навчального набору даних

Визначивши параметри для побудови моделі штучної нейронної мережі (формула 2.12), можна графічно зобразити структуру штучної нейронної мережі зі зворотним поширенням похибки на рис. 3.1. Вона складається з вхідного шару з чотирма нейронами, прихованого шару з N_h нейронів та вихідного шару з одним нейроном.

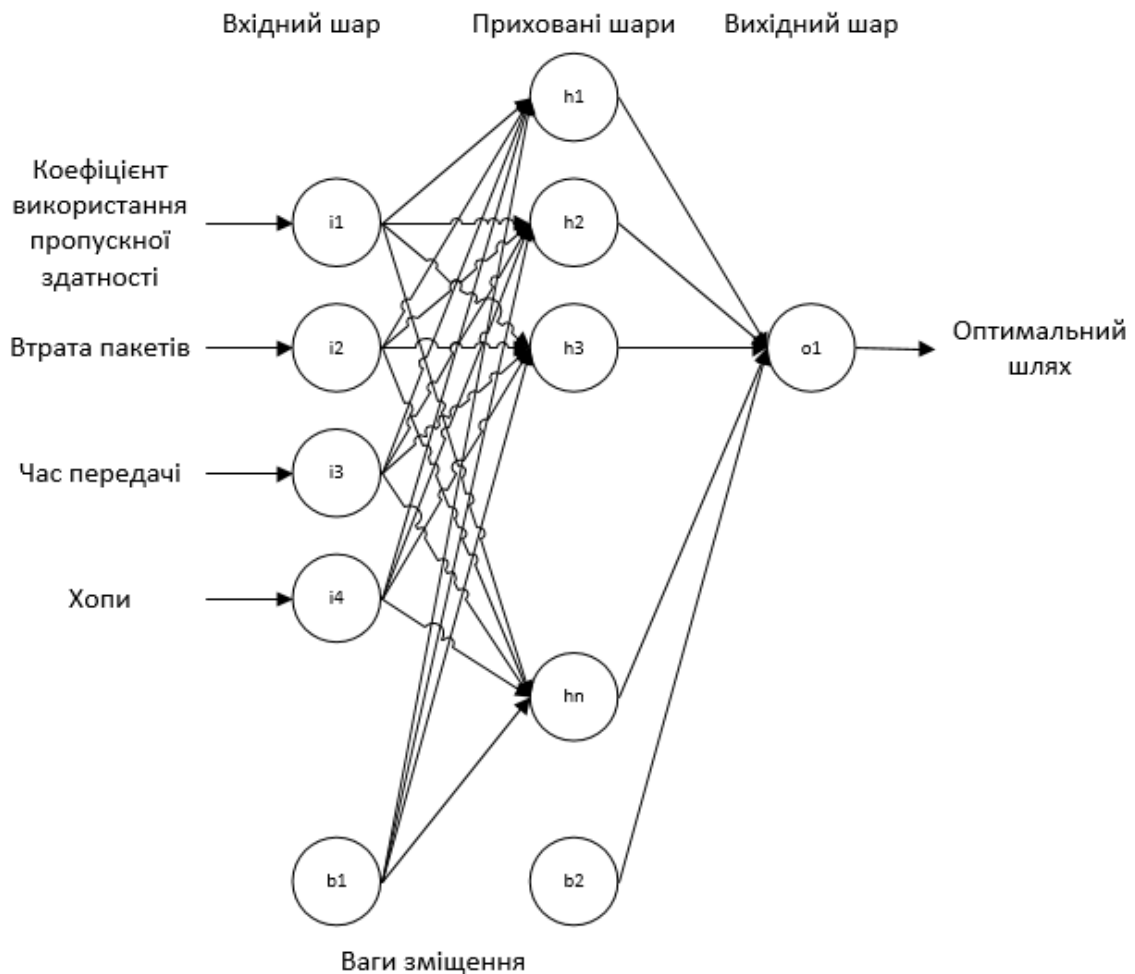


Рис. 3.1. Структура нейронної мережі для балансування навантаження

Одним із етапів у розробці системи штучної нейронної мережі є її навчання на значному наборі даних для досягнення найменшої можливої помилки прогнозування.

У цьому розділі здійснюється попередня обробка набору даних щоб полегшити його подальше використання для навчання нейронної мережі для балансування навантаження в SDN. Спочатку набір даних уточнюється шляхом видалення помилкових і дефектних прикладів. Наступним етапом є розділення набору даних на дві окремі підмножини: навчальну та тестову вибірки. Далі проводиться відбір ознак, що передбачає ідентифікацію підмножини відповідних ознак для подальшого використання в моделюванні.

Процес очищення даних передбачає видалення або виправлення записів, які містять помилки чи дефекти, що потенційно може поставити під загрозу правильність прогнозування розробленої моделі машинного навчання. Поділ даних на навчальний та тестовий набори дозволяє оцінити продуктивність моделі на даних, які не використовувалися під час фази навчання, забезпечуючи об'єктивність оцінювання. Вибір ознак є важливим етапом, оскільки він спрямований на визначення змінних, які підвищують ефективність і точність моделі.

Для навчання нейронної мережі було обрано набір даних IP Network Traffic Flows Labeled with 75 Apps. Представлені тут дані були зібрані на ділянці мережі в Університеті Дель-Каука, Попаян, Колумбія, шляхом захоплення пакетів у різний час, вранці та вдень, протягом шести днів (26, 27, 28 квітня та 9, 11 і 15 травня) 2017 року. Набір даних складається з 87 характеристик. Кожен блок містить інформацію, що стосується IP-потoku, створеного мережевим пристроєм. Це включає IP-адреси джерела та адресата, порти, час зворотного зв'язку та інші дані. Більшість атрибутів має числовий тип, але також присутні номінальні типи та тип дат, виражені як мітки часу. Доцільність використання запропонованого набору даних для побудови моделі машинного навчання обґрунтовано в [40, 68, 69].

Для використання цього набору даних потрібно виконати попередню обробку. Спочатку було виконано видалення помилкових та дефектних даних з набору. Далі було відібрано вибірку для використання в навчанні моделі. Після етапу обробки були видалені всі неповні записи.

Крім того, після попередньої перевірки було виявлено ще одну невідповідність у стовпчиках ознак. Набір даних складається з 87 стовпців, які визначають характеристики потоку, включаючи Flows.ID, Source.IP та Source.Port. Однак стовпець, позначений як «Fwd.Header.Length», присутній у трьох випадках, а саме у 40-му, 41-му

та 61-му стовпчиках. Цю невідповідність було виправлено шляхом видалення стовпців, що повторюються (рис. 3.2).

```
# Зчитування даних
pd.set_option('display.float_format', '{:.2f}'.format)
dataset = pd.read_csv("../input/ip-network-traffic-flows-labeled-with-87-apps/Dataset-Unicauca-Version2-87Atts.csv")

# Видалення дублюючих колонок
dataset = dataset.loc[:, ~dataset.columns.duplicated()]
```

Рис. 3.2. Видалення дублюючих колонок

У цьому коді метод *dataset.columns.duplicated()* повертає логічний масив, який вказує, які стовпці в наборі даних є дублікатами. Щоб інвертувати логічний масив, необхідно використати оператор логічного заперечення "~". Після виконання методу *dataset.loc[:, ~dataset.columns.duplicated()]* залишаються лише унікальні стовпці.

Ще одним важливим етапом попередньої обробки даних є перетворення ознак з категоріальними та текстовими значеннями (такими як Timestamp, Label та ProtocolName) у числові формати, для їх подальшого використання в алгоритмах машинного навчання. Це перетворення можна здійснити за допомогою класу LabelEncoder з бібліотеки Sklearn. Отже, текстові значення, які безпосередньо не застосовуються в алгоритмі машинного навчання, замінюються числовими значеннями в діапазоні від 0 до n-1. Це використовується для підвищення їх придатності для подальшої обробки (рис. 3.3).

```
# Створення об'єкту LabelEncoder
le = LabelEncoder()

# Перетворення категоріальних ознак в числові
for column in ['Timestamp', 'Label', 'ProtocolName']:
    dataset[column] = le.fit_transform(dataset[column])
```

Рис. 3.3. Перетворення ознак

У цьому випадку створюється об'єкт *LabelEncoder()*. Згодом використовується цикл для перетворення кожної категоріальної ознаки з використанням *fit_transform()* для кожного стовпця. Це дозволяє перетворювати текстові значення на числові.

Крім того, стовпці перевіряються на наявність значень нескінченності та NaN, які виправляються на -1 та 0 відповідно, щоб гарантувати точну обробку даних.

```
# Перетворення значень Infinity на -1 та NaN на 0
dataset.replace([np.inf, -np.inf, np.nan], [-1, -1, 0], inplace=True)
```

Рис. 3.4. Перевірка на наявність значень Infinity та NaN та їх виправлення

Тут використовується вбудована функція бібліотеки pandas – *replace*. Вона оптимізована для роботи з великими обсягами даних і ефективніша за ітераційні методи. Її можна використовувати для обробки великих наборів даних, виконуючи заміну за один прохід. Однак цей метод менш гнучкий для складних перетворень.

Функція *replace([np.inf, -np.inf, np.nan], [-1, -1, 0], inplace=True)* замінює всі значення *np.inf*, *-np.inf* на -1, а всі значень *np.nan* на 0 у один прохід.

Крім того була додана функція `pd.set_option('display.float_format', '{:.2f}'.format)`. Вона дозволяє конфігурувати формат відображення чисел з плаваючою комою в `pandas DataFrame`. Це особливо корисно, коли метою є відображення числових значень із заданою кількістю знаків після коми, наприклад, два знаки після коми.

Оброблений датасет представлений на рисунку 3.5.

[17]:

	Source.IP	Source.Port	Destination.IP	Destination.Port	Flow.Bytes.s	Total.Fwd.Packets	Total.Backward.Packets	Flow.Duration	Flow.IAT.Mean	TTL
0	172.19.1.46	52422	10.200.7.7	3128	2428354.90	22	18	45523	598.99	5
1	10.200.7.7	3128	172.19.1.46	52422	12000000.00	2	0	1	1.00	5
2	50.31.185.39	80	10.200.7.217	38848	674000000.00	3	2	1	0.50	7
3	50.31.185.39	80	10.200.7.217	38848	0.00	1	0	217	72.33	7
4	192.168.72.43	55961	10.200.7.7	3128	13782.86	5	0	78068	19517.00	13
5	10.200.7.6	3128	172.19.1.56	50004	2984267.48	136	46	105069	778.29	17
6	192.168.72.43	55963	10.200.7.7	3128	10302.27	5	2	104443	26110.75	13
7	192.168.10.47	51848	10.200.7.6	3128	354117.43	3	2	11002	785.86	24
8	68.67.178.197	443	10.200.7.217	57300	75629.25	10	9	108503	7233.53	19
9	192.168.72.43	55977	10.200.7.7	3128	18663.18	7	2	118415	19735.83	13
10	10.200.7.4	3128	192.168.180.51	57740	46860.83	32	24	205118	5860.51	21
11	10.200.7.4	3128	192.168.180.51	57740	3330333333.33	5	3	3	0.75	21
12	10.200.7.4	3128	192.168.180.51	57740	42832061.07	3	1	131	65.50	21
13	10.200.7.4	3128	192.168.180.51	57740	935166666.67	3	2	6	3.00	21
14	10.200.7.6	3128	172.19.1.45	50227	1779818.72	123	120	108338	888.02	22
15	212.124.124.94	443	10.200.7.194	44447	16348.67	4	0	202096	22455.11	10
16	10.200.7.4	3128	192.168.180.51	57741	24630.10	31	30	202151	5945.62	21
17	10.200.7.4	3128	192.168.180.51	57741	998000000.00	2	1	1	1.00	21
18	10.200.7.4	3128	192.168.180.51	57741	998000000.00	2	0	1	1.00	21
19	172.217.30.13	443	10.200.7.217	41526	0.00	2	0	1	1.00	7

+ Code + Markdown

Рис. 3.5. Вивід датасету після обробки

На рисунку 3.5 представлено десять ознак, які використовуються для аналізу мережевого трафіку, а саме: `Source.IP`, `Source.Port`, `Destination.IP`, `Destination.Port`, `Flow.Bytes.s`, `Total.Fwd.Packets`, `Total.Backward.Packets`, `Flow.Duration`, `Flow.IAT.Mean` та `TTL`.

- `Source.IP` (Source Internet Protocol Address) – IP-адреса пристрою, який ініціює передачу даних у мережі. Вказує на те, звідки надсилаються дані.

- `Source.Port` (Source Port Number) – номер вихідного порту, що позначає порт на пристрої джерела, через який здійснюється передача

даних.

- Destination.IP (Destination Internet Protocol Address) – IP-адреса пристрою, на який надсилаються дані. Вказує на кінцеву точку, де мають бути отримані дані.

- Destination.Port (Destination Port Number) – номер порту призначення. Позначає порт на пристрої призначення, куди надходять дані.

- Flow.Duration (Flow Transmission Duration) – час, протягом якого тривав мережевий потік, зазвичай вимірюється в мілісекундах. Це різниця між часом надсилання першого та останнього пакета в потоці.

- Total.Fwd.Packets (Total Forward Packets) – загальна кількість пакетів, які були відправлені від джерела до пристрою призначення.

- Total.Backward.Packets (Total Backward Packets) – загальна кількість пакетів, надісланих із пристрою призначення до джерела. Ці пакети йдуть у зворотному напрямку — від отримувача до відправника.

- Flow.Bytes.s (Flow Bytes per Second) – метрика, яка вказує на кількість байтів, переданих у потоці даних за одну секунду. Вона дозволяє виміряти швидкість передачі даних для конкретного потоку в реальному часі.

- Flow.IAT.Mean (Flow Inter-Arrival Time Mean) – середній час між прибуттям послідовних пакетів у потоці. IAT (Inter-Arrival Time) вимірює час між двома підряд отриманими пакетами. Середнє значення цього часу дозволяє оцінити регулярність або варіабельність у часі прибуття пакетів. Високі значення може вказувати на затримки або нерівномірність в передачі.

- TTL (Time to Live) – поле в заголовку IP-пакета, яке визначає максимальну кількість маршрутизаторів або "стрибків" (hops), через які може пройти пакет, перш ніж його буде видалено. TTL зазвичай зменшується на одиницю кожного разу, коли пакет проходить через

маршрутизатор. Коли TTL знижується до 0, пакет видаляється, щоб уникнути безкінечного циклу в мережі.

Для наповнення датасету були використані синтетичні дані, а саме TTL та maxBandwidth. Для TTL було згенеровано значення для кожного запису в таблиці від 0 до 63, в той час як для maxBandwidth використовувалось дефолтне значення в 1 гігабайт.

Для успішного функціонування алгоритмів машинного навчання необхідний навчальний набір для навчання моделі. У свою чергу, тестовий набір необхідний для остаточної оцінки продуктивності алгоритму та перевірки адекватності моделі на невідомих даних. Датасет "IP Network Traffic Flows Labeled with 75 Apps", який застосовується у даній роботі, не містить окремих спеціальних наборів даних для навчання та тестування, а надає лише єдиний розмічений набір даних. Тому необхідно вручну розділити датасет на навчальний та тестовий набори.

Для цього використовується функція *train_test_split* з бібліотеки Sklearn, яка дозволяє розділити дані на дві частини відповідно до вказаного користувачем розміру. Зазвичай оптимальним співвідношенням для поділу є 80% даних для навчального набору та 20% даних для тестового набору. Така ж пропорція використовується і в цій роботі.

```
# Вибір ознак для моделі
features = ['Bandwidth', 'Packets.Lost', 'Transmission.Delay',
'Hop.Count']

dataX = dataset[features]

# Перетворення цільової змінної на числову
# le_optimal_path = LabelEncoder()
```

```

dataY = dataset['Bandwidth']

#le_optimal_path.fit_transform(dataset['Flow.ID'])

# Поділ даних на навчальну та тестову вибірку
trainX, testX, trainY, testY = train_test_split(dataX, dataY,
test_size=0.2, random_state=42)

```

Рис. 3.6. Підготовка даних

В цьому коді відбувається вибір ознак та поділ датасету на навчальний та тестовий набори.

Для обрахування ознак потрібно створити нові колонки, куди буде записувати обраховані ознаки. Для обрахунку пропускної здатності використовується значення з *Flow.Bytes.s* яке представляє кількість байтів, що передаються за секунду в потоці поділену на максимальну пропускну здатність шляху:

$$\text{dataset['Bandwidth']} = \text{dataset['Flow.Bytes.s']} / \text{maxBandwidth}$$

Для обрахунку коефіцієнту втрати пакетів (*Packets.Lost*) обраховуємо різницю між кількістю переданих пакетів (*Total.Fwd.Packets*) і кількістю підтверджених пакетів (*Total.Backward.Packets*) поділену на кількість переданих пакетів (*Total.Fwd.Packets*):

$$\text{dataset['Packets.Lost']} = (\text{dataset['Total.Fwd.Packets']} - \text{dataset['New.Total.Backward.Packets']}) / \text{dataset['Total.Fwd.Packets']}$$

Для обрахунку часу передачі обраховуємо суму тривалості потоку та середнього значення часу передачі потоку:

$$\text{dataset['Transmission.Delay']} = \text{dataset['Flow.Duration']} + \text{dataset['Flow.IAT.Mean']}$$

Для обрахунку кількості переходів до кінцевого вузла потрібно від початкового значення TTL, яке залежить від ОС та конфігурації

системи, відняти кінцеве значення TTL.

```
dataset['Hop.Count'] = initial_ttl - dataset['TTL']
```

Далі потрібно обрати тільки ті колонки, які будуть використовуватися як вхідні ознаки для моделі. Обираємо чотири нові колонки: *Bandwidth*, *Packets.Lost*, *Transmission.Delay* та *Hop.Count* і зберігаємо їх у змінну *dataX*:

```
features = ['Bandwidth', 'Packets.Lost', 'Transmission.Delay',  
'Hop.Count']
```

```
dataX = dataset[features]
```

Далі використовується колонка *'Bandwidth'* для формування рядка, що представляє маршрут передачі пакетів. Потім ця колонка зберігається у змінну *dataY*, яка буде використовуватися як цільова змінна:

```
dataY = dataset['Bandwidth']
```

На останок, використовуємо функцію *train_test_split()*, яка розділяє дані на навчальний і тестовий набори. У нашому випадку задаємо *test_size=0.2* - 20% даних буде використано для тестування, а 80% для навчання.

Після цього дані необхідно нормалізувати. Це важливий попередній крок, особливо якщо набір даних складається з різних типів ознак та різними діапазонами значень. Нормалізація даних необхідна для того, щоб досягти більш збалансованого розподілу ваг ознак. Це дозволить вирішити проблему ознак з різними діапазонами значень, що дасть змогу системі розпізнавати та керувати лише тими ознаками, які мають більші діапазони значень, ігноруючи інші, та сприяти прискоренню процесу навчання.

Стандартна нормалізація, або стандартизація, передбачає перетворення значень ознак у стандартний нормальний розподіл із середнім значенням 0 і стандартним відхиленням 1 (рис. 3.7).

Формула 3.1 використовуються для нормалізації кожної ознаки x_i .

$$x'_i = \frac{x_i - \mu}{\sigma} \quad (3.1)$$

, де μ - середнє значення ознаки в навчальному наборі, σ - стандартне відхилення ознаки в навчальному наборі.

```
# Нормалізація даних
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(trainX)
X_test_scaled = scaler.transform(testX)
```

Рис. 3.7. Нормалізація даних

У цій частині коду створюється екземпляр `StandardScaler`, який стандартизує ознаки шляхом віднімання середнього значення та ділення на стандартне відхилення.

Потім навчальні дані підганяються та трансформуються. Функція підгонки `fit` обчислює середнє значення та стандартне відхилення для кожної ознаки в навчальному наборі даних. Розраховані значення потім використовуються функцією `transform` для стандартизації навчального набору. Далі кожне значення ознаки віднімається від відповідного середнього значення і ділиться на стандартне відхилення. Метод `fit_transform` надає зручне рішення, поєднуючи попередні два кроки в одному виклику функції. Результатом є стандартизовані навчальні дані, які зберігаються у змінній `trainX_scaled`.

Далі відбувається перетворення тестових даних. Метод `scaler.transform(testX)` реалізує стандартизацію тестового набору даних `testX`, використовуючи обчислені середнє значення та стандартне відхилення з набору даних `trainX`.

Таким чином, обидва набори даних (навчальний і тестовий)

містять нормалізовані значення, що полегшує навчання моделі та забезпечує точне порівняння між ознаками з різними масштабами.

3.3. Побудова та навчання моделі нейронної мережі

Після формування та обробки набору даних було отримано два датасети: навчальний і тестовий. Вони використовуються для навчання моделі нейронної мережі та перевірки її точності.

Побудова моделі. Модель нейронної мережі складається з трьох компонентів: вхідного шару, прихованих шарів та вихідного шару.

Вхідний шар відповідає за отримання вхідних даних з навчального набору. Кількість нейронів у вхідному шарі дорівнює кількості ознак, присутніх у наборі навчальних даних. У нашому випадку навчальний набір містить чотири ознаки, отже, вхідний рівень складатиметься з чотирьох нейронів. Слід зазначити, що вхідний рівень не виконує жодної обробки даних; його єдиною функцією є передача даних на наступний шар.

Приховані шари розташовані між вхідним і вихідним шарами і відповідають за первинну обробку даних. Кількість нейронів у прихованих шарах, а також функції активації можуть змінюватись залежно від конкретного завдання та структури моделі нейронної мережі. Розроблена модель використовує три прихованих шари, кожен з яких реалізує функцію активації ReLU (Rectified Linear Unit) та складається з N_h нейронів (формула 2.12).

Вихідний шар призначений для виводу остаточного результату моделі. У задачах регресії вихідний шар зазвичай містить один нейрон з лінійною функцією активації, яка забезпечує безперервне числове значення як результат передбачення. У нашій моделі вихідний шар складається з одного нейрону, який видає прогнозоване значення.

```

# Побудова моделі нейронної мережі
model = Sequential()
model.add(Input(shape=(X_train_scaled.shape[1],)))
model.add(Dense(7, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(7, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='linear')) # Вихідний шар для регресії

# Компіляція моделі
model.compile(loss='mean_squared_error',
optimizer=Adam(learning_rate=0.001), metrics=['mean_absolute_error'])

```

Рис. 3.8. Побудова моделі нейронної мережі

Побудова моделі відбувається викликом метода *Sequential()*. В даному випадку створюється порожня модель послідовного типу, яка дозволяє додавати шари один за одним. Далі додається вхідний шар за допомогою `model.add(Input(shape=(X_train_scaled.shape[1],)))` та повнозв'язний шар з 7 нейронами за допомогою `model.add(Dense(7, activation='relu'))`. Функція *Input()* визначає розмірність вхідних даних, яка дорівнює кількості ознак у навчальному наборі (*X_train_scaled.shape[1]*). Функція активації *relu* (Rectified Linear Unit) використовується для додавання нелінійності. Потім додається *Dropout* шар з рівнем випадкового вимикання 50% нейронів під час навчання. Це допомагає запобігти перенавчанню (*overfitting*). Аналогічно додаємо другий прихований шар та *Dropout* шари.

У кінці додаємо вихідний шар з одним нейроном за допомогою `Dense(1, activation='linear')`. Для забезпечення безперервного числового

виходу використовується функція активації `linear`.

Далі проходить компіляція побудованої моделі. Для цього вказується функція втрат, оптимізатор і метрика для оцінки. Для функції втрат використовується середньоквадратична помилка `loss='mean_squared_error'`, що є стандартом для задач регресії. Як оптимізатор, був використаний Adam з початковою швидкістю навчання 0.001: `optimizer=Adam(learning_rate=0.001)`. У вигляді метрики вказується середня абсолютна помилка `metrics=['mean_absolute_error']`.

Наступним кроком проводиться навчання та збереження моделі. У процесі навчання використовується метод `fit`, який приймає дані навчання, кількість епох, розмір батчу та параметри перевірки.

Після завершення процесу навчання для моделі нейронної мережі важливо зберегти її параметри, включаючи ваги та структуру, для подальшого використання. Збереження моделі слугує для того, щоб уникнути необхідності перенавчання, що є важким і трудомістким процесом, особливо коли маємо справу з об'ємними наборами даних.

Щоб зберегти модель, використовується функція `model.save()`, яка дозволяє записувати всі компоненти моделі, включаючи архітектуру, зважені параметри та конфігурацію оптимізатора, в один файл HDF5. Цей формат широко використовується для зберігання значної кількості даних і складних структур для ефективного зберігання як даних моделі, так і метаданих (рис. 3.9).

```
# Навчання моделі
history = model.fit(X_train_scaled, trainY, epochs=30, batch_size=32,
validation_split=0.2)

# Збереження моделі
model.save('optimal_path_model.h5')
```

Рис. 3.9. Навчання та збереження моделі

В цьому коді *trainX_scaled* і *trainY* - навчальні дані та відповідні цільові значення. Параметр *epochs=50* визначає кількість ітерацій по всьому набору навчальних даних, які буде виконувати модель. Для визначення кількості зразків, які обробляються перед оновленням параметрів моделі задаємо параметр *batch_size=32*. Параметр *validation_split=0.2* вказує, що 20% навчальних даних буде використано для перевірки.

Виклик функції *model.save()* зберігає модель у файлі з розширенням *.h5*. Цей процес зберігає не тільки параметри нейрона (ваги та зміщення), а й архітектуру моделі, включаючи кількість шарів, їх типи, функції активації та інші параметри. Крім того, внутрішні налаштування навчання, такі як оптимізатор і його поточний стан, також зберігаються. Це дає змогу, за потреби, продовжувати навчання моделі з того самого моменту.

Після етапу навчання модель оцінюється на тестовому наборі даних за допомогою методу *evaluate*. Потім вона застосовується до тестового набору даних, який включає як тестові дані, так і відповідні цільові значення. На основі цих даних обчислюються отримані втрати і середня абсолютна похибка (MAE) (рис. 3.10).

```
# Оцінка моделі
```

```
loss, mae = model.evaluate(X_test_scaled, testY)
print(f'Mean Absolute Error: {mae}')
```

Рис. 3.10. Оцінка моделі

Після навчання моделі можна візуалізувати зміну функції втрат з плином часу як для навчальних, так і для валідаційних наборів даних. Це дає змогу визначити ступінь навчання моделі та виявити будь-які

випадки перенавчання (рис. 3.11).

```
# Візуалізація кривих навчання
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

Рис. 3.11. Візуалізація результатів

3.4. Реалізація користувацького інтерфейсу

У рамках дисертаційної роботи було розроблено інтерфейс користувача (GUI) для полегшення взаємодії користувача з моделлю нейронної мережі та контролером SDN з метою прогнозування оптимального мережевого маршруту. Основні функції інтерфейсу користувача такі: по-перше, користувач може вибрати пріоритети серед показників маршрутизації (таких як пропускна здатність, втрата пакетів, час передачі та кількість хопів); по-друге, користувач може підключитися до контролера SDN, щоб отримати поточні характеристики шляху; і по-третє, користувач може розрахувати оптимальний шлях на основі цих характеристик за допомогою моделі нейронної мережі.

З огляду на важливість ефективної взаємодії між мережею та кінцевим користувачем, розроблений додаток використовує принципи модульного дизайну, де кожен компонент виконує свою специфічну роль у загальній архітектурі системи. Ключовими функціональними елементами додатку є менеджер сесій, який керує підключенням до контролера SDN, та GUI, що забезпечує інтуїтивно зрозумілий інтерфейс

для взаємодії користувача з системою. Цей підхід дає змогу забезпечити гнучкість у додаванні нових функцій, таких як вибір показників для визначення оптимального шляху, експорт журналів сеансів і керування поточними з'єднаннями.

Клас *PathOptimizerGUI* забезпечує зручний інтерфейс користувача, який дозволяє керувати сеансами SDN контролера та налаштовувати показники вибору шляху. Цей клас виконує роль основного компонента для створення графічного інтерфейсу користувача (GUI), що забезпечує взаємодію з SDN контролером, керує сесіями та підключеннями. Він відповідає за ініціалізацію всіх елементів інтерфейсу, їхнє налаштування та керування діями користувача.

```
class PathOptimizerGUI(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.session_manager = SessionManager()
        self.active_sessions_checkboxes = {}
```

Рис. 3.12. Ініціалізація класу PathOptimizerGUI

Клас починається з ініціалізації, де створюється екземпляр класу *SessionManager*. Цей екземпляр відповідає за керування сесіями між додатком і SDN контролером, зокрема їх створення, видалення та логування. Окрім цього, створюється словник *active_sessions_checkboxes*, який використовується для збереження стану активних сесій. Він допомагає користувачу відстежувати активні підключення під час керування сесіями за допомогою інтерфейсу чекбоксів (рис. 3.12).

Інтерфейс користувача поділений на дві основні частини: бокова панель та головне вікно. Бокова панель виконує роль навігаційного інструмента для переміщення між різними вкладками програми. Ці

вкладки відповідають за різні функції, такі як створення нових підключень до SDN контролера, перегляд активних сесій та отримання інформації про програму.

Для створення бокової панелі використовується метод *create_sidebar*, який генерує елемент інтерфейсу *CTkFrame* та кнопки. Кожна з кнопок відповідає за зміну вкладок або виклик певної функції для полегшення доступ до основних можливостей програми. Наприклад, натискання кнопки "New Connection" відкриває вкладку, яка дозволяє користувачу створити нове підключення до контролера, ввівши IP-адрес та порт (рис. 3.13).

```
def create_sidebar(self):
    self.sidebar_frame = ctk.CTkFrame(self, width=140, corner_radius=0)
    self.sidebar_frame.grid(row=0, column=0, rowspan=4,
sticky="nsew")
```

Рис. 3.13. Створення бокової панелі

У вкладці "New Connection" реалізована форма для введення параметрів підключення до контролера SDN. Поля для введення IP-адреси та порту створені за допомогою методу *create_labeled_entry*, який додає підписані текстові поля для забезпечення зрозумілого та зручного інтерфейсу користувача (рис. 3.14).

```
self.ip_entry = self.create_labeled_entry(new_connection_frame, "IP
Address:", 1, 0)
self.port_entry = self.create_labeled_entry(new_connection_frame,
"Port:", 2, 0)
```

Рис. 3.14. Конфігурація полів для введення IP-адреси та порту SDN контролера

Крім того, користувач має можливість вибрати показники для оптимізації маршрутів, а саме пропускну здатність, кількість втрачених пакетів, часу передачі та кількість хопів. Ці параметри представлені у вигляді випадаючих меню, кожне з яких налаштовується за допомогою масиву *optionmenu_vars* (рис. 3.15).

```
option_values = ['Bandwidth', 'Packets Lost', 'Transmission Delay', 'Hop
Count']

self.optionmenu_vars = [ctk.StringVar(value=val) for val in
option_values]
```

Рис. 3.15. Конфігурація вибору показників шляху

Керування активними сесіями здійснюється через вкладку "Active Sessions", де відображаються всі поточні підключення до контролера SDN. Для цього використовується компонент *CTkScrollableFrame*, який дозволяє відображати список сесій у вигляді чекбоксів, що дозволяє користувачу обирати ті сесії, з якими він хоче працювати. Крім того, цей компонент полегшує видалення сеансу або експортування логів. Його використання є зручним для відображення великої кількості активних підключень (рис. 3.16).

```
self.active_sessions_scrollable_frame =
ctk.CTkScrollableFrame(self.active_sessions_frame)

self.active_sessions_scrollable_frame.grid(row=0, column=0, padx=10,
pady=10, sticky="nsew")
```

Рис. 3.16. Конфігурація вкладки для керування активними сесіями

Для покращення взаємодії з користувачем додаток містить систему

діалогових вікон для відображення різноманітних повідомлень. Для прикладу, якщо користувач вводить помилкові дані під час додавання нового з'єднання (наприклад, неправильну IP-адресу), система генерує повідомлення про помилку за допомогою діалогового вікна. Це забезпечує своєчасний зворотний зв'язок і допомагає користувачу уникнути помилкових дій (рис. 3.17).

```
if not self.validate_ip(ip):
    messagebox.showerror("Error", "Invalid IP address.")
    return
```

Рис. 3.17. Налаштування діалогового вікна

Клас *SDNControllerSession* служить посередником між інтерфейсом користувача та контролером ONOS SDN. Він забезпечує стабільний зв'язок через REST API для обробки запитів на маршрутизацію в реальному часі. Цей клас автоматизує вибір оптимального маршруту між хостоми відправника та отримувача. Вибір відбувається на основі інформації про доступні маршрути та обраних користувачем пріоритетів характеристик.

Цей клас використовує нейронну мережу, яка завантажується на етапі ініціалізації. Модель попередньо навчена на основі обраних характеристик мережевого шляху та здатна прогнозувати оптимальний маршрут для передачі даних. Це дозволяє не тільки враховувати основні показники, але й адаптувати вибір відповідно до динамічних змін у мережі.

```
class SDNControllerSession:
    def __init__(self, ip, port, logger, priority_vars):
        self.ip = ip
```

```

self.port = port
self.logger = logger
self.priority_vars = priority_vars
self.paths_data = []
self.active = True
self.model = tf.keras.models.load_model('optimal_path_model.h5')

```

Рис. 3.18. Ініціалізація класу *SDNControllerSession*

У процесі створення екземпляра класу *SDNControllerSession*, задаються наступні параметри:

- *ip* — IP-адреса контролера.
- *port* — порт, на якому контролер слухає запити.
- *logger* — об'єкт класу *Logger*, який відповідає за логування всіх запитів і відповідей.
- *priority_vars* — змінні, що визначають показники вибору оптимального шляху.

Також ініціалізується список *paths_data*, який містить інформацію про шляхи, та змінна *active*, яка визначає стан сеансу.

Модель нейронної мережі завантажується з файлу *optimal_path_model.h5* у змінну *model* (рис. 3.18).

Підключення до контролера SDN встановлюється за допомогою метода *connect*, який виконує HTTP-запит до контролера, щоб отримати дані щодо доступних маршрутів. У разі успішного підключення відповідь від контролера обробляється та зберігається. Потім дані трансформуються у формат JSON, що полегшує подальшу обробку. Далі процес підключення логується шляхом виклику методу *log_request* об'єкта *Logger*. Успішне підключення також ініціює окремий потік, який відповідає за постійне отримання оновленої інформації про маршрут від контролера в режимі реального часу.

```

def connect(self):
    try:
        url = f'http://{self.ip}:{self.port}/onos/v1/route/getPaths'
        response = requests.get(url)
        response.raise_for_status()
        self.paths_data = response.json()
        self.logger.log_request(self.ip, "Initial Request", self.paths_data)
        threading.Thread(target=self.continuous_requesting,
daemon=True).start()
        return True
    except requests.RequestException as e:
        self.logger.log_request(self.ip, "Connection Error", str(e))
        return False

```

Рис. 3.19. Модуль підключення до SDN контролера

З'єднання встановлюється за допомогою методу *requests.get*, який передає HTTP-запит до контролера за вказаною IP-адресою та портом. У разі успішного запиту дані шляху зберігаються в змінній *paths_data*, а відповідь логується за допомогою методу *log_request*.

У випадку виникнення помилки підключення (наприклад, недоступний сервер або мережеві проблеми), відповідне повідомлення також записується до журналу (рис. 3.19).

Метод *continuous_requesting* відповідає за моніторинг контролера SDN з метою забезпечення доступності оновленої інформації щодо маршрутів. Він працює у фоновому режимі, забезпечуючи безперервний потік даних від контролера. У разі виникнення помилки під час запиту до контролера, наприклад, внаслідок збою мережі або неправильної відповіді, відповідна інформація належним чином логується. Отже,

ведення логів дозволяє відстежувати всі важливі події у роботі системи (рис. 3.20).

```
def continuous_requesting(self):
    while self.active:
        try:
            url = f'http://{self.ip}:{self.port}/onos/v1/route/getPaths'
            response = requests.get(url)
            response.raise_for_status()
            self.paths_data = response.json()
            self.logger.log_request(self.ip, "Path Request", self.paths_data)

            # Розраховуємо оптимальний шлях
            optimal_path = self.predict_optimal_path(self.paths_data)
            self.logger.log_request(self.ip, "Optimal Path Sent",
optimal_path)

            post_url =
f'http://{self.ip}:{self.port}/onos/v1/route/setOptimalPath'
            requests.post(post_url, json={"pathId": optimal_path['pathId']})
        except requests.RequestException as e:
            self.logger.log_request(self.ip, "Error", str(e))

        time.sleep(1)
```

Рис. 3.20. Отримання даних про характеристики шляхів від SDN контролера для подальшого вибору шляху

Для визначення оптимального шляху використовується метод

predict_optimal_path. Дані про всі доступні маршрути перетворюються у формат, сумісний з нейромережевою моделлю. На початковому етапі збираються дані про різні параметри, включаючи пропускну здатність, кількість втрачених пакетів, часу передачі та кількість хопів для кожного маршруту. Потім дані впорядковуються відповідно до визначених користувачем пріоритетів для враховування відносного пріоритету кожного показнику (рис. 3.21).

```
def predict_optimal_path(self, paths_data):
    input_data = np.array([[path['Bandwidth'], path['Packets.Lost'],
path['Transmission.Delay'], path['Hop.Count']]
        for path in paths_data])
    selected_priorities = [self.priority_vars[i].get() for i in range(4)]
    priority_mapping = {
        'Bandwidth': 0,
        'Packets Lost': 1,
        'Transmission Delay': 2,
        'Hop Count': 3
    }

    reordered_input_data = np.array([
        [path[priority_mapping[selected_priorities[0]]],
        path[priority_mapping[selected_priorities[1]]],
        path[priority_mapping[selected_priorities[2]]],
        path[priority_mapping[selected_priorities[3]]]]
        for path in input_data
    ])
```

```

predictions = self.model.predict(reordered_input_data)
optimal_index = np.argmin(predictions)
return paths_data[optimal_index]

```

Рис. 3.21. Модуль визначення оптимального шляху

Після визначення оптимального маршруту необхідна інформація передається назад на контролер. Це досягається завдяки використанню HTTP POST-запиту, який полегшує передачу структури маршруту на контролер для його використання у пересиланні нового потоку даних. Крім того, процес логується, що дозволяє відстежувати вибір маршруту на кожному етапі.

Для завершення сеансу і припинення пов'язаної з ним діяльності передбачений метод *disconnect*. Завершення сеансу запобігає безперервному отриманню даних, а сама ця подія логується. Цей елемент гарантує належне завершення зв'язку з контролером і запобігає помилковій обробці даних у разі завершення сесії (рис. 3.22).

```

def disconnect(self):
    self.active = False
    self.logger.log_request(self.ip, "Disconnected", "Session
disconnected")

```

Рис. 3.22. Завершення сесії

Клас *SessionManager* був розроблений з метою надання централізованого рішення для керування сеансами з різними контролерами SDN. Це дозволяє користувачам використовувати певний набір функцій, зокрема додавати, видаляти та експортувати логи. Основна мета класу полягає в тому, щоб інкапсулювати механізм керування підключеннями до контролерів, забезпечуючи надійне

керування сеансами та їх логами, одночасно надаючи користувачеві інструменти для зручного керування мережею.

```
class SessionManager:
```

```
    def __init__(self):
```

```
        self.sessions = {}
```

```
        self.loggers = {}
```

Рис. 3.23. Завершення сесії

Після створення об'єкта класу *SessionManager* ініціалізуються дві порожні структури даних: *sessions* та *loggers*. Словник *sessions* використовується для зберігання активних сеансів підключень до контролерів SDN. Ключем є рядок у форматі *ip:port*, який служить для унікальної ідентифікації кожного сеансу. Другий словник, *loggers*, зберігає об'єкти класу *Logger*, кожен з яких відповідає за логування конкретного сеансу. Це дає змогу відстежувати всі запити, відповіді та інші дії, пов'язані зі встановленням з'єднання з певним контролером (рис. 3.23).

Основною функцією класу є можливість додавати нові сесії. Це реалізується через метод *add_session*, який виконує кілька важливих перевірок і дій. Спочатку метод перевіряє, чи існує вже сесія з вказаною IP-адресою та портом. Якщо таке підключення вже активне, нова сесія не створюється, і метод повертає *False*. Це дозволяє уникнути дублювання з'єднань. Якщо ж сесія відсутня, створюється новий журнал (об'єкт класу *Logger*), який зберігатиме логи для нової сесії (рис. 3.24).

```
def add_session(self, ip, port, priority_vars):
```

```
    if f"{ip}:{port}" in self.sessions:
```

```
        return False
```

```

logger = Logger(f"{ip}:{port}")
session = SDNControllerSession(ip, port, logger, priority_vars)
if session.connect():
    self.sessions[f"{ip}:{port}"] = session
    self.loggers[f"{ip}:{port}"] = logger
    return True
return False

```

Рис. 3.24. Завершення сесії

Наступним кроком є створення нового об'єкта класу *SDNControllerSession*, який інкапсулює всі деталі підключення до контролера. Об'єкт сесії отримує IP-адресу, порт, логер та пріоритети для пошуку оптимального шляху в мережі (ці пріоритети передаються через змінну *priority_vars*). Після цього метод *connect* намагається встановити з'єднання з контролером. Якщо підключення проходить успішно, нова сесія додається до словника *sessions*, а відповідний логер — до словника *loggers*. Якщо сесія додана успішно, метод повертає *True*, в іншому випадку — *False*.

Крім додавання сесій, клас *SessionManager* також дозволяє видаляти активні з'єднання через метод *remove_session*. Для цього спочатку перевіряється, чи існує сесія з вказаним ключем (*session_key*, у форматі *"ip:port"*). Якщо така сесія існує, викликається метод *disconnect*, що відповідає за коректне завершення підключення. Після цього сесія видаляється з обох словників (*sessions* і *loggers*), що означає, що більше немає активного підключення до цього контролера, а лог також більше не ведеться (рис. 3.25).

```

def remove_session(self, session_key):
    if session_key in self.sessions:

```

```

self.sessions[session_key].disconnect()
del self.sessions[session_key]
del self.loggers[session_key]

```

Рис. 3.25. Завершення сесії

Однією з реалізованих функцій класу є можливість експортувати логи взаємодії з контролером у текстовий файл. Ця функція доступна через метод *export_logs*, який дозволяє користувачеві зберігати останні N записів із зазначеного журналу сеансу. Перед експортом перевіряється, чи існують логи для відповідного сеансу. У випадку, якщо логи вже існують, викликається метод *export_logs* і останні N записів зберігаються у файл за вказаним шляхом (рис. 3.26).

```

def export_logs(self, session_key, file_path, last_n=100):
    if session_key in self.loggers:
        self.loggers[session_key].export_logs(file_path, last_n)

```

Рис. 3.26. Завершення сесії

Клас *Logger* відповідає за запис та зберігання логів активності кожного окремого сеансу взаємодії з контролером SDN. Основною функцією є запис даних, що стосуються всіх запитів, переданих до контролера, та відповідей, отриманих у відповідь. Кожен лог містить інформацію про час запиту, сам запит і відповідь на нього. Це дозволяє відстежувати дії та стан мережі в кожен конкретний момент часу. Ще однією ключовою особливістю класу є можливість контролювати кількість записів, що дозволяє уникнути перевантаження системи та надмірного використання пам'яті.

```
class Logger:
    def __init__(self, session_key):
        self.session_key = session_key
        self.logs = []
```

Рис. 3.27. Завершення сесії

Після створення об'єкта класу *Logger* викликається конструктор, який приймає як вхідні дані ідентифікатор сеансу *session_key*. Він використовується з метою ідентифікації відповідної діяльності, яка пов'язана з певним підключенням до контролера SDN. Логи зберігаються у вигляді списку, який спочатку встановлений у порожній стан. Список служить для зберігання всіх записів запитів і відповідей від контролера (рис. 3.27).

Одним із основних методів класу є *log_request*. Він зберігає кожен запит, який був переданий контролеру, разом із відповідною відповіддю. Крім того, кожен запис супроводжується міткою часу, яка точно фіксує момент взаємодії. Це досягається за допомогою функції *datetime.now()*, яка повертає поточний час, і методу *strftime('%Y-%m-%d %H:%M:%S')*, який форматує час як дата й час із точністю до секунд. Це дозволяє точно визначити момент часу, коли було ініційовано запит, що має важливе значення для перевірки роботи мережі та виявлення потенційних або існуючих проблем (рис. 3.28).

```
def log_request(self, session_key, request, response):
    timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    if len(self.logs) >= 100:
        self.logs.pop(0)
    self.logs.append(f"{timestamp} - {request}: {response}")
```

Рис. 3.28. Завершення сесії

Для уникнення накопичення надмірної кількості записів і надмірного використання пам'яті у системі, клас обмежує кількість збережених логів. Це реалізується шляхом обмеження списку логів до 100 записів. Коли кількість записів у списку перевищує цей поріг, найстаріші з них автоматично видаляються за допомогою методу *pop(0)*. Це забезпечує збереження лише актуальних даних, що стосуються останніх запитів до контролера, і дозволяє запобігти ситуації, коли журнал логів переповнюється.

Крім запису логів у пам'ять, клас *Logger* також дозволяє експортувати їх у вигляді текстового файлу. Це робиться за допомогою методу *export_logs*, який дозволяє зберігати останні записи журналу в окремий файл. За замовчуванням експорт включає останні 100 записів, однак користувач може змінити це значення, вказавши іншу кількість логів для збереження. Функція екпортує останні записи за допомогою стандартної операції запису у файл. Спочатку потрібні записи об'єднуються в один текстовий рядок шляхом з'єднання елементів списку за допомогою методу *join*. Після цього файл відкривається у режимі запису, і логи зберігаються у ньому. Файл автоматично закривається після завершення операції завдяки використанню конструкції *with*, яка забезпечує безпечне керування файлами і запобігає витоку ресурсів (рис. 3.29).

```
def export_logs(self, file_path, last_n = 100):
    logs_to_export = ".join(self.logs[-last_n:])
    with open(file_path, 'w') as file:
        file.write(logs_to_export)
```

Рис. 3.29. Завершення сесії

3.5. Модифікація SDN контролера ONOS

Для інтеграції ONOS з інтерфейсом користувача, який використовується для вибору оптимальні маршрути передачі даних у SDN мережі, створено додаток ONOS під назвою LBBNN Application. Він використовує REST API для взаємодії з компонентами маршрутизації ONOS, надаючи доступ до можливих маршрутів між хостами та вибираючи оптимальний шлях для передачі даних.

Розробка передбачає створення REST API з двома основними кінцевими точками: */getPaths* для отримання списку доступних маршрутів між вказаними хостами та */setOptimalPath* для встановлення оптимального шляху на основі отриманих даних.

Розробка починається зі створення проекту додатка ONOS, який буде реалізовувати зазначений функціонал. Команда *onos-create-app* створює структуру проекту та необхідні конфігураційні файли. Ім'я проекту та його унікальний ідентифікатор (*org.onosproject.lbbnn*) визначаються під час ініціалізації для зручного відстеження та керування (рис. 3.30).

```
onos-create-app org.onosproject.lbbnn lbbnn 1.0.0
```

Рис. 3.30. Створення структури проекту

Клас *LbbnnResource* відповідає за обробку всіх HTTP-запитів, спрямованих до REST API. Клас містить методи для пошуку та модифікації маршрутів. Зокрема, метод *getPaths* використовується для пошуку всіх потенційних маршрутів між двома вказаними хостами, використовуючи їхні відповідні ідентифікатори (*srcHostId* та *dstHostId*).

У класі також ініціалізуються служби, необхідні для роботи з мережею, зокрема *HostService*, *PathService* та *PacketService*. Кожна з цих служб відіграє важливу роль у зборі та обробці інформації про мережу.

HostService відповідає за керування хостами в мережі, *PathService* надає функціонал для отримання інформації про маршрути між хостами, а *PacketService* дозволяє обробляти пакети даних, які проходять через мережу (рис. 3.31).

```
@Path("route")

public class LbbnnResource extends AbstractWebResource {
    private final Logger log = LoggerFactory.getLogger(getClass());

    private final HostService hostService = get(HostService.class);
    private final PathService pathService = get(PathService.class);
    private final PacketService packetService = get(PacketService.class);
    private final PacketProcessor packetProcessor = new
InternalPacketProcessor();

    public LbbnnResource() {
        packetService.addProcessor(packetProcessor,
PacketProcessor.director(2));
    }
```

Рис. 3.31. Створення основного класу

Розглянутий клас містить внутрішній клас *InternalPacketProcessor*, який реалізує інтерфейс *PacketProcessor*. Він використовується для обробки нових пакетів, які надходять у мережу. Після отримання нового пакету метод *process* цього класу витягує MAC-адреси з Ethernet-фрейму та встановлює значення для змінних *currentSrc* і *currentDst*, які представляють джерело і адресат пакету відповідно. Це дозволяє здійснювати подальшу обробку пакетів (рис. 3.32).

```

private class InternalPacketProcessor implements PacketProcessor {
    static HostId currentSrc;
    static HostId currentDst;

    @Override
    public void process(PacketContext context) {
        if (context.isHandled()) return;

        Ethernet ethernetFrame = context.inPacket().parsed();
        if (ethernetFrame == null) return;

        currentSrc = HostId.hostId(ethernetFrame.getSourceMAC());
        currentDst = HostId.hostId(ethernetFrame.getDestinationMAC());

        log.info("New packet processed - Source: {}, Destination: {}",
currentSrc, currentDst);
    }
}

```

Рис. 3.32. Створення основного класу

Метод `getPaths`, реалізований у вищезазначеному класі, розроблено спеціально для обробки запитів GET для маршрутів між хостами. Він перевіряє, чи є *src* та *dst* ідентифікаторами хоста. Після цього викликається метод `pathService.getPaths` для отримання списку доступних маршрутів. Для кожного маршруту метод обчислює наступні характеристики: пропускна здатність, кількість втрачених пакетів, час передачі та кількість хопів. Ці значення повертаються у форматі JSON, що дозволяє користувачеві або клієнтській програмі отримувати

структуровану інформацію про маршрути (рис. 3.33).

```

@GET
@Path("/getPaths")
@Produces(MediaType.APPLICATION_JSON)
public Response getPaths() {
    try {
        // src та dst, збережені при обробці пакету
        HostId src = InternalPacketProcessor.currentSrc;
        HostId dst = InternalPacketProcessor.currentDst;

        if (src == null || dst == null) {
            log.warn("Source or Destination HostId is null.");
            return
                Response.status(Response.Status.BAD_REQUEST).entity("Source or
                Destination HostId is null.").build();
        }

        List<Path> paths = pathService.getPaths(
            hostService.getHost(src).location(),
            hostService.getHost(dst).location());

        List<Map<String, Object>> pathsData = paths.stream().map(path -
    > {
        Map<String, Object> pathData = new HashMap<>();
        pathData.put("pathId", path.id());
        pathData.put("Bandwidth", calculateBandwidth(path));
        pathData.put("PacketsLost", calculatePacketsLost(path));
    }

```

```

        pathData.put("TransmissionDelay",
calculateTransmissionDelay(path));
        pathData.put("HopCount", path.links().size());
        return pathData;
    }).collect(Collectors.toList());

    log.info("Paths with characteristics between {} and {}: {}", src, dst,
pathsData);

    return Response.ok(pathsData).build();
} catch (Exception e) {
    log.error("Failed to retrieve paths", e);
    return
Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
}
}

```

Рис. 3.33. Метод *getPaths*

Розрахунок характеристик маршруту реалізовано за допомогою використання приватних методів *calculateBandwidth*, *calculatePacketsLost*, *calculateTransmissionDelay* та *calculateHopCount*. Метод *calculateBandwidth* використовується для визначення коефіцієнту пропускної здатності відповідного маршруту за формулою 2.1. Подібним чином, методи *calculatePacketsLost* і *calculateTransmissionDelay* використовують формули, розглянуті у 2 розділі (формули 2.3 та 2.4), щоб обчислити кількість втрачених пакетів і часу передачі, використовуючи зібрані дані про стан мережі.

Метод *calculateBandwidth* відповідає за обрахунок коефіцієнта пропускної здатності маршруту. Основною метою цього методу є

визначення кількості даних, які було передано через усі складові ланки вказаного маршруту. Метод приймає аргумент типу *Path*, який представляє собою маршрут, що складається з декількох ланок. Внутрішня логіка методу передбачає ітерацію по всіх каналах маршруту, при цьому кількість байтів, переданих по кожному каналу, отримується за допомогою виклику методу *getBytesTransferred(link)*. Після цього метод отримує статистику для кожного каналу і обчислює загальну кількість переданих байтів з розрахунку на біт в секунду. Потім загальна кількість байтів ділиться на максимальну пропускну здатність, таким чином отримуючи нормалізоване значення пропускну здатності в діапазоні від 0 до 1 (рис. 3.34).

```
private double calculateBandwidth(Path path) {
    double totalBytes = 0.0;
    double maxBandwidth = path.maxBandwidth();

    for (Link link : path.links()) {
        double bytes = getBytesTransferred(link);
        totalBytes += bytes;
    }
    return totalBytes / maxBandwidth;
}

private double getBytesTransferred(Link link) {
    DeviceId srcDeviceId = link.src().deviceId();
    DeviceId dstDeviceId = link.dst().deviceId();

    Load load = statisticService.load(link);
    double bytesTransferred = load.rate();
}
```

```

        log.debug("Bytes transferred on link {}-{}: {}", srcDeviceId,
dstDeviceId, bytesTransferred);
        return bytesTransferred / 8.0;
    }

```

Рис. 3.34. Метод *calculateBandwidth*

Метод *calculatePacketsLost* використовується для обчислення швидкості втрати пакетів на заданому маршруті. Метод приймає аргумент типу *Path* і, як і у випадку з попереднім методом, виконує обхід всіх ланок маршруту. Для кожної ланки обчислюється кількість пакетів, що пройшли в прямому і зворотному напрямках, шляхом виклику методів *getForwardPackets(link)* та *getBackwardPackets(link)* відповідно. Кількість втрачених пакетів обчислюється відповідно до формули 2.2. Це дозволяє користувачеві отримати показник, який відображає відсоток втрат пакетів. Важливою особливістю методу є можливість представлення показника в діапазоні від 0 до 1, де 0 означає відсутність втрат пакетів, а 1 - повну втрату всіх пакетів (рис. 3.35).

```

private double calculatePacketsLost(Path path) {
    double totalForwardPackets = 0.0;
    double totalBackwardPackets = 0.0;
    for (Link link : path.links()) {
        totalForwardPackets += getForwardPackets(link);
        totalBackwardPackets += getBackwardPackets(link);
    }

    return (totalForwardPackets - totalBackwardPackets) /
totalForwardPackets;
}

```

```

private double getForwardPackets(Link link) {
    DeviceId srcDeviceId = link.src().deviceId();

    return flowRuleService.getFlowEntries(srcDeviceId).stream()
        .mapToDouble(FlowEntry::packets) // Загальна кількість
пакетів вперед
        .sum();
}

private double getBackwardPackets(Link link) {
    DeviceId dstDeviceId = link.dst().deviceId();

    return flowRuleService.getFlowEntries(dstDeviceId).stream()
        .mapToDouble(FlowEntry::packets).sum();
}

```

Рис. 3.35. Метод *calculatePacketsLost*

Метод *calculateTransmissionDelay* використовується для визначення загального часу передачі даних через маршрут. Цей показник має важливе значення для оцінки загальної продуктивності мережі, враховуючи, що час передачі може мати значний вплив на якість передачі даних. Як і в попередніх випадках, метод приймає аргумент типу *Path* і виконує ітерацію по каналам маршруту. Для кожного каналу розраховується загальна тривалість передачі даних за допомогою методу *getFlowDuration(link)*, а також середній міжпакетний час (IAT) за допомогою методу *getFlowIATMean(link)*. Загальний час передачі розраховується додаванням загальної тривалості та середньої затримки,

що дає точну оцінку часу передачі по всьому маршруту. У результаті отримаємо показник, який відображає час, необхідний для доставки даних від джерела до місця призначення (рис. 3.36).

```
private double calculateTransmissionDelay(Path path) {
    double totalDuration = 0.0;
    double meanIAT = 0.0;

    for (Link link : path.links()) {
        totalDuration += getFlowDuration(link);
        meanIAT += getFlowIATMean(link);
    }

    return totalDuration + meanIAT;
}

private double getFlowDuration(Link link) {
    DeviceId deviceId = link.src().deviceId();

    return flowRuleService.getFlowEntries(deviceId).stream()
        .mapToDouble(FlowEntry::life)
        .average().orElse(0.0);
}

private double getFlowIATMean(Link link) {
    DeviceId deviceId = link.src().deviceId();

    return flowRuleService.getFlowEntries(deviceId).stream()
```



```

        .mapToDouble(flow -> {
            double packets = flow.packets();
            double duration = flow.life();
            return packets > 0 ? (duration / packets) : 0.0;
        }).average().orElse(0.0);
    }

```

Рис. 3.36. Метод *calculateTransmissionDelay*

Метод *setOptimalPath*, у свою чергу, приймає JSON-запит з ID оптимального маршруту, отриманого з інтерфейсу користувача, та використовує внутрішній метод *applyOptimalPath* для його встановлення. Спочатку цей метод знаходить обраний маршрут на основі ID, а потім застосовує його у мережі через ONOS API (рис. 3.37).

```

@POST
@Path("/setOptimalPath")
@Consumes(MediaType.APPLICATION_JSON)
public Response setOptimalPath(OptimalPathRequest request) {
    try {
        Path optimalPath = findPathById(request.getPathId());
        if (optimalPath != null) {
            applyOptimalPath(optimalPath);
            log.info("Optimal path {} applied.", request.getPathId());
            return Response.ok().build();
        } else {
            log.warn("Optimal path not found: {}", request.getPathId());
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }
}

```

```

    } catch (Exception e) {
        log.error("Failed to apply optimal path", e);
        return
    }
    Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
}
}

```

Рис. 3.37. Метод *setOptimalPath*

Клас *OptimalPathRequest* реалізовано для обробки даних, отриманих у форматі JSON при побудові оптимального маршруту. Єдиним атрибутом цього класу є *pathId*, який слугує для ідентифікації обраного маршруту. Конструктори та методи *getPathId* та *setPathId* полегшують отримання та присвоєння значення параметру, тим самим гарантуючи безперебійну передачу даних у форматі JSON між інтерфейсом користувача та додатком ONOS (рис. 3.38).

```

package org.onosproject.lbbnn;

public class OptimalPathRequest {
    private String pathId;

    public String getPathId() {
        return pathId;
    }

    public void setPathId(String pathId) {
        this.pathId = pathId;
    }
}

```

Рис. 3.38. Клас обробки даних

В головному класі додатка, *LbbnnApp*, виконано налаштування компонентів для реєстрації *LbbnnResource* як REST API. Використовуючи методи життєвого циклу *activate* та *deactivate*, забезпечується початкове ініціалізування додатка та коректне відключення його функціональності при зупинці. *LbbnnApp* реєструється в ONOS за допомогою *coreService.registerApplication*, що забезпечує йому унікальний ID і дозволяє обробляти запити від інших модулів ONOS, а також від зовнішніх клієнтів через REST API (рис. 3.39).

```
@Component(immediate = true)
@Service
public class LbbnnApp extends AbstractWebApplication {
    private final Logger log = LoggerFactory.getLogger(getClass());
    private ApplicationId appld;
    @Override
    protected Set<Class<?>> getClasses() {
        return getClasses(LbbnnResource.class);
    }
    protected void activate() {
        appld = coreService.registerApplication("org.onosproject.lbbnn");
        log.info("Started LBBNN Application with ID {}", appld.id());
    }
    protected void deactivate() {
        log.info("Stopped LBBNN Application");
    }
}
```

Рис. 3.39. Налаштування компонентів для реєстрації *LbbnnResource* як REST API

Щоб забезпечити оптимальне функціонування програми, важливо включити залежності у файл *pom.xml* з метою полегшення взаємодії з *HostService*, *PathService* та іншими службами ONOS. Це забезпечить повну інтеграцію з основними компонентами мережі, включаючи хости та топологію.

Після налаштування необхідних залежностей проект компілюється та збирається за допомогою команди *mvn clean install*, яка генерує необхідні артефакти для встановлення програми ONOS у формі пакета *.oar*. Далі програма встановлюється та активується в ONOS за допомогою *onos-app localhost install!*, що дозволяє програмі взаємодіяти з існуючими службами ONOS.

Висновки до розділу 3

В даному розділі було розроблено модель нейронної мережі для прогнозування оптимального шляху передачі пакетів у програмно-конфігурованій мережі. Було досягнуто ряд важливих етапів та результатів:

1. Зібрано і попередньо оброблено реальні мережеві дані, які містять інформацію про різні параметри мережевих потоків, такі як пропускна здатність, втрата пакетів, час передачі пакетів та кількість переходів;

2. На основі наявних даних розраховано основні ознаки, які впливають на якість передачі даних у мережі. Пропускна здатність, втрата пакетів, загальний час передачі та кількість хопів були використані як ключові змінні;

3. Дані були розділені на навчальну та тестову вибірки у співвідношенні 80:20. Для кожної вибірки були нормалізовані значення ознак, що забезпечило їх масштабування та підвищення ефективності навчання моделі;

4. Розроблено модель нейронної мережі зі структурою, що включає вхідний рівень, два прихованих шари з функцією активації ReLU та вихідний рівень з лінійною функцією активації. Додатково, для запобігання перенавчанню, використовувалися Dropout шари.

РОЗДІЛ 4

ТЕСТУВАННЯ ТА АНАЛІЗ МЕТОДУ КОНСТРУЮВАННЯ ТРАФІКУ У ПРОГРАМНО-КОНФІГУРОВАНИХ МЕРЕЖАХ НА ОСНОВІ ШТУЧНОГО ІНТЕЛЕКТУ

У даному розділі дисертації проводиться аналіз тестування методу конструювання трафіку у програмно-конфігурованих мережах, заснованого на штучному інтелекті. Проводиться оцінка ефективності впровадження штучного інтелекту в контексті оптимізації трафіку та керування мережевими ресурсами. Аналіз засновується на результатах експериментальних досліджень та симуляцій, проведених з метою визначення впливу запропонованого методу на загальну продуктивність мережі.

4.1. Розробка тестового середовища

Для того, щоб оцінити ефективність розробленої нейронної мережі на системі, яка наближена до реальних умов експлуатації, було використано інструмент моделювання мережі Mininet.

Щоб гарантувати оптимальне функціонування тестового середовища в Mininet, важливо налаштувати як топологію мережі, так і параметри контролера SDN. Топологія мережі побудована за допомогою мови програмування Python, що дозволяє конфігурувати компоненти мережі та їх взаємодію.

Тестовий набір даних генерується на основі створеної топології, як показано на малюнку 4.1. У цій топології хости випадково передають мережевий трафік один одному. Такий підхід забезпечує моделювання мережевого трафіку, наближеного до реальних умов, дозволяючи імітувати різноманітні сценарії навантаження. Генерація випадкового трафіку забезпечує динамічність мережевого навантаження, таким

чином наближаючи умови, які б виникли під час роботи мережі в реальному середовищі.

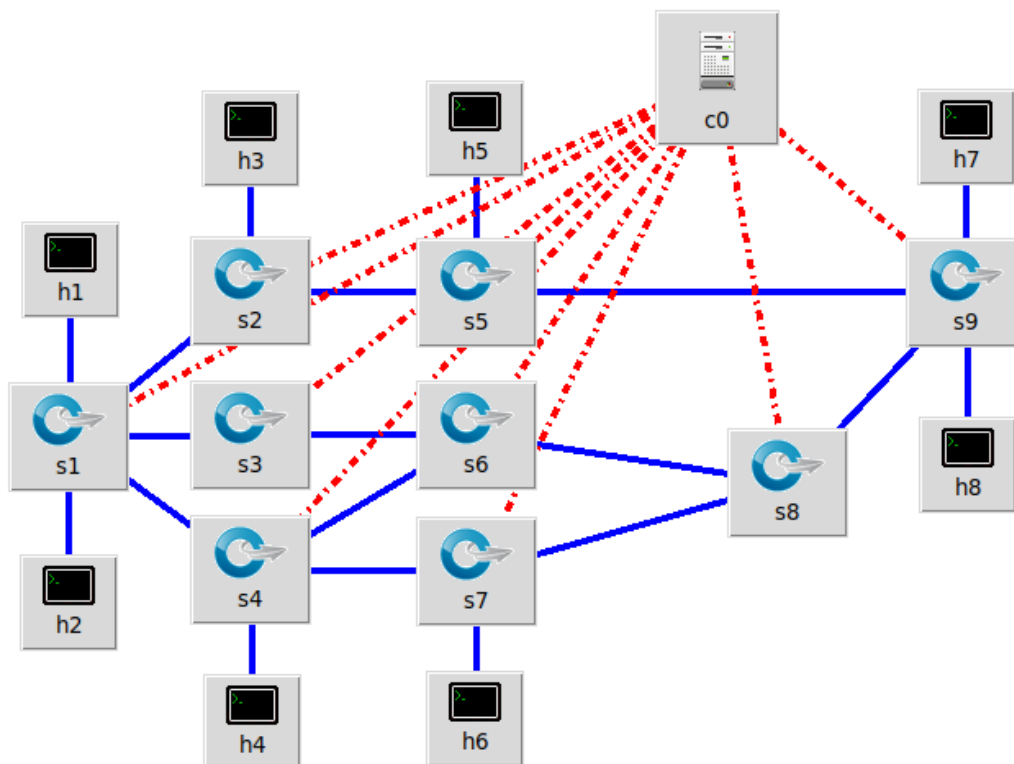


Рис. 4.1. Слабко зв'язана топологія

Контролер SDN, у свою чергу, використовує протокол OpenFlow для визначення характеристик мережевого навантаження. Протокол OpenFlow дозволяє контролеру відстежувати навантаження на мережеві шляхи, що є важливим кроком у процесі аналізу та оптимізації маршрутів передачі трафіку даних. Це дозволяє аналізувати потоки даних і визначати продуктивність мережі в умовах змінного навантаження.

Для створення тестового набору даних для тестування нейронних мереж, характеристики навантаження шляхів збирались протягом 180 секунд. Цей період часу дозволив зібрати достатню кількість даних, що охоплюють діапазон умов навантаження, які можна спостерігати в

реальних мережах. Набір даних містить ряд ключових параметрів, включаючи час передачі, пропускну здатність і втрату пакетів. Дані були використані для оцінки ефективності нейронної мережі для балансування навантаження.

Розглянемо роботу запропонованого методу на повнозв'язному графі, представленому на рисунку 4.2, та графі де Бруйна (рис. 4.3), який розглядався в роботах [70-72].

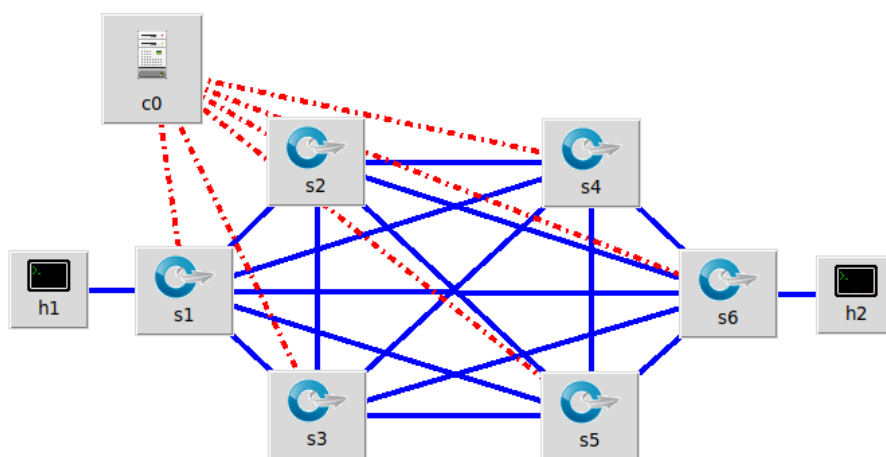


Рис. 4.2. Повнозв'язна топологія

Генерація тестового набору для топологій, зображених на рис. 4.2-4.3 відбувалась аналогічно до топології 4.1. Характеристики завантаження шляхів збиралися протягом 180 секунд та включають ключові характеристики мережі.

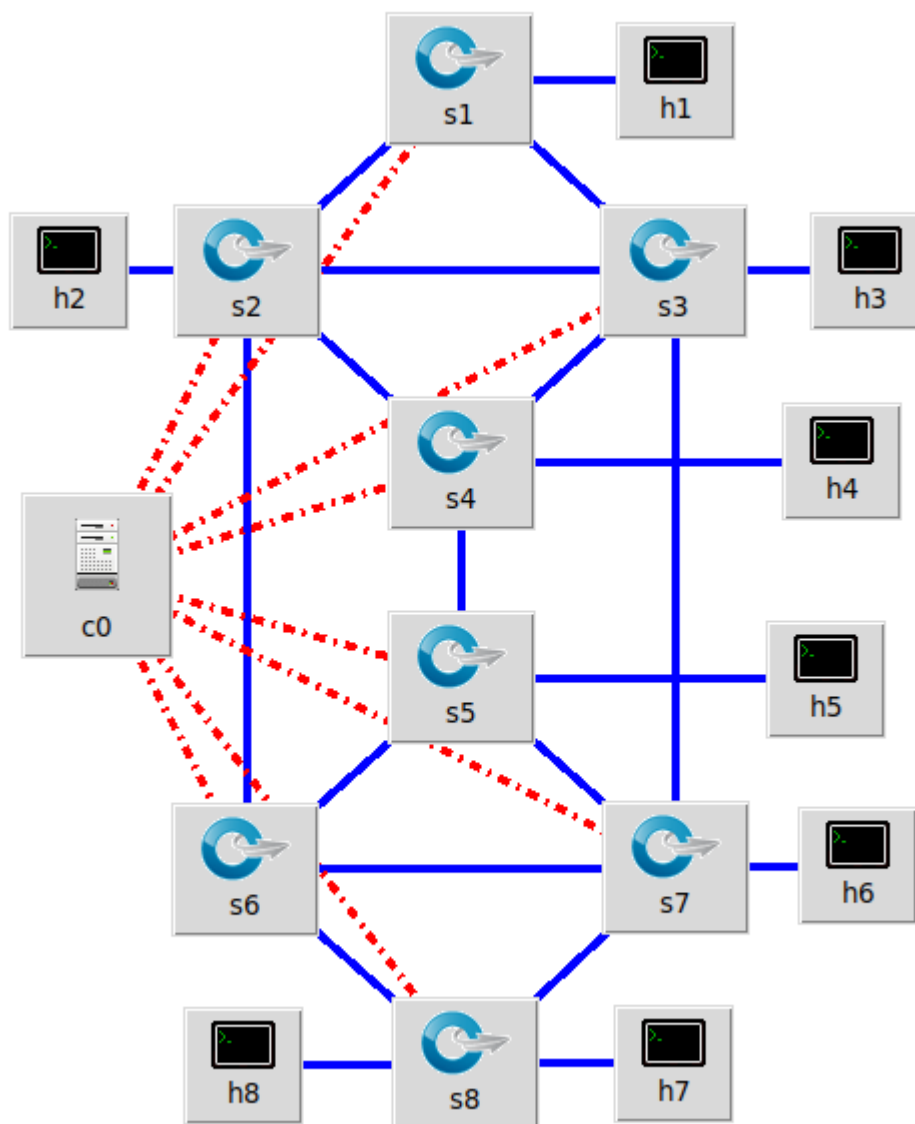


Рис. 4.3. Граф де Бруйна

На основі зібраних даних для обраних топологій (рис. 4.1 - 4.3) проводиться аналіз ефективності розробленого методу балансування навантаження у порівнянні з іншими методами, а також залежність ефективності конструювання трафіку від обраної топології.

4.2. Аналіз результатів тестування

Для оцінки ефективності різних конфігурацій нейронної мережі, її

навчання проводилося з різною кількістю нейронів у прихованому шарі (3, 5, 7, 9 та 11). Їх було обрано для пошуку оптимальної конфігурації для задач балансування навантаження.

Рисунок 4.4 ілюструє середню абсолютну похибку для п'яти нейронних мереж з різною кількістю нейронів у прихованому шарі. Як можна побачити, коли кількість нейронів у прихованому шарі занадто мала (а саме 3), похибка навченої нейронної мережі є відносно найбільшою. При збільшенні кількості нейронів у прихованому шарі зменшується і похибка навченої нейронної мережі. За результатами проведеного експерименту, найбільшу відносну похибку демонструють нейронні мережі з найменшою (3) та найбільшою (11) кількістю нейронів у прихованому шарі. Для 7 нейронів у прихованому шарі, значення похибки є відносно найменшим та залишається стабільним.

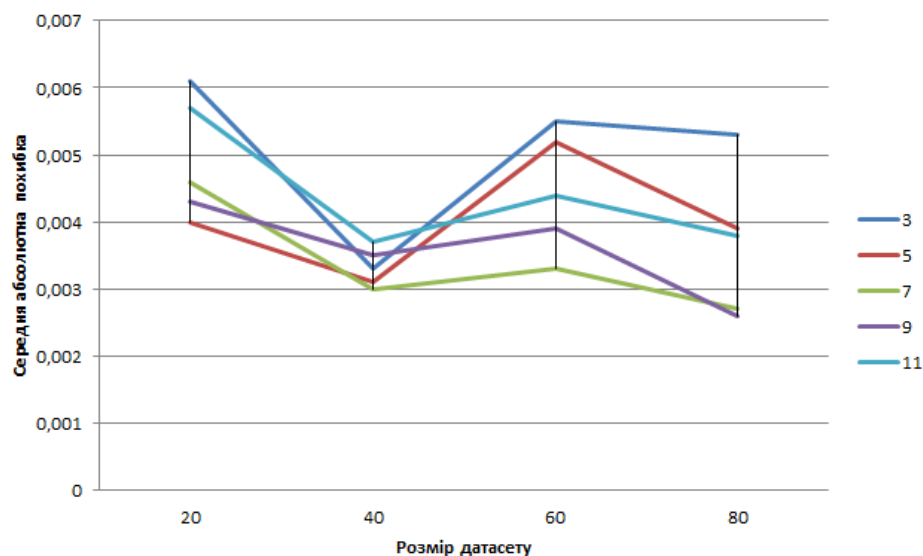


Рис. 4.4. Середня абсолютна похибка для різних нейронних мереж

Враховуючи результати експерименту, було обрано нейронну мережу зворотного поширення з 7 нейронами в прихованому шарі. Ця конфігурація є найбільш оптимальною для балансування навантаження в мережі SDN. Це пояснюється тим, що нейронна мережа з 7 нейронами в

прихованому шарі забезпечує найменшу та стабільну середню абсолютну похибку, що є важливим пунктом для точного прогнозування.

Для оцінки ефективності розробленої стратегії балансування навантаження на основі нейронної мережі, було використано для порівняння три інші стратегії балансування навантаження. Це метод динамічного балансування навантаження (DLB), запропонований у роботі [10], модифікований ECMP [65] та стратегія статичного балансування навантаження Round Robin.

Протягом перших 90 секунд експерименту всі хости в тестовій топології, за винятком h1, випадково передавали дані один одному, таким чином створюючи трафік по кількох шляхах у мережі. Це дозволило імітувати реалістичне навантаження на мережу та перевірити здатність кожної стратегії ефективно керувати трафіком. Далі h1 передавав трафік на h8 для імітації нового вхідного потоку даних. Це було зроблено для оцінки реакції мережі на раптове збільшення потоку даних.

Для полегшення роботи з розробленим методом було реалізовано інтерфейс користувача. Він призначений для зручної взаємодії з нейронною мережею, надаючи користувачу доступ до функцій налаштування параметрів нових підключень та відображенню активних сесій.

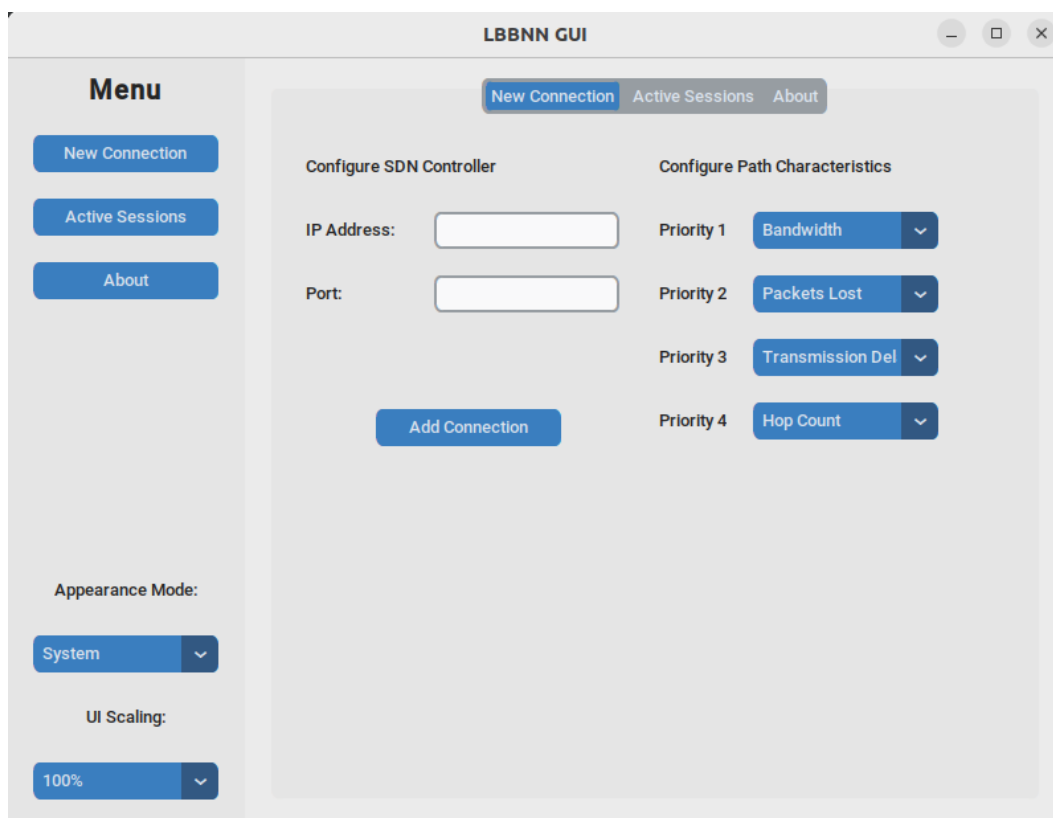


Рис. 4.5. Початкова сторінка користувацького інтерфейсу

Для встановлення нового сеансу в системі користувач повинен ввести IP-адресу та номер порту контролера, з яким він хоче встановити з'єднання, щоб розпочати взаємодію. Після встановлення параметрів підключення користувач може перейти до вибору показників шляху, за якими буде визначено оптимальний шлях. Вищезазначені показники охоплюють ряд параметрів, включаючи пропускну здатність, кількість втрачених пакетів, час передачі та кількість хопів. Вибрані показники безпосередньо впливають на алгоритмічний процес розрахунку оптимального шляху, надаючи користувачеві можливість регулювати рівновагу між продуктивністю та надійністю маршруту відповідно до конкретних вимог до QoS.

За замовчуванням, як показано на рисунку 4.5, кожному показнику присвоюється попередньо визначене значення пріоритету, яке можна змінити відповідно до конкретних вимог досліджуваної мережі. Це

полегшує вибір шляхів для попереднього тестування, забезпечуючи швидку конфігурацію процесу оптимізації та фундаментальну оцінку функціональності нейронної мережі.

У разі введення користувачем помилкових параметрів підключення, таких як неправильна IP-адреса (рис. 4.6) або порт контролера (рис. 4.7), система відобразить діалогове вікно з повідомленням про помилку. Діалогове вікно попереджає користувача про необхідність переконатися в правильності IP-адреси або порту, щоб підключитись до контролера.



Рис. 4.6. Неправильно введений IP-адрес контролера

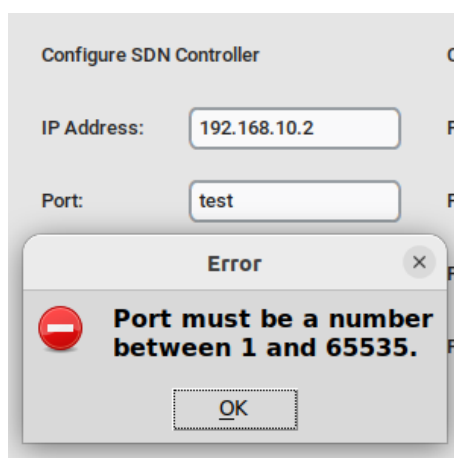


Рис. 4.7. Неправильно введений порт контролера

У випадку, якщо вибрані показники оптимізації шляху мають однакові пріоритети, таким чином порушуючи принцип унікальності, приписуваний кожному показнику (рис. 4.8), буде створене діалогове вікно з повідомленням про помилку, яке вказує на конфлікт пріоритетів. Це повідомлення допомагає користувачеві змінювати послідовність пріоритетів таким чином, щоб кожному показнику призначався окремий пріоритет, що має важливе значення для функціонування алгоритму пошуку маршруту.

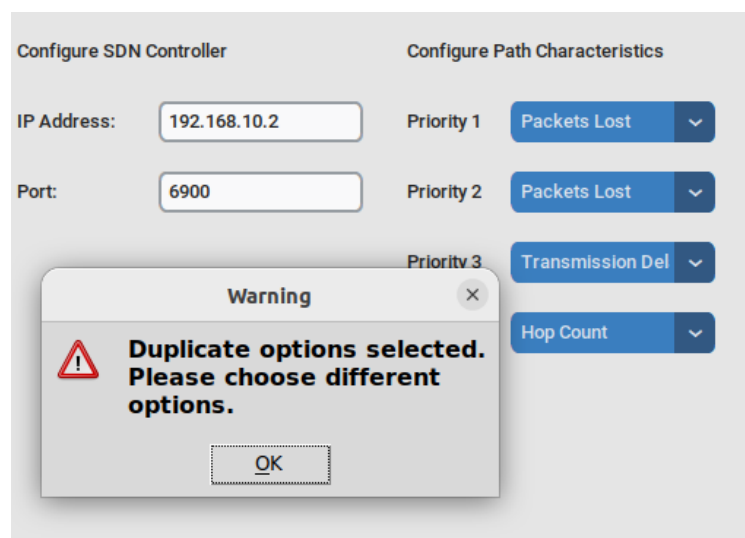


Рис. 4.8. Дублювання обраних показників шляху

Якщо всі параметри підключення введені правильно, система перейде до створення нового сеансу з контролером. Успішне з'єднання вказується на вкладці «Active Sessions», де користувач може переглянути відповідну інформацію щодо активних сеансів, включаючи IP-адресу та порт контролера, з яким встановлюється з'єднання. Ця вкладка дозволяє користувачеві спостерігати за станом сеансу в режимі реального часу, що полегшує моніторинг з'єднання (рис. 4.9).

Крім того, інтерфейс відображає підтвердження успішного з'єднання, таким чином надаючи користувачеві зворотний зв'язок щодо стабільності встановленого з'єднання. У разі розриву зв'язку

інформація, що відображається на вкладці «Active Sessions», буде автоматично оновлена відповідно до поточного статусу сеансу.

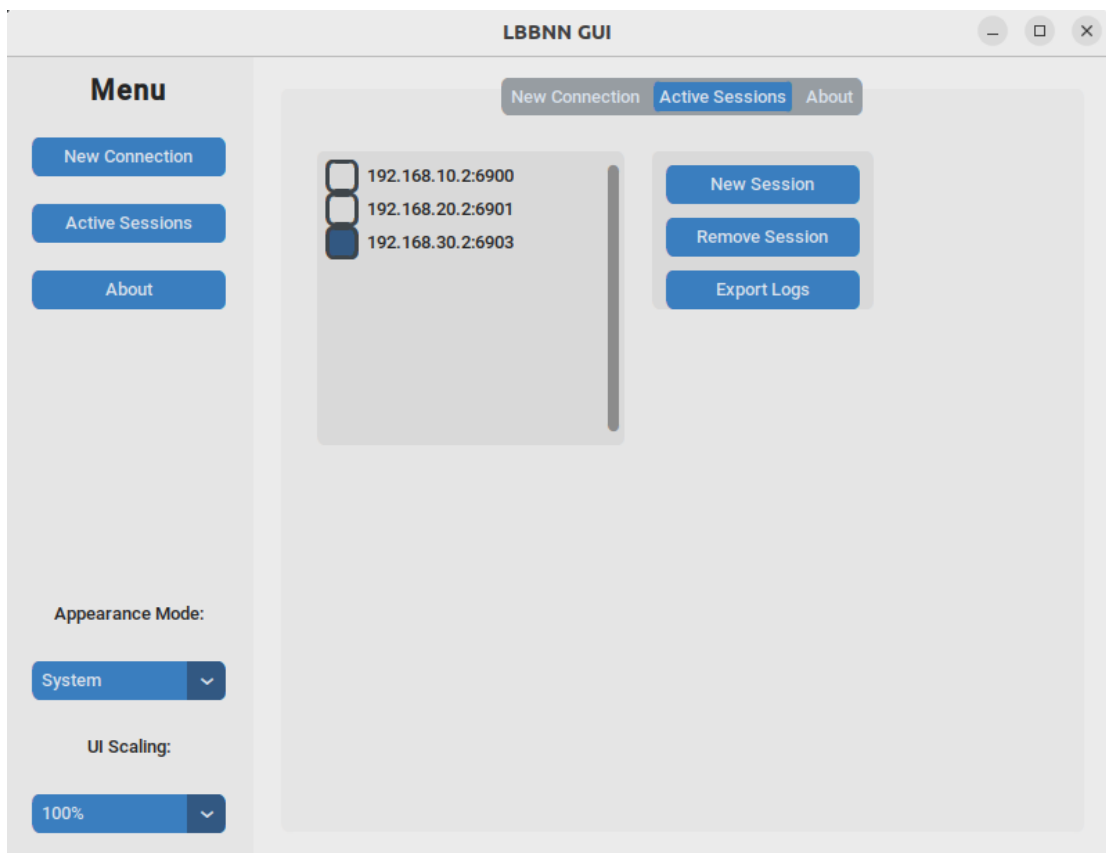


Рис. 4.9. Активні сесії

Розроблений інтерфейс користувача забезпечує можливість встановлення одночасних з'єднань з кількома контролерами в різних топологіях, тим самим помітно розширюючи сферу експериментів. Це дає змогу конфігурувати незалежні сеанси з контролерами, які керують розрізненими мережевими конфігураціями, включаючи слабкозв'язний граф, повнозв'язний граф та граф де Бруїна. Гнучкість налаштувань забезпечує можливість проведення експериментів для різних топологій, що необхідно для комплексної оцінки розробленого методу побудови трафіку та його здатності адаптуватися до конкретних умов мережевого середовища.

Початковий експеримент порівнює середній коефіцієнт

використання пропускної здатності між запропонованим методом і альтернативними методами балансування трафіку, а саме Dynamic Load Balancing (DLB) [10], модифікований Equal-Cost Multi-Path (ECMP) [65] та Round Robin. Експеримент демонструє роботу кожного методу в балансуванні навантаження за різними топологіями та його вплив на середнє використання пропускної здатності в мережі. Це дає змогу оцінити переваги розробленого методу та його конкурентоспроможність з існуючими алгоритмами.

Під час експерименту було зібрано середній коефіцієнт використання пропускної здатності та середній час передачі даних. Ці показники використовуються для оцінки стратегії балансування навантаження, оскільки відображають рівномірність розподілу трафіку та швидкість передачі даних по мережі. Результати експериментів представлені на рисунках 4.10 - 4.13.

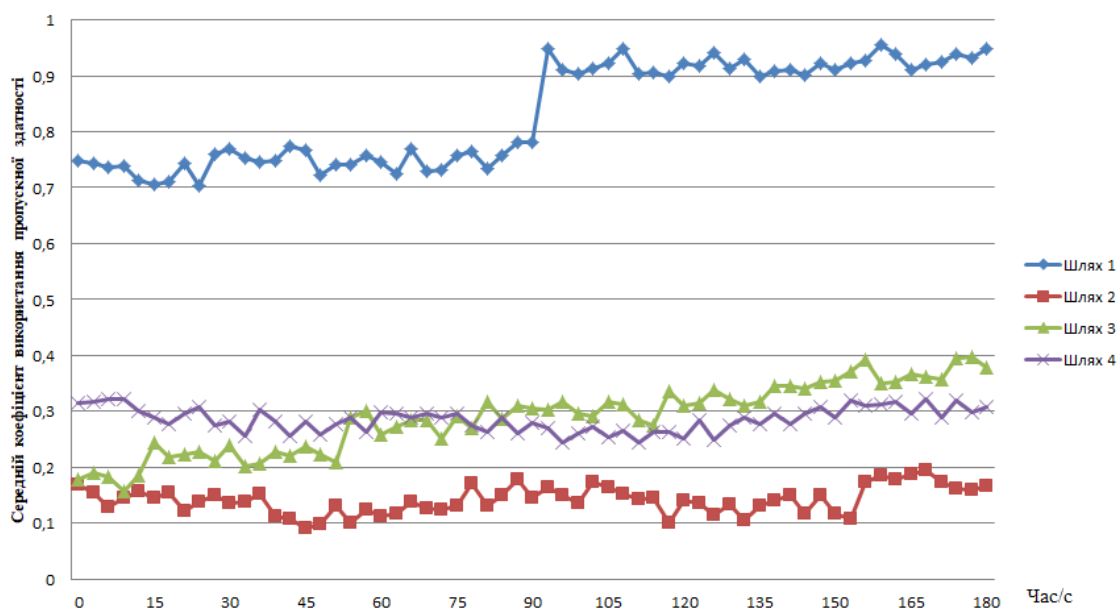


Рис. 4.10. Середній коефіцієнт використання пропускної здатності для стратегії Round Robin

На рисунку 4.10 показано експериментальні результати стратегії

Round Robin. Враховуючи статичний характер цієї стратегії балансування навантаження, вона не може врахувати стан з'єднання в реальному часі. Отже, для передачі нового потоку даних обирається шлях 1, який демонструє високий коефіцієнт використання пропускної здатності.

Стратегія Round Robin працює на основі чергування шляхів, незалежно від їх поточного стану або навантаження. Це означає, що вона не здатна реагувати на зміни стану мережі, такі як перевантаження або обмежена пропускна здатність. І як видно по результатам експерименту, для передачі нового потоку даних обирається шлях 1, який вже був перевантаженим.

Це призводить до негативного впливу на стан навантаження 1 шляху, тим самим збільшуючи ймовірність перевантаження. Перевантаження на одному з основних шляхів негативно впливає на загальну продуктивність мережі, що призводить до збільшення затримок і підвищення ймовірності втрати даних. Основним недоліком стратегії балансування навантаження Round Robin є те, що в ній відсутній механізм динамічного реагування на мережеві умови. Цей недолік робить її менш ефективною, ніж динамічні стратегії балансування навантаження, які аналізують стан мережі в режимі реального часу.

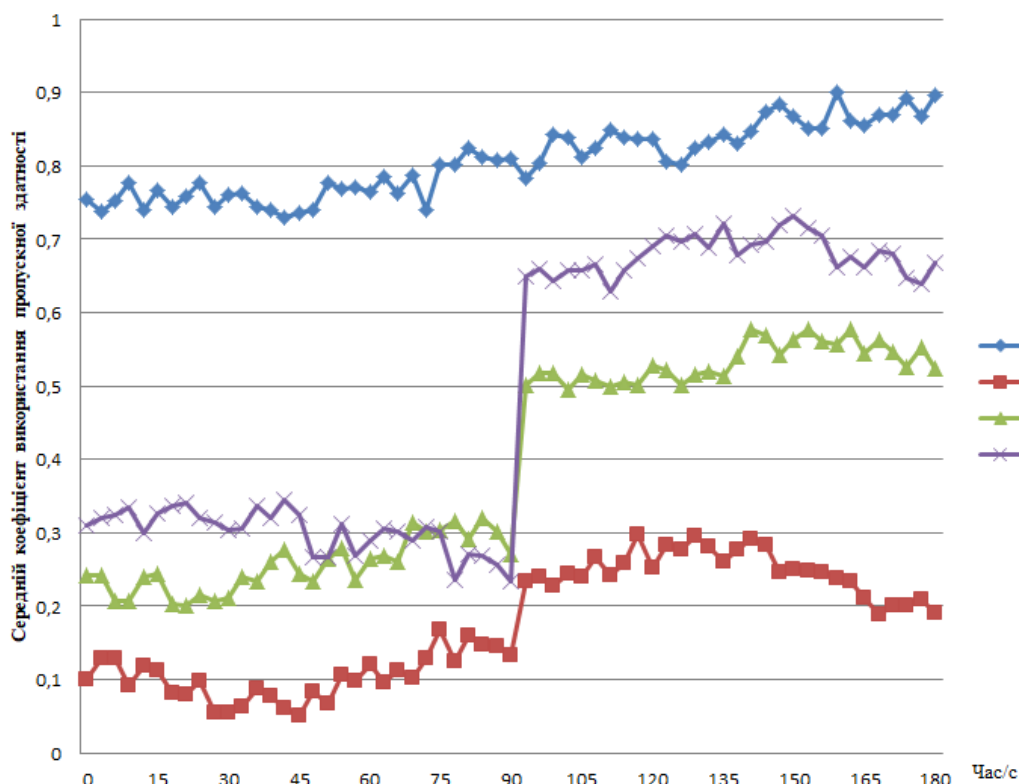


Рис. 4.11. Середній коефіцієнт використання пропускної здатності для стратегії DLB

На рисунку 4.11 показано, що при передачі нового потоку даних метод динамічного балансування навантаження (DLB), запропонований у роботі [10], враховує лише стан підключення наступного хопу. Це призводить до помилкового вибору шляху 4 для передачі нового потоку даних.

Через обмежену максимальну пропускну здатність цього каналу, швидко відбувається перевантаження 4 шляху. Коефіцієнт використання пропускної здатності на ньому швидко зростає, що негативно впливає на якість передачі даних. Одночасно перевантаження на шляху 4 також впливає на шляхи 2 та 3, які мають спільні канали зв'язку. Це, відповідно, збільшує ймовірність перевантаження мережі в цілому.

Основною проблемою цього методу є те, що він не враховує глобальний стан мережі, що призводить до неоптимального розподілу

трафіку. Це є суттєвим недоліком, оскільки призводить до збільшення затримок і втрати даних. Як наслідок, знижується ефективність мережі і збільшується ризик перевантаження в критичних зонах.

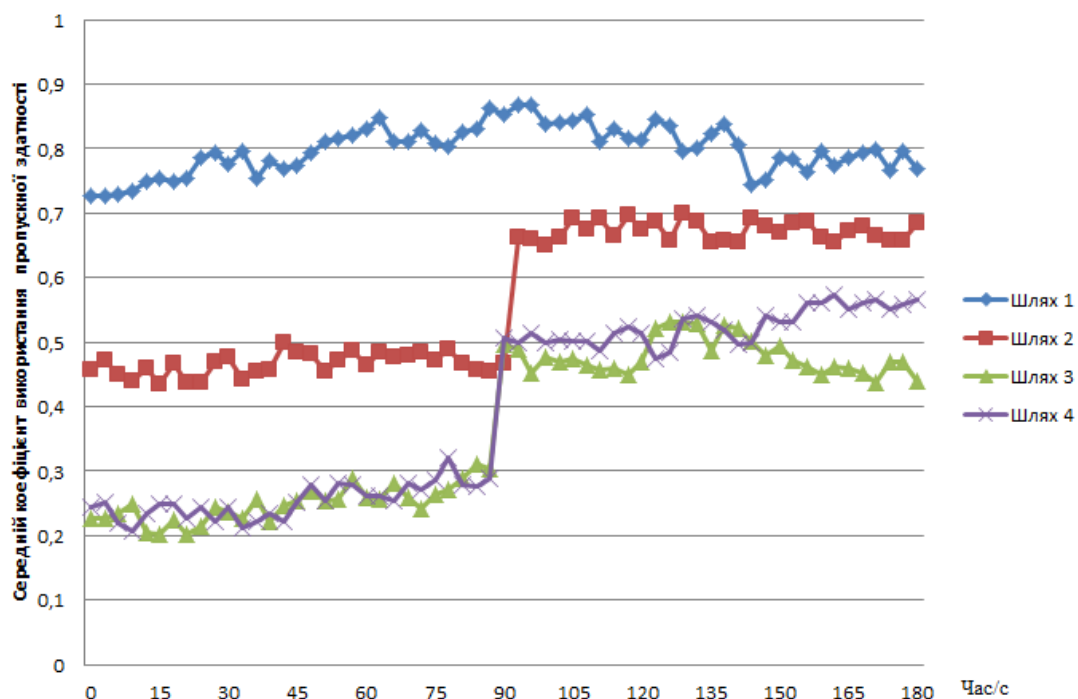


Рис. 4.12. Середній коефіцієнт використання пропускної здатності для стратегії модифікованого ЕСМР [65]

На рис. 4.12 показано, що при передачі нового потоку даних модифікований метод динамічного балансування навантаження ЕСМР, що був запропонований у роботі [65], реалізується централізовано в контролері SDN. Це дозволяє оперативно коригувати маршрутну інформацію, враховуючи взаємний вплив мережевих потоків, та суттєво зменшує обсяг службової інформації. Крім того, модифікований підхід враховує надійність шляху, включаючи ймовірність відмови для всіх вузлів на кожному шляху. Тому більша кількість пакетів передається більш надійним шляхом, що, відповідно, зменшує відсоток втрати інформації, яка відбувається в процесі передачі. Тим не менш, такий

розподіл трафіку може бути недостатньо рівномірним.

У порівнянні зі стандартним методом, модифікований ЕСМР зменшує час, необхідний для перемаршрутизації, і втрату пакетів на 10-15%. Однак цей підхід не враховує глобальний стан мережі, що може призвести до неефективного розподілу трафіку. Це є суттєвим недоліком, оскільки може призвести до неефективного використання ресурсів системи. Як наслідок, знижується ефективність мережі та збільшується ризик перевантаження в критичних зонах [65].

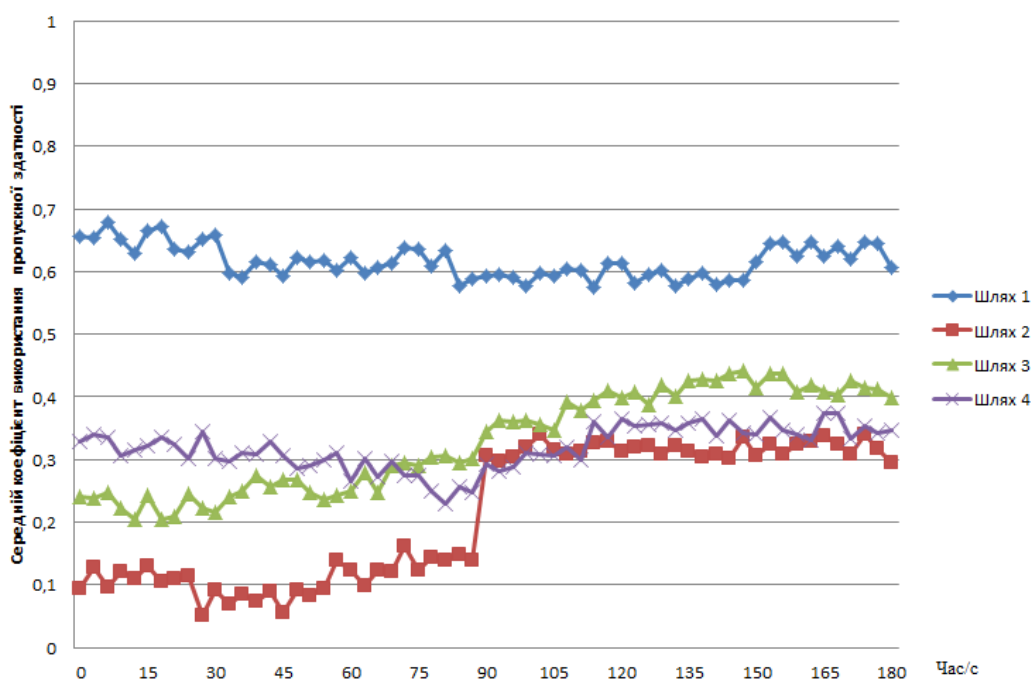


Рис. 4.13. Середній коефіцієнт використання пропускної здатності для розробленого методу LBBNN

Як показано на рисунку 4.13, при передачі нового потоку даних, розроблена стратегія балансування навантаження LBBNN обирає шлях 2 як основний шлях передачі. Цей шлях обрано через його здатності забезпечити оптимальне навантаження на мережу, що дозволяє досягти високої ефективності передачі даних.

Метод LBBNN аналізує поточний стан мережі для вибору

оптимального маршруту для передачі даних. Він враховує такі показники, як коефіцієнт пропускної здатності, часу передачі даних, кількість хопів та коефіцієнт втрати пакетів. За цими показниками 2 шлях був визначений оптимальним для вхідного потоку даних.

Крім того, для всіх альтернативних шляхів підтримуються умови оптимального використання ресурсів. Це означає, що навіть у випадку перевантаження на шляху 2, інші шляхи можуть бути задіяні для безперебійної роботи мережі без значних затримок і втрат даних.

Такий підхід дозволяє не тільки ефективно розподіляти навантаження в мережі, але й зменшити ймовірність виникнення перевантажень, що позитивно впливає на загальну продуктивність мережі. Результати експерименту демонструють, що запропонована стратегія сприяє стабільному та ефективному балансуванню навантаження, тим самим дозволяючи досягти вищої якості обслуговування в SDN мережах.

На відміну від методів DLB, Round Robin та модифікованого ECMP, запропонований підхід дозволяє ефективно розподіляти навантаження в мережі та зменшує ймовірність виникнення перевантажень, що позитивно впливає на загальну продуктивність мережі. Результати експериментів демонструють, що запропонована стратегія сприяє стабільному та ефективному балансуванню навантаження, тим самим дозволяючи досягти вищої якості обслуговування в SDN мережах.

Таблиця 4.1 ілюструє середній коефіцієнт використання пропускної здатності та середньоквадратичне відхилення для розглянутих стратегій балансування навантаження.

Таблиця 4.1. Середньоквадратичне відхилення для слабкозв'язного графа

	Шлях 1	Шлях 2	Шлях 3	Шлях 4	Середньоквадратичне відхилення
LBBNN	0.616	0.214	0.329	0.319	0.172
DLB	0.804	0.175	0.391	0.485	0.262
Модифікований ECMP	0.799	0.567	0.366	0.392	0.2
Round Robin	0.832	0.143	0.29	0.284	0.307

Як показано в таблиці 4.1, середньоквадратичне відхилення методу LBBNN є найнижчим серед усіх розглянутих стратегій, що свідчить про те, що використання пропускну здатності всіх шляхів є відносно однаковим. Завдяки використанню стратегії балансування навантаження на основі штучної нейронної мережі досягається сприятливий результат.

Оскільки стратегія DLB не враховує глобальний стан завантаження каналу, що призводить до вибору неоптимального шляху передачі та погіршення стану альтернативних шляхів. Це також стає причиною нерівномірного розподілу навантаження, що може призвести до потенційних перевантажень в мережі. Тим не менш, ця стратегія балансування навантаження доволі активно використовується, хоча і з меншою ефективністю, ніж запропонована стратегія LBBNN на основі нейронній мережі.

Стратегія Round Robin - це статична стратегія балансування навантаження, тому вона не може враховувати стан навантаження на шляхи в реальному часі. Це призводить до вибору неоптимальних шляхів передачі даних, що значно збільшує ймовірність перевантаження. Отже, ця стратегія є найменш ефективною з вищезгаданих методів.

Модифікований ECMP реалізується централізовано в контролері SDN, що дає змогу оперативно коригувати маршрутну інформацію та суттєво зменшити обсяг службової інформації. Він зменшує час, необхідний для перемаршрутизації, і втрату пакетів на 10-15% але не враховує глобальний стан мережі, що призводить до неефективного використання ресурсів та розподілу трафіку.

Рисунок 4.14 ілюструє вплив різних стратегій балансування навантаження на час передачі нового потоку даних.

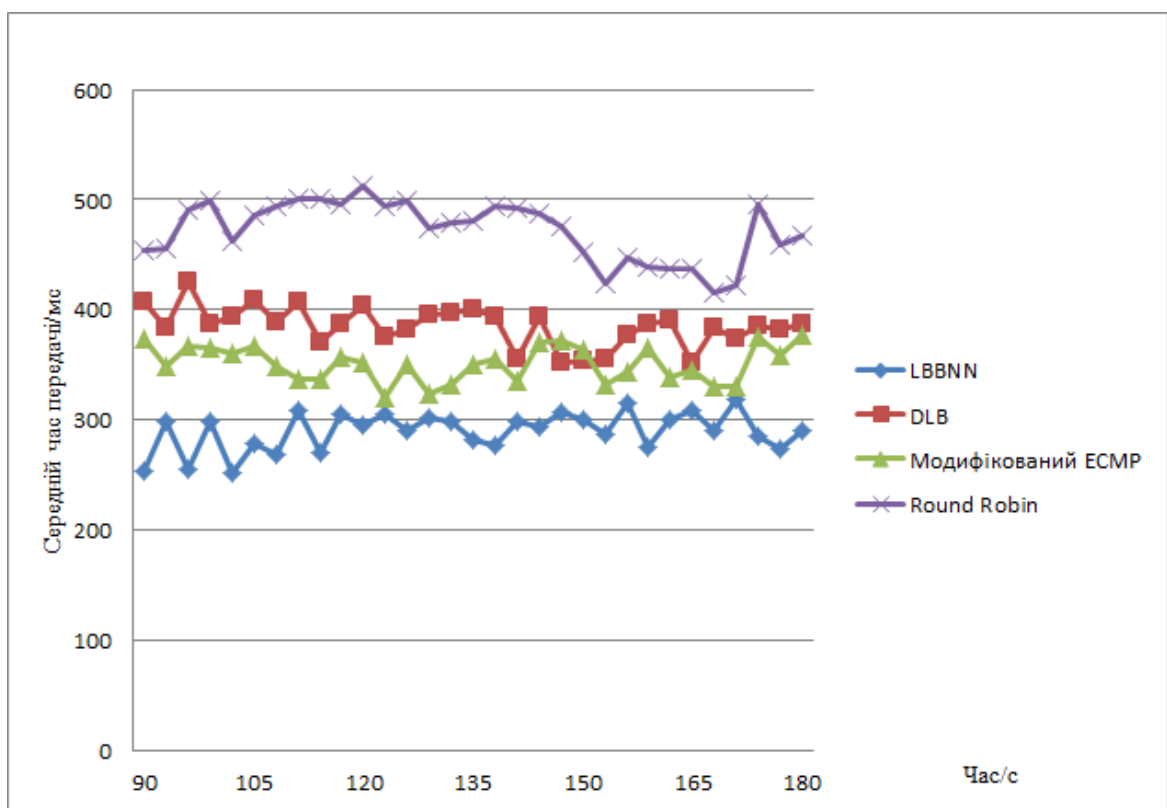


Рис. 4.14. Середній час передачі нового вхідного потоку даних для різних стратегій

Середній час передачі нового вхідного потоку даних при використанні розробленої стратегії балансування трафіку на основі нейронної мережі становить 291 мс. Це найменший час передачі серед усіх розглянутих стратегій, що свідчить про високу ефективність та швидкість реакції на зміни в умовах мережі.

Середній час стратегії DLB становить 385 мс. Незважаючи на те, що ця стратегія включає в себе певні динамічні аспекти, вона розглядає виключно стан наступного з'єднання при виборі шляху передачі, що призводить до більшого часу передачі пакетів порівняно із запропонованою стратегією.

Середній час передачі пакетів модифікованого ЕСМР становить 349 мс. Ця стратегія, хоч і враховує певні динамічні аспекти мережі, в основному зосереджений на зменшенні відсотку втрати пакетів та швидкості переадресації. У результаті більша кількість пакетів надсилається більш надійним шляхом, що зменшує відсоток втрати інформації під час передачі даних. Але, в той же час, такий розподіл трафіку не є достатньо рівномірним що призводить до більш високого часу передачі порівняно з запропонованою стратегією.

Середній час передачі Round Robin становить 469 мс. Це найбільший час серед усіх розглянутих стратегій, що вказує на неефективність статичного підходу до балансування навантаження та відсутність врахування реального стану мережі.

Результати, представлені на рисунку 4.14, демонструють, що запропонована стратегія на основі нейронної мережі забезпечує найменший час передачі даних. Здатність розробленої стратегії адаптуватися до мінливих умов у режимі реального часу та ефективно використання наявних ресурсів дозволяють їй досягати значно кращих результатів, ніж інші методи. Результати застосування розробленого методу демонструють підвищення продуктивності мережі та зменшення часу передачі пакетів у мережі на 14% у порівнянні з модифікованим ЕСМР та більш ніж на 20% у порівнянні з DLB.

Для перевірки розробленої системи балансування навантаження на топологіях, зображених на рисунках 4.2 та 4.3, був використаний ідентичний алгоритм, що і для першої топології. Перші 90 секунд експерименту всі хости другої топології, окрім h1, випадково передавали

дані один одному, щоб створити деякий трафік кількома шляхами в мережі. Це дозволило імітувати реалістичне навантаження на мережу та перевірити здатність кожної стратегії ефективно керувати трафіком. Потім h1 передавав трафік на h8 для імітації нового вхідного потоку даних у домені SDN.

Другий експеримент проводиться для вивчення ефективності роботи розробленого методу конструювання трафіку для різних типів топологій, включаючи слабкозв'язний граф, повнозв'язний граф та граф де Бруїна. Для цього використовується можливість паралельного підключення до контролерів у кожній з зазначених топологій та здійснення вимірювання за певними показниками: швидкості передачі даних та втрати пакетів. Вибір різних топологій дозволяє виявити переваги та недоліки методу в умовах різної структури зв'язності та підтвердити його ефективність у широкому спектрі мережевих умов.

Хід експерименту включав поетапне підключення до контролерів з кожною з обраних топологій. Після встановлення з'єднання з контролером інтерфейс автоматично активував сесію, починаючи запис статистики, що відображала середнє завантаження каналів, середній час передачі та показник втрат пакетів. У кожній з мереж розраховувались часові характеристики для методів балансування трафіку: запропонованого методу конструювання трафіку LBBNN, а також методів DLB, модифікованого ECMP та Round Robin. Таке порівняння дозволить оцінити не тільки ефективність запропонованого методу, а й його продуктивність щодо часу конструювання, що є критично важливим для масштабних мережевих топологій.

Для кожної топології контролери були налаштовані на моніторинг усіх необхідних характеристик з'єднання в режимі реального часу. Швидкість передачі даних і втрата пакетів були кількісно визначені для кожного методу розподілу трафіку за подібних умов навантаження. Експеримент проводився шляхом встановлення початкової та кінцевої

точок передачі даних у мережі, після чого реєструвалися значення цих параметрів. Такий підхід дозволяє збирати дані з метою аналізу ефективності запропонованого методу в порівнянні з іншими методами, а також для оцінки продуктивності як для випадкових шляхів, так і для обраного оптимального шляху.

Нижче наведені результати експерименту, які показують середню пропускну здатність, втрати пакетів і час передачі для кожного з методів конструювання трафіку та для кожної топології (таблиця 4.2).

Таблиця 4.2. Середній коефіцієнт використання пропускну здатності та втрата пакетів

Топологія	Метод	Пропускна здатність (біт/с)	Втрата пакетів (%)
Слабкозв'язний граф	LBBNN	4974	4
	DLB	3715	9
	Модифікований ECMP	4247	5
	Round Robin	1782	14
Повнозв'язний граф	LBBNN	277	2
	DLB	220	7
	Модифікований ECMP	269	4
	Round Robin	178	10
Граф де Бруйна	LBBNN	1416	2
	DLB	1027	12
	Модифікований ECMP	1339	4
	Round Robin	995	10

Графічне порівняння, зображене на рисунках 4.15–4.16, демонструє стабільну перевагу методу LBBNN у всіх трьох топологіях. Зокрема, спостерігається підвищення пропускної здатності та зменшення втрат пакетів, особливо на слабкозв'язних графах, де розроблений метод забезпечує найвищу пропускну здатність 4974 біт/с, значно випереджаючи інші методи. В умовах повнозв'язного графа та графа де Бруйна метод також демонструє знижений час передачі порівняно з іншими розглянутими методами, що вказує на його ефективність у балансуванні навантаження на складних топологіях.

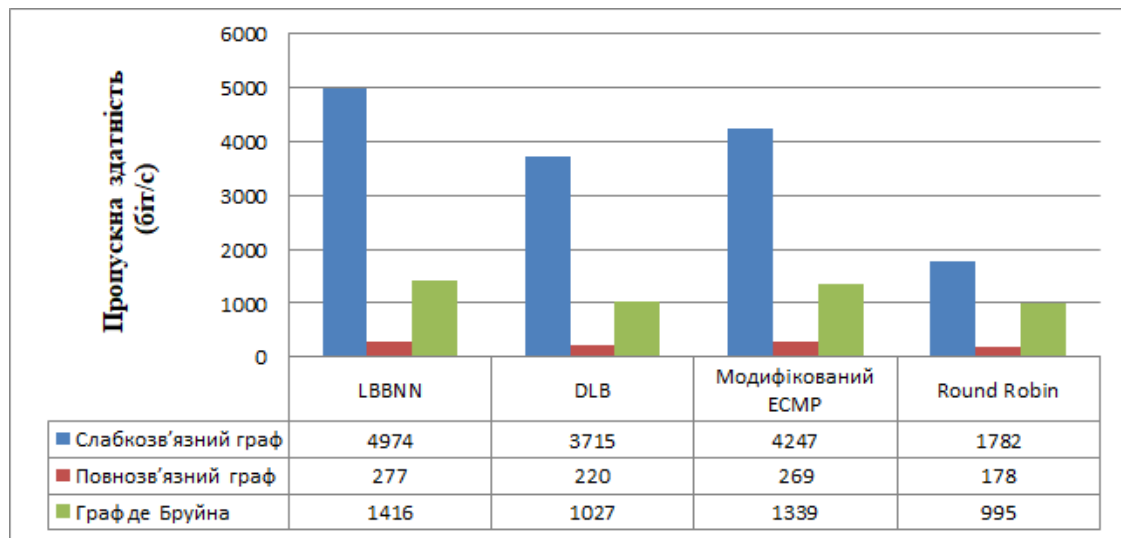


Рис. 4.15. Середня пропускну здатність для різних стратегій конструювання трафіку

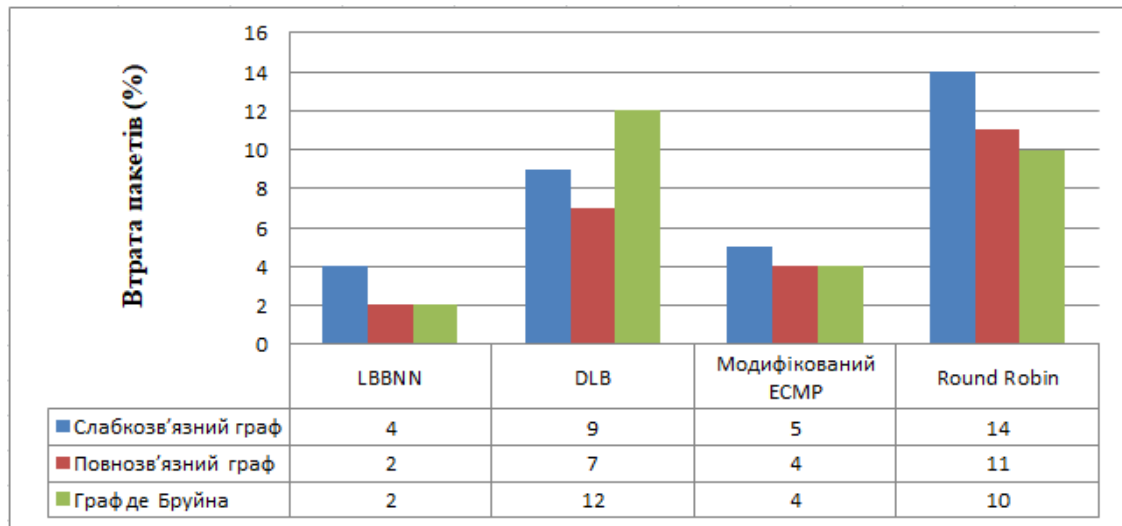


Рис. 4.16. Відсоток втрати пакетів для різних стратегій конструювання трафіку

Що стосується перевантаженості маршруту, результати експериментів показали, що конструювання трафіку в запропонованому алгоритмі LBBNN, як і у модифікованому ECMP, враховує параметри надійності маршруту. Це дає змогу забезпечити рівномірне завантаження каналів і, водночас, знизити ймовірність втрати пакетів під час маршрутизації. У традиційних алгоритмах DLB та Round Robin і втрата пакетів, і перевантаження маршрутів вищі, ніж у запропонованому алгоритмі LBBNN. На підставі наведених вище результатів моделювання можна зробити висновок, що запропонований метод конструювання трафіку забезпечує відповідні параметри якості обслуговування.

Також, через велику кількість можливих шляхів, особливо у повнозв'язних графах та графах де Бруїна, стандартні методи конструювання трафіку, такі як DLB та Round Robin, демонструють високу часову складність. Це призводить до збільшення загальної вартості планування та створення маршрутів у мережі, що може бути значною проблемою в контексті великих SDN мереж.

Запропонований метод LBBNN дозволяє суттєво скоротити часову складність маршрутизації, незалежно від топології мережі та рівня підключення вузлів. Таким чином, метод LBBNN підходить для реалізації в мережах SDN з високим рівнем зв'язку, де ключовими є відмовостійкість, ефективне балансування навантаження та висока якість обслуговування. Отже, LBBNN здатний гарантувати рівномірне завантаження каналів мережі та скорочувати час і ресурси, необхідні для конструювання трафіку, тим самим зберігаючи задані показники продуктивності навіть за наявності високої щільності мережевих з'єднань.

Висновки до розділу 4

У данному розділі представлено аналіз результатів застосування розробленого методу балансування навантаження LBBNN.

Ефективність розробленого методу була оцінена на основі результатів його тестування на наборах даних, зібраних з різних типів топологій, та порівняна з іншими методами. Розроблений метод базується на використанні нейронної мережі для визначення оптимального шляху передачі нового потоку даних з урахуванням динамічних змін стану SDN мережі. Це дозволяє покращити результати балансування навантаження порівняно з існуючими стратегіями. Тестування розробленого методу балансування навантаження на трьох різних топологіях дало наступні результати:

- зменшення середнього коефіцієнту використання пропускну здатності
- зменшення ймовірності перевантаження шляху у випадку передачі нового потоку даних
- досяглося оптимальне використання ресурсів з урахуванням динамічного стану мережі

Результати аналізу показали, що запропонований метод балансування навантаження LBBNN на основі штучного інтелекту продемонстрував кращі результати серед усіх протестованих стратегій. Найменше значення середньоквадратичного відхилення свідчить про те, що навантаження розподіляється рівномірно з оптимальним використанням доступних мережевих ресурсів. Це свідчить про те, що коефіцієнт використання пропускну здатності є однорідним для кожного маршруту, що забезпечує кращу продуктивність мережі. Крім того, середній час передачі нових потоків даних при використанні LBBNN для слабозв'язної топології становить 315 мс, що є найнижчим показником серед усіх протестованих методів для цієї топології, тим самим підтверджуючи високу швидкість реакції на зміни в мережі.

Крім того, результати тестування продемонстрували перевагу динамічних стратегій балансування навантаження порівняно зі статичними аналогами. Стратегія на основі штучного інтелекту (LBBNN) не тільки забезпечує рівномірний розподіл навантаження і мінімізує час передачі пакетів, але й ефективно реагує на зміни в стані мережі. Це особливо важливо для SDN мереж, де швидкість і надійність передачі даних є ключовими показниками продуктивності.

Таким чином, запропонована стратегія балансування навантаження на основі штучного інтелекту явно перевершує інші методи, особливо в динамічному мережевому середовищі. Реалізація LBBNN дозволяє оптимально використовувати пропускну здатність і зменшити час передачі пакетів у мережі на 14%, що робить цей підхід ефективним засобом керування трафіком в програмно-конфігурованих мережах.

ВИСНОВКИ

В дисертаційній роботі розв'язане актуальне наукове завдання використання штучного інтелекту в основі методу конструювання трафіку в програмно-конфігурованих мережах. При цьому, отримано наступні наукові та практичні результати:

1. Проведено аналіз існуючих методів керування трафіком у програмно-конфігурованих мережах, зокрема статичних та динамічних стратегій балансування навантаження, що дозволило виявити основні недоліки та переваги кожного з підходів.

2. Досліджено вплив динамічних стратегій балансування навантаження, включаючи стратегію балансування навантаження LBBNN на основі штучного інтелекту, на ефективність процедури конструювання трафіку.

3. Запропоновано та обґрунтовано модифікований метод конструювання трафіку на основі штучного інтелекту у SDN мережах з урахуванням їх особливостей та вимог до них, який враховує глобальний стан мережі та дозволяє досягти більш рівномірного розподілу трафіку між каналами передачі даних.

4. Запропоновано та обґрунтовано удосконалену архітектуру системи конструювання трафіку в програмно-конфігурованих мережах на основі штучного інтелекту, яка, на відміну від існуючих методів, забезпечує можливість використання інтегрального показника для балансування навантаження в залежності від типу трафіка.

5. Отримав подальший розвиток метод обрахунку показників для вибору шляху в програмно-конфігурованих мережах з урахуванням особливостей мереж та вимог, висунутих до них, що надає можливість використовувати комплексний метод конструювання трафіку, який включає у себе метод обрахунку показників шляху та балансування навантаження. Це, на відміну від існуючих підходів, дозволяє

адаптувати модель нейронної мережі до вимог, які висунуті до мережі, та враховувати глобальний стан мережі для конструювання трафіку.

6. Проведено тестування запропонованого методу конструювання трафіку LBBNN у порівнянні з іншими методами (DLB, Round Robin, модифікований ЕСМР), що підтвердило його переваги у мінімізації затримок та збалансованому використанні мережевих ресурсів.

7. Проведено аналіз результатів тестування, що підтвердив перевагу запропонованого методу LBBNN за ключовими показниками: середнім часом передачі пакетів, рівномірністю розподілу навантаження та мінімізацією ймовірності перевантаження мережі. За результатами застосування розробленого методу було досягнуто підвищення продуктивності мережі за рахунок більш рівномірного розподілу трафіку та оптимального вибору маршрутів. Зокрема, середній час передачі пакетів у мережі зменшився на 14% порівняно з традиційними алгоритмами маршрутизації. Це стало можливим завдяки динамічному аналізу параметрів трафіку, зокрема пропускної здатності та завантаженості мережевих вузлів, що дозволило швидше адаптувати маршрутизацію до змін у реальному часі.

Розроблена модель має високу ефективність у задачі прогнозування оптимального шляху передачі пакетів. Вона може бути використана як основа для подальших досліджень та вдосконалення мережевих алгоритмів маршрутизації, а також для розробки нових методів оптимізації передачі даних у сучасних комп'ютерних мережах.

ЖИТЕПАТҮПА

1. Nunes, B. A. A., Mendonca, M., Nguyen, X.-N., Obraczka, K., & Turetti, T. (2014). A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3), 1617–1634.
2. Raza, S., Huang, G., Chuah, C.N. et al: ‘Measurouting: a framework for routing assisted traffic monitoring’, *IEEE/ACM Trans. Netw.*, 2012, 20, (1), pp. 45–56
3. Heorhiadi, V., Reiter, M.K., Sekar, V.: ‘Simplifying software-defined network optimization using SOL’. *Proc. of the 13th Usenix Conf. on Networked Systems Design and Implementation*, California, USA, March 2016, pp. 223–237
4. Martini, B., Adami, D., Sgambelluri, A. et al: ‘An SDN orchestrator for resources chaining in cloud data centers’. *Proc. of 2014 European Conf. on Networks and Communications (EuCNC)*, Bologna, Italy, June 2014, pp. 1–5
5. Akyildiz, I.F., Lee, A., Wang, P. et al: ‘A roadmap for traffic engineering in SDN-OpenFlow networks’, *Comput. Netw.*, 2014, 71, pp. 1–30
6. Feamster, N., Rexford, J., Zegura, E.: ‘The road to SDN: an intellectual history of programmable networks’. *ACM SIGCOMM Comput. Commun. Rev.*, 2014, 44, (2), pp. 87–98
7. Nunes, B.A.A., Mendonca, M., Nguyen, X.N. et al: ‘A survey of software-defined networking: past, present, and future of programmable networks’, *IEEE Commun. Surv. Tutorials*, 2014, 16, (3), pp. 1617–1634
8. Kulakov, Y., & Korenko, D. (2021). Modified Method of Traffic Engineering in DCN with a Ramified Topology. *International Journal of Advanced Computer Science and Applications*, 12(12).

9. Korenko, D., Cherevatenko, O., Rusinov, V., & Kulakov, Y. (2022). Creation of the method of multipath routing using known paths in software-defined networks. *Technology audit and production reserves*, 4(2/66), 19–24.
10. Jain, R., Paul, S.: ‘Network virtualization and software defined networking for cloud computing: a survey’, *IEEE Commun. Mag.*, 2013, 51, (11), pp. 24–31
11. Bera, S., Misra, S., Vasilakos, A.V.: ‘Software-defined networking for internet of things: a survey’, *IEEE Internet Things J.*, 2017, 4, (6), pp. 1994–2008
12. Lin, P., Bi, J., Wolff, S. et al: ‘A west-east bridge based SDN inter-domain testbed’, *IEEE Commun. Mag.*, 2015, 53, (2), pp. 190–197
13. Jain, S., Kumar, A., Mandal, S. et al: ‘B4: experience with a globally-deployed software defined WAN’, *ACM SIGCOMM Comput. Commun. Rev.*, 2013, 43, (4), pp. 3–14
14. Hong, C., Kandula, S., Mahajan, R. et al: ‘Achieving high utilization with software-driven WAN’. *Proc. of ACM SIGCOMM Conf.*, Hong Kong, China, August 2013, pp. 15–26
15. Singh, A. K., & Srivastava, S. (2018). A survey and classification of controller placement problem in SDN. *International Journal of Network Management*, 28, 1–25.
16. Masoudi, R., & Ghafari, A. (2016). Software defined networks: A survey. *Journal of Network and Computer Applications*, 67, 1–25.
17. Alshnta, A. M., Abdollah, M. F., & Al-Haiqi, A. (2018). SDN in the home: A survey of home network solutions using Software Defined Networking. *Cogent Engineering*, 5, 1–40.
18. Karakus, M., & Duresi, A. (2017). Quality of Service (QoS) in Software Defined Networking (SDN): A survey. *Journal of Network and Computer Applications*, 80, 200–218.
19. Wang, W., Tian, Y., Gong, X., Qi, Q., & Hu, Y. (2015). Software

defined autonomic QoS model for future Internet. *The Journal of Systems and Software*, 110, 122–135.

20. Diego, F. M. V., Rothenberg, C., & Azodolmolky, S. (2014). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), 14–76.

21. Wibowo, F. X. A., Gregory, M. A., Ahmed, K., & Gomez, K. M. (2017). Multi-domain software defined networking: Research status and challenges. *Journal of Network and Computer Applications*, 87, 32–45.

22. Nippon Telegraph and Telephone Corporation. (2012). Ryu Network Operating System. <http://osrg.github.com/ryu/>

23. Floodlight is a Java-based OpenFlow controller. (2012). <http://foodlight.openflowhub.org/>

24. OpenDaylight. (2013). OpenDaylight: A Linux Foundation Collaborative Project. <http://www.opendaylight.org>

25. Krishnaswamy, U., Berde, P., Hart, J., Kobayashi, M., Radoslavov, P., Lindberg, T., Sverdlov, R., Zhang, S., Snow, W., & Parulkar, G. (2013). ONOS: An open source distributed SDN OS. <http://www.slideshare.net/umeshkrishnaswamy/open-network-operating-system>.

26. HP VAN SDN Controller Software. <http://h20195.www2.hp.com/v2/getpdf.aspx/4AA4-9827ENW.pdf>

27. Sheinbein, D., Weber, R.P.: ‘Stored program controlled network: 800 service using spc network capability’, *Bell Labs Tech. J.*, 1982, 61, (7), pp. 1737–1744

28. Kreutz, D., Ramos, F.M., Verissimo, P.E. et al: ‘Software-defined networking: a comprehensive survey’, *Proc. IEEE*, 2015, 103, (1), pp. 14–76

29. Ferguson, A.D., Guha, A., Liang, C. et al: ‘Participatory networking: an API for application control of SDNs’, *ACM SIGCOMM Comput. Commun. Rev.*, 2013, 43, (4), pp. 327–338

30. Voellmy, A., Kim, H., Feamster, N.: ‘Procera: a language for high-

level reactive network control’. Proc. of the first workshop on Hot Topics in Software Defined Networks, Helsinki, Finland, August 2012, pp. 43–48

31. Monsanto, C., Foster, N., Harrison, R. et al: ‘A compiler and run-time system for network programming languages’, ACM SIGPLAN Not., 2012, 47, pp. 217–230

32. Foster, N., Harrison, R., Freedman, M.J. et al: ‘Frenetic: a network programming language’, ACM SIGPLAN Not., 2011, 46, (9), pp. 279–291

33. ["RESDN: A Novel Metric and Method for Energy Efficient Routing in Software Defined Networks"](#). IEEE Transactions on Network and Service Management. 17 (2): 736–749. [arXiv:1905.12219](#). [doi:10.1109/TNSM.2020.2973621](#). [S2CID 199442001](#).

34. AppNeta. ["Rate Limiting Detection: Bandwidth and Latency"](#). Appneta. Retrieved 2020-07-23.

35. Zhai, Yahong, Yong Lv, and Longyan Xu. "Optimized QoS Routing in Software-Defined In-Vehicle Networks." INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL 19.1 (2024).

36. Mouhoub, Nouredine, Mohamed Lamine Lamali, and Damien Magoni. "A Transitive Closure Algorithm for Routing With Automatic Tunneling." IEEE/ACM Transactions on Networking (2024).

37. Alhilali, Ahmed Hazim, and Ahmadreza Montazerolghaem. "Artificial intelligence based load balancing in SDN: A comprehensive survey." Internet of Things 22 (2023): 100814.

38. Al-Dunainawi, Yousif, Bilal R. Al-Kaseem, and Hamed S. Al-Raweshidy. "Optimized artificial intelligence model for DDoS detection in SDN environment." IEEE Access (2023).

39. Abdi, Abdinasir Hirsi, et al. "Security Control and Data Planes of SDN: A Comprehensive Review of Traditional, AI and MTD Approaches to Security Solutions." IEEE Access (2024).

40. Rojas, Juan Sebastián, Alvaro Rendón Gallón, and Juan Carlos Corrales. "Personalized service degradation policies on OTT applications based on the consumption behavior of users." *Computational Science and Its Applications–ICCSA 2018: 18th International Conference, Melbourne, VIC, Australia, July 2–5, 2018, Proceedings, Part III* 18. Springer International Publishing, 2018.

41. Masood, Fahad, et al. "AI-enabled traffic control prioritization in software-defined IoT networks for smart agriculture." *Sensors* 23.19 (2023): 8218.

42. Russell, S., Norvig, P.: ‘Artificial intelligence (a modern approach)’ (Prentice Hall, New Jersey, 1995, 3rd edn.), 1152p

43. Negnevitsky, M.: ‘Artificial intelligence - a guide to intelligent systems’ (Addison-Wesley, Essex, 2005, 2nd edn.), 415p

44. Kulakov, Y. O., & Korenko, D. V. Methods of applying artificial intelligence in software-defined networks. *Problems of Informatization and Control*, 1(73), 2023, 23-27.

45. Кулаков Ю. О., Коренко Д. В. Метод балансування навантаження в мережах SDN з використанням штучного інтелекту. *Проблеми інформатизації та управління*, 2(78)б 2024.

46. Nguyen, T.T., Armitage, G.: ‘A survey of techniques for internet traffic classification using machine learning’. *IEEE Commun. Surv. Tutor.*, 2008, 10, (4), pp. 56–76

47. Zhang, G.P.: ‘Neural networks for classification: a survey’, *IEEE Trans. Syst. Man Cybern. C, Appl. Rev.*, 2000, 30, (4), pp. 451–462

48. Rokach, L.: ‘Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography’, *Comput. Stat. Data Anal.*, 2008, 53, (12), pp. 4046–4072

49. LeCun, Y., Bengio, Y., Hinton, G.: ‘Deep learning’, *Nature*, 2016, 521, (7553), pp. 436–444

50. Deng, L.: ‘A tutorial survey of architectures, algorithms, and

applications for deep learning’, *APSIPA Trans. Signal Inf. Process.*, 2014, 3, (e2), pp. 1–19

51. Mohammadi, M., Al-Fuqaha, A., Sorour, S. et al: ‘Deep learning for IoT big data and streaming analytics: a survey’, *IEEE Commun. Surv. Tutorials*, 2018, 20, (4), pp. 2923–2960

52. Tang, T., Zaidi, S.A.R., McLernon, D. et al: ‘Deep recurrent neural network for intrusion detection in SDN-based networks’. 2018 IEEE Int. Conf. on Network Softwarization (NetSoft 2018), Montreal, Canada, June 2018

53. Zhang, Q., Yang, L.T., Chen, Z. et al: ‘A survey on deep learning for big data’, *Inf. Fusion*, 2018, 42, pp. 146–157

54. MacQueen, J.B.: ‘Some methods for classification and analysis of multivariate observations’. *Proc. of 5th Berkeley Symp. on Mathematical Statistics and Probability*, Berkeley, USA, 1967, pp. 281–297

55. Khan, S.S., Ahmad, A.: ‘Cluster center initialization algorithm for K-means clustering’, *Pattern Recognit. Lett.*, 2004, 25, (11), pp. 1293–1302

56. Zhang, C., Xia, S.: ‘K-means clustering algorithm with improved initial center’. *IEEE Second Int. Workshop on Knowledge Discovery and Data Mining*, Moscow, Russia, January 2009, pp. 790–792

57. Pal, N.R., Bezdek, J.C.: ‘On cluster validity for the fuzzy c-means model’, *IEEE Trans. Fuzzy Syst.*, 1995, 3, (3), pp. 370–379

58. Kohonen, T.: ‘The self-organizing map’, *Neurocomputing*, 1988, 21, (1-3), pp. 1–6

59. Kohonen, T.: ‘The self-organizing map’, *Proc. IEEE*, 1990, 78, (9), pp. 1464–1480

60. Phan, T.V., Bao, N.K., Park, M.: ‘Distributed-SOM: a novel performance bottleneck handler for large-sized software-defined networks under flooding attacks’, *J. Netw. Comput. Appl.*, 2017, 91, pp. 14–25

61. Fan, Z., Xiao, Y., Nayak, A. et al: ‘An improved network security situation assessment approach in software defined networks’, *Peer-to-Peer*

Netw. Appl., 2017, <https://doi.org/10.1007/s12083-017-0604-2>

62. Arulkumaran, K., Deisenroth, M.P., Brundage, M. et al: 'Deep reinforcement learning: A brief survey', IEEE Signal Process. Mag., 2017, 34, (6), pp. 26–38

63. Watkins, C.J., Dayan, P.: 'Q-learning', Mach. Learn., 1992, 8, (3-4), pp. 279–292

64. Fan, X., Guo, Z.: 'A semi-supervised text classification method based on incremental EM algorithm'. WASE Int. Conf. on Information Engineering, Beidaihe, China, August, 2010, pp. 211–214

65. Artem Volokyta, Alla Kogan, Oleksii Cherevatenko, Dmytro Korenko, Dmytro Oboznyi, Yurii Kulakov, "Traffic Engineering with Specified Quality of Service Parameters in Software-defined Networks", International Journal of Computer Network and Information Security(IJCNIS), Vol.16, No.5, pp.1-13, 2024. DOI:10.5815/ijcnis.2024.05.01

66. Zhang, Y., Cui, L., Wang, W., & Zhang, Y. (2018). A survey on software defined networking with multiple controllers. Journal of Network and Computer Applications, 103, 101–118.

67. Karakus, M., & Durresi, A. (2017). A survey: Control plane scalability issues and approaches in Software-Defined Networking. Computer Networks, 112, 279–293.

68. Rojas, Juan Sebastián, Alvaro Rendon, and Juan Carlos Corrales. "Consumption behavior analysis of over the top services: Incremental learning or traditional methods?." IEEE Access 7 (2019): 136581-136591.

69. Rojas, Juan Sebastián, et al. "Smart user consumption profiling: Incremental learning-based OTT service degradation." IEEE access 8 (2020): 207426-207442.

70. Loutskii, H., Volokyta, A., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & Korenko, D. (2021). Topology synthesis method based on excess de bruijn and dragonfly. In Advances in Computer Science

for Engineering and Education IV (pp. 315-325). Springer International Publishing.

71. Volokyta A., Loutskii H., Rehida P., Honcharenko O., Korenko D., Rusinov V., Ivanishchev B., Kaplunov A. Convolutionary neural networks regarding problem of monitoring data balancing in de bruijn topology. Bulgarian Journal for Engineering Design, 2021, Mechanical Engineering Faculty, Technical University-Sofia. ISSN 1313-7530

72. Volokyta, A., Loutskii, H., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & Korenko, D. (2022). Extended DragonDeBruijn topology synthesis method. International Journal of Computer Network and Information Security, 9(6), 23.

ДОДАТОК А

Частина програмного коду

Лістинг А.1 — Модуль LbbnnResource.java

```
import org.onosproject.net.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.ws.rs.*;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Path("route")
public class LbbnnResource extends AbstractWebResource {
    private final Logger log = LoggerFactory.getLogger(getClass());

    private final HostService hostService = get(HostService.class);
    private final PathService pathService = get(PathService.class);
    private final PacketService packetService = get(PacketService.class);

    // Ініціалізація PacketProcessor для обробки нових пакетів
    private final PacketProcessor packetProcessor = new InternalPacketProcessor();
```

```

public LbbnnResource() {
    packetService.addProcessor(packetProcessor, PacketProcessor.director(2));
}

@GET
@Path("/getPaths")
@Produces(MediaType.APPLICATION_JSON)
public Response getPaths() {
    try {
        // src та dst, збережені при обробці пакету
        HostId src = InternalPacketProcessor.currentSrc;
        HostId dst = InternalPacketProcessor.currentDst;

        if (src == null || dst == null) {
            log.warn("Source or Destination HostId is null.");
            return
Response.status(Response.Status.BAD_REQUEST).entity("Source or Destination
HostId is not set.").build();
        }

        // Отримання всіх маршрутів між джерелом та призначенням
        List<Path> paths = pathService.getPaths(
            hostService.getHost(src).location(),
            hostService.getHost(dst).location());

        // Генерація JSON з характеристиками кожного шляху

```

```

List<Map<String, Object>> pathsData = paths.stream().map(path -> {
    Map<String, Object> pathData = new HashMap<>();
    pathData.put("pathId", path.id());
    pathData.put("Bandwidth", calculateBandwidth(path));
    pathData.put("PacketsLost", calculatePacketsLost(path));
    pathData.put("TransmissionDelay", calculateTransmissionDelay(path));
    pathData.put("HopCount", path.links().size());
    return pathData;
}).collect(Collectors.toList());

log.info("Paths with characteristics between {} and {}: {}", src, dst,
pathsData);

return Response.ok(pathsData).build();
} catch (Exception e) {
    log.error("Failed to retrieve paths", e);
    return
Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
}
}

private double calculateBandwidth(Path path) {
    double totalBytes = 0.0;
    double maxBandwidth = path.maxBandwidth()

    for (Link link : path.links()) {

```

// getBytesTransferred(link) повертає кількість байтів, переданих по
лінку

```
double bytes = getBytesTransferred(link);
totalBytes += bytes;
}
```

// Обчислюємо середню пропускну здатність

```
return totalBytes / maxBandwidth;
}
```

```
private double calculatePacketsLost(Path path) {
```

```
double totalForwardPackets = 0.0;
```

```
double totalBackwardPackets = 0.0;
```

```
for (Link link : path.links()) {
```

// getBackwardPackets(link) повертає кількість пакетів, що пройшли
назад

```
totalForwardPackets += getForwardPackets(link);
```

```
totalBackwardPackets += getBackwardPackets(link);
```

```
}
```

// Обчислюємо втрачені пакети

```
return (totalForwardPackets - totalBackwardPackets) / totalForwardPackets;
```

```
}
```

```
private double calculateTransmissionDelay(Path path) {
```

```

double totalDuration = 0.0;

double meanIAT = 0.0;

for (Link link : path.links()) {
    // getFlowIATMean(link) повертає середній міжпакетний час (IAT)
    totalDuration += getFlowDuration(link);
    meanIAT += getFlowIATMean(link);
}

// Обчислюємо загальний час передачі пакетів
return totalDuration + meanIAT;
}

// Допоміжні методи для отримання даних по лінках
private double getBytesTransferred(Link link) {
    DeviceId srcDeviceId = link.src().deviceId();
    DeviceId dstDeviceId = link.dst().deviceId();

    // Отримуємо статистику завантаження між пристроями
    Load load = statisticService.load(link);

    double bytesTransferred = load.rate(); // Отримуємо швидкість передачі у
біт/с

    log.debug("Bytes transferred on link {}-{}: {}", srcDeviceId, dstDeviceId,
bytesTransferred);

    return bytesTransferred / 8.0; // Перетворюємо біт/с в байти/с

```

```
}
```

```
private double getForwardPackets(Link link) {
    DeviceId srcDeviceId = link.src().deviceId();

    // Отримуємо всі потоки на пристрої та рахуємо пакети
    return flowRuleService.getFlowEntries(srcDeviceId).stream()
        .mapToDouble(FlowEntry::packets) // Загальна кількість пакетів
        вперед
        .sum();
}
```

```
private double getBackwardPackets(Link link) {
    DeviceId dstDeviceId = link.dst().deviceId();

    // Отримуємо всі потоки на пристрої та рахуємо пакети
    return flowRuleService.getFlowEntries(dstDeviceId).stream()
        .mapToDouble(FlowEntry::packets) // Загальна кількість пакетів
        назад
        .sum();
}
```

```
private double getFlowDuration(Link link) {
    DeviceId deviceId = link.src().deviceId();

    // Отримуємо всі потоки на пристрої та обчислюємо тривалість потоку
```

```

return flowRuleService.getFlowEntries(deviceId).stream()

    .mapToDouble(FlowEntry::life) // Час життя потоку у секундах

    .average()

    .orElse(0.0); // Значення за замовчуванням
}

private double getFlowIATMean(Link link) {
    DeviceId deviceId = link.src().deviceId();

    // Отримуємо середнє міжпакетне значення IAT (час)
    return flowRuleService.getFlowEntries(deviceId).stream()

        .mapToDouble(flow -> {

            // Обчислюємо середній міжпакетний час

            double packets = flow.packets();

            double duration = flow.life(); // у секундах

            return packets > 0 ? (duration / packets) : 0.0;

        })

        .average()

        .orElse(0.0); // Значення за замовчуванням
}

// Внутрішній клас для обробки нових пакетів і встановлення src та dst
private class InternalPacketProcessor implements PacketProcessor {
    static HostId currentSrc;

    static HostId currentDst;

```



```

@Override

public void process(PacketContext context) {

    if (context.isHandled()) return;

    Ethernet eth = context.inPacket().parsed();

    if (eth == null) return;

    // Визначаємо src та dst на основі MAC-адрес з Ethernet-фрейму
    currentSrc = HostId.hostId(eth.getSourceMAC());
    currentDst = HostId.hostId(eth.getDestinationMAC());

    log.info("New packet processed - Source: {}, Destination: {}", currentSrc,
currentDst);

    }

}

```

```

@POST

```

```

@Path("/setOptimalPath")

```

```

@Consumes(MediaType.APPLICATION_JSON)

```

```

public Response setOptimalPath(OptimalPathRequest request) {

```

```

    try {

```

```

        // Знаходимо та застосовуємо обраний маршрут

```

```

        Path optimalPath = findPathById(request.getPathId());

```

```

        if (optimalPath != null) {

```

```

            applyOptimalPath(optimalPath);

```

```

        log.info("Optimal path {} applied.", request.getPathId());
        return Response.ok().build();
    } else {
        log.warn("Optimal path not found: {}", request.getPathId());
        return Response.status(Response.Status.NOT_FOUND).build();
    }
} catch (Exception e) {
    log.error("Failed to apply optimal path", e);
    return
Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
}
}
}

```

Лістинг Б.1 — Модуль нейронної мережі

```

import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import tensorflow as tf

```

Налаштування TensorFlow для використання CPU

```

tf.config.threading.set_intra_op_parallelism_threads(4)
tf.config.threading.set_inter_op_parallelism_threads(4)

# Виведення списку файлів у директорії input
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# Зчитування даних
pd.set_option('display.float_format', '{:.2f}'.format)
dataset = pd.read_csv("../input/ip-network-traffic-flows-labeled-with-87-
apps/Dataset-Unicauca-Version2-87Atts.csv")

# Видалення дублюючих колонок
dataset = dataset.loc[:, ~dataset.columns.duplicated()]

# Перетворення значень Infinity на -1 та NaN на 0
dataset.replace([np.inf, -np.inf, np.nan], [-1, -1, 0], inplace=True)

# Створення об'єкту LabelEncoder
le = LabelEncoder()

# Перетворення категоріальних ознак в числові
for column in ['Timestamp', 'Label', 'ProtocolName']:
    dataset[column] = le.fit_transform(dataset[column])

```

```
# Обчислення ознак
```

```
maxBandwidth = 1000000000
```

```
initial_ttl = 64
```

```
dataset['Bandwidth'] = dataset['Flow.Bytes.s'] / maxBandwidth
```

```
dataset['Packets.Lost'] = (dataset['Total.Fwd.Packets'] -  
dataset['New.Total.Backward.Packets']) / dataset['Total.Fwd.Packets']
```

```
dataset['Transmission.Delay'] = dataset['Flow.Duration'] +  
dataset['Flow.IAT.Mean']
```

```
dataset['Hop.Count'] = initial_ttl - dataset['TTL']
```

```
# Вибір ознак для моделі
```

```
features = ['Bandwidth', 'Packets.Lost', 'Transmission.Delay', 'Hop.Count']
```

```
dataX = dataset[features]
```

```
# Перетворення цільової змінної на числову
```

```
# le_optimal_path = LabelEncoder()
```

```
dataY = dataset['Bandwidth']
```

```
#le_optimal_path.fit_transform(dataset['Flow.ID'])
```

```
# Поділ даних на навчальну та тестову вибірки
```

```
trainX, testX, trainY, testY = train_test_split(dataX, dataY, test_size=0.2,  
random_state=42)
```

```
# Нормалізація даних
```

```
scaler = StandardScaler()
```

```
X_train_scaled = scaler.fit_transform(trainX)
X_test_scaled = scaler.transform(testX)

# Побудова моделі нейронної мережі
model = Sequential()
model.add(Input(shape=(X_train_scaled.shape[1],)))
model.add(Dense(7, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(7, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='linear')) # Вихідний шар для регресії

# Компіляція моделі
model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.001),
metrics=['mean_absolute_error'])

# Навчання моделі
history = model.fit(X_train_scaled, trainY, epochs=25, batch_size=32,
validation_split=0.2)

# Збереження моделі
model.save('optimal_path_model.h5')

# Оцінка моделі
loss, mae = model.evaluate(X_test_scaled, testY)
print(f'Mean Absolute Error: {mae}')
```

```
# Візуалізація кривих навчання
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

Лістинг В.1 — Модуль користувацького інтерфейсу

gui.py

```
import customtkinter as ctk

from tkinter import messagebox, filedialog
from session_manager import SessionManager

class PathOptimizerGUI(ctk.CTk):

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

        self.session_manager = SessionManager() # Додано менеджер сесій
        self.active_sessions_checkboxes = {}

        ctk.set_appearance_mode("System")
        ctk.set_default_color_theme("dark-blue")
```

```
self.title("LBBNN GUI")
```

```
self.geometry(f'{800}x580')
```

```
self.configure_layout()
```

```
self.create_sidebar()
```

```
self.create_tabview()
```

```
self.appearance_mode_optionmenu.set("System")
```

```
self.scaling_optionmenu.set("100%")
```

```
def configure_layout(self):
```

```
    self.grid_columnconfigure(1, weight=1)
```

```
    self.grid_rowconfigure((0, 1, 2), weight=1)
```

```
def create_sidebar(self):
```

```
    self.sidebar_frame = ctk.CTkFrame(self, width=140, corner_radius=0)
```

```
    self.sidebar_frame.grid(row=0, column=0, rowspan=4, sticky="nsew")
```

```
    self.sidebar_frame.grid_rowconfigure(4, weight=1)
```

```
    self.add_sidebar_label("Menu", 0, "bold")
```

```
    self.add_sidebar_button("New Connection", 1,  
self.show_new_connection_tab)
```

```
    self.add_sidebar_button("Active Sessions", 2, self.show_active_sessions_tab)
```

```
    self.add_sidebar_button("About", 3, self.show_about_tab)
```

```

self.add_sidebar_options()

def add_sidebar_label(self, text, row, font=None):
    ctk.CTkLabel(self.sidebar_frame, text=text, font=ctk.CTkFont(size=20,
weight=font) if font else None) \
        .grid(row=row, column=0, padx=20, pady=(10, 10))

def add_sidebar_button(self, text, row, command=None):
    ctk.CTkButton(self.sidebar_frame, text=text, command=command) \
        .grid(row=row, column=0, padx=20, pady=(10, 10))

def add_sidebar_options(self):
    self.add_sidebar_label("Appearance Mode:", 5)

    self.appearance_mode_optionemenu =
ctk.CTkOptionMenu(self.sidebar_frame, values=["Light", "Dark", "System"],
command=self.change_appearance_mode_event)

    self.appearance_mode_optionemenu.grid(row=6, column=0, padx=20,
pady=(10, 10))

    self.add_sidebar_label("UI Scaling:", 7)

    self.scaling_optionemenu = ctk.CTkOptionMenu(self.sidebar_frame,
values=["80%", "90%", "100%", "110%", "120%"],
command=self.change_scaling_event)

    self.scaling_optionemenu.grid(row=8, column=0, padx=20, pady=(10, 20))

def create_tabview(self):

```



```

self.tabview = ctk.CTkTabview(self, width=250)

self.tabview.grid(row=0, rowspan=3, column=1, columnspan=3, padx=(20,
20), pady=(5, 20), sticky="nsew")

self.tabview.add("New Connection")

self.tabview.add("Active Sessions")

self.tabview.add("About")


self.setup_new_connection_tab()

self.setup_active_sessions_tab()

self.setup_about_tab()


def setup_new_connection_tab(self):

    new_connection_frame = self.create_frame_in_tab("New Connection")


    self.add_config_label(new_connection_frame, "Configure SDN Controller",
0, 0)


    self.ip_entry = self.create_labeled_entry(new_connection_frame, "IP
Address:", 1, 0)

    self.port_entry = self.create_labeled_entry(new_connection_frame, "Port:", 2,
0)


    self.add_config_label(new_connection_frame, "Configure Path
Characteristics", 0, 3)


    option_values = ['Bandwidth', 'Packets Lost', 'Transmission Delay', 'Hop
Count']

```

```

self.optionmenu_vars = [ctk.StringVar(value=val) for val in option_values]

for i, var in enumerate(self.optionmenu_vars, start=1):
    self.create_labeled_optionmenu(new_connection_frame, f"Priority {i}", i,
3, var, option_values)

    ctk.CTkButton(new_connection_frame, text="Add Connection",
command=self.add_connection).grid(row=4, column=0, columnspan=2, padx=10,
pady=(10, 0))

def setup_active_sessions_tab(self):
    self.active_sessions_frame = self.create_frame_in_tab("Active Sessions")
    self.active_sessions = []

    self.active_sessions_scrollable_frame =
ctk.CTkScrollableFrame(self.active_sessions_frame)

    self.active_sessions_scrollable_frame.grid(row=0, column=0, padx=10,
pady=10, sticky="nsew")

    self.buttons_frame = ctk.CTkFrame(self.active_sessions_frame)
    self.buttons_frame.grid(row=0, column=1, padx=10, pady=10, sticky="n")

    self.add_button_to_frame("New Session", self.show_new_connection_tab,
self.buttons_frame, 0)

    self.add_button_to_frame("Remove Session", self.remove_session,
self.buttons_frame, 1)

    self.add_button_to_frame("Export Logs", self.export_logs,
self.buttons_frame, 2)

```

```

def setup_about_tab(self):

    about_frame = self.create_frame_in_tab("About")

    ctk.CTkLabel(about_frame, text="This is where you will provide your
instructions.", anchor="w").grid(row=0, column=0, padx=10, pady=10,
sticky="w")


def create_frame_in_tab(self, tab_name):

    frame = ctk.CTkFrame(self.tabview.tab(tab_name))

    frame.grid(row=0, column=0, padx=10, pady=10, sticky="nsew")

    frame.grid_columnconfigure(0, weight=1)

    return frame


def create_labeled_entry(self, parent, label_text, row, col):

    ctk.CTkLabel(parent, text=label_text).grid(row=row, column=col, padx=10,
pady=10, sticky="w")

    entry = ctk.CTkEntry(parent)

    entry.grid(row=row, column=col + 1, padx=20, pady=10, sticky="ew")

    return entry


def create_labeled_optionmenu(self, parent, label_text, row, col, var, values):

    ctk.CTkLabel(parent, text=label_text).grid(row=row, column=col, padx=10,
pady=10, sticky="w")

    ctk.CTkOptionMenu(parent, variable=var, dynamic_resizing=False,
values=values).grid(row=row, column=col + 1, padx=10, pady=10)


def add_config_label(self, parent, text, row, col):

```

```

    ctk.CTkLabel(parent, text=text, anchor="center").grid(row=row, column=col,
columnspan=2, padx=10, pady=10, sticky="w")

```

```

def add_button_to_frame(self, text, command, frame, row):

```

```

    ctk.CTkButton(frame, text=text, command=command).grid(row=row,
column=0, padx=10, pady=(10, 0))

```

```

def add_connection(self):

```

```

    ip = self.ip_entry.get()

```

```

    port = self.port_entry.get()

```

```

    options = [var.get() for var in self.optionmenu_vars]

```

```

    if not self.validate_ip(ip):

```

```

        messagebox.showerror("Error", "Invalid IP address.")

```

```

        return

```

```

    if not port.isdigit() or not (0 < int(port) < 65536):

```

```

        messagebox.showerror("Error", "Port must be a number between 1 and
65535.")

```

```

        return

```

```

    if len(options) != len(set(options)):

```

```

        messagebox.showwarning("Warning", "Duplicate options selected. Please
choose different options.")

```

```

        return

```

```

# Створення сесії

```

```

if self.session_manager.add_session(ip, port, self.optionmenu_vars):
    self.add_active_session_checkbox(f'{ip}:{port}')
    messagebox.showinfo("Success", "Connection established.")
else:
    messagebox.showerror("Error", "Failed to establish connection.")

def add_active_session_checkbox(self, session_key):
    checkbox = ctk.CTkCheckBox(self.active_sessions_scrollable_frame,
text=session_key)

    checkbox.grid(sticky="w")

    self.active_sessions_checkboxes[session_key] = checkbox

def validate_ip(self, ip):
    parts = ip.split('.')

    return len(parts) == 4 and all(part.isdigit() and 0 <= int(part) <= 255 for part
in parts)

def change_appearance_mode_event(self, new_appearance_mode: str):
    ctk.set_appearance_mode(new_appearance_mode)

def change_scaling_event(self, new_scaling: str):
    ctk.set_widget_scaling(int(new_scaling.replace("%", "")) / 100)

def show_new_connection_tab(self):
    self.tabview.set("New Connection")

```

```
def show_active_sessions_tab(self):
    self.tabview.set("Active Sessions")

def show_about_tab(self):
    self.tabview.set("About")

def remove_session(self):
    selected_session = self.get_selected_session()
    if not selected_session:
        messagebox.showwarning("Warning", "No session selected for removal.")
        return
    self.session_manager.remove_session(selected_session)
    self.active_sessions_checkboxes[selected_session].destroy()
    del self.active_sessions_checkboxes[selected_session]
    messagebox.showinfo("Success", f"Session {selected_session} removed.")

def get_selected_session(self):
    for session_key, checkbox in self.active_sessions_checkboxes.items():
        if checkbox.get() == 1: # Якщо чекбокс вибраний
            return session_key
    return None

def export_logs(self):
    selected_session = self.get_selected_session()
    if not selected_session:
```

```
        messagebox.showwarning("Warning", "No session selected for log
export.")
```

```
        return
```

```
        # Запитуємо шлях до файлу
```

```
        file_path = filedialog.asksaveasfilename(defaultextension=".txt",
                                                filetypes=[("Text files", "*.txt"), ("All files",
                                                "**.*")])
```

```
        if file_path:
```

```
            self.session_manager.export_logs(selected_session, file_path)
```

```
            messagebox.showinfo("Success", f"Logs exported to {file_path}.")
```

```
sdn_controller.py
```

```
import requests
```

```
import threading
```

```
import time
```

```
import numpy as np
```

```
import tensorflow as tf
```

```
class SDNControllerSession:
```

```
    def __init__(self, ip, port, logger, priority_vars):
```

```
        self.ip = ip
```

```
        self.port = port
```

```
        self.logger = logger
```

```
        self.priority_vars = priority_vars
```

```
        self.paths_data = []
```

```

self.active = True

self.model = tf.keras.models.load_model('optimal_path_model.h5')

def connect(self):
    try:
        url = f'http://{self.ip}:{self.port}/onos/v1/route/getPaths'
        response = requests.get(url)
        response.raise_for_status()
        self.paths_data = response.json()
        self.logger.log_request(self.ip, "Initial Request", self.paths_data)
        threading.Thread(target=self.continuous_requesting, daemon=True).start()
        return True
    except requests.RequestException as e:
        self.logger.log_request(self.ip, "Connection Error", str(e))
        return False

def continuous_requesting(self):
    while self.active:
        try:
            url = f'http://{self.ip}:{self.port}/onos/v1/route/getPaths'
            response = requests.get(url)
            response.raise_for_status()
            self.paths_data = response.json()
            self.logger.log_request(self.ip, "Path Request", self.paths_data)

```



```

# Розраховуємо оптимальний шлях
optimal_path = self.predict_optimal_path(self.paths_data)
self.logger.log_request(self.ip, "Optimal Path Sent", optimal_path)

post_url = f'http://{self.ip}:{self.port}/onos/v1/route/setOptimalPath'
requests.post(post_url, json={"pathId": optimal_path['pathId']})
except requests.RequestException as e:
    self.logger.log_request(self.ip, "Error", str(e))

time.sleep(1)

def predict_optimal_path(self, paths_data):
    input_data = np.array([
        [path['Bandwidth'], path['PacketsLost'], path['TransmissionDelay'],
        path['HopCount']]
        for path in paths_data
    ])
    selected_priorities = [self.priority_vars[i].get() for i in range(4)]
    priority_mapping = {
        'Bandwidth': 0,
        'Packets Lost': 1,
        'Transmission Delay': 2,
        'Hop Count': 3
    }

    reordered_input_data = np.array([

```

```

        [path[priority_mapping[selected_priorities[0]]],
         path[priority_mapping[selected_priorities[1]]],
         path[priority_mapping[selected_priorities[2]]],
         path[priority_mapping[selected_priorities[3]]]]
    for path in input_data
])

```

```

predictions = self.model.predict(reordered_input_data)
optimal_index = np.argmin(predictions)
return paths_data[optimal_index]

```

```

def disconnect(self):
    self.active = False
    self.logger.log_request(self.ip, "Disconnected", "Session disconnected")

```

session_manager.py

```

from sdn_controller import SDNControllerSession

```

```

from logger import Logger

```

```

class SessionManager:

```

```

    def __init__(self):
        self.sessions = {}
        self.loggers = {}

```

```

    def add_session(self, ip, port, priority_vars):

```

```

if f"{ip}:{port}" in self.sessions:
    return False

logger = Logger(f"{ip}:{port}")
session = SDNControllerSession(ip, port, logger, priority_vars)
if session.connect():
    self.sessions[f"{ip}:{port}"] = session
    self.loggers[f"{ip}:{port}"] = logger
    return True
return False

def select_route_and_send_packet(self, session_key, src_host, dst_host):
    session = self.sessions.get(session_key)
    if session:
        session.select_route_and_send_packet(src_host, dst_host)

def remove_session(self, session_key):
    if session_key in self.sessions:
        self.sessions[session_key].disconnect()
        del self.sessions[session_key]
        del self.loggers[session_key]

def export_logs(self, session_key, file_path, last_n=100):
    if session_key in self.loggers:
        self.loggers[session_key].export_logs(file_path, last_n)

```

ДОДАТОК Б

Список публікацій здобувача

1. Kulakov, Y., & **Korenko, D.** (2021). Modified Method of Traffic Engineering in DCN with a Ramified Topology. *International Journal of Advanced Computer Science and Applications*, 12(12).
2. Loutskii, H., Volokyta, A., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & **Korenko, D.** (2021). Topology synthesis method based on excess de bruijn and dragonfly. In *Advances in Computer Science for Engineering and Education IV* (pp. 315-325). Springer International Publishing.
3. Volokyta A., Loutskii H., Rehida P., Honcharenko O., **Korenko D.**, Rusinov V., Ivanishchev B., Kaplunov A. Convolutionary neural networks regarding problem of monitoring data balancing in de bruijn topology. *Bulgarian Journal for Engineering Design*, 2021, Mechanical Engineering Faculty, Technical University-Sofia. ISSN 1313-7530
4. Volokyta, A., Loutskii, H., Rehida, P., Kaplunov, A., Ivanishchev, B., Honcharenko, O., & **Korenko, D.** (2022). Extended DragonDeBruijn topology synthesis method. *International Journal of Computer Network and Information Security*, 9(6), 23.
5. **Korenko, D.**, Cherevatenko, O., Rusinov, V., & Kulakov, Y. (2022). Creation of the method of multipath routing using known paths in software-defined networks. *Technology audit and production reserves*, 4(2/66), 19-24.
6. Kulakov, Y. O., & **Korenko, D. V.** Methods of applying artificial intelligence in software-defined networks. *Problems of Informatization and Control*, 1(73), 2023, 23-27.
7. Artem Volokyta, Alla Kogan, Oleksii Cherevatenko, **Dmytro Korenko**, Dmytro Oboznyi, Yurii Kulakov, "Traffic Engineering with

Specified Quality of Service Parameters in Software-defined Networks", International Journal of Computer Network and Information Security(IJCNIS), Vol.16, No.5, pp.1-13, 2024. DOI:10.5815/ijcnis.2024.05.01

8. Кулаков Ю. О., Коренко Д. В. Метод балансування навантаження в мережах SDN з використанням штучного інтелекту. Проблеми інформатизації та управління, 2(78), 2024, ст. 31-39. DOI: <https://doi.org/10.18372/2073-4751.78.18959> ISSN 2073-4751