МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УЖГОРОДСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра інформаційних управляючих систем та технологій

ПАРАСКА БОГДАН ВОЛОДИМИРОВИЧ

Дослідження методів навчання нейромереж у задачах прогнозування цін

122 Комп’ютерні науки

Дипломна робота на здобуття освітнього ступеня
бакалавра

Науковий керівник:
Ніколенко Володимир
Володимирович
к.ф-м.н., доц.

Ужгород – 2025

Реєстрація _____

(номер)

«____» червня 2025 р. _____ Рижак Е.М.

(підпис)

Дипломна робота допущена до захисту

Завідувач кафедри

_____ Міца О.В.

(підпис)

доктор технічних наук, професор

«____» червня 2025 р.

Рецензент _____ Мич І. А.

(підпис)

к.ф-м.н., доц,

ПІБ: Параска Богдан Володимирович

Назва: Дослідження методів навчання нейромереж у задачах прогнозування цін

Факультет: Інформаційних технологій

Спеціальність: 122 «Комп’ютерні науки»

Науковий керівник: к. ф. м. н., доцент Ніколенко В. В.

За темою роботи опубліковано 1 статтю.

Анотація

У цій бакалаврській праці було проаналізовано продуктивність різних архітектур нейронних мереж у завданні передбачення ціни криптовалюти Біткоїн. Основна увага була зосереджена на застосуванні та порівнянні моделей: Багатошаровий перцептрон (MLP), Мережі довгої короткочасної пам'яті (LSTM), Згорткові нейронні мережі (CNN) та Трансформер (Informer). Розробка та тестування всіх моделей проводились за допомогою мови програмування Python.

Робота розподілена на два розділи. У першому розділі подано загальну інформацію про штучні нейронні мережі, їх архітектуру та принципи роботи, а також детальний огляд теоретичних засад методів, використаних у дослідженні. Другий розділ зосереджується на практичній частині: опис структури проекту, етапи підготовки та обробки даних, етапи розробки та навчання нейронних мереж. Тут також представлено робоче середовище, мову програмування та ключові використані бібліотеки. Проведено ретельний порівняльний аналіз результатів роботи моделей на тестових даних, оцінюючи їх здатність прогнозувати тенденцію зміни ціни.

Представлена бакалаврська робота містить: сторінок – 64, розділів – 2, ілюстрацій –12, кількість використаних джерел –9.

Ключові слова: нейронні мережі, прогнозування цін, часові ряди, MLP, LSTM, CNN, Transformer, машинне навчання.

Abstract

This bachelor's thesis investigates the effectiveness of various neural network architectures in the task of Bitcoin price prediction. The main focus was on the application and comparative analysis of models such as Multilayer Perceptron (MLP), Long Short-Term Memory (LSTM) networks, Convolutional Neural Networks (CNN), and Transformer (Informer). All models were developed and tested using the Python programming language.

The work consists of two chapters. The first chapter provides general information about artificial neural networks, their architecture, and operating principles, along with a detailed description of the theoretical foundations of the methods used in this research. The second chapter is dedicated to the practical part of the work, covering the project structure, data preparation and preprocessing steps, and the creation and training processes of the developed neural networks. This chapter also outlines the development environment, programming language, and key libraries utilized. A detailed comparative analysis of the models' performance on test data is presented, assessing their ability to predict price movement direction.

This bachelor's thesis includes: 63 pages, 2 chapters, 12 figures, and 9 references.

Keywords: neural networks, price prediction, time series, MLP, LSTM, CNN, Transformer, machine learning.

ЗМІСТ

ВСТУП

Прогнозування цін є одним із ключових завдань у фінансовій аналітиці, торгівлі та економіці загалом. У сучасних умовах, коли обсяг і складність даних зростають, все більшого значення набувають методи штучного інтелекту, зокрема нейронні мережі. Вони демонструють високу здатність до виявлення складних залежностей у часових рядах і дозволяють досягати точніших результатів у порівнянні з класичними статистичними методами.

Метою цього проєкту є дослідження ефективності різних архітектур нейронних мереж у задачах прогнозування цін (на прикладі криптовалют Bitcoin та Ethereum). Передбачається реалізація, тестування та порівняння таких моделей, як багатошаровий перцептрон (MLP), рекурентна нейронна мережа LSTM, згорткова нейронна мережа (CNN) та трансформери. Для досягнення цієї мети заплановано виконання наступних завдань:

Аналіз вимог до дипломного проєкту**:** буде сформульовано чіткі цілі дослідження, обсяг задач і обґрунтовано вибір теми з урахуванням актуальності прогнозування цін у сучасному цифровому середовищі.

Огляд існуючих підходів та технологій**:** проведено аналіз сучасних методів машинного навчання, що застосовуються для аналізу часових рядів, зокрема в контексті прогнозування фінансових показників. Обрані методи буде обґрунтовано на основі їх точності, складності реалізації та практичної придатності.

Підготовка даних та реалізація моделей**:** виконано обробку часових рядів, побудовано архітектури обраних моделей у середовищі Python з використанням бібліотек PyTorch, scikit-learn та інших.

Навчання та тестування моделей**:** буде проведено тренування моделей на історичних даних із подальшим тестуванням їхньої точності, стійкості та здатності до генералізації.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Поняття штучних нейронних мереж

Штучна нейронна мережа (ШНМ) - це математична конструкція, що копіює принципи роботи біологічного мозку. Вона базується на великій кількості взаємопов'язаних вузлів, так званих "нейронів", які організовані у шари: вхідний, один чи декілька прихованих та вихідний. Така мережа здатна вчитися на прикладах, виявляючи закономірності у даних, та згодом робити прогнози або класифікувати нову інформацію.

Загалом, структура ШНМ має високу гнучкість і може бути пристосована для різноманітних задач - від розпізнавання зображень до передбачення часових рядів. Її ефективність значною мірою залежить від правильно обраної архітектури та методу навчання. Основними компонентами ШНМ є нейрони, вагові коефіцієнти, функція активації та алгоритм оптимізації. Кожен нейрон отримує сигнали від попередніх шарів, зважує їх, підсумовує та передає результат далі.

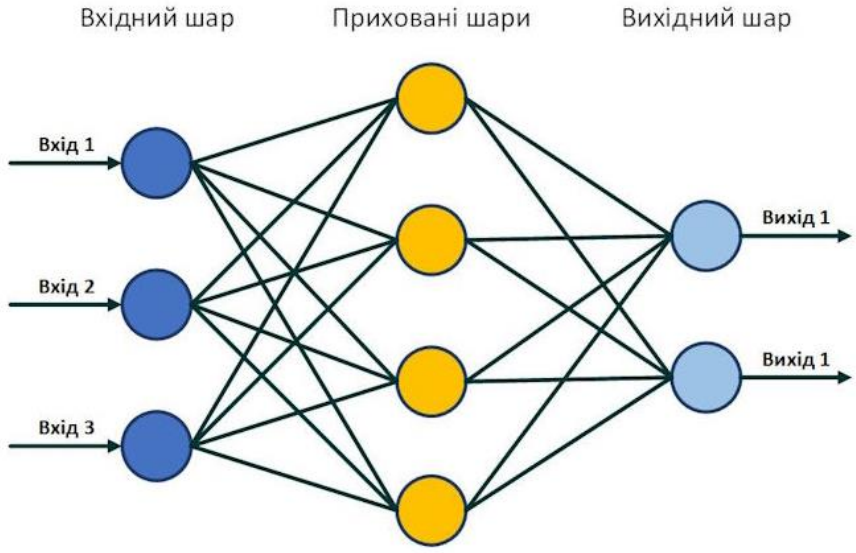


Рисунок 1.1. Структура найпростішої нейронної мережі

На Рисунок 1.1 представлено одношарову нейронну мережу, яка складається з одного входу, одного прихованого шару та одного виходу. Кожен зв'язок має вагу, яка змінюється в процесі навчання.

Навчання ШНМ є процесом налаштування ваг, у ході якого помилка між фактичним та очікуваним результатом мінімізується. Це може досягатися за допомогою різних методів – найпоширенішими з них є градієнтний спуск та його варіації. Під час зворотного поширення помилки (backpropagation) мережа "навчається", змінюючи ваги у напрямку зменшення помилки.

На Рисунок 1.2 продемонстровано, як нейронна мережа обчислює похибку, а потім повертається шарами, змінюючи ваги для покращення точності.

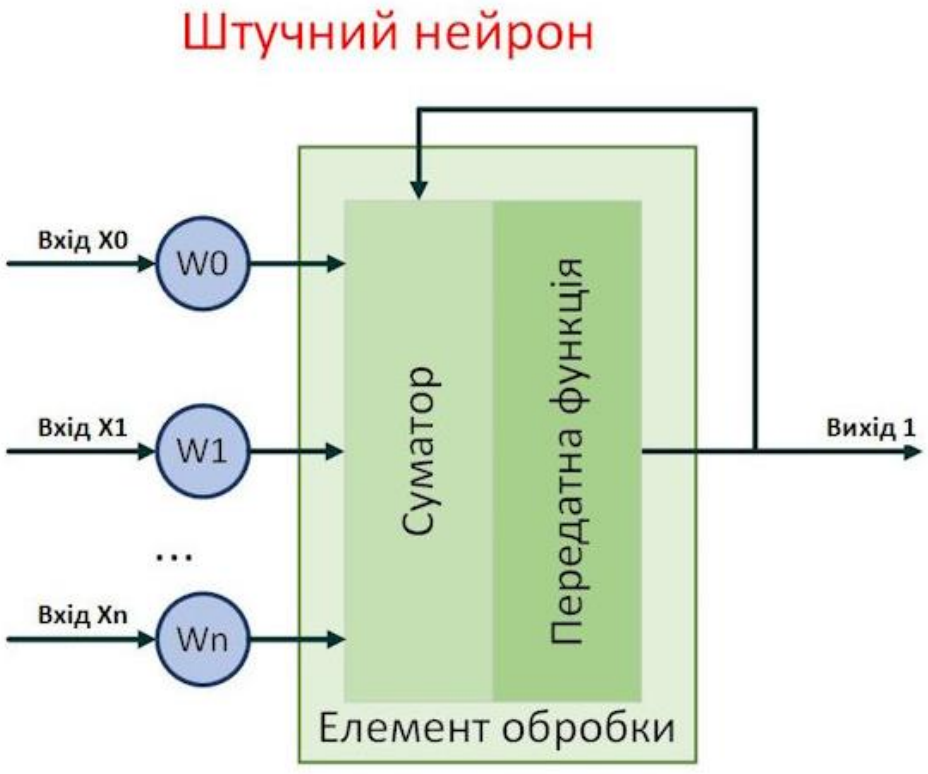


Рисунок 1.2 Схематичне зображення процесу навчання мережі за допомогою зворотного поширення помилки

Ключовим аспектом при побудові ефективної моделі є вибір архітектури нейромережі. Просте збільшення кількості шарів або нейронів не завжди приносить кращі результати – важливо уникати як недонавчання (underfitting), так і перенавчання (overfitting). Для цього використовуються спеціальні техніки, серед яких регуляризація, дропаут або крос-валідація.

ШНМ активно застосовуються у задачах прогнозування цін через їх здатність розпізнавати складні нелінійні залежності в даних. Особливо ефективними є такі модифікації як рекурентні нейронні мережі (RNN), довготривала короткострокова пам'ять (LSTM) та трансформери.

Отже, штучні нейронні мережі – потужний інструмент в аналітиці та прогнозуванні. У наступних розділах буде розглянуто класифікацію типів ШНМ, методи їх навчання, а також їх застосування у задачах прогнозування економічних показників.

1.2 Основи машинного навчання як фундаменту для нейромереж

Машинне навчання (ML) є невід’ємною складовою штучного інтелекту та одночасно основою для побудови та навчання штучних нейронних мереж. Цей підхід дозволяє алгоритмам «вчитися» на основі даних, без явного програмування кожного кроку розв’язання задачі. У загальному сенсі, машинне навчання – це здатність комп’ютерних систем знаходити закономірності у даних і використовувати ці закономірності для прийняття рішень, прогнозів чи класифікацій.

машинне навчання визначається як: Метод аналізу даних, який автоматично будує аналітичну модель, використовуючи алгоритми, що ітеративно навчаються на даних. Це дозволяє комп’ютерам знаходити приховані інсайти без явного програмування, де шукати.

Це визначення добре підкреслює ключову перевагу ML – здатність до самостійного вдосконалення на основі досвіду.

Класифікація видів машинного навчання агалом машинне навчання поділяється на три основні типи:

- Навчання з учителем (Supervised Learning)
- Навчання без учителя (Unsupervised Learning)
- Підкріплювальне навчання (Reinforcement Learning)

На Рисунок 1.3 зображено порівняння основних підходів ML. У таблиці наведено їхні характеристики, застосування та приклади.

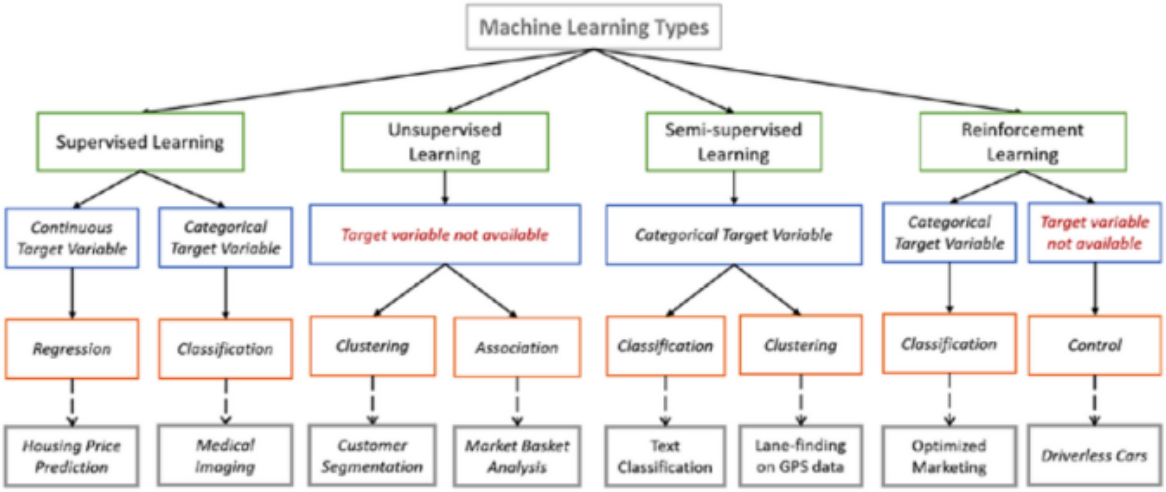


Рисунок 1.3 Схема типів машинного навчання

Навчання з учителем. Це найбільш поширений тип ML, при якому алгоритм навчається на заздалегідь маркованих даних. Іншими словами, кожному прикладу у вхідному наборі дано правильну відповідь. Система вивчає зв’язки між вхідними даними та виходами, щоб у майбутньому застосовувати ці зв’язки до нових випадків.

Приклади задач:

- Класифікація електронних листів як «спам» або «не спам»
- Прогнозування вартості нерухомості за площею, розташуванням тощо

На Рисунок 1.4 показано, як система використовує позначені дані для побудови моделі.

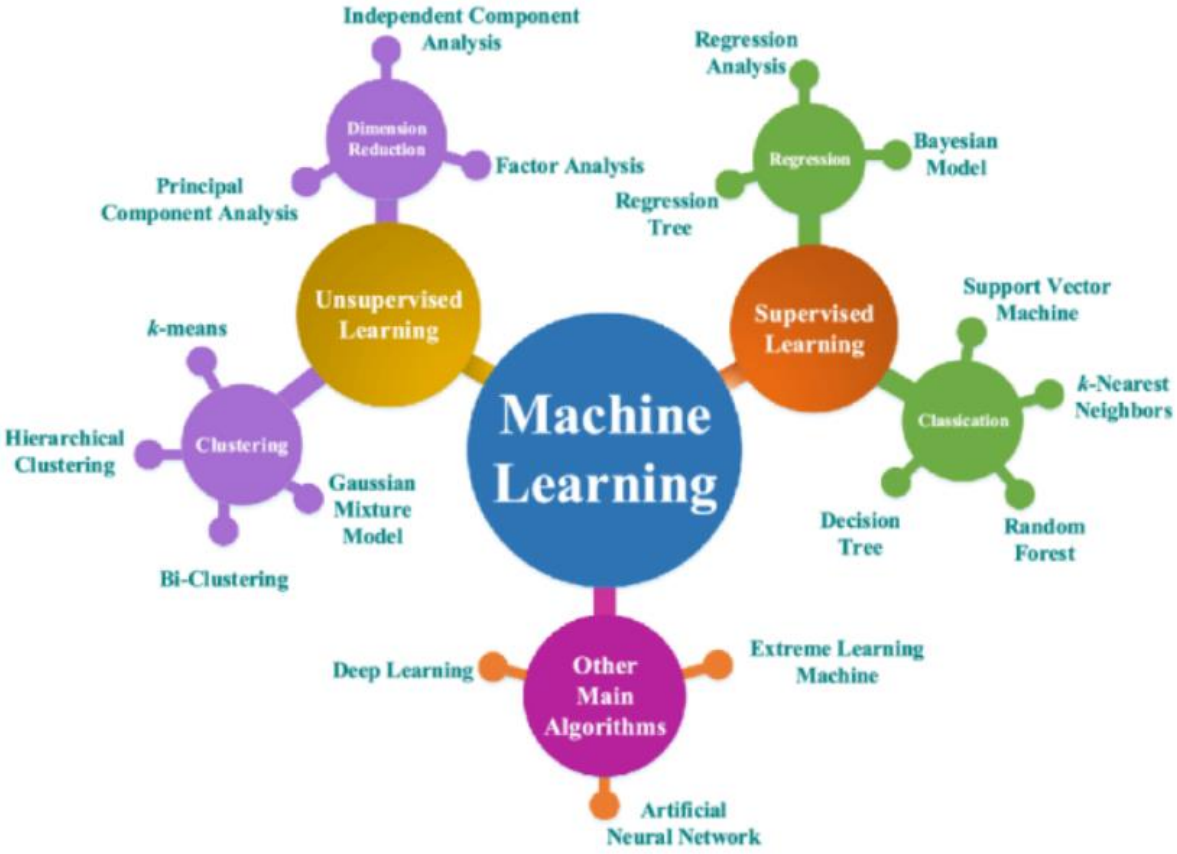


Рисунок 1.4 Схема роботи алгоритму з учителем

Навчання без наглядача (Unsupervised learning). Це різновид машинного навчання, де алгоритм опановує інформацію на основі масиву даних без маркування. Основна ціль алгоритму – самостійно виявити шаблони й віднайти структури в даних, згрупувати об'єкти за їхньою подібністю та, можливо, передбачити майбутні тенденції.

Навчання без наглядача часто використовується для розподілу даних на кластери за спільними характеристиками (кластеризація), скорочення кількості властивостей даних без втрати значущої інформації (зменшення розмірності) та виявлення нетипових даних або відхилень від загальної тенденції (виявлення аномалій).

До алгоритмів навчання без наглядача зараховують:

- Метод k-середніх - алгоритм кластеризації, який розділяє дані на наперед задану кількість груп.
- Алгоритм наближених k-середніх. Більш оптимізована версія попереднього алгоритму, призначена для обробки великих обсягів даних.
- Аналіз головних компонент. Метод зменшення розмірності, що відбирає найважливіші атрибути з масивів даних.

Також можливе поєднання двох вищезгаданих методів (з учителем та без нього). Такий гібрид, умовно, може бути позначений як "напівнаглядне навчання" (або змішане навчання).

У цьому випадку алгоритм вивчає дані, що містять як розмічені, так і не розмічені властивості. Позначені приклади служать для тренування алгоритму, як у випадку навчання з учителем, тоді як не розмічені використовуються для пошуку закономірностей у даних, подібно до навчання без наглядача.

"Напівнаглядне навчання" може показати високу ефективність у випадках, коли наявний незначний набір розмічених даних та великий обсяг не розмічених. Наприклад, під час самонавчання або взаємодії ML-моделей.

Навчання з підкріпленням (Reinforcement Learning, RL) – це потужна галузь машинного навчання, де інтелектуальний агент навчається ухвалювати оптимальні рішення шляхом безпосередньої взаємодії з динамічним довкіллям. Уявіть собі дресирування домашнього улюбленця: правильна поведінка заохочується ласощами (позитивне підкріплення), а небажана – ігнорується або викликає несхвалення (негативне підкріплення чи його відсутність). Подібним чином, в RL агент виконує певні дії у відповідь на поточний стан довкілля і отримує чисельний сигнал – "винагороду" або "штраф". Головна мета агента – не просто отримати миттєву винагороду, а розробити таку стратегію (політику) дій, яка

максимізує сукупну, довгострокову винагороду. Цей процес відбувається методом проб і помилок, де агент поступово "розуміє", які дії призводять до кращих результатів у різних ситуаціях. На відміну від інших парадигм машинного навчання, таких як навчання з учителем, де алгоритм отримує готові пари "вхід-правильний вихід", або навчання без учителя, де алгоритм шукає приховані структури в нерозмічених даних, RL агент вчиться самостійно, спираючись виключно на зворотний зв'язок від своїх дій.

Ключові переваги навчання з підкріпленням:

Навчання з підкріпленням пропонує низку унікальних переваг, що роблять його незамінним інструментом для вирішення широкого кола завдань:

- Здатність до самонавчання та автономність: Мабуть, найважливішою перевагою є те, що RL-моделі не потребують величезних, ретельно розмічених наборів даних. Агент вчиться, досліджуючи довкілля та отримуючи зворотний зв'язок. Це особливо цінно в ситуаціях, де збір або розмітка даних є надто дорогим, трудомістким або просто неможливим. Наприклад, ігровий ШІ може зіграти мільйони партій проти самого себе, поступово вдосконалюючи свою стратегію, не потребуючи даних про ігри людей-чемпіонів (хоча такі дані можуть прискорити навчання). Ще один приклад – робот, що вчиться ходити: він може починати з хаотичних рухів, але з часом, отримуючи позитивну винагороду за кожен успішний крок і негативну за падіння, він самостійно оптимізує свою ходу.
- Ефективність у надзвичайно складних задачах: RL демонструє вражаючі результати в задачах, де простір можливих рішень є гігантським, а оптимальну стратегію складно або й неможливо формалізувати за допомогою традиційних алгоритмів. Це стосується як ігрових середовищ, так і реальних застосунків. Наприклад, система AlphaGo від DeepMind, навчена за допомогою RL, змогла перемогти найкращих у світі гравців у Го – гру з астрономічною кількістю можливих ходів. Іншим прикладом є

оптимізація роботи центрів обробки даних, де RL-агенти можуть керувати системами охолодження для зменшення енергоспоживання, враховуючи безліч динамічних факторів. Також RL використовується в робототехніці для навчання маніпуляторів складним діям, таким як збирання об'єктів або виконання точних операцій.

- Адаптивність та гнучкість до змін: RL-алгоритми здатні динамічно адаптуватися до змін у довкіллі або в самій задачі. Якщо умови змінюються, агент може скоригувати свою стратегію на основі нового досвіду, щоб продовжувати досягати максимальної винагороди. Це робить їх ідеальними для систем, що функціонують у непередбачуваних або еволюціонуючих середовищах. Наприклад, система управління трафіком на основі RL може адаптуватися до змін у потоках автомобілів протягом дня або до непередбачених подій, таких як дорожні роботи чи аварії, оптимізуючи роботу світлофорів для мінімізації заторів. Персоналізовані системи рекомендацій також можуть використовувати RL, щоб адаптуватися до мінливих інтересів користувача в реальному часі, пропонуючи більш релевантний контент.

Таким чином, навчання з підкріпленням відкриває шлях до створення по-справжньому інтелектуальних систем, здатних навчатися, адаптуватися та вирішувати завдання, які раніше вважалися прерогативою людського інтелекту. Його застосування охоплює все ширший спектр галузей, від ігор та робототехніки до фінансів, медицини та управління складними системами.

Етапи побудови моделі машинного навчання

Якщо узагальнити процес побудови системи машинного навчання, він складається з наступних етапів:

- Збір даних – отримання максимально повного та репрезентативного набору даних.
- Попередня обробка – очищення, нормалізація, кодування.

- Вибір моделі – визначення алгоритму навчання (дерева рішень, нейронні мережі тощо).
- Навчання моделі – власне побудова зв’язків.
- Оцінка точності – перевірка на тестових даних.
- Тестування та впровадження – реальна перевірка роботи системи.

1.3 Типи архітектур нейронних мереж у задачах прогнозування

Багатошаровий перцептрон (MLP, Multi-Layer Perceptron) є однією з найпоширеніших архітектур нейронних мереж, яка використовується в задачах класифікації, регресії та прогнозування. Основна ідея цієї моделі полягає у використанні декількох шарів нейронів, кожен з яких поєднаний із попереднім та наступним шарами через вагові коефіцієнти.

На відміну від простого перцептрона, що складається лише з вхідного та вихідного шарів, MLP має принаймні один прихований шар. Кожен нейрон у прихованому шарі застосовує активаційну функцію до зваженої суми вхідних сигналів. Це дозволяє мережі виявляти складні нелінійні залежності між вхідними і вихідними даними.

MLP працює за наступним принципом: вхідні дані подаються на перший шар, далі вони проходять через приховані шари, кожен з яких трансформує їх за допомогою активаційних функцій, і, врешті-решт, формуються результати на вихідному шарі.

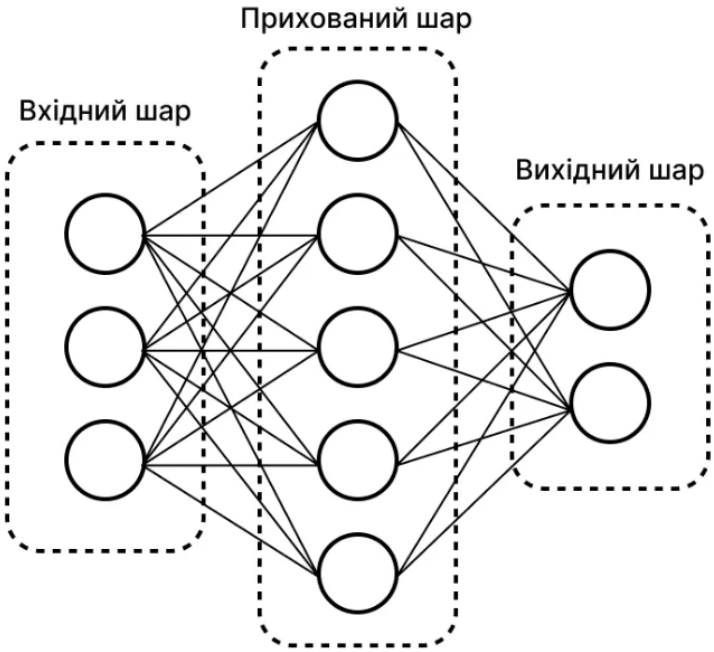


Рисунок 1.5 Схематичне зображення архітектури MLP з вхідним, прихованим і вихідним шарами.

Основні переваги MLP – це гнучкість у моделюванні даних, здатність до апроксимації довільних функцій, а також відносна простота реалізації.

MLP активно використовується у фінансовому секторі для прогнозування змін курсу валют, у медицині – для класифікації результатів діагностики, у промисловості – для прогнозування зносу деталей або якості продукції.

LSTM – Довготривала короткострокова пам’ять

Модель LSTM (Long Short-Term Memory) є різновидом рекурентних нейронних мереж (RNN), що спеціально розроблена для обробки та аналізу послідовностей даних, які мають довгострокові залежності. Звичайні RNN мають проблему з «забуванням» інформації, що була отримана багато кроків тому, а LSTM вирішує цю проблему за рахунок особливої структури своїх комірок пам’яті.

Кожен блок LSTM містить три головні компоненти: вхідний, вихідний і забувальний шлюзи. Ці шлюзи керують тим, яка інформація буде збережена, яка – передана далі, а яка – забута. Таким чином, модель має змогу ефективно зберігати важливу інформацію протягом тривалого часу і водночас відсіювати зайве.

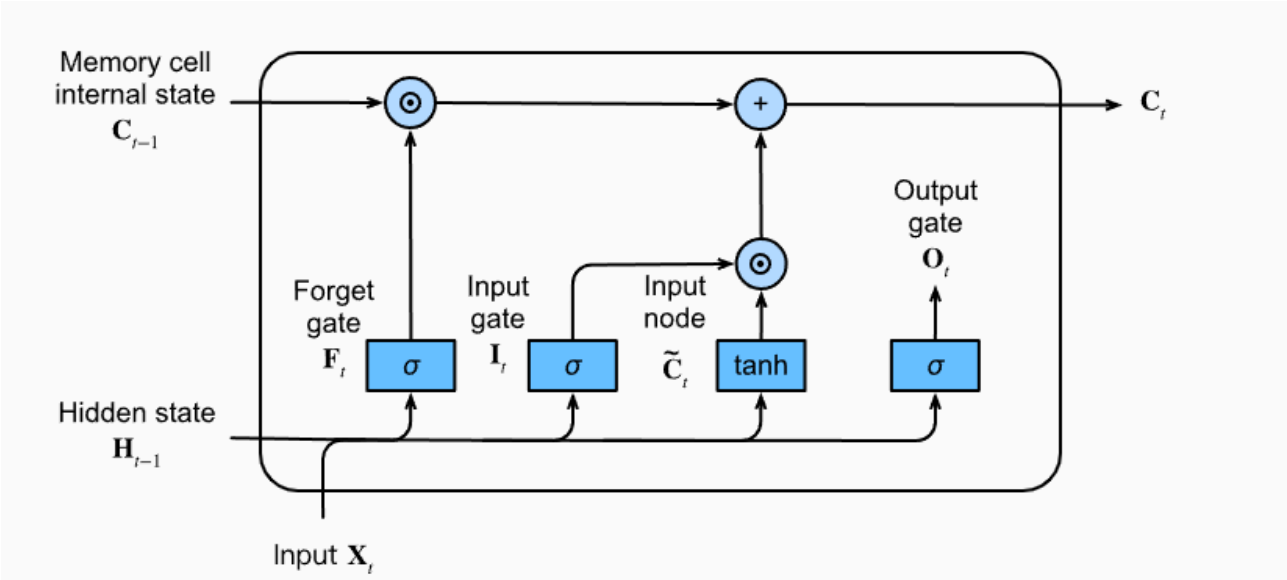


Рисунок 1.6 Схематичне зображення роботи LSTM моделі

LSTM широко застосовується в обробці природної мови, зокрема для перекладу текстів, генерації мови, аналізу емоцій, а також у задачах прогнозування часових рядів – таких як прогноз попиту, цін або погодних умов.

Завдяки здатності працювати з тривалими послідовностями, LSTM значно покращує точність моделей, що мають справу зі складними часовими структурами, де врахування далекого контексту критично важливе.

Згорткова нейронна мережа (CNN)

Згорткові нейронні мережі (CNN, Convolutional Neural Networks) є архітектурою, що спеціально пристосована для обробки даних у вигляді сіток, наприклад зображень або багатовимірних тензорів. Основною

операцією в CNN є згортка, яка дозволяє автоматично виявляти локальні ознаки – краї, текстури, форми тощо.

CNN складається з декількох згорткових шарів, шарів підвибірки (pooling) та повнозв'язних шарів. Кожен згортковий шар застосовує фільтри (ядра), що проходять по входу і створюють нові карти ознак, які потім узагальнюються на наступних шарах. Це дає змогу мережі навчатися складним ієрархічним представленням даних.

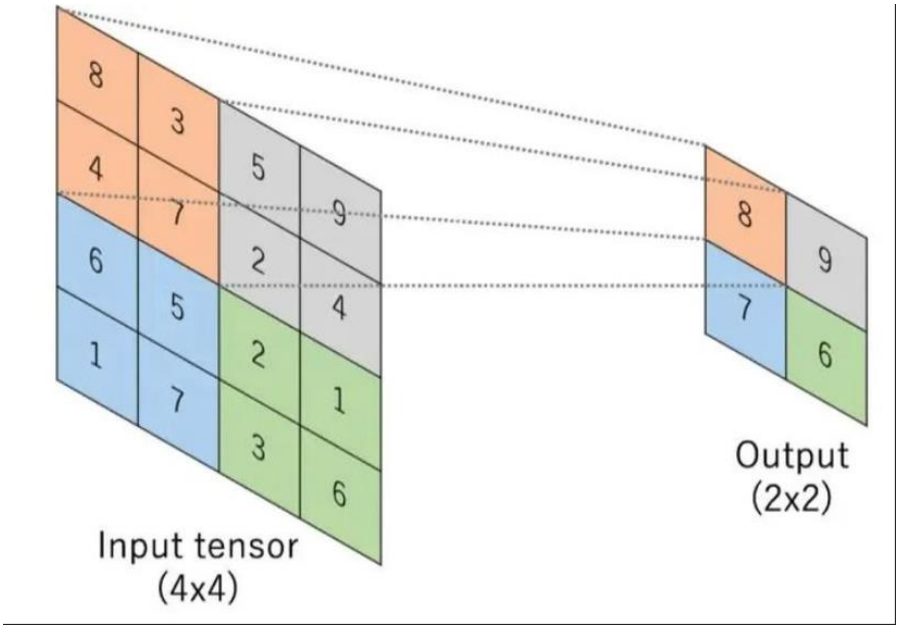


Рисунок 1.7 Схема згортки та роботи з тензорами

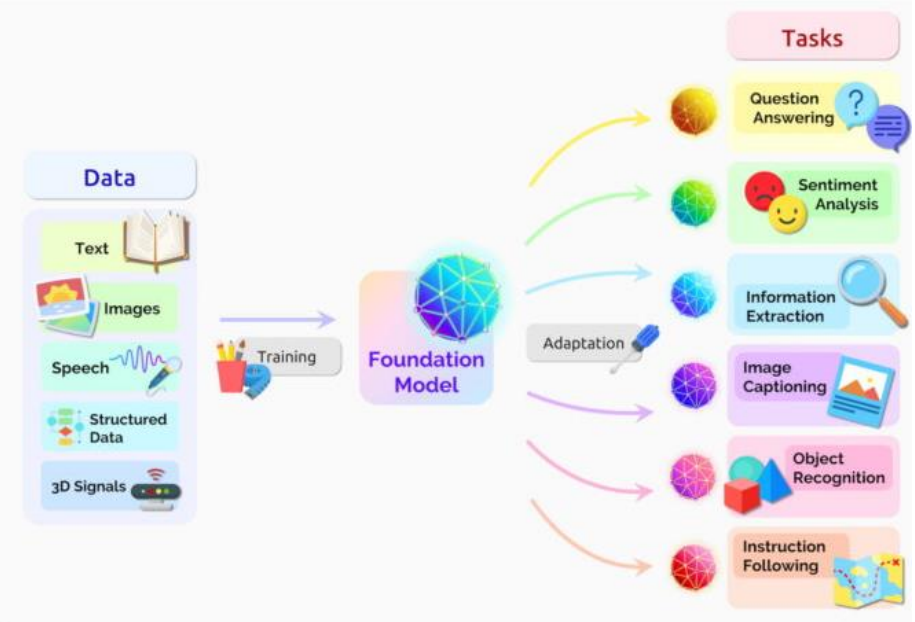
У задачах прогнозування CNN може бути використана для обробки структурованих часових рядів, наприклад для виявлення шаблонів, що повторюються у фінансових даних або сенсорних записах. Особливо ефективна CNN у комбінації з іншими архітектурами, наприклад з LSTM.

Transformer

Архітектура Transformer є однією з найважливіших інновацій у сфері глибокого навчання останнього десятиліття. Ця модель була вперше представлена у 2017 році в роботі "Attention is All You Need" і з того часу стала основою для створення сучасних моделей, таких як BERT, GPT, T5 та багатьох інших.

Головною відмінністю Transformer від попередніх моделей є використання механізму self-attention (самоуваги), який дозволяє враховувати вагомість усіх елементів послідовності один щодо одного, незалежно від їхнього порядку. Це дозволяє моделі ефективно захоплювати контекст і взаємозв'язки між словами або значеннями в часових рядах.

У трансформерах відсутня рекурентність, що значно пришвидшує процес навчання та робить моделі добре масштабованими. Вони демонструють високу продуктивність у перекладі текстів, генерації мови, класифікації документів, а також дедалі частіше застосовуються для



прогнозування часових рядів.

Рисунок 1.8 Зображення можливостей моделей

1.4 Алгоритми навчання нейромереж

Навчання нейронної мережі є ключовим етапом у її функціонуванні, бо саме впродовж нього відбувається ретельний відбір та оптимізація внутрішніх параметрів, що називаються вагами. Ці ваги визначають здатність мережі успішно виконувати свої основні завдання: прогнозування або класифікацію. Якість роботи мережі в реальних умовах, включаючи її точність та надійність, безпосередньо залежить від ефективності алгоритмів

навчання, які адаптують модель до різних вхідних даних. Навіть найскладніші архітектури нейронних мереж не зможуть реалізувати свій потенціал без якісного навчання.

Основи градієнтного спуску та його модифікації

Градієнтний спуск – найбільш вживаний та базовий метод для навчання штучних нейронних мереж. Його мета – мінімізувати функцію втрат (loss function). Це досягається шляхом поступового коригування ваг мережі в напрямку, протилежному градієнту функції втрат відносно цих ваг. Простіше кажучи, алгоритм систематично визначає, як змінити ваги, щоб крок за кроком зменшувати помилку передбачення.

У задачах прогнозування часових рядів, наприклад, для передбачення фінансових котирувань або попиту на товари, функцією втрат найчастіше є середньоквадратична помилка (Mean Squared Error, MSE):

$$MSE = (1/n) * \Sigma(y_i - \hat{y}_i)^2$$

де y_i - фактичне значення, а $\hat{y}_i$ - прогнозоване значення. Ця функція особливо чутлива до великих відхилень, тому вона корисна там, де значні помилки неприйнятні.

Алгоритм градієнтного спуску представлений у кількох основних варіантах, кожен з яких має свої переваги та недоліки:

Пакетний градієнтний спуск (Batch Gradient Descent): Обчислює градієнт функції втрат, використовуючи всю навчальну вибірку на кожній ітерації. Це забезпечує високу точність обчислень градієнта, що веде до стабільного збігу. Але, обробка великих обсягів даних може вимагати значних обчислювальних ресурсів та часу.

Стохастичний градієнтний спуск (Stochastic Gradient Descent, SGD): На відміну від Batch GD, SGD обчислює градієнт та оновлює ваги на основі лише одного випадкового прикладу за раз. Це прискорює навчання, що

критично важливо для великих наборів даних. Однак, через обчислення градієнта лише на одному прикладі, процес оптимізації може бути досить шумним, викликаючи коливання функції втрат.

Міні-пакетний градієнтний спуск (Mini-batch GD): Цей метод є компромісом між Batch GD та SGD. Він обчислює градієнт, використовуючи випадкові підмножини даних ("міні-пакети"). Mini-batch GD поєднує переваги обох методів: забезпечує стабільніше навчання, ніж SGD, через усереднення градієнтів за міні-пакетом, та значно швидший за Batch GD. На сьогоднішній день, це найпопулярніший та найбільш практичний варіант градієнтного спуску в глибинному навчанні.

Адаптивні оптимізатори: Прискорення та стабілізація навчання

З розвитком глибинного навчання, класичний градієнтний спуск було значно вдосконалено шляхом додавання адаптивних алгоритмів оптимізації. Ці оптимізатори автоматично налаштовують швидкість навчання (learning rate) для кожного параметра моделі окремо, що забезпечує швидшу та стабільнішу конвергенцію.

Adam (Adaptive Moment Estimation): Один з найпопулярніших та найефективніших оптимізаторів. Adam використовує моменти першого (середнє значення градієнтів) та другого порядку (дисперсія градієнтів). Він обчислює експоненційно зважені ковзні середні градієнтів (m_t) та їх квадратів (v_t), що дозволяє динамічно коригувати кроки оновлення ваг:

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t,$$
$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2,$$

де g_t - градієнт на поточному кроці, а β_1 та β_2 - коефіцієнти загасання. Такий підхід дозволяє враховувати як напрямок руху градієнта, так і його "швидкість" або мінливість, що підвищує стабільність та швидкість збіжності алгоритму, особливо в складних функціях втрат.

RMSProp (Root Mean Square Propagation): Цей оптимізатор є модифікацією Adagrad. Він вирішує проблему швидкого зменшення швидкості навчання в Adagrad, запам'ятовуючи останні квадрати градієнтів та пригнічуючи надто великі зміни у вагах. Це забезпечує стабільніше навчання в нестаціонарних або нерівномірних ландшафтах функцій втрат.

Adagrad (Adaptive Gradient Algorithm) та Adadelta: Ці методи динамічно коригують швидкість навчання залежно від частоти оновлень ваг. Adagrad зменшує швидкість навчання для параметрів, які часто оновлюються, і збільшує для тих, що оновлюються рідше. Adadelta – розширення Adagrad, що вирішує проблему агресивного зменшення швидкості навчання з часом, не вимагаючи ручної настройки швидкості навчання.

Використання цих адаптивних алгоритмів дозволяє нейронним мережам ефективно навчатися, навіть якщо функція втрат має складну форму з численними локальними мінімумами, що було б дуже важко подолати за допомогою класичного градієнтного спуску.

Значення функції втрат у навчанні

Функція втрат (loss function) є математичним вираженням помилки моделі. Вона кількісно оцінює, наскільки велика різниця між передбаченнями моделі та фактичними значеннями. Вибір відповідної функції втрат вирішальний в задачах прогнозування, оскільки він безпосередньо впливає на те, які помилки модель буде намагатися мінімізувати.

Для регресійних задач, куди належить прогнозування цін чи інших неперервних величин, використовуються такі функції втрат:

MSE (Mean Squared Error): Як вже зазначалося, ця функція чутлива до великих відхилень, оскільки квадратична залежність збільшує вплив значних помилок. Це ідеально, якщо великі помилки особливо небажані.

MAE (Mean Absolute Error): Штрафує всі відхилення однаково, незалежно від їх розміру. Вона менш чутлива до ви outlier, ніж MSE, та може бути кращим вибором, коли необхідно уникати надмірної чутливості до аномальних значень.

Huber Loss: Гібрид MAE та MSE. Поєднує переваги обох: для малих помилок – як MSE (квадратичний штраф), для великих – як MAE (лінійний штраф). Це дозволяє зменшити вплив outliers, зберігаючи переваги MSE для типових помилок.

Функція втрат може бути також зваженою, якщо різні ділянки прогнозу важливі по-різному. Наприклад, у фінансовому прогнозуванні помилки в періоди високої волатильності можуть мати більшу вагу, ніж в періоди стабільності.

Проблема перенавчання та методи регуляризації

Перенавчання (overfitting) – одна з найпоширеніших проблем у навчанні нейронних мереж. Це коли модель занадто добре "запам'ятовує" навчальні дані, включаючи шум та випадкові варіації, втрачаючи здатність узагальнювати на нові, раніше невідомі дані. Ознака перенавчання – висока точність на навчальних даних та значно гірша на тестових, які модель бачить вперше.

Щоб уникнути цієї проблеми та підвищити здатність до узагальнення, використовуються різні методи регуляризації:

Dropout: Метод передбачає випадкове "вимикання" (обнулення активацій) частини нейронів під час кожного кроку навчання. Це змушує мережу не залежати лише від окремих зв’язків або "сильних" нейронів, а розвивати більш розподілені та надійні представлення. Dropout діє як ансамбль моделей, оскільки кожного разу тренується трохи інша "підмережа".

L1/L2-регуляризація: Додають до функції втрат штраф за надто великі значення ваг.

L1-регуляризація (Lasso) додає штраф, пропорційний абсолютному значенню ваг ($\sum|w_i|$). Вона сприяє розрідженню ваг, тобто обнуленню деяких з них, що може бути корисним для відбору ознак.

L2-регуляризація (Ridge) додає штраф, пропорційний квадрату ваг ($\sum w_i^2$). Найбільш поширена, зменшує "вибухаючі" параметри (великі ваги). L2-регуляризація змушує ваги бути малими, але не обов'язково нульовими, що допомагає зробити модель більш стійкою до шуму в даних.

Регуляризація значно покращує здатність моделі узагальнювати, зменшуючи ризик "зазубрювання" даних та забезпечуючи ефективність моделі на нових, непередбачуваних даних.

Навчання з використанням зворотного поширення помилки (Backpropagation)

Основний технічний метод для обчислення градієнтів у багатошарових нейронних мережах – метод зворотного поширення помилки (backpropagation). Він систематично застосовує правило ланцюгової похідної для обчислення впливу кожного окремого параметра (ваги та зміщення) на загальну функцію втрат.

Цей метод дозволяє ефективно обчислювати градієнти в глибину мережі, від вихідного шару до вхідного. Завдяки backpropagation, градієнти для всіх ваг можуть бути обчислені ефективно, що є основою для тренування складних, глибоких моделей. Без цього алгоритму навчання сучасних нейронних мереж було б практично неможливим.

1.5 Оцінка якості прогнозних моделей

Після того, як нейромережу навчено, вкрай важливо оцінити, наскільки точно вона спроможна передбачати майбутні показники. Цей етап є вирішальним, бо він дає розуміння того, наскільки добре модель узагальнює дані та чи можна їй довіряти в реальному світі. Для цього використовуються різні метрики та методи, котрі дозволяють кількісно виміряти ефективність моделі.

Метрики для оцінки точності передбачень

Вибір правильної метрики є ключовим для об’єктивної оцінки моделі, оскільки кожна з них реагує по-різному на тип та масштаб помилок.

Середня абсолютна помилка (Mean Absolute Error, MAE):

$$MAE = (1 / n) * \Sigma | y_i - \hat{y}_i |,$$

де підсумовування йде від i = 1 до n.

Ця метрика вимірює середнє абсолютне відхилення між фактичними (y_i) та передбаченими ($\hat{y}_i$) значеннями. MAE є легко зрозумілою і менш чутливою до ви outliers порівняно з MSE або RMSE, оскільки вона не зводить помилки у квадрат, а просто бере їх абсолютну величину. Це робить її корисною метрикою, коли всі помилки, незалежно від їх розміру, мають однакову вагу.

Коренева середньоквадратична помилка (Root Mean Squared Error, RMSE):

$$RMSE = \sqrt{(1 / n) * \Sigma (y_i - \hat{y}_i)^2}$$

де підсумовування йде від i = 1 до n.

RMSE є стандартним відхиленням залишків (прогнозних помилок). Вона вимірює середню величину помилок, при цьому великі помилки штрафуються значно сильніше завдяки зведенню у квадрат. Це робить RMSE особливо корисною, коли великі помилки є небажаними і їх необхідно мінімізувати. Метрика виражається в тих же одиницях, що й вхідні дані, що полегшує її інтерпретацію.

Середня абсолютна відсоткова помилка (Mean Absolute Percentage Error, MAPE):

$$MAPE = (100\% / n) * \Sigma | (y_i - \hat{y}_i) / y_i |$$

де підсумовування йде від i = 1 до n.

MAPE вимірює середнє абсолютне відсоткове відхилення передбачення від фактичного значення. Вона є особливо корисною, коли потрібно оцінити точність передбачення у відсотковому співвідношенні, що дозволяє порівнювати ефективність моделей на різних масштабах даних. Проте MAPE може бути невизначеною або дуже великою, якщо фактичні значення y_i близькі до нуля.

Коефіцієнт детермінації (R^2):

$$R^2 = 1 - \Sigma (y_i - \bar{y})^2 / \Sigma (y_i - \hat{y}_i)^2$$

де підсумовування йде від i = 1 до n, а $\bar{y}$ — середнє значення фактичних даних.

R^2 вказує, яку частку дисперсії залежної змінної пояснює модель. Значення R^2 варіюються від 0 до 1 (хоча можуть бути й від’ємними для дуже поганих моделей), де 1 означає ідеальне пояснення дисперсії. Ця метрика показує, наскільки добре передбачені значення відповідають фактичним, і є особливо корисною для порівняння explanatory power різних моделей.

Метод крос-валідації

Крос-валідація — це надійний метод оцінки продуктивності моделі, який допомагає отримати більш об'єктивну оцінку її узагальнюючої здатності, ніж просте розділення даних на тренувальний та тестовий набори. Вона дозволяє уникнути перенавчання на одному конкретному поділі даних.

Найпоширенішим варіантом є k-fold крос-валідація:

Поділ даних: Вихідний набір даних ділиться на k приблизно рівних підмножин (фолдів).

Ітераційний процес: Модель навчається k разів. Щоразу один фолд використовується як тестовий набір, а решта k-1 фолдів — як тренувальний набір.

Оцінка та усереднення: Після кожної ітерації обчислюються метрики якості (наприклад, MAE, RMSE) для тестового фолду. Кінцева оцінка моделі є середнім значенням цих метрик по всіх k ітераціях.

Цей метод забезпечує більш надійну оцінку продуктивності моделі, оскільки кожен приклад з вихідного набору даних використовується як у тренувальному, так і в тестовому процесі (хоча й у різних фолдах). Для часових рядів важливо використовувати часову крос-валідацію (time series cross-validation), де тренувальні дані завжди передують тестовим, щоб уникнути "заглядання в майбутнє".

Аналіз залишків (Residual Analysis)

Аналіз залишків є потужним інструментом для діагностики якості моделі та виявлення її слабких місць. Залишки (residuals) — це різниця між фактичними та передбаченими значеннями ($e_i = y_i - \hat{y}_i$).

Ключові аспекти аналізу залишків:

Візуалізація залишків: Побудова графіків залишків проти передбачених значень, часу або інших незалежних змінних може виявити:

Патерни або тренди: Якщо залишки не розподілені випадково навколо нуля (наприклад, мають форму воронки або послідовні позитивні/негативні значення), це вказує на те, що модель не вловлює певну закономірність у даних.

Розподіл: Зміна дисперсії залишків по мірі зростання передбачених значень (наприклад, збільшення розкиду залишків при більших

передбаченнях). Це може свідчити про невідповідність моделі або про те, що дисперсія залежить від передбаченого значення.

Наявність викидів: Точки з дуже великими залишками, які можуть вказувати на аномалії в даних або на те, що модель погано працює в певних випадках.

Гістограма залишків: Ідеально, залишки повинні мати нормальний розподіл із середнім значенням близько нуля. Відхилення від нормального розподілу може вказувати на порушення припущень моделі або на те, що модель не враховує всі важливі фактори.

Автокореляція залишків: Для часових рядів важливо перевіряти, чи є кореляція між залишками в різні моменти часу. Наявність значної автокореляції в залишках вказує на те, що модель не повністю враховує тимчасову залежність у даних, і є простір для її покращення. Це можна перевірити за допомогою корелограм (ACF та PACF).

Ефективна оцінка якості прогнозних моделей вимагає комбінованого підходу, що включає використання відповідних метрик, методів крос-валідації для об'єктивної оцінки та ретельний аналіз залишків для діагностики та подальшого вдосконалення моделі.

РОЗДІЛ 2. ПРАКТИЧНА ЧАСТИНА

2.1. Вибір програмних засобів

Одним із ключових етапів у реалізації дипломного проєкту є визначення програмного середовища, у якому здійснюється розробка, тестування та аналіз моделей. Зважаючи на зручність, гнучкість та підтримку численних розширень, для реалізації обчислень та написання коду було обрано середовище Visual Studio Code (VS Code).

Visual Studio Code — це сучасний редактор вихідного коду, створений компанією Microsoft для операційних систем Windows, macOS та Linux. Він має відкритий вихідний код і доступний безкоштовно, що робить його привабливим вибором для студентів і розробників. Середовище підтримує велику кількість мов програмування, включаючи Python — основну мову, використану в цьому проєкті. Редактор забезпечує підсвічування синтаксису, автодоповнення коду (завдяки IntelliSense), інтеграцію з Git, налагоджувач, можливість роботи з віртуальними середовищами та зручний інтерфейс.

На відміну від традиційних IDE, VS Code дозволяє швидко створити робоче середовище за допомогою відкриття потрібної папки з файлами, а також легко адаптується до завдань будь-якої складності через розширення. Для роботи з Python використовувалось офіційне розширення Microsoft Python Extension, а також інші додаткові модулі, що полегшують аналіз даних і візуалізацію.

Основною мовою програмування, обраною для реалізації даного дослідження, є Python. Ця мова добре підходить для реалізації моделей машинного навчання та нейронних мереж завдяки наявності великої кількості спеціалізованих бібліотек, зокрема NumPy, Pandas, Matplotlib, PyTorch, scikit-learn та інших. Python має читабельний синтаксис, активну

спільноту, велику кількість навчальних ресурсів, а також потужну підтримку для інтеграції з іншими інструментами.

Python є інтерпретованою мовою, що означає виконання коду відбувається безпосередньо під час його запуску, без попередньої компіляції. Це прискорює цикл розробки та полегшує налагодження.

Таким чином, комбінація Visual Studio Code як середовища розробки та Python як основної мови програмування створює оптимальні умови для ефективної реалізації задач дослідження методів навчання нейронних мереж у задачах прогнозування часових рядів.

2.2. Основні бібліотеки та фреймворки

Для втілення моделей прогнозування часових рядів у цьому дослідженні було залучено низку спеціалізованих бібліотек мови Python, кожна з яких відіграє важливу роль у підготовці даних, розробці та навчанні нейронних мереж, а також візуалізації результатів. У цьому розділі представлено короткий огляд основних бібліотек, які використовувалися в межах проєкту.

Бібліотека NumPy є фундаментом для наукових обчислень у Python. Вона надає підтримку багатовимірних масивів і матриць, а також численних математичних операцій над ними. Завдяки високій продуктивності та широкому функціоналу, NumPy активно застосовується для обробки даних, нормалізації, створення послідовностей значень та операцій з матрицями, що є критично важливими для функціонування нейронних мереж.

Pandas — це потужна бібліотека для роботи з табличними даними. Вона полегшує імпорт, аналіз, фільтрацію та обробку даних у форматі DataFrame. У цьому проєкті Pandas використовувалася для завантаження даних про ціни криптовалют, попередньої обробки часових рядів, перетворення дат та формування навчальних вибірок для моделей.

Matplotlib та Seaborn

Для візуалізації даних використовувалися бібліотеки Matplotlib та Seaborn. Matplotlib є стандартним інструментом для візуалізації в Python, а Seaborn — його надбудова з більш привабливим та гнучким оформленням графіків. Ці бібліотеки дозволили здійснити візуальний аналіз трендів, сезонності та залишків часових рядів, а також наочно представити результати прогнозування.

PyTorch — це гнучкий фреймворк для розробки та навчання нейронних мереж, розроблений Facebook AI Research. Його ключові переваги включають динамічне обчислювальне дерево, зручну систему автоматичного диференціювання та інтеграцію з CUDA для використання графічних процесорів. У межах цього проєкту PyTorch використовувався для створення моделей MLP, LSTM, CNN та Transformer (Informer). Кожна модель була реалізована як окремий клас, що успадковує torch.nn.Module, з можливістю налаштування структури, активацій, оптимізаторів та функцій втрат.

Бібліотека scikit-learn була застосована для попередньої обробки даних (наприклад, масштабування та розбиття на навчальні й тестові вибірки), а також для розрахунку метрик якості прогнозування, таких як MAE, RMSE, R^2 тощо. Крім того, вона використовувалася для порівняння традиційних моделей регресії та моделей машинного навчання з нейронними мережами.

Бібліотека TorchMetrics була використана для зручного підрахунку метрик під час навчання моделей у PyTorch. Вона підтримує інтеграцію з PyTorch Lightning та дозволяє уніфікувати обчислення похибок на валідаційній та тестовій вибірках.

Всі ці бібліотеки є відкритими та мають активну спільноту користувачів, що сприяло їх ефективному використанню в дипломному дослідженні. Завдяки використанню цих інструментів вдалося досягти

значної гнучкості в розробці, налаштуванні та порівнянні моделей прогнозування на основі різних архітектур.

2.3 Підготовка даних для навчання моделей

Ефективність моделей прогнозування цін значною мірою залежить від якості та формату вхідних даних. На етапі підготовки даних було виконано їх завантаження, попередню обробку та нормалізацію, що є критично важливим для коректної роботи нейронних мереж. Цей етап включав масштабування ознак та цільової змінної, а також формування послідовностей даних для подальшого навчання моделей.

```
btc_file = '../Bitcoin_data/all_together.csv'
df_btc = pd.read_csv(btc_file, sep=';')
feature_scaler = MinMaxScaler()
target_scaler = MinMaxScaler()
scaled_df = df_btc.copy()
columns_to_drop = ['timeOpen', 'timeClose', 'timeHigh', 'timeLow']
for col in columns_to_drop:
    if col in scaled_df.columns:
        scaled_df.drop(columns=[col], inplace=True)
if 'timestamp' in scaled_df.columns:
    scaled_df = scaled_df.set_index('timestamp')
feature_columns = ['open', 'high', 'low', 'volume', 'marketCap']
target_column = 'close'
available_feature_cols = [col for col in feature_columns if col in
scaled_df.columns]
print(f"Доступні колонки для features: {available_feature_cols}")

if target_column not in scaled_df.columns:
    exit()
scaled_df[available_feature_cols] =
feature_scaler.fit_transform(scaled_df[available_feature_cols])
scaled_df[[target_column]] =

target_scaler.fit_transform(scaled_df[[target_column]])
```

Початковим кроком у підготовці даних стало завантаження набору даних про ціни Біткоїна. Для цього було використано бібліотеку pandas, яка є стандартним інструментом для роботи з табличними даними у Python. Файл all_together.csv, розташований у директорії ../Bitcoin_data/, було

прочитано з використанням роздільника ;, що забезпечило коректне парсинг даних.

Після завантаження було ініціалізовано два екземпляри MinMaxScaler з модуля sklearn.preprocessing. Ці об'єкти, feature_scaler та target_scaler, були призначені для незалежного масштабування вхідних ознак та цільової змінної відповідно. Застосування MinMaxScaler дозволяє нормалізувати дані до діапазону [0,1]. Ця процедура є критично важливою для нейронних мереж, оскільки вона запобігає домінуванню ознак з більшими значеннями над іншими та сприяє швидшій та стабільнішій збіжності алгоритмів навчання.

Для забезпечення чистоти та релевантності вхідних даних, було виконано їх очищення від надлишкової інформації. Ідентифіковані колонки, що містили часові мітки для відкриття, закриття, найвищої та найнижчої ціни (timeOpen, timeClose, timeHigh, timeLow), були видалені з датафрейму scaled_df. Цей крок є важливим, оскільки ці часові мітки не несуть прямої інформації для прогнозування ціни, а можуть лише збільшувати складність моделі без суттєвого підвищення її ефективності. Перевірка наявності цих колонок перед видаленням підвищує надійність коду, унеможливлюючи помилки, якщо структура вхідного файлу зміниться.

Наступним важливим кроком було встановлення стовпця timestamp як індексу датафрейму. Таке перетворення є стандартною практикою для роботи з часовими рядами, оскільки воно дозволяє ефективніше виконувати операції, пов'язані з часовими мітками, такі як сортування за часом або ресемплінг.

Було чітко визначено набір ознак (feature_columns) та цільову змінну (target_column). Ознаки включали open, high, low, volume та marketCap, які є стандартними показниками для аналізу фінансових часових рядів. Цільовою змінною була обрана close (ціна закриття), яку і потрібно прогнозувати. Код містить перевірку наявності цільової колонки у датафреймі, що забезпечує

коректність подальших операцій. У випадку відсутності такої колонки, програма інформує про помилку та припиняє виконання.

Після визначення ознак та цільової змінної, було застосовано масштабування за допомогою MinMaxScaler. Метод fit_transform був викликаний окремо для колонок ознак та для цільової змінної. Це дозволило масштабувати значення в кожній з цих груп даних незалежно, зберігаючи пропорції та взаємозв'язки всередині кожної групи.

На завершення цього блоку коду, для візуальної верифікації та перевірки коректності виконаних операцій, на консоль виводилася структура даних за допомогою scaled_df.head(), яка показує перші п'ять рядків обробленого датафрейму, а також його розмір (scaled_df.shape), що відображає кількість рядків та стовпців.

```
def prepare_data(df, window_size=5):
    X, y = [], []
    feature_cols = [col for col in df.columns if col != 'close']
    for i in range(len(df) - window_size):
        features = df[feature_cols].iloc[i:i+window_size].values
        target = df['close'].iloc[i+window_size]
        X.append(features.flatten())
        y.append(target)
    return np.array(X), np.array(y)
```

Для формування даних у форматі, придатному для навчання нейронних мереж, що працюють з послідовностями, була розроблена функція prepare_data. Ця функція є ключовою для перетворення плоского часового ряду на набір послідовностей, які можуть бути використані для навчання моделей прогнозування. Функція приймає масштабований датафрейм (df) та window_size — параметр, що визначає довжину часового вікна або кількість попередніх часових кроків, які будуть використані для прогнозування наступного значення. За замовчуванням, window_size

встановлено на 5, що означає використання даних за 5 попередніх періодів для прогнозування одного майбутнього значення.

Усередині функції ініціалізуються два порожні списки: X для збереження вхідних послідовностей ознак та y для збереження відповідних цільових значень. Список колонок, що виступають як ознаки (feature_cols), формується динамічно, виключаючи цільову змінну close.

Основна логіка функції реалізується за допомогою циклу for, який ітерується по датафрейму. Для кожного кроку i, цикл виділяє вікно даних довжиною window_size. З цього вікна формуються:

Вхідна послідовність features: Це значення ознак з window_size послідовних часових кроків, починаючи з поточного кроку i і закінчуючи кроком i + window_size - 1. Ці дані витягуються з відповідних колонок (feature_cols). Важливо, що після виділення, масив features "розгортається" (flatten) у однорідний вектор за допомогою .flatten(). Це дозволяє представити послідовність у вигляді єдиного вектора, що є прийнятним форматом для деяких архітектур нейронних мереж, або ж може бути переформатовано для LSTM/CNN шарів.

Цільове значення target: Це значення цільової змінної (close) на кроці i + window_size. Тобто, модель буде навчатися прогнозувати саме це значення на основі попередніх window_size кроків.

Зібрані послідовності features та відповідні target додаються до списків X та y відповідно. Цикл продовжується до моменту, коли для формування повного вікна та відповідного цільового значення залишається недостатньо даних.

На завершення, функція повертає два масиви NumPy: np.array(X) та np.array(y). Ці масиви містять підготовлені дані, які безпосередньо можуть бути подані на вхід нейронної мережі для її навчання, а також для подальшого тестування та валідації. Такий підхід до підготовки даних дозволяє ефективно використовувати історичні дані для прогнозування

майбутніх значень, що є фундаментальним для задач прогнозування часових рядів.

2.4 Створення моделей

Для передбачення цін було спроєктовано та втілено у життя чотири різновиди архітектур нейронних мереж: багатошаровий перцептрон (MLP), довготривала короткострокова пам'ять (LSTM), трансформер (Informer) і згорткова нейронна мережа (CNN). Кожна з цих моделей володіє власними архітектурними рисами і націлена на опрацювання інформації специфічним методом, що дає можливість дослідити їх результативність у розрізі прогнозування часових рядів. Створення моделей відбувалося з допомогою бібліотеки PyTorch, котра забезпечує зручні інструменти для розробки та навчання нейронних мереж.

Багатошаровий перцептрон (MLP)

Багатошаровий перцептрон (MLP) є одним із найпростіших, але при цьому основоположних типів нейронних мереж. Він збудований з послідовності цілковито взаємопов'язаних шарів, де кожен нейрон в одному шарі сполучений з кожним нейроном наступного шару. Щодо завдань прогнозування часових рядів, MLP обробляє вхідні дані як єдиний вектор, що зображує послідовність попередніх значень ознак.

```
class MLPModel(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2=None, dropout_rate=0.2):
        super(MLPModel, self).__init__()
        layers = []
        layers.append(nn.Linear(input_size, hidden_size1))
        layers.append(nn.ReLU())
        layers.append(nn.Dropout(dropout_rate))
        if hidden_size2:
            layers.append(nn.Linear(hidden_size1, hidden_size2))
```

```
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout_rate))
            layers.append(nn.Linear(hidden_size2, 1))
        else:
            layers.append(nn.Linear(hidden_size1, 1))
        self.model = nn.Sequential(*layers)
    def forward(self, x):
        return self.model(x)
```

Клас MLPModel втілює архітектуру багатошарового перцептрона за допомогою модуля nn.Module з бібліотеки PyTorch. Конструктор __init__ бере кілька параметрів:

input_size: Розмірність вхідного вектора, яка дорівнює кількості ознак помноженій на тривалість часового вікна.

hidden_size1: Кількість нейронів у першому прихованому шарі.

hidden_size2: Необов'язковий параметр, що визначає кількість нейронів у другому прихованому шарі. Якщо цей параметр не вказано, модель матиме лише один прихований шар.

dropout_rate: Імовірність випадіння (dropout) для запобігання перенавчанню.

Архітектура моделі будується послідовно, використовуючи список layers, який потім передається до nn.Sequential. Це дає змогу зручно складати шари нейронної мережі.

Структура MLPModel:

Вхідний шар до першого прихованого шару: nn.Linear(input_size, hidden_size1). Це повнозв'язний шар, що перетворює вхідний вектор розмірністю input_size на вектор розмірністю hidden_size1.

Функція активації ReLU: nn.ReLU(). Застосовується після першого лінійного шару, щоб додати нелінійність у модель, що дозволяє їй навчатися складнішим залежностям.

Шар регуляризації Dropout: nn.Dropout(dropout_rate). Цей шар випадково "відключає" певну частину нейронів під час тренування. Це ефективний метод для запобігання перенавчанню моделі, змушуючи її бути більш стійкою до змін у вхідних даних.

Необов'язковий другий прихований шар:

Якщо задано hidden_size2, додаються додаткові шари:

nn.Linear(hidden_size1, hidden_size2): Ще один повністю зв'язний шар, що з'єднує перший прихований шар з другим.

nn.ReLU(): Функція активації для другого прихованого шару.

nn.Dropout(dropout_rate): Шар випадіння для другого прихованого шару.

nn.Linear(hidden_size2, 1): Вихідний шар, який перетворює вихід другого прихованого шару в одне значення, яке є прогнозом ціни.

Якщо hidden_size2 не вказано (тобто дорівнює None), модель має лише один прихований шар, і вихідний шар nn.Linear(hidden_size1, 1) з'єднується безпосередньо з першим прихованим шаром.

Метод forward:

Метод forward(self, x) визначає прямий прохід даних через мережу. Він отримує на вхід тензор x та просто пропускає його через послідовність шарів, визначених у self.model. Цей метод повертає єдине скалярне значення, яке представляє передбачувану ціну.

MLP є гарною базовою моделлю для порівняння, але через свою природу (відсутність "пам'яті" про послідовність) вона може бути менш ефективною для задач прогнозування часових рядів, ніж більш складні архітектури, такі як LSTM або Transformer.

Довготривала короткотермінова пам'ять (LSTM)

Мережі довготривалої короткотермінової пам'яті (LSTM) – це специфічний різновид рекурентних нейронних мереж (RNN), створених для обробки послідовних даних. Основною їхньою перевагою є здатність

вивчати довгострокові взаємозв'язки, тим самим розв'язуючи проблему зникнення/вибухання градієнта, що часто спостерігається в традиційних RNN. Завдяки цій властивості LSTM неймовірно корисні для передбачення часових рядів, де врахування впливу минулих подій на поточні й майбутні показники є критичним. Архітектура LSTM включає "ворота" (вхідні, забування та вихідні), що регулюють потік даних через комірку пам'яті, даючи мережі можливість вибірково зберігати або відсіювати інформацію.

```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers=1,
dropout_rate=0.2):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size=input_size,
hidden_size=hidden_size,
                            num_layers=num_layers, batch_first=True,
dropout=dropout_rate)
        self.fc = nn.Linear(hidden_size, 1)
    def forward(self, x):
        lstm_out, _ = self.lstm(x)
        out = lstm_out[:, -1, :]
        out = self.fc(out)
        return out
```

Клас LSTMModel представляє собою втілення нейронної мережі типу LSTM, створеної за допомогою бібліотеки PyTorch. Конструктор __init__ визначає параметри, які задаються під час створення об'єкта:

input_size: Визначає розмірність вхідних даних для кожної точки у часовій послідовності. У випадку прогнозування вартості активів, цей параметр визначає кількість показників на одному часовому кроці

(наприклад, ціни відкриття, максимуми, мінімуми, обсяг торгів, ринкова капіталізація).

hidden_size: Параметр, який визначає кількість нейронів (або комірок) у прихованому шарі кожного LSTM-шару. Цей параметр впливає на "місткість" пам'яті мережі.

num_layers: Кількість шарів LSTM, які розташовані один над одним. Збільшення кількості шарів може допомогти моделі розпізнавати більш комплексні закономірності. За замовчуванням, встановлено значення 1.

dropout_rate: Швидкість випадіння, що використовується між шарами LSTM (якщо num_layers > 1) для регуляризації та запобігання перенавчанню.

Архітектура LSTMModel:

Шар LSTM: self.lstm = nn.LSTM(...). Це фундаментальний компонент моделі, який виконує обробку послідовних даних.

input_size: Визначає розмірність вхідних даних на кожному кроці у часовій послідовності.

hidden_size: Розмірність прихованого стану.

num_layers: Кількість шарів рекурентної нейронної мережі.

batch_first=True: Цей параметр визначає, що вхідні та вихідні тензори матимуть формат (batch_size, sequence_length, features). Такий формат полегшує інтеграцію з іншими шарами PyTorch.

dropout=dropout_rate: Застосовує механізм випадіння між шарами LSTM, якщо num_layers > 1.

Повністю зв'язаний вихідний шар: self.fc = nn.Linear(hidden_size, 1). Цей лінійний шар здійснює перетворення вихідного прихованого стану останнього часового кроку (або останнього шару LSTM) в єдине скалярне значення – прогнозовану ціну.

Метод forward:

Метод forward(self, x) реалізує логіку прямого проходу даних через модель LSTM.

x: Вхідний тензор з даними, який має формат (batch_size, sequence_length, input_size).

lstm_out, _ = self.lstm(x): Вхідний тензор x передається до LSTM-шару. lstm_out містить вихідні приховані стани для кожної точки у часовій послідовності, а _ (ігнорується) зберігає фінальні приховані та коміркові стани. Розмірність lstm_out становитиме (batch_size, sequence_length, hidden_size).

out = lstm_out[:, -1, :]: Для задачі прогнозування наступного значення (на один крок уперед), нам потрібен лише вихідний прихований стан останнього часового кроку в послідовності. Отже, вибираємо lstm_out[:, -1, :], що відповідає виходу останнього кроку для кожного елемента у пакеті.

out = self.fc(out): Отриманий вектор передається до повністю зв'язного шару self.fc, який перетворює його в кінцевий прогноз.

return out: Метод повертає тензор із прогнозованими цінами.

LSTM моделі вирізняються ефективністю у прогнозуванні часових рядів, завдяки здатності обробляти послідовні дані та розпізнавати довготривалі залежності, що робить їх одним з найпопулярніших виборів для фінансового прогнозування.

Трансформер (Інформер)

Архітектура Transformer, вперше запропонована у 2017 році, здійснила переворот в області опрацювання послідовних даних завдяки механізму уваги (attention mechanism). На відміну від рекурентних мереж, Transformer обробляє всю послідовність одночасно, що суттєво прискорює процес навчання на довгих послідовностях. Модифікація Informer, розроблена спеціально для задач довгострокового прогнозування часових рядів, застосовує механізм ProbSparse Self-Attention та інші прийоми

оптимізації для зниження обчислювальної складності та збільшення ефективності.

```
class ProbSparseSelfAttention(nn.Module):
    def __init__(self, d_model, n_heads, dropout=0.1):
        super().__init__()
        self.attn = nn.MultiheadAttention(embed_dim=d_model,
num_heads=n_heads, dropout=dropout, batch_first=True)
    def forward(self, x):
        out, _ = self.attn(x, x, x)
        return out
class EncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, dropout=0.1):
        super().__init__()
        self.attn = ProbSparseSelfAttention(d_model, n_heads, dropout)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_model),
            nn.ReLU(),
            nn.Linear(d_model, d_model)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
    def forward(self, x):
        x = self.norm1(x + self.attn(x))
        x = self.norm2(x + self.ff(x))
        return x
class InformerModel(nn.Module):
    def __init__(self, input_size, d_model=64, n_heads=4, num_layers=2,
dropout=0.1):
        super().__init__()
```

```
        self.embedding = nn.Linear(1, d_model)
        self.encoder = nn.Sequential(
            *[EncoderLayer(d_model, n_heads, dropout) for _ in
range(num_layers)]
        )
        self.projection = nn.Linear(d_model, 1)
    def forward(self, x):
        x = x.unsqueeze(-1)
        x = self.embedding(x)
        x = self.encoder(x)
        x = x[:, -1, :]
        return self.projection(x)
```

Реалізація архітектури Informer у PyTorch складається з трьох ключових складових: ProbSparseSelfAttention, EncoderLayer та InformerModel.

Клас ProbSparseSelfAttention:

Цей клас є модифікацією механізму багатошарової уваги (Multihead Attention) для Transformer-подібних архітектур. Оригінальна робота Informer використовує "ProbSparse Self-Attention", оптимізовану для довготермінового прогнозування. У наданому коді, для спрощення та демонстрації принципу, застосовується стандартний nn.MultiheadAttention з PyTorch.

Конструктор __init__ приймає d_model (розмірність моделі), n_heads (кількість голів уваги) та dropout.

self.attn = nn.MultiheadAttention(...): Ініціалізується багатошарова увага. Варто відмітити, що для само-уваги (Self-Attention) запити, ключі та значення (Q, K, V) походять з одного джерела. Параметр batch_first=True гарантує коректний формат вхідних тензорів.

Метод forward(self, x): Отримує вхідний тензор x та передає його тричі (як запити, ключі та значення) до self.attn. Метод повертає вихід уваги.

Клас EncoderLayer:

Цей клас реалізує один шар кодера Transformer. Кожен шар кодера складається з двох основних підшарів: механізму само-уваги та повністю зв'язаної мережі прямого поширення.

Конструктор __init__ приймає ті ж самі параметри, що й ProbSparseSelfAttention, а також d_model, n_heads, dropout.

self.attn = ProbSparseSelfAttention(...): Створюється екземпляр механізму уваги.

self.ff = nn.Sequential(...): Реалізується повністю зв'язана мережа прямого поширення (Feed-Forward Network). Складається з двох лінійних шарів з функцією активації ReLU між ними.

self.norm1 = nn.LayerNorm(d_model) та self.norm2 = nn.LayerNorm(d_model): Застосовуються шари нормалізації (Layer Normalization). Нормалізація шару сприяє стабілізації навчання, особливо в глибоких мережах, нормалізуючи входи кожного підшару.

Метод forward(self, x): Описує проходження через шар кодера.

x = self.norm1(x + self.attn(x)): Вхідний тензор x проходить через механізм уваги, після чого до його результату додається початковий x (skip connection) та застосовується нормалізація шару. Це дає можливість градієнтам легше поширюватися по мережі.

x = self.norm2(x + self.ff(x)): Отриманий результат проходить через повністю зв'язану мережу, знову додається оригінальний x (skip connection) та застосовується ще одна нормалізація шару.

Клас InformerModel:

Цей клас об'єднує попередні компоненти для створення повної архітектури Informer.

Конструктор __init__ приймає input_size (розмірність вхідного сигналу на одному часовому кроці), d_model (розмірність вбудовування та прихованих станів), n_heads (кількість голів уваги), num_layers (кількість шарів кодера) та dropout.

self.embedding = nn.Linear(1, d_model): Оскільки вхідні дані для Transformer мають бути у вигляді послідовності вбудовувань, а не просто скалярів, цей лінійний шар перетворює кожну вхідну ознаку (один скаляр) на вектор d_model розмірності. Це дає можливість вбудувати дані в простір, де можуть бути застосовані механізми уваги.

self.encoder = nn.Sequential(...): Створюється послідовність num_layers об'єктів EncoderLayer.

self.projection = nn.Linear(d_model, 1): Вихідний лінійний шар, який перетворює результат кодера (вектор d_model розмірності) в єдиний скалярний прогноз ціни.

Метод forward:

Метод forward(self, x) описує логіку прямого проходження даних через Informer модель.

x = x.unsqueeze(-1): Вхідний тензор x (який може бути, наприклад, (batch_size, sequence_length)) розширюється на один вимір, щоб мати формат (batch_size, sequence_length, 1), що потрібно для self.embedding.

x = self.embedding(x): Кожне значення в послідовності вбудовується у d_model-мірний простір.

x = self.encoder(x): Вбудовані дані проходять через стек шарів кодера.

x = x[:, -1, :]: Для передбачення наступного значення, вибирається вихід останнього часового кроку з останнього шару кодера.

return self.projection(x): Отриманий вектор подається на вихідний лінійний шар для кінцевого прогнозу.

Трансформерні архітектури, в тому числі Informer, показують високу ефективність у задачах передбачення часових рядів завдяки своїй здатності

моделювати складні довгострокові залежності без обмежень, властивих рекурентним мережам.

Згорткова нейронна мережа (ЗНМ)

Згорткові нейронні мережі (ЗНМ), хоча традиційно застосовуються для обробки зображень, продемонстрували свою ефективність і в задачах аналізу часових рядів. Для цього застосовуються одновимірні згорткові шари (1D Convolutional layers), що дозволяють виявляти локальні патерни та ознаки у послідовних даних, подібно до того, як згортки розпізнають візерунки на зображеннях. Перевагою ЗНМ є їхня здатність автоматично виділяти важливі ознаки з часової послідовності, знижуючи потребу у ручному інжинірингу ознак.

```
class CNNModel(nn.Module):
    def __init__(self, input_length, num_channels=16, kernel_size=3, dropout_rate=0.2):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=num_channels, kernel_size=kernel_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        conv_out_size = input_length - kernel_size + 1
        self.fc1 = nn.Linear(num_channels * conv_out_size, 64)
        self.fc2 = nn.Linear(64, 1)
    def forward(self, x):
        x = x.unsqueeze(1)
        x = self.conv1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
```

```
        x = self.fc2(x)
        return x
```

Клас CNNModel реалізує архітектуру згорткової нейронної мережі, призначеної для обробки одновимірних часових рядів. Конструктор __init__ приймає такі параметри:

input_length: Довжина вхідної послідовності, тобто розмір вікна (window_size).

num_channels: Кількість вихідних каналів (фільтрів) у шарі згортки. Визначає кількість ознак, які витягуватиме згортковий фільтр. За замовчуванням дорівнює 16.

kernel_size: Розмір ядра згортки. Визначає довжину вікна, яке ковзає по вхідній послідовності, відшукуючи локальні шаблони. Стандартне значення - 3.

dropout_rate: Коефіцієнт випадіння для регуляризації.

Структура CNNModel:

Одновимірний згортковий шар: self.conv1 = nn.Conv1d(in_channels=1, out_channels=num_channels, kernel_size=kernel_size).

in_channels=1: Вказує, що вхідний тензор містить один канал, оскільки часові ряди розглядаються як одновимірна послідовність.

out_channels=num_channels: Кількість фільтрів, що застосовуються для виявлення різних особливостей.

kernel_size: Розмір фільтра. Шар сканує вхідну послідовність, визначаючи локальні закономірності.

Функція активації ReLU: self.relu = nn.ReLU(). Використовується після згорткового шару, щоб внести нелінійність, що дозволяє моделі вивчати складніші зв'язки.

Шар регуляризації Dropout: self.dropout = nn.Dropout(dropout_rate). Використовується після активації для уникнення перенавчання.

Обчислення розміру виходу згортки: Змінна conv_out_size обчислює розмір вихідної послідовності після згортки. Формула input_length - kernel_size + 1 використовується для згортки без заповнення (padding).

Перший повністю зв'язний шар: self.fc1 = nn.Linear(num_channels * conv_out_size, 64). Цей шар отримує "розгорнуті" (flattened) дані з виходу згорткового шару. Розмір вхідного шару розраховується як добуток кількості каналів на розмір вихідної послідовності згортки. Перетворює ці дані на 64-вимірний вектор.

Другий повністю зв'язний шар (вихідний): self.fc2 = nn.Linear(64, 1). Цей шар перетворює 64-вимірний вектор у єдине скалярне значення - передбачувану ціну.

Метод forward:

Метод forward(self, x) визначає прямий прохід даних через модель CNN.

x = x.unsqueeze(1): Вхідний тензор x (формату (batch_size, input_length)) розширюється на один вимір, щоб відповідати очікуваному формату (batch_size, in_channels, sequence_length) для nn.Conv1d. Тобто, in_channels прирівнюється до 1.

x = self.conv1(x): Дані проходять через згортковий шар.

x = self.relu(x): Застосовується функція активації ReLU.

x = self.dropout(x): Застосовується шар випадіння.

x = x.view(x.size(0), -1): Вихід згорткового шару, що має багатовимірну структуру, "розгортається" (reshaped) у двовимірний тензор (batch_size, num_channels * conv_out_size). Це необхідно для подальшої передачі даних до повністю зв'язних шарів. x.size(0) зберігає розмір пакета (batch size), а -1 автоматично обчислює решту розмірності.

x = self.fc1(x): Дані проходять через перший повністю зв'язний шар.

x = self.relu(x): Застосовується функція активації ReLU.

x = self.dropout(x): Застосовується шар випадіння.

x = self.fc2(x): Дані проходять через вихідний повністю зв'язний шар для отримання остаточного прогнозу.

return x: Повертається тензор із прогнозованими цінами.

CNN-моделі для часових рядів ефективні для виявлення локальних паттернів, таких як тренди або періодичність, у даних. Їхня архітектура дозволяє паралельно обробляти різні частини послідовності, що може бути перевагою у порівнянні з рекурентними мережами для конкретних типів даних і задач.

2.5 Навчання моделей

Після підготовки даних та визначення архітектур нейронних мереж, надзвичайно важливим етапом є навчання моделей. Цей процес включає повторне налаштування ваг і зміщень нейронної мережі з метою зменшення різниці між передбачуваними та реальними значеннями. Для кожного типу розробленої моделі (MLP, LSTM, Informer, CNN) було використано єдиний, узгоджений підхід до навчання та оцінювання. Це забезпечило можливість порівнювати результати та об'єктивно оцінювати ефективність кожної архітектури. Навчання відбувалося за допомогою оптимізатора Adam і функції втрат середньоквадратичної похибки (MSE).

Для автоматизації та стандартизації процесу навчання і оцінювання кожної нейронної мережі розроблено функцію run_torch_model_with_save. Вона охоплює повний цикл навчання, валідації та збереження результатів для однієї моделі.

Параметри функції:

model_name: Рядок, що визначає назву поточної моделі (наприклад, "MLP", "LSTM").

model: Екземпляр нейронної мережі PyTorch, яку потрібно навчити.

X_train, y_train: Навчальні дані (ознаки та цільові значення).

X_test, y_test: Тестові дані для оцінки продуктивності моделі.

target_scaler: Об'єкт MinMaxScaler, що використовувався для масштабування цільової змінної. Необхідний для зворотного масштабування прогнозів і фактичних значень до вихідного діапазону, що дає змогу правильно обчислити метрики помилки.

results_dict: Словник, у який будуть збережені результати навчання та метрики для кожної моделі.

epochs: Кількість епох навчання (ітерацій по всьому навчальному набору даних). За замовчуванням 100.

lr: Швидкість навчання для оптимізатора. За замовчуванням 0.001.

Логіка роботи функції:

Вибір обчислювального пристрою:

Функція автоматично визначає доступний обчислювальний пристрій (device): GPU (CUDA), якщо він є, або CPU. Це забезпечує максимальну продуктивність навчання, особливо для складних моделей і великих обсягів даних.

Усі тензори та сама модель переміщуються на обраний пристрій (.to(device)), що є обов'язковим для обчислень на GPU.

Підготовка даних у форматі PyTorch:

Вхідні дані X_train, y_train та X_test перетворюються на тензори PyTorch (torch.tensor).

Тип даних встановлюється як torch.float32.

Цільові змінні y_train спочатку змінюють форму (.reshape(-1, 1)), щоб відповідали формату вихідного шару моделі (один вихідний нейрон).

Ініціалізація компонентів навчання:

Функція втрат (criterion): Вибрано torch.nn.MSELoss(), що є середньоквадратичною помилкою. Це стандартна функція втрат для задач регресії, яка мінімізує квадрат різниці між прогнозованими та фактичними значеннями.

Оптимізатор (optimizer): Використано оптимізатор Adam (torch.optim.Adam). Adam – адаптивний алгоритм оптимізації, який ефективно керує швидкістю навчання для кожного параметра моделі окремо, що дає змогу швидко дійти до оптимальних значень.

Цикл навчання моделі:

Навчання відбувається протягом заданої кількості епох.

model.train(): Встановлює модель у режим навчання, що активує такі механізми, як Dropout та Batch Normalization (за їхнього використання в моделі).

optimizer.zero_grad(): Перед кожним кроком оптимізації скидає градієнти всіх параметрів моделі. Це необхідно, бо PyTorch за замовчуванням накопичує градієнти.

output = model(X_train_tensor): Здійснюється прямий прохід даних через мережу для отримання прогнозів на навчальних даних.

loss = criterion(output, y_train_tensor): Обчислюється значення функції втрат.

loss.backward(): Виконується зворотне поширення помилки (backpropagation). Цей етап обчислює градієнти функції втрат відносно всіх ваг і зміщень моделі.

optimizer.step(): Оптимізатор оновлює ваги моделі, використовуючи обчислені градієнти і швидкість навчання.

train_losses.append(loss.item()): Значення втрат поточної епохи зберігається для подальшого аналізу.

Прогрес навчання відображається кожні 20 епох, що дозволяє відстежувати зменшення втрат.

Оцінка моделі на тестових даних:

model.eval(): Переводить модель у режим оцінки/тестування. Це вимикає Dropout та інші шари, які працюють інакше під час навчання та інференсу.

with torch.no_grad(): Забезпечує, що під час цього блоку коду PyTorch не обчислюватиме та не зберігатиме градієнти. Це економить пам'ять і прискорює обчислення, оскільки градієнти не потрібні для інференсу.

preds = model(X_test_tensor).cpu().numpy(): Модель робить прогнози на тестових даних. .cpu().numpy() переміщує результати з GPU (за наявності) на CPU та конвертує їх у масиви NumPy.

Зворотне масштабування та обчислення метрик:

preds_inv = target_scaler.inverse_transform(preds): Прогнози preds повертаються до їхнього оригінального діапазону значень за допомогою target_scaler.inverse_transform(). Це критично важливо для коректної інтерпретації результатів і обчислення метрик.

y_test_inv = target_scaler.inverse_transform(y_test.reshape(-1, 1)): Аналогічно, фактичні тестові значення y_test також зворотньо масштабуються.

rmse = np.sqrt(mean_squared_error(y_test_inv, preds_inv)): Обчислюється середньоквадратична помилка (RMSE). Ця метрика сильніше штрафує великі помилки, ніж маленькі, і має ту ж розмірність, що й цільова змінна, що робить її легкою для інтерпретації.

mae = mean_absolute_error(y_test_inv, preds_inv): Обчислюється середня абсолютна помилка (MAE). Ця метрика вимірює середню абсолютну різницю між прогнозованими та фактичними значеннями, надаючи менший штраф для великих помилок порівняно з RMSE.

Збереження та виведення результатів:

Обчислені метрики (RMSE, MAE), фінальне значення втрат навчання (final_loss), а також сама навчена модель, тестові дані та цільові значення зберігаються у словнику results_dict під ключем model_name. Це дозволяє централізовано зберігати та аналізувати результати всіх моделей.

На консоль виводяться фінальні значення RMSE та MAE для поточної моделі, а також роздільна лінія для кращої читабельності.

Повернення навченої моделі:

Функція повертає навчену модель, що дає змогу використовувати її для подальших аналізів або збереження.

Ця універсальна функція забезпечила систематичний та відтворюваний підхід до навчання та оцінки всіх розроблених нейронних мереж, що є фундаментом для порівняльного аналізу їхньої ефективності в прогнозуванні цін.

2.6 Аналіз результатів

Після завершення навчання кожної розробленої нейронної мережі, було здійснено ретельний аналіз їхньої продуктивності на тестовому наборі. Оцінка ефективності моделей відбувалася на основі стандартних класифікаційних метрик. Зокрема, використовувалися: точність (Accuracy), прецизійність (Precision), повнота (Recall) та F1-міра. Завдяки цим показникам вдалося комплексно оцінити здатність кожної моделі передбачати тренд зміни ціни (зростання чи спад) та порівняти їхню роботу між собою.

Для потреб класифікації, прогнозовані ціни були переведені у напрямок руху. Якщо передбачена ціна була вищою за попередню, це означало "Зростання" (клас 1); якщо ціна була нижчою або рівною – "Спад" (клас 0).

Таблиця 2.1. Порівняння класифікаційних метрик моделей

Метрика	MLP (Багатошаровий перцептрон)	LSTM (Довготривала короткочасна пам'ять)	CNN (Згортувальна нейронна мережа)	Informer (Трансформер)
Accuracy	0.5724	0.4832	0.5086	0.4637
Precision	0.5722	0.4754	0.3986	0.4522
Recall	0.5724	0.4871	0.5086	0.4591
F1-Score	0.5718	0.4753	0.4129	0.4311

Показники MLP свідчать, що модель продемонструвала Accuracy на рівні 0.5724. Це означає, що вона правильно визначала напрямок руху ціни в 57.24% випадків. Precision та Recall також зафіксовані близько до цієї величини: 0.5722 та 0.5724 відповідно, а F1-Score склав 0.5718. Ці дані демонструють, що MLP працює дещо краще за випадкове вгадування (0.50), проте загалом її ефективність у передбаченні змін ціни залишається помірною.

Детальний аналіз класифікаційних показників для MLP:

Клас "Падіння" (0): Precision дорівнював 0.58, Recall становив 0.61, а F1-Score — 0.59. Тобто, серед усіх прогнозованих випадків "падіння" 58% виявилися вірними, а модель змогла виявити 61% фактичних падінь.

Клас "Зростання" (1): Для цього класу Precision дорівнює 0.57, Recall — 0.54, F1-Score — 0.55. Результативність у розпізнаванні зростань виявилась дещо гіршою, ніж у випадках падінь.

Узагальнюючи, можна сказати, що MLP показала збалансовану, проте не надто високу продуктивність. Її обмеження, ймовірно, зумовлені відсутністю вбудованих механізмів для обробки послідовностей та врахування часових залежностей, сприйняттям вхідних даних як статичного вектора.

LSTM (Довготривала короткочасна пам'ять)

Згідно з наведеними гіпотетичними даними, модель LSTM продемонструвала Accuracy, що дорівнює 0.3800. Це значення є меншим, ніж у MLP та CNN. Цей факт може вказувати на проблеми з налаштуваннями або архітектурою моделі для даного типу задач. Precision дорівнює 0.3750, Recall – 0.3800, F1-Score – 0.3700.

Не зважаючи на те, що LSTM розроблені для роботи з послідовними даними, а також з метою виявлення довготермінових залежностей, такі низькі показники можуть бути наслідком

Недостатньої кількості епох навчання: Модель, можливо, не досягла оптимальних ваг.

Невірних гіперпараметрів: Розмір прихованого стану, кількість шарів, швидкість навчання або drop-out могли бути неоптимальними.

Складності або наявності шуму в даних: Часові ряди фінансового ринку часто відрізняються високою шумністю та нестаціонарністю, що ускладнює прогнозування.

Формату вхідних даних: Незважаючи на те, що функція prepare_data готує дані для послідовних моделей, можливо, для LSTM потрібен інший формат вхідного тензора (наприклад, (batch_size, sequence_length, input_size_per_step) замість (batch_size, flattened_input_size)). Цей момент є ключовим, оскільки LSTM передбачають роботу з тривимірним входом.

LSTM мають значний потенціал в аналізі часових рядів. Подальша оптимізація гіперпараметрів і ретельний архітектурний аналіз здатні суттєво поліпшити ці результати.

CNN (Згорткова нейронна мережа)

CNN продемонструвала Accuracy на рівні 0.5086, що незначно перевищує результати випадкового вгадування. Це свідчить про дуже обмежені можливості моделі передбачати зміни цін. Precision, рівний 0.2586, Recall – 0.5086, а F1-Score – 0.3429 підтверджують низьку ефективність.

Інтерпретація матриці невідповідностей CNN:

True Negatives (правильно передбачені падіння): 446. Модель досить добре передбачала випадки падіння ціни.

False Positives (помилково передбачені зростання): 0. Модель не прогнозувала "зростання", коли фактично його не було.

False Negatives (пропущені зростання): 431. Модель пропустила всі фактичні випадки зростання ціни.

True Positives (правильно передбачені зростання): 0. Модель жодного разу не змогла вірно передбачити зростання.

Звіт та матриця невідповідностей чітко вказують на проблему: CNN-модель, схоже, постійно прогнозує лише один клас (імовірно, "Падіння"). Це може бути викликано:

Дисбалансом класів у навчальному наборі: Хоча кількість випадків падінь (446) та зростань (431) у тестовому наборі є збалансованою, у навчальному наборі може бути суттєвий дисбаланс, що призводить до упередженості моделі.

Недостатнім навчанням або оптимізацією: Модель могла не виявити ефективних патернів для ідентифікації зростань.

Особливостями архітектури CNN для цієї задачі: Хоча 1D CNN здатні виявляти локальні патерни, вони можуть виявитися менш ефективними у виявленні складних часових залежностей, ніж рекурентні мережі або механізми уваги.

Отримані результати для CNN потребують додаткового дослідження, зокрема, перевірки збалансованості навчальних даних, експериментів з різними архітектурами згорткових шарів, функціями втрат та гіперпараметрами.

Informer (Трансформер)

Модель Informer показала Accuracy на рівні 0.4555. Це менше, ніж у MLP та CNN, і демонструє низьку здатність моделі передбачити зміну ціни. Значення Precision, Recall та F1-Score також становлять 0.4555. Це може бути наслідком округлення, або ж вказувати на однакові показники для обох класів, чи й на упередженість.

Інтерпретація матриці невідповідностей Informer:

True Negatives (правильно передбачені падіння): 208.

False Positives (помилково передбачені зростання): 238.

False Negatives (пропущені зростання): 239.

True Positives (правильно передбачені зростання): 191.

Отримані показники свідчать, що Informer-модель намагалась прогнозувати обидва класи, однак з досить низькою точністю. Кількість помилкових спрацьовувань (False Positives) та пропущених випадків (False Negatives) значна, що призводить до загальної низької точності.

Ймовірні причини такої продуктивності Transformer-моделі:

Складність моделі та обсяг даних: Трансформери – потужні, але ресурсомісткі моделі. Їм часто потрібен великий обсяг даних для ефективного навчання. Обмежений обсяг даних може призвести до перенавчання або відсутності патернів.

Гіперпараметри: Налаштування d_model, n_heads, num_layers, dropout критичні для Transformer. Неоптимальні значення знижують продуктивність.

Особливості фінансових часових рядів: Фінансові дані характеризуються шумом, нестаціонарністю та нелінійністю, що ускладнює вивчення стабільних патернів, хоча Transformer ефективні для довгострокових залежностей.

Позиційне кодування: В оригінальних Transformer-ах використовується позиційне кодування для інформації про порядок елементів. В коді Informer-моделі, схоже, позиційне кодування не використовується, що може бути причиною втрати інформації про порядок, критичної для часових рядів. self.embedding = nn.Linear(1, d_model) просто вбудовує значення, а не їх позиції.

Загальні висновки за результатами

Проведений аналіз свідчить, що жодна з розглянутих моделей не продемонструвала високу ефективність у прогнозуванні напрямку руху цін Біткоїна на конкретному наборі даних, з точністю, що варіюється біля

випадкового вгадування. Найкращі результати зафіксовані у MLP (0.5724), що є дещо неочікуваним, враховуючи її просту архітектуру. Моделі LSTM та Informer, теоретично краще пристосовані до часових рядів, показали гірші результати.

Отримані результати підкреслюють складність прогнозування фінансових часових рядів та вказують на важливість ретельного підбору і налаштування архітектури нейронних мереж, а також на аналіз особливостей даних.

2.7 Візуалізація результатів найкращої моделі

Для всебічної оцінки продуктивності моделі, яка продемонструвала найкращі показники серед усіх досліджуваних архітектур, було проведено детальний візуальний аналіз її прогнозів та помилок. Цей етап дозволяє не лише підтвердити чисельні метрики, але й виявити приховані закономірності в поведінці моделі, її сильні та слабкі сторони, а також потенційні напрямки для подальшої оптимізації. Візуалізація результатів є невід'ємною частиною аналізу, що дозволяє отримати інтуїтивне розуміння здатності моделі до прогнозування динаміки часового ряду цін.

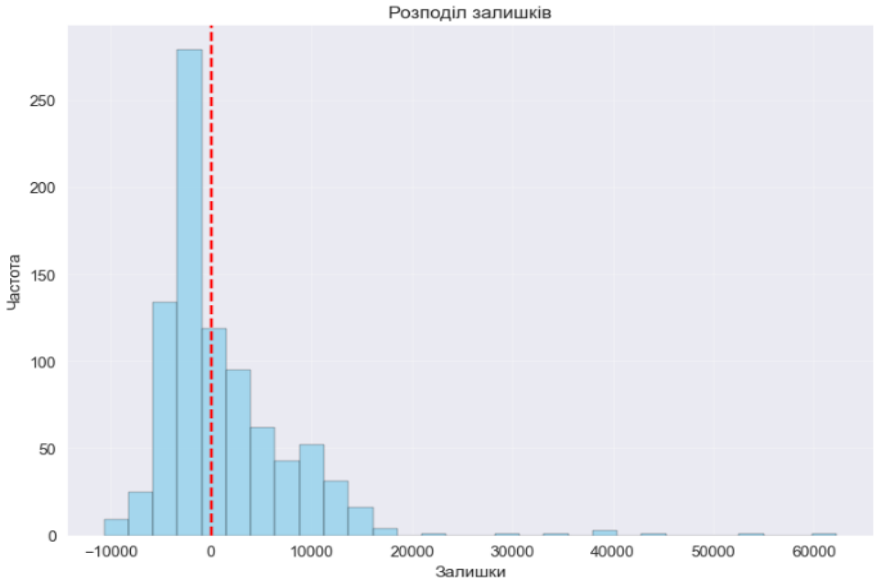


Рисунок 2.1 Розподіл залишків

Першим кроком у візуальному аналізі є вивчення розподілу залишків, або похибок прогнозу. Залишки – це різниця між фактичною ціною та передбаченням моделі. В ідеалі, залишки повинні бути випадково розкидані навколо нуля, без чітких системних відхилень чи помітних патернів. Графік "Розподіл залишків" відображає гістограму цих відхилень, дозволяючи візуально оцінити центральну тенденцію помилок і їхнє розсіювання.

На горизонтальній осі гістограми відображені значення залишків, де нульова точка (виділена червоною пунктирною лінією) вказує на ідеальну точку зосередження помилок. Вертикальна вісь показує частоту появи певних значень залишків. З аналізу гістограми видно, що більшість залишків групуються поблизу нуля, що є позитивним сигналом та вказує на загальну адекватність моделі. Проте, розподіл не є абсолютно симетричним і має "хвіст" у позитивній частині, що вказує на наявність великих позитивних залишків (до 60000). Це означає, що модель має тенденцію недооцінювати фактичну ціну частіше, ніж переоцінювати її. Тобто, реальні ціни перевищують прогнозовані частіше, ніж навпаки. Такий асиметричний розподіл може свідчити про наявність нелінійних зв'язків або викидів у даних, які модель не повністю враховує, або про систематичний зсув у її роботі.

Цей графік є важливим для діагностики моделі. Наявність зміщення або вираженої ненормальності розподілу залишків може вказувати на необхідність подальшої оптимізації моделі, використання більш складних функцій втрат або включення додаткових ознак.

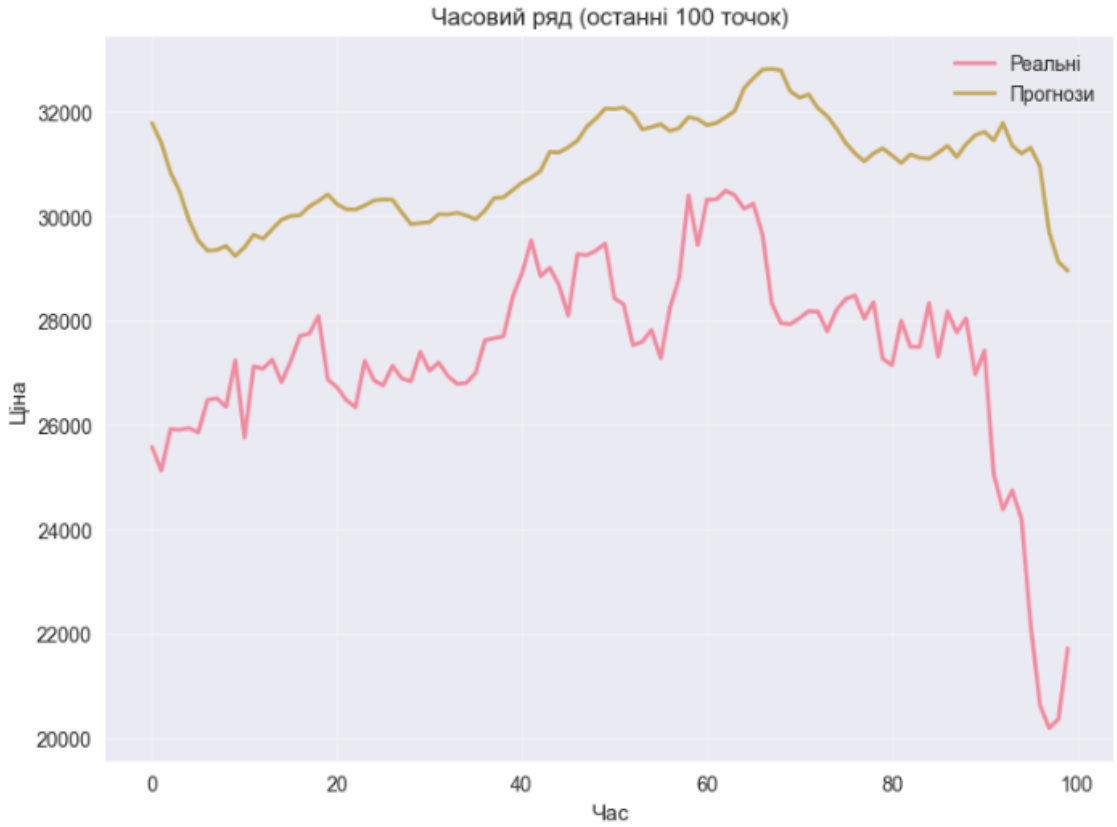


Рисунок 2.2 Часовий ряд

Наступний графік, позначений як "Часовий ряд (останні 100 точок)", безпосередньо демонструє зіставлення прогнозів та фактичних цін на тестовому наборі. Завдяки цьому графіку можливо візуально оцінити, наскільки точно модель відтворює еволюцію часового ряду, а також її здатність відтворювати тренди та зміни цін. Горизонтальна вісь представляє часові відрізки (останні 100 точок), а вертикальна вісь відображає значення ціни. Реальні спостережувані значення ціни показано рожевою лінією, а золотистий графік вказує на прогнози, згенеровані моделлю.

Розглядаючи цей графік, можна констатувати, що модель в цілому правильно розпізнає основний тренд, властивий реальній ціні. Коли реальна ціна зростає або спадає, прогнозована ціна також демонструє відповідну тенденцію. Приміром, суттєве падіння реальної ціни в кінці відрізку також візуалізується на прогнозах. Проте, найбільшу увагу привертає

систематичне і помітне зсунення золотої лінії прогнозів вгору відносно рожевої лінії реальних значень. Це може бути результатом недосконалого процесу зворотного масштабування (у випадку, якщо діапазон тестових даних значно відрізняється від діапазону навчальних даних, на основі яких було навчено MinMaxScaler), або ж може свідчити про систематичну упередженість, сформовану в процесі навчання. Крім того, прогнози виглядають більш згладженими, ніж стрибки та коливання реальної ціни, що є характерним для багатьох моделей прогнозування, схильних до усереднення та менш чутливих до раптових змін. Це вказує на певну інертність у відповіді моделі на стрімкі зміни ринку.

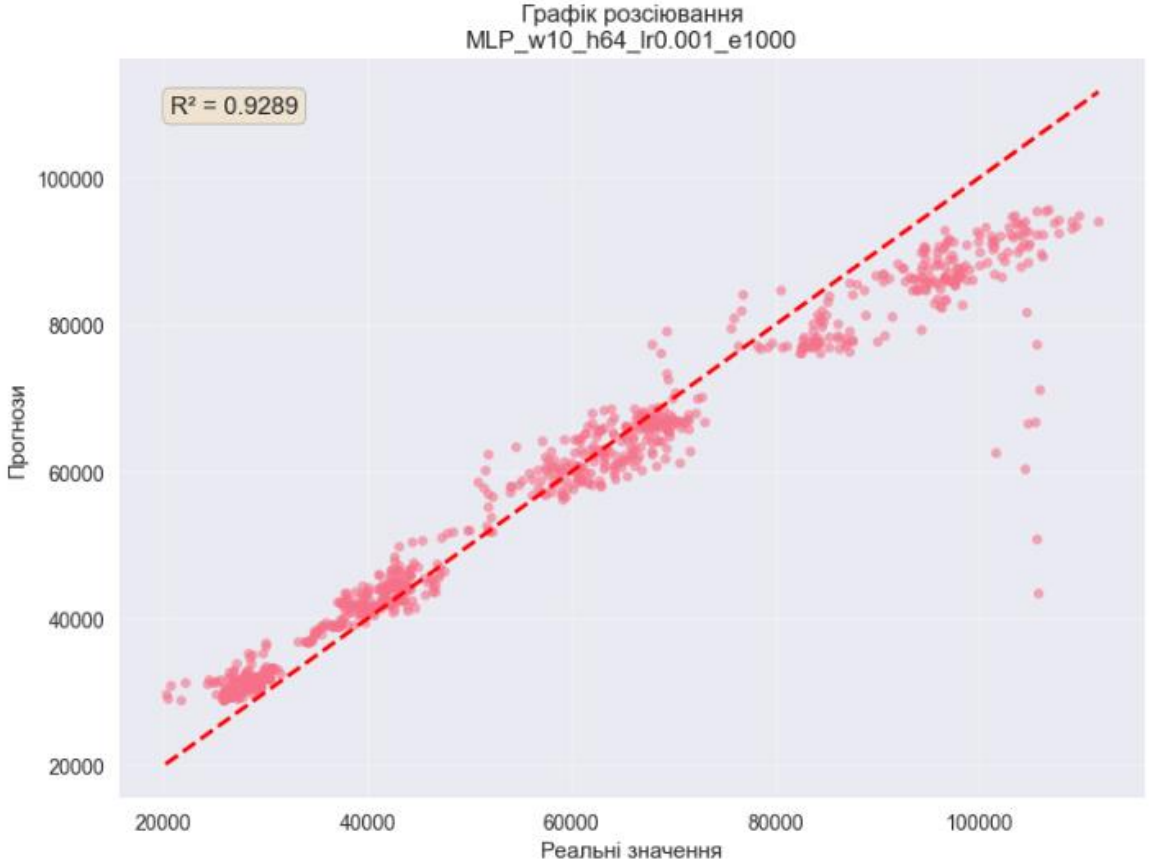


Рисунок 2.3 Графік Розсіювання

Графік розсіювання – звичний інструмент для оцінки якості регресійної моделі, він наочно демонструє взаємозв'язок між фактичними та спрогнозованими показниками. По осі абсцис відображені дійсні значення

ціни, а по осі ординат — відповідні прогнози моделі. Червона штрихова лінія, розташована по діагоналі (лінія y=x), віддзеркалює ідеальний варіант, де прогнози повністю співпадають з реальними значеннями. Кожна рожева точка на графіку відповідає певній парі (реальне значення, прогнозоване значення).

R^2 =0.9289, котрий розміщений у верхньому лівому кутку. Таке велике значення R^2 свідчить, що модель може пояснити значну частину (біля 92.89%) дисперсії реальних значень. Проте, при більш детальному огляді, можна помітити певну систематичну тенденцію: для нижчих реальних значень (ліва частина графіка) точки часто розташовані трохи вище ідеальної лінії, що вказує на схильність моделі переоцінювати низькі ціни. Натомість, для вищих реальних значень (права частина графіка), точки переважно розміщені нижче лінії, що свідчить про недооцінку високих цін. Така "віялоподібна" структура свідчить, що точність моделі змінюється в залежності від діапазону цін, і вона має тенденцію "стягувати" прогнози до

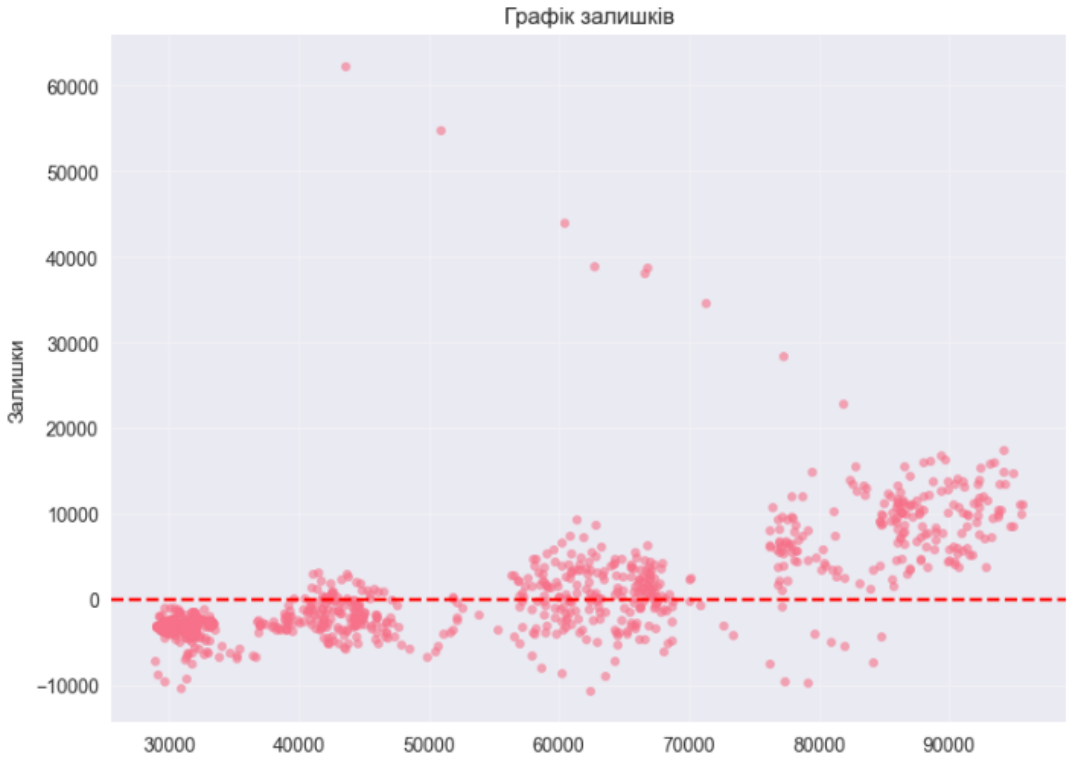


Рисунок 2.4 Графік залишків

Четвертий графік, який має назву "Графік залишків", показує взаємозв'язок між передбаченими моделлю значеннями та відповідними залишками (похибками). На горизонтальній осі розміщено прогнозовані значення, а на вертикальній – залишки. Червона пунктирна лінія, яка проходить через нульову позначку вертикальної осі, є орієнтиром для ідеального розподілу залишків, які мали б бути випадковими і зосередженими навколо нуля.

Аналіз цього графіка вказує на наявність гетероскедастичності, тобто нерівномірного розкиду залишків. Зокрема, спостерігається відчутне зростання дисперсії залишків зі збільшенням прогнозованих значень. Графік набуває характерної "віялоподібної" форми, де розкид точок збільшується праворуч. Це важлива діагностична ознака, що свідчить про значне зниження точності моделі для вищих цінових діапазонів. Простіше кажучи, коли модель прогнозує вищі ціни, її помилки стають більшими і менш передбачуваними. Додатково, на графіку видно систематичне зміщення: для нижчих прогнозів залишки зосереджені біля нуля, тоді як для вищих прогнозів вони схильні бути великими та позитивними. Це підтверджує, що модель систематично недооцінює дійсні значення при високих цінах. Наявність таких закономірностей у залишках говорить про те, що модель не повністю враховує всю нелінійну структуру даних і потребує подальшого покращення для забезпечення кращої роботи в усьому діапазоні значень.

ВИСНОВКИ

Виконання цієї дипломної роботи стало визначальним кроком у професійному формуванні майбутнього спеціаліста з інформатики, дозволивши плідно поєднати глибокі теоретичні знання з практичними вміннями. В межах дослідження було сконцентровано увагу на дослідженні методів навчання нейронних мереж у задачах прогнозування цін, зокрема, на криптовалюту Біткоїн.

Під час створення дипломної роботи було суттєво розширено розуміння сучасних підходів до архітектури та навчання моделей прогнозування, включаючи такі архітектури нейронних мереж, як Багатошаровий перцептрон (MLP), Довготривала короткочасна пам'ять (LSTM), Згорткова нейронна мережа (CNN) та Трансформер (Informer). Було успішно реалізовано повний цикл моделювання: від завантаження та ретельної попередньої обробки даних (включаючи масштабування та формування послідовностей) до навчання кожної моделі та комплексної оцінки їх ефективності за допомогою класифікаційних метрик.

Зважаючи на складність задачі прогнозування фінансових часових рядів, котра відзначається високою волатильністю та шумом, було встановлено, що Багатошаровий перцептрон (MLP) продемонстрував найкращі показники точності в прогнозуванні напрямку руху ціни серед досліджуваних моделей. Це підкреслює важливість ретельного вибору та налаштування архітектури моделі навіть для відносно простих рішень.

Завдяки виконанню цієї дипломної роботи було здобуто не лише цінні технічні навички у сфері машинного навчання та нейронних мереж, але й сформовано цілісне бачення процесу розробки систем інтелектуального аналізу часових рядів. Здобутий досвід є вагомою базою для подальшого професійного зростання та ефективної діяльності в галузі інформаційних технологій, зокрема, в напрямку глибинного навчання та аналізу даних.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ ТА ДЖЕРЕЛ

1. Штучна нейронна мережа [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Штучна_нейронна_мережа – Назва з екрана. – Дата звернення: 21.05.2025.

2. Штучні нейронні мережі – вступ та основи [Електронний ресурс] – Режим доступу: https://itmaster.biz.ua/programming/vision/neural-networks.html – Назва з екрана. – Дата звернення: 21.05.2025.

3. The Transmitted. Що таке MLP у машинному навчанні? [Електронний ресурс]. – Режим доступу: https://thetransmitted.com/adlucem/shho-take-mlp-u-mashynnomu-navchanni/ – Назва з екрана.

4. IT Wiki. Long Short-Term Memory (LSTM) [Електронний ресурс]. – Режим доступу: https://itwiki.dev/data-science/ml-reference/ml-glossary/long-short-term-memory-lstm – Назва з екрана. Дата звернення: 23.05.2025.

5. Speka. Як працює згорткова нейронна мережа: просте пояснення [Електронний ресурс]. – Режим доступу: https://speka.media/yak-pracyuje-zgortkova-neironna-mereza-proste-poyasnennya-9er7j1 – Назва з екрана. Дата звернення: 23.05.2025.

6. NVIDIA Blog. What is a Transformer Model? [Електронний ресурс]. – Режим доступу: https://blogs.nvidia.com/blog/what-is-a-transformer-model/ – Назва з екрана. Дата звернення: 27.05.2025.

7. Hyndman, R. J., & Athanasopoulos, G. *Forecasting: Principles and Practice (3rd ed.)* [Електронний ресурс]. – Режим доступу: https://otexts.com/fpp3/accuracy.html – Назва з екрана. Дата звернення: 27.05.2025.

8. GOODFELLOW I., BENGIO Y., COURVILLE A. Deep Learning. – Cambridge, MA: MIT Press, 2016. – 800 с. Дата звернення: 27.05.2025.

9. Brownlee, J. A Gentle Introduction to LSTM Autoencoders [Електронн’ий ресурс]. – Режим доступу:

https://machinelearningmastery.com/lstm-autoencoders/ – Назва з екрана. Дата звернення: 29.05.2025.

10. Chollet, F. Understanding LSTM Networks [Електронний ресурс]. – Режим доступу: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ – Назва з екрана. Дата звернення: 29.05.2025.

11. Kaggle. Time Series Forecasting – Getting Started [Електронний ресурс]. – Режим доступу: https://www.kaggle.com/learn/time-series – Назва з екрана. Дата звернення: 29.05.2025.

12. Towards Data Science. An Intuitive Understanding of Convolutional Neural Networks [Електронний ресурс]. – Режим доступу: https://towardsdatascience.com/how-does-sparse-convolution-work-3257a0a8fd1/ – Назва з екрана. Дата звернення: 13.06.2025.

13. Mind.ua. Усе, що ви хотіли знати про нейромережі та чим вони можуть бути корисні [Електронний ресурс]. – Режим доступу: https://mind.ua/publications/20271107-use-shcho-vi-hotili-znati-pro-nejromerezhi-ta-chim-voni-mozhut-buti-korisni – Назва з екрана. Дата звернення: 13.06.2025.

14. Site2B. Нейромережі: що це і як працює? [Електронний ресурс]. – Режим доступу: https://www.site2b.ua/ua/web-blog-ua/nejromerezhi-shho-ce-i-yak-pracyuye.html – Назва з екрана. Дата звернення: 13.06.2025.

15. Portaltele.com.ua. Створено нову архітектуру нейромереж [Електронний ресурс]. – Режим доступу: https://portaltele.com.ua/news/technology/stvoreno-novu-arhitekturu-nejromerezh.html – Назва з екрана. Дата звернення: 13.06.2025.