НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

До захисту допущено:
Завідувач кафедри
Оксана ТИМОЩУК
«___»_____ 2026 р.

Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системний аналіз і управління»
спеціальності 124 Системний аналіз

на тему: «Сурогатні моделі для відтворення динаміки цін фінансових активів»

Виконав:
студент IV курсу, групи КА-21
Прийма Владислав Михайлович _____

Керівник:
професор кафедри ММСА, д.т.н.,
проф. Дмитрієва Ольга Анатоліївна _____

Консультант з економічного розділу:
доцент кафедри ЕК, к.е.н.,
доц. Рощина Надія Василівна _____

Консультант з нормоконтролю:
асистент кафедри ММСА, д. філос.
Канцедал Георгій Олегович _____

Рецензент:
доцент кафедри Штучного інтелекту НН ІПСА,
к.т.н., доц. Шахновський Аркадій Маркусович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2026 р.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 124 Системний аналіз
Освітня програма «Системний аналіз і управління»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Оксана ТИМОЩУК
«__» _____ 2026 р.

ЗАВДАННЯ
на дипломну роботу студенту
Приймі Владиславу Михайловичу

1. Тема роботи «Сурогатні моделі для відтворення динаміки цін фінансових активів», керівник роботи Дмитрієва Ольга Анатоліївна, доктор технічних наук, професор кафедри ММСА, затверджені наказом по університету від «27» травня 2026 р. № 1944

2. Термін подання студентом роботи «10» червня 2026 р.

3. Вихідні дані до роботи: історичні котирування фінансових активів, навчальні, валідаційні та тестові вибіркові сукупності динаміки цін фінансових активів, нейромережеві архітектури для моделювання часових рядів, стохастичні диференціальні рівняння, модель Heston, метод Ейлера-Маруями, матеріали переддипломної практики.

4. Зміст роботи: аналіз сучасних підходів до побудови сурогатних моделей відтворення динамічних процесів, розробка алгоритмів генерування навчальних даних, обґрунтування методів чисельних симуляцій, проєктування архітектури нейромережевої сурогатної моделі, реалізація процедури навчання моделі та підбір гіперпараметрів для апроксимації еволюції цін акцій, оцінювання здатності моделі відтворювати статистичні характеристики процесу, аналіз отриманих результатів.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): схема архітектури сурогатних моделей, криві навчання, гістограми розподілів дохідностей, теплові карти метрик якості, згенеровані траєкторії цін, Парето-фронт «швидкодія × якість», презентація до виступу.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Н.В., доц., к.е.н.		

7. Дата видачі завдання: «12» березня 2026

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Аналіз наукових джерел з сурогатного моделювання для відтворення динаміки цін фінансових активів	15.02.26-25.02.26	Виконано
2	Дослідження актуальності задачі застосування нейромережевих сурогатних моделей, реалізації процедури навчання моделі та підбору гіперпараметрів	26.02.26-02.03.26	Виконано
3	Проєктування архітектури нейромережевої сурогатної моделі для апроксимації еволюції цін акцій	3.03.26-05.03.26	Виконано
4	Реалізація процедури навчання моделі та підбір гіперпараметрів на основі згенерованих або/та історичних даних	06.03.26-30.03.26	Виконано
5	Розробка програмної підтримки проведення експериментальних досліджень, оцінювання здатності моделі відтворювати статистичні характеристики процесу	31.03.26-19.04.26	Виконано
6	Підготовка розділів під час переддипломної практики	20.04.26-17.05.26	Виконано
7	Оформлення записки та розділів в дипломній роботі відповідно до нормоконтролю	18.05.26-31.05.26	Виконано
8	Попередній захист дипломної роботи	08.06.26	
9	Захист дипломної роботи	15.06.2026 – 19.06.2026	

Студент _____ ПРИЙМА В.М.

Керівник роботи _____ Дмитрієва О.А.

РЕФЕРАТ

Дипломна робота: 117 с., 25 рис., 6 табл., 2 додатки, 35 джерел.

СУРОГАТНІ МОДЕЛІ, ДИНАМІКА ЦІН, ФІНАНСОВІ АКТИВИ, НЕЙРОННІ МЕРЕЖІ, СТОХАСТИЧНІ ДИФЕРЕНЦІАЛЬНІ РІВНЯННЯ, ФІНАНСОВІ ЧАСОВІ РЯДИ

Об’єктом дослідження є багатовимірні динамічні процеси, еволюція яких описується стохастичними диференціальними рівняннями з корельованими шумами.

Предметом дослідження є методи побудови нейромережевих сурогатних моделей для відтворення розподілових та часових характеристик дохідностей фінансових активів.

Метою роботи є розробка, обґрунтування і реалізація сурогатних моделей для дослідження поведінки основних фінансових показників на основі нейромережевої апроксимації стохастичної динаміки логарифмічних дохідностей.

Методи дослідження базуються на основних положеннях теорії стохастичних диференціальних рівнянь, чисельних схемах інтегрування, теорії ймовірності та математичної статистики, методах глибокого навчання (рекурентні мережі, трансформери, Neural SDE, WGAN-GP) та статистичних тестах оцінювання якості розподілів (KS, Wasserstein, MMD).

Практичне значення роботи полягає в тому, що запропоновані нейромережеві сурогати можуть замінити обчислювально дорогий симулятор Гестона у задачах, де критичною є швидкодія: оцінюванні ринкового ризику, стрес-тестуванні портфелів, ціноутворенні похідних інструментів і навчанні моделей машинного навчання на синтетичних даних. У роботі запропоновано і програмно реалізовано в середовищі Python з використанням бібліотеки PyTorch архітектуру Neural SDE-GAN з двовимірним станом Heston-типу та змагальним навчанням.

ABSTRACT

Diploma thesis: 117 p., 25 figures, 6 tables, 2 appendices, 35 references.

SURROGATE MODELS, PRICE DYNAMICS, FINANCIAL ASSETS, NEURAL NETWORKS, STOCHASTIC DIFFERENTIAL EQUATIONS, FINANCIAL TIME SERIES

The object of the study is multidimensional dynamic processes whose evolution is described by stochastic differential equations with correlated noise.

The subject of the study is methods for constructing neural network surrogate models for reproducing the distributional and temporal characteristics of returns of financial assets.

The aim of the work is the development, justification, and implementation of surrogate models for studying the behavior of key financial indicators based on neural network approximation of the stochastic dynamics of log-returns.

The research methods are based on the fundamentals of the theory of stochastic differential equations, numerical integration schemes, probability theory and mathematical statistics, deep learning methods (recurrent networks, transformers, Neural SDE, WGAN-GP), and statistical tests for distribution quality evaluation (KS, Wasserstein, MMD).

The practical significance of the work lies in the fact that the proposed neural network surrogates can replace the computationally expensive Heston simulator in tasks where speed is critical: market risk assessment, portfolio stress testing, derivative pricing, and training machine learning models on synthetic data. The work proposes and implements in software in Python with PyTorch the Neural SDE-GAN architecture with a two-dimensional Heston-type state and adversarial training.

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

СДР (SDE) — стохастичне диференціальне рівняння

GBM — geometric Brownian motion, геометричний броунівський рух

MDN — Mixture Density Network, нейронна мережа щільності суміші

LSTM — long short-term memory, рекурентна нейронна мережа з довгою короткостроковою пам'яттю

BiLSTM — bidirectional LSTM, двонаправлена LSTM

MLP — multi-layer perceptron, багатошаровий персептрон

Neural SDE (NSDE) — нейромережеве стохастичне диференціальне рівняння

GAN — generative adversarial network, генеративна змагальна мережа

ACF — autocorrelation function, автокореляційна функція

ARCH — autoregressive conditional heteroskedasticity, авторегресивна умовна гетероскедастичність

ARCH-LM — ARCH Lagrange Multiplier, тест на наявність ARCH-ефекту

MMD — Maximum Mean Discrepancy, максимальна середня розбіжність

CDF — cumulative distribution function, кумулятивна функція розподілу

CIR — Cox-Ingersoll-Ross, стохастичний процес Кокса-Інгерсолла-Росса

VaR — Value at Risk, показник максимально допустимих втрат портфеля при заданій ймовірності

Adam — Adaptive Moment Estimation, алгоритм оптимізації параметрів нейронної мережі

GPU — graphics processing unit, графічний процесор

CUDA — Compute Unified Device Architecture, архітектура паралельних обчислень NVIDIA

CSV — comma-separated values, формат зберігання табличних даних із розділенням значень комами

ВСТУП

Фінансові ринки є одним із ключових об’єктів сучасного математичного моделювання [1]. Динаміка цін активів формує основу для оцінки ринкового ризику, ціноутворення похідних інструментів, портфельного аналізу та розробки стратегій управління капіталом [2-5]. Особливістю фінансових часових рядів є їхня стохастична природа: ціни змінюються під впливом великої кількості випадкових і взаємопов’язаних факторів — інформаційних шоків, ринкових настроїв, макроекономічних подій [2, 3]. Тому коректний імовірнісний опис динаміки цін є важливим прикладним завданням, яке має застосування у банківській сфері, інвестиційній діяльності, регулюванні фінансового сектору та квантитативному фінансовому аналізі [2].

Класичним інструментом моделювання цін активів є стохастичні диференціальні рівняння. Найвідомішими прикладами є геометричний броунівський рух, що лежить в основі моделі Блека-Шоулза [4], та модель Heston [3], яка враховує стохастичну волатильність. Модель Heston дозволяє відтворювати низку стилізованих фактів реальних фінансових рядів — кластеризацію волатильності, важкі хвости розподілу дохідностей, leverage-ефект [2, 3]. Однак практичне використання таких моделей часто потребує великих обсягів обчислень: для оцінки показника Value-at-Risk або ціни складних похідних інструментів необхідно генерувати десятки тисяч траєкторій методом Монте-Карло, що для багатовимірних задач і довгих горизонтів стає ресурсовитратним [5].

У зв’язку з цим перспективним напрямом є побудова сурогатних моделей — обчислювально дешевих наближень класичних стохастичних моделей. Сурогат після етапу навчання здатен швидко генерувати траєкторії, які статистично відтворюють властивості істинного процесу. Особливий інтерес становлять нейромережеві сурогати, оскільки сучасні архітектури (рекурентні мережі, трансформери, нейромережеві диференціальні рівняння)

показали здатність апроксимувати складні залежності у часових рядах. Окремим перспективним напрямом є моделі Neural SDE, у яких дрейф і дифузія параметризуються нейронними мережами, а саме рівняння інтегрується чисельно під час навчання.

Актуальність роботи зумовлена зростаючою потребою у швидких і статистично коректних інструментах генерації сценаріїв ринкової динаміки для задач ризик-менеджменту, стрес-тестування портфелів, ціноутворення похідних інструментів та навчання моделей машинного навчання. Існуючі підходи на основі Heston-симулятора або параметризованих GARCH-моделей [3] мають обмежену гнучкість і повільну швидкодію, що мотивує дослідження нейромережевих альтернатив.

Об’єктом дослідження є стохастичні процеси динаміки цін фінансових активів та методи їх чисельного моделювання.

Предметом дослідження є методи побудови нейромережевих сурогатних моделей для відтворення розподілових та часових характеристик логарифмічних дохідностей фінансових активів.

Для досягнення поставленої мети у роботі вирішено такі задачі.

1. Досліджено і систематизовано сучасні підходи до сурогатного моделювання, проведено огляд класичних стохастичних моделей цін.
2. Реалізовано еталонний симулятор моделі Heston за схемою Ейлера-Маруями з технікою повного зрізання дисперсії.
3. Спроєктовано множину нейромережевих архітектур з імовірнісними виходами: LSTM з MDN-головою, Transformer з MDN-головою та двома варіаціями Neural SDE (без енкодера контексту та з LSTM-енкодером).
4. Обґрунтувано комбіновану функцію втрат NLL + λ·ACF-loss та процедуру мультиактивного навчання.
5. Проведено гіперпараметричний аналіз та запропоновано порівняльну оцінку архітектури за широким набором метрик (моменти, KS, Wasserstein, MMD, ARCH-LM, leverage effect, coverage probability).

6. Визначено обчислювальну ефективність сурогатів та здійснено побудову Парето-фронту «швидкодія × якість».

7. Виконано функціонально-вартісний аналіз програмного продукту.

Методи дослідження базувалися на теорії стохастичних диференціальних рівнянь, чисельних схемах інтегрування СДР (метод Ейлера-Маруями з декомпозицією Холецького), теорії ймовірності та математичної статистики, методах глибокого навчання (рекурентні мережі, трансформери, Neural SDE), теорії ядерних методів (MMD), а також статистичних тестах оцінювання якості розподілів (KS, Wasserstein-1, ARCH-LM).

Наукова новизна роботи полягає у застосуванні для задачі сурогатного моделювання фінансових цін різнотипних нейромережевих архітектур (рекурентної LSTM-MDN, трансформерної Transformer-MDN, варіацій одновимірних Neural SDE та архітектури Neural SDE-GAN з двовимірним Heston-станом, що поєднує явну компоненту волатильності з adversarial-навчанням) у єдиній експериментальній постановці; запропоновано комбіновану функцію втрат $NLL + \lambda \cdot ACF - loss$ та експериментально обґрунтовано вибір ваги регуляризатора $\lambda = 0{,}1$; продемонстровано, що архітектура Neural SDE-GAN з 2D-станом (Y, v) досягла найкращих результатів за маргінальними метриками — 30–37% покращення Wasserstein-відстані порівняно з найкращою одновимірною Neural SDE архітектурою; параметр кореляції ρ між шумами ціни і волатильності було автоматично вивчено зі значення −0,7 до −0,775, що підтверджує здатність моделі відтворювати leverage effect.

Практичне значення роботи полягає у демонстрації того, що нейромережеві сурогати на основі Neural SDE можуть замінити класичний симулятор Heston у застосуваннях, де критичною є швидкодія (ризик-менеджмент, стрес-тестування, навчання моделей машинного навчання на синтетичних даних). Виявлене обмеження архітектур щодо кластеризації волатильності окреслює напрям подальших досліджень.

Структура роботи. Дипломна робота складається зі вступу, чотирьох розділів, висновків, переліку джерел посилання та двох додатків. У першому розділі викладено теоретичний апарат: концепція сурогатного моделювання, модель Heston, чисельна схема Ейлера-Маруями, стилізовані факти фінансових часових рядів та загальний огляд нейромережевих архітектур, що використовуються для часових рядів. У другому розділі обґрунтовано власні теоретичні положення роботи: вибір архітектур, комбінованої функції втрат і стратегії мультиактивного навчання. У третьому розділі описано програмну реалізацію та представлено результати експериментального дослідження. У четвертому розділі проведено функціонально-вартісний аналіз розробленого програмного продукту.

РОЗДІЛ 1 СУЧАСНІ ПІДХОДИ ДО ПОБУДОВИ НЕЙРОМЕРЕЖЕВИХ СУРОГАТНИХ МОДЕЛЕЙ ДИНАМІКИ ЦІН ФІНАНСОВИХ АКТИВІВ

1.1 Концепція сурогатного моделювання та класичні стохастичні моделі ціноутворення фінансових активів

Сурогатна модель (surrogate model, emulator) — це обчислювально ефективна апроксимація дорогої у виконанні базової моделі [6-9]. Концепція виникла у задачах інженерного моделювання, де повне розв'язання рівнянь у часткових похідних або великих систем диференціальних рівнянь є ресурсовитратним, і її поступово було адаптовано для задач фінансової математики [5]. У фінансовому контексті як «дорога» модель виступає стохастичний процес зі складною структурою (наприклад, модель Heston з її стохастичною волатильністю або моделі зі стрибками), а як сурогат — швидка нейромережева чи статистична модель, що навчена видавати траєкторії з тим самим розподілом.

Принципова відмінність сурогатної моделі від прогнозної полягає в тому, що сурогат не намагається передбачити конкретну майбутню ціну. Завданням сурогата є відтворення розподілу можливих майбутніх траєкторій, тобто закону стохастичного процесу. Якщо позначити істинний процес як $S_{t_{t\geq 0}}$ із умовним розподілом $P(S_{t+1:t+H}|F_t)$, то задача побудови сурогата формулюється як знаходження такого генератора $G_\theta(h)$, який при подачі контексту h повертає синтетичну траєкторію $\tilde{S}_{t+1:t+H}$ таку, що розподіл $G_\theta(h)$ статистично нерозрізнюваний з істинним умовним розподілом процесу. Така постановка підкреслює генеративний, а не прогнозний характер задачі [10, 11].

Області застосування сурогатних моделей у фінансах є численними [12, 13]. По-перше, це ризик-менеджмент: для оцінки показника Value-at-Risk або очікуваної втрати (Expected Shortfall) на портфелі активів необхідно

генерувати десятки тисяч траєкторій майбутніх змін цін, що при використанні класичних стохастичних моделей може займати години обчислень. По-друге, це ціноутворення похідних інструментів методом Монте-Карло: ціна європейського або азійського опціону, бар'єрного деривативa чи структурованого продукту обчислюється як середнє за виплатами на N згенерованих траєкторіях, де N зазвичай становить від 10^4 до 10^6. По-третє, це генерація стрес-сценаріїв та синтетичних даних для тестування торгових стратегій і навчання моделей машинного навчання, для яких реальних історичних даних може бути недостатньо [5].

Найпростішою стохастичною моделлю динаміки ціни активу є геометричний броунівський рух (GBM), що лежить в основі моделі Блека-Шоулза [4]. Він описується стохастичним диференціальним рівнянням:

$$dS_t = \mu \cdot S_t \cdot dt + \sigma \cdot S_t \cdot dW_t, \quad (1.1)$$

де S_t — ціна активу в момент часу t;

μ — детерміновний коефіцієнт дрейфу (середня очікувана дохідність);

σ — стала волатильність;

W_t — стандартний вінерівський процес.

Розв'язок цього рівняння має замкнуту форму:

$$S_t = S_0 \cdot exp\big((\mu - \sigma^2/2) \cdot t + \sigma \cdot W_t\big), \quad (1.2)$$

що дає можливість моделювати траєкторії аналітично, без чисельного інтегрування. GBM лежить в основі формули Блека-Шоулза для ціноутворення європейських опціонів. Однак GBM має принципове обмеження: стала волатильність σ не відповідає емпіричним даним, де волатильність змінюється в часі, демонструє кластеризацію (періоди високої і низької волатильності змінюють один одного) та залежить від рівня цін [2].

Для подолання цього обмеження було запропоновано модель стохастичної волатильності Гестона, яка дотепер залишається одним із базових інструментів кількісної фінансової математики [3]. У моделі Heston

ціна активу і миттєва дисперсія описуються системою двох корельованих стохастичних диференціальних рівнянь:

$$dS_t = \mu \cdot S_t \cdot dt + \sqrt{v_t} \cdot S_t \cdot dW_t^{(1)}, \quad (1.3)$$

$$dv_t = \kappa(\theta - v_t) \cdot dt + \sigma_v \cdot \sqrt{v_t} \cdot dW_t^{(2)}, \quad (1.4)$$

$$corr\left(dW^{(1)}, dW^{(2)}\right) = \rho, \quad (1.5)$$

де v_t — миттєва дисперсія (квадрат волатильності) у момент часу t;

κ — швидкість повернення дисперсії до довгострокового середнього значення θ;

σ_{v} — так звана «волатильність волатильності»;

ρ — кореляція між випадковими шоками ціни і дисперсії.

Параметр ρ зазвичай від'ємний для акцій ($\rho \approx -0{,}7$), що відтворює leverage effect — емпіричне спостереження, що падіння цін зазвичай супроводжується зростанням волатильності, а зростання — навпаки. Для забезпечення невід'ємності дисперсії на параметри моделі накладається умова Феллера: $2\kappa\theta \geq {\sigma_{\text{v}}}^2$, яка гарантує, що процес v_t залишається строго додатним. На практиці ця умова часто порушується, тому при чисельній реалізації використовуються спеціальні техніки запобігання від'ємним значенням v_t (повне зрізання, метод відбиття), що детально описано у підрозділі 1.2.

Модель Heston має кілька переваг порівняно з GBM. Її компонента волатильності побудована як квадратно-кореневий дифузійний процес (CIR-процес), вперше введений у роботі Кокса, Інгерсолла, Росса [14] для моделювання структури процентних ставок. По-перше, вона дозволяє відтворювати кластеризацію волатильності — періоди стабільності змінюються періодами підвищеної турбулентності, що відповідає реальній поведінці ринку. По-друге, від'ємна кореляція ρ створює лівосторонню асиметрію розподілу дохідностей і важчі ліві хвости, що відповідає емпіричному факту: великі падіння на ринку трапляються частіше, ніж великі зростання тієї ж величини. По-третє, модель Heston має квазіаналітичну

формулу для ціни європейських опціонів через характеристичну функцію, що робить її зручною для калібрування до ринкових даних [3].

Існують також розширення моделі Heston, які додають стрибки в ціні або у дисперсії (модель Bates [15], jump-diffusion models), а також моделі з пам’яттю (rough volatility models [16], rough Bergomi), які дозволяють точніше відтворювати окремі стилізовані факти. Однак для цілей цієї роботи модель Heston є оптимальним вибором: вона достатньо складна, щоб демонструвати основні стилізовані факти, і водночас достатньо проста, щоб її параметри мали зрозумілу фінансову інтерпретацію. Окрім того, для моделі Heston існує добре вивчена аналітика, яку можна використовувати як еталон при оцінюванні якості нейромережевих сурогатів [2, 3].

У контексті побудови нейромережевих сурогатних моделей класична модель Heston виконує роль так званої «синтетичної істини». Подальші розширення моделі вивчають питання опціонних бульбашок та аномалій у динаміці [17]. Її можна симулювати за відомими параметрами, отримати довільно великі вибірки траєкторій з відомими розподіловими властивостями, а далі навчати нейромережу на цих даних і перевіряти, наскільки точно вона відтворює властивості істинного процесу. Такий підхід дозволяє контрольовано оцінити якість сурогата у ситуації, коли «справжній» процес повністю відомий [5, 10].

Поряд із дослідженням на синтетичних даних важливим є експеримент на реальних фінансових часових рядах, де істинний породжуючий процес невідомий і може відрізнятися від будь-якої параметризованої моделі. У цьому випадку якість сурогата оцінюється за непрямими критеріями — відтворенням маргінального розподілу дохідностей, автокореляційної структури, стилізованих фактів. Саме така комбінація — навчання на синтетичних даних з відомою істиною та валідація на реальних активах з оцінкою стилізованих фактів — використовується у цій роботі для всебічного аналізу якості нейромережевих сурогатів [2, 10, 18].

1.2 Чисельне інтегрування стохастичних диференціальних рівнянь моделі Heston за схемою Ейлера-Маруями

Стохастичні диференціальні рівняння моделі Heston (1.3)–(1.5) у загальному випадку не мають замкнутого аналітичного розв'язку, тому траєкторії процесу отримуються чисельно. Найпоширенішою чисельною схемою для інтегрування СДР є схема Ейлера-Маруями — стохастичний аналог явного методу Ейлера для звичайних диференціальних рівнянь [2, 19]. Її простота, доведена теоретично збіжність і обчислювальна ефективність роблять її стандартним інструментом для побудови навчальних вибірок у задачах сурогатного моделювання.

Розглянемо узагальнене скалярне стохастичне диференціальне рівняння Ito:

$$dX_t = f(X_t, t) \cdot dt + g(X_t, t) \cdot dW_t, \tag{1.6}$$

де $f(X_t, t)$ — дрейф (детермінована складова);

$g(X_t, t)$ — дифузія (інтенсивність випадкового збурення);

W_t — стандартний вінерівський процес.

Схема Ейлера-Маруями полягає у дискретизації часового інтервалу [0, T] на N рівних кроків розміру $\Delta t = T/N$ та ітеративному обчисленні значень процесу за рекурентною формулою:

$$X_{n+1} = X_n + f(X_n, t_n) \cdot \Delta t + g(X_n, t_n) \cdot \Delta W_n, \tag{1.7}$$

де $\Delta W_n = \sqrt{(\Delta t)} \cdot \xi_n$;

$\xi_n \sim N(0,1)$ — незалежні стандартні нормальні випадкові величини.

Збіжність схеми у сильному сенсі має порядок $O\left(\sqrt{(\Delta t)}\right)$, у слабкому — порядок $O(\Delta t)$. Для генерації навчальних даних, де важлива відповідність розподілів, а не точність окремих траєкторій, слабкої збіжності зазвичай достатньо [2, 19].

Застосування схеми Ейлера-Маруями до моделі Heston має одну принципову складність: при чисельній дискретизації миттєва дисперсія v_t

може стати від'ємною, що суперечить її фізичній інтерпретації і призводить до взяття квадратного кореня від від'ємного числа у формулі дифузії ціни. Цю проблему вирішують за допомогою однієї з трьох технік [2]:

1) повне зрізання (full truncation): при обчисленні дрейфу і дифузії використовується значення $v_t^+ = max(v_t, 0)$, а саме v_t зберігається зі знаком; ця техніка вважається найстабільнішою і використовується у цій роботі;

2) часткове зрізання (partial truncation): v_t^+ використовується тільки у квадратному корені, але не в дрейфі;

3) метод відбиття (reflection): від'ємні значення замінюються на їх абсолютне значення.

Для двох корельованих вінерівських процесів $W_t^{(1)}$ та $W_t^{(2)}$ із кореляцією ρ прирости генеруються через декомпозицію Холецького:

$$\Delta W^{(1)} = \sqrt{\Delta t} \cdot Z_1, \Delta W^{(2)} = \sqrt{\Delta t} \cdot \left(\rho \cdot Z_1 + \sqrt{1 - \rho^2} \cdot Z_2\right), \quad (1.8)$$

де $Z_1, Z_2 \sim N(0,1)$ — незалежні стандартні нормальні випадкові величини.

Така схема гарантує правильну коваріаційну структуру шумів між ціною і дисперсією, що є критично важливим для відтворення leverage-ефекту.

З урахуванням техніки full truncation повна дискретизація системи Heston (1.3)–(1.5) у вигляді ітеративних оновлень логарифма ціни та дисперсії має вигляд:

$$v_n^+ = max(v_n, 0), \quad (1.9)$$

$$S_{n+1} = S_n \cdot exp\left((\mu - v_n^+/2) \cdot \Delta t + \sqrt{v_n^+} \cdot \Delta W_n^{(1)}\right), \quad (1.10)$$

$$v_{n+1} = v_n + \kappa(\theta - v_n^+) \cdot \Delta t + \sigma_v \cdot \sqrt{v_n^+} \cdot \Delta W_n^{(2)}. \quad (1.11)$$

Перехід до логарифма ціни у формулі (1.10) дозволяє точніше відтворювати геометричну природу руху ціни і запобігає від'ємним значенням $S_n + 1$, які можуть виникнути при звичайному застосуванні схеми Ейлера до самого S_t. Цей підхід є стандартом у фінансовій інженерії і використовується у переважній більшості систем для генерації траєкторій моделі Heston [2, 5].

Альтернативні чисельні схеми (схема QE Andersen [20]) забезпечують кращу точність при високій кореляції шуми ціни і дисперсії, проте для цілей цієї роботи Ейлер-Маруями з повним зрізанням залишається оптимальним балансом простоти і точності. Кінцевим результатом застосування описаної схеми є двовимірний масив траєкторій (S, v) розмірності N_paths times (N_steps + 1), де N_paths — кількість незалежних реалізацій процесу, N_steps — кількість часових кроків. Для практичного застосування у цій роботі використовуються типові параметри: $\mu = 0{,}05$, $\kappa = 2{,}0$, $\theta = 0{,}04$, $\sigma_{\mathrm{v}} = 0{,}3$, $\rho = -0{,}7$, $v_0 = 0{,}04$, $S_0 = 100$. Ці значення відповідають типовим параметрам акцій великих корпорацій (річна волатильність близько 20%) і забезпечують виражений leverage-ефект і кластеризацію волатильності у згенерованих траєкторіях [2, 3].

Перевагою методу Ейлера-Маруями у контексті побудови сурогатних моделей є його повна векторизація: при реалізації на сучасних чисельних бібліотеках (NumPy для CPU, PyTorch для GPU) усі N_paths траєкторій оновлюються паралельно на кожному часовому кроці, що дозволяє за прийнятний час генерувати сотні тисяч траєкторій. Це є необхідною умовою для формування достатньо великих навчальних вибірок для нейромережевих сурогатів і для проведення Монте-Карло порівнянь у задачах оцінки якості [5].

1.3 Стилізовані статистичні факти дохідностей фінансових активів

Стилізованими фактами (stylized facts) у фінансовій економетриці називають емпіричні закономірності, які стабільно спостерігаються у часових рядах дохідностей різних активів, різних ринків і різних періодів. На відміну від точкових властивостей конкретного активу, стилізовані факти є універсальними характеристиками всього класу фінансових процесів. Найбільш цитованою роботою, де ці закономірності систематизовано, є стаття Р. Конта 2001 року, у якій сформульовано одинадцять основних стилізованих

фактів [2, 21]. Здатність моделі відтворити ці факти є ключовим критерієм її якості у генеративному контексті: модель, що повторює лише два-три моменти розподілу, але не відтворює часову структуру або хвости, вважається недостатньо точною для практичного застосування.

Розглянемо основні стилізовані факти, які є важливими для цієї роботи.

Перший факт — відсутність лінійної автокореляції дохідностей. Якщо позначити логарифмічну дохідність як $r_t = ln(S_t/S_{t-1})$, то емпірична автокореляційна функція $\rho(k) = Corr(r_t, r_{t-k})$ є близькою до нуля для всіх лагів $k \geq 1$. Це є наслідком гіпотези ефективного ринку: якби існувала стійка лінійна залежність між поточною і минулою дохідностями, інвестори б її експлуатували і знищили б арбітражними операціями. На практиці на коротких часових масштабах (1 хвилина, 5 хвилин) спостерігається слабка від'ємна автокореляція через мікроструктурні ефекти, але на щоденних і вищих масштабах ACF фактично нульова [2].

Другий факт — важкі хвости розподілу дохідностей. Емпіричний розподіл r_t суттєво відхиляється від нормального: ексцес (excess kurtosis) типово становить 3–10, що означає значно більшу імовірність екстремальних значень, ніж передбачає гауссіан. Розподіл апроксимується степеневою функцією у хвостах: $P(|r_t| > x) \sim x^{-\alpha}$ з показником α між 2 і 4. Це означає, що так звані «події 6-сигма», які за нормальним розподілом мали б траплятися раз на мільйон років, на фінансових ринках відбуваються кілька разів на десятиліття. Важкі хвости є критичною властивістю для коректного розрахунку Value-at-Risk і Expected Shortfall [2].

Третій факт — кластеризація волатильності (volatility clustering). Незважаючи на відсутність автокореляції самих дохідностей, їх абсолютні значення $|r_t|$ та квадрати r_t^2 демонструють виражену додатну автокореляцію, що повільно загасає з лагом. Це означає, що великі рухи цін зазвичай супроводжуються великими рухами (як вгору, так і вниз), а періоди низької волатильності так само групуються. Періоди ринкової турбулентності (наприклад, криза 2008 року, COVID-шок березня 2020 року) можна

спостерігати як стійкі кластери високої волатильності. Стандартним статистичним тестом на наявність ARCH-ефекту (кластеризації) є тест Енгла, у якому перевіряється статистична значущість регресії r_t^2 на лагові значення $r_{t-1}^2, \dots, r_{t-p}^2$ [2, 22]. Кластеризація волатильності лежить в основі цілого класу моделей GARCH і є визначальною ознакою, яку повинна відтворювати будь-яка адекватна модель фінансового процесу.

Четвертий факт — повільне загасання ACF абсолютних значень (long memory). Якщо подивитися на автокореляційну функцію $|r_t|$ або r_t^2 на великих лагах (до 100–500 кроків), вона не падає до нуля експоненціально, як у класичних коротко-залежних процесах ARMA, а зменшується степенево. Цей факт неможливо описати лінійними моделями і вимагає або довгопам'ятних моделей (FIGARCH), або моделей зі стохастичною волатильністю, де повільне загасання дисперсії природно вбудоване [2].

П'ятий факт — leverage effect. Для більшості акцій спостерігається від'ємна кореляція між поточною дохідністю і майбутньою волатильністю: $Corr(r_t, |r_{t+k}|) < 0$ для невеликих лагів k. Економічно це означає, що падіння цін зазвичай супроводжується підвищенням волатильності у наступні кілька днів, тоді як зростання — навпаки, стабілізує ринок. Цей факт пояснюється кількома механізмами: по-перше, падіння ціни активу збільшує борг-до-капітал компанії і робить її більш ризиковою; по-друге, падіння створює паніку інвесторів і неоднозначність очікувань; по-третє, при падінні відключаються алгоритмічні стратегії, що збільшує шум. У моделі Heston leverage-ефект вбудовано через від'ємну кореляцію ρ між шоками ціни і дисперсії (1.5) [2, 3].

Шостий факт — aggregational gaussianity. При збільшенні масштабу часової агрегації (від денних дохідностей до тижневих, місячних, річних) розподіл поступово наближається до нормального. Це проявляється у зниженні ексцесу: для денних дохідностей акцій типовий ексцес становить 5–10, для місячних — 1–3, для річних — близько 0. Така залежність випливає з центральної граничної теореми за умови достатньої некорельованості

дохідностей на різних масштабах і є додатковим тестом адекватності моделі — модель, яка відтворює денний розподіл, але дає неправильний ексцес на місячному масштабі, не пройде цей тест [2].

Окрім перерахованих, до стилізованих фактів зараховують також асиметрію розподілу (від'ємну скошеність для більшості акцій), волатильність-обсяг кореляцію (Karpoff 1987 [23]), ефект «коарс-файн» волатильності (Müller et al. 1997) та інші. Однак для цілей переддипломної практики достатньо розглянути перші шість фактів — вони охоплюють найважливіші властивості розподілу і часової структури, які мають відтворюватися сурогатною моделлю.

З точки зору побудови нейромережевих сурогатів стилізовані факти виконують подвійну функцію. По-перше, вони слугують емпіричною ціллю: модель має генерувати траєкторії, у яких ці факти проявляються кількісно так само, як у реальних даних. По-друге, вони дозволяють діагностувати слабкі сторони архітектур: модель, побудована на основі простого MSE-loss над логарифмічними дохідностями, зазвичай добре відтворює моменти, але провалюється на ARCH-ефекті, тоді як модель з явним стохастичним інтегруванням (Neural SDE) краще ловить кластеризацію волатильності, але може втрачати точність у хвостах. Тому повне оцінювання сурогата має включати не один зведений показник, а набір метрик, кожна з яких відповідає одному стилізованому факту [2, 10, 11].

У третьому розділі цієї роботи для кожного з чотирьох досліджуваних активів (S&P 500, Apple, JPMorgan, Tesla) обчислюються і візуалізуються основні стилізовані факти як на реальних, так і на згенерованих траєкторіях. Для кількісної оцінки відповідності використовуються стандартні статистичні тести: ARCH-LM тест Енгла на кластеризацію волатильності, тест Колмогорова-Смирнова на відмінність розподілів, абсолютні відхилення моментів. Це дозволяє об'єктивно ранжувати моделі за здатністю відтворювати специфічні властивості фінансових часових рядів [2, 24, 25].

Типову картину перелічених стилізованих фактів для дохідностей індексу S&P 500 наведено на рис. 1.1.

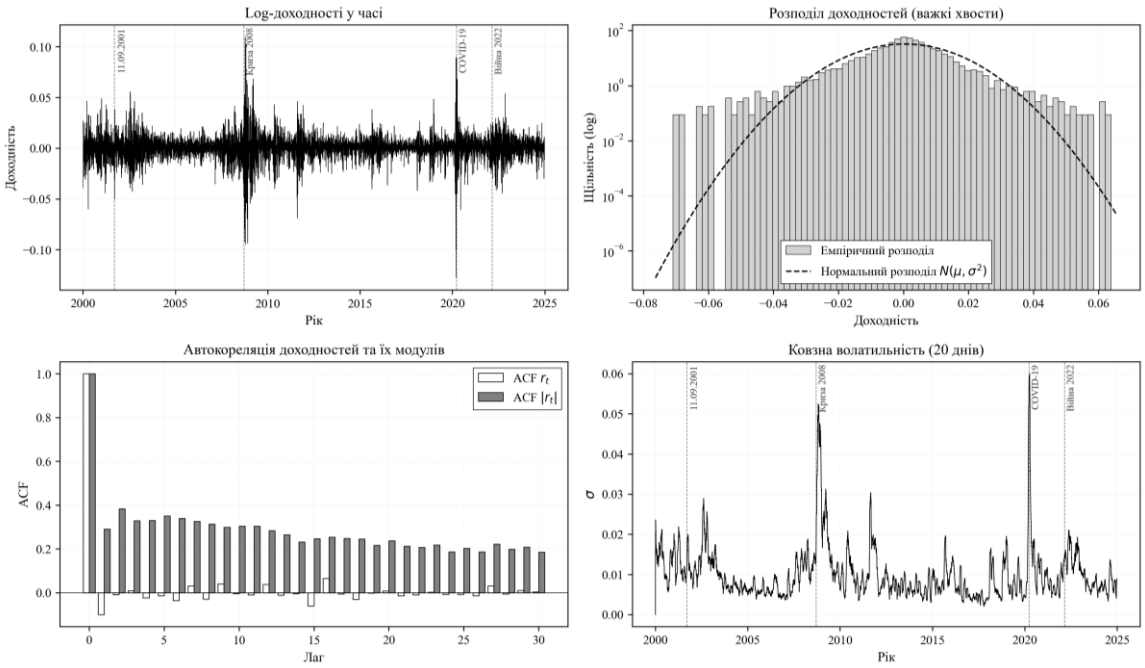


Рисунок 1.1 — Стилізовані факти дохідностей S&P 500

1.4 Нейромережеві архітектури та функції втрат для ймовірнісного моделювання часових рядів

Для побудови сурогатних моделей динаміки фінансових цін у цій роботі використовується чотири нейромережеві архітектури з ймовірнісним виходом. Усі вони мають спільну мету — за вхідним контекстом $h = (r_{t-L+1}, \dots, r_t)$ довжини L генерувати наступні H значень логарифмічних дохідностей, що мають той самий розподіл, що й істинний процес. Архітектури відрізняються тим, як вони параметризують умовний розподіл $p_\theta(r_{t+1:t+H}|h)$ та як використовують контекст [26, 27].

Базовим елементом перших двох архітектур є Mixture Density Network (MDN) — ймовірнісна голова, яку запропонував К. Бішоп у 1994 році [25].

Замість того щоб видавати одне детерміноване значення, MDN параметризує умовну щільність ймовірності як суміш K гаусіан:

$$p_\theta(r|h) = \Sigma_{k=1}^{K} \pi_k(h) \cdot N\left(r; \mu_k(h), \sigma_k^2(h)\right), \quad (1.12)$$

де $\pi_\text{k}(h)$ — ваги компонент, що задовольняють умови $\pi_\text{k} \geq 0, \Sigma_\text{k}\, \pi_\text{k} = 1$ (забезпечується softmax-активацією);

$\mu_\text{k}(h)$ — центри компонент;

$\sigma_\text{k}(h) > 0$ — стандартні відхилення (забезпечується експоненційною активацією з обмеженням зверху).

Усі три набори параметрів отримуються лінійним перетворенням останнього прихованого стану мережі. У роботі використано $K = 3$ компонентів та обмеження $\sigma_\text{k} \leq 0{,}05$, що відповідає реалістичному діапазону денних волатильностей. MDN-голова дозволяє моделювати мультимодальні і важкохвості розподіли, що є важливим для фінансових дохідностей.

Перша архітектура — LSTM + MDN. Рекурентна мережа з довгою короткостроковою пам’яттю (Long Short-Term Memory, LSTM), запропонована Хохрайтером і Шмідгубером у 1997 році [27], є класичним рішенням для моделювання часових рядів. У LSTM-комірці на кожному кроці зберігається прихований стан h_t і так званий cell state c_t, оновлення яких регулюється трьома гейтами — забування, входу і виходу. Це дозволяє мережі вибірково запам’ятовувати або стирати інформацію з минулих кроків, уникаючи проблеми зникаючого градієнта. У цій роботі використано двошарову LSTM з h_dim = 64 нейронами; на вхід подається послідовність логарифмічних дохідностей $(r_{t-L+1}, \ldots, r_t)$, а останній прихований стан другого шару подається у MDN-голову, яка видає параметри суміші для усіх H майбутніх кроків. Загальна кількість параметрів LSTM-моделі — близько 60 тисяч, що робить її найшвидшою у тренуванні серед розглянутих архітектур.

Друга архітектура — Transformer + MDN. Запропонована у роботі А. Васвані та співавторів у 2017 році [18], трансформер є архітектурою, що базується на механізмі self-attention. На відміну від LSTM, де інформація

поширюється послідовно, у трансформері кожна позиція може напряму взаємодіяти з будь-якою іншою через обчислення зваженої суми значень за формулою:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V, \quad (1.13)$$

де Q, K, V — матриці запитів, ключів і значень, отримані лінійними перетвореннями вхідної послідовності;

d_k — розмірність ключів.

Multi-head attention обчислює кілька таких карт паралельно і об'єднує їх. У цій роботі використано Transformer-Encoder з 2 шарами, 8 головами уваги, розмірністю моделі d_model = 96 і feed-forward шаром розміром 4 · d_model = 384. До вхідної послідовності додається синусоїдальне позиційне кодування для збереження порядку елементів. Останній прихований стан подається на MDN-голову з K = 3 компонентами.

Третя і четверта архітектури — Neural SDE (концептуальна основа: Neural ODE [28], розширена стохастичним членом) (нейромережеве стохастичне диференціальне рівняння) у двох варіантах. Концепція полягає в тому, щоб параметризувати дрейф і дифузію стохастичного диференціального рівняння нейронними мережами [11]:$f_\theta g_\theta$

$$dY_t = f_\theta(t, Y_t, h) \cdot dt + g_\theta(t, Y_t, h) \cdot dW_t, \quad (1.14)$$

де Y_t — внутрішня прихована змінна (кумулятивна log-дохідність);

h — опціональний контекст від енкодера.

Інтегрування рівняння виконується чисельно методом Ейлера-Маруями (1.7), а градієнти за параметрами θ обчислюються через автоматичне диференціювання у бібліотеці torchsde [29]. Згенеровані прирости ΔY_n інтерпретуються як log-дохідності r_n.

Варіант A (NSDE-A) — без енкодера контексту. Дрейф і дифузія залежать лише від поточного часу і стану Y_t, контекст не використовується. Така модель навчається загальній стохастичній динаміці ринку і не адаптується до конкретного активу чи історичного періоду. Її перевага —

простота і мала кількість параметрів (близько 25 тисяч); недолік — відсутність умовності.

Варіант B (NSDE-B) — з енкодером контексту. Перед інтегруванням SDE рекурентний енкодер (одношарова LSTM з h_enc = 32) обробляє вхідний контекст h і видає вектор стану ринку $c \in \mathbb{R}^{16}$. Цей вектор додається до входу нейронних мереж дрейфу і дифузії як додаткова умова. Таким чином, динаміка SDE стає залежною від історичного контексту, що дозволяє моделі по-різному поводитися у режимах високої і низької волатильності. Загальна кількість параметрів — близько 35 тисяч.

Для всіх чотирьох архітектур основною функцією втрат є негативна логарифмічна правдоподібність (negative log-likelihood, NLL). Для MDN-моделей NLL обчислюється напряму:

$$L_{NLL} = -\Sigma_t log \Sigma_k \pi_k(h) \cdot N\left(r_t; \mu_k(h), \sigma_k^2(h)\right). \quad (1.15)$$

Для Neural SDE використовується step-NLL — наближення NLL через дискретний перехідний розподіл: на кожному кроці інтегрування $r_n \sim N\left(f_\theta \cdot \Delta t, g_\theta^2 \cdot \Delta t\right)$ і відповідна логарифмічна щільність підсумовується. Мінімізація NLL еквівалентна мінімізації KL-дивергенції між модельним і емпіричним розподілом, що відповідає задачі максимально точного відтворення розподілу [26].

Для покращення відтворення часової структури (зокрема кластеризації волатильності) у функцію втрат додано допоміжний ACF-loss — диференційований штраф за невідповідність автокореляційної функції квадратів згенерованих і реальних дохідностей на перших L_max лагах:

$$L_{ACF} = \frac{1}{L_{max}} \Sigma_{k=1}^{L_{max}} \left(\rho_{gen}(r^2, k) - \rho_{real}(r^2, k)\right)^2, \quad (1.16)$$

де $\rho_{gen}(r^2, k)$ і $\rho_{real}(r^2, k)$ — автокореляції квадратів дохідностей у згенерованій і реальній вибірках відповідно на лагу k.

У цій роботі $L_{max} = 5$ і коефіцієнт ACF-loss установлено $\lambda_{ACF} = 0{,}1$ (обґрунтування цього вибору наведено у підрозділі 2.2 на основі ablation-експерименту). Сукупна функція втрат має вигляд $L = L_{NLL} + \lambda_{ACF} \cdot L_{ACF}$ [10, 11].

Тренування здійснюється методом Adam [30] зі швидкістю навчання 10^{-3} і косинусним розкладом (cosine annealing) протягом 20 епох. Для запобігання вибуховим градієнтам у Neural SDE використано gradient clipping з порогом 1.0 по нормі. Розмір батчу — 256 вікон, кожне вікно містить $L = 60$ днів контексту і $H = 20$ днів прогнозу.

Оцінювання якості сурогата проводиться за допомогою набору метрик, що покривають різні аспекти відтворення процесу. Похибки моментів — абсолютні відхилення середнього, стандартного відхилення, скошеності та ексцесу: $|\Delta\mu|$, $|\Delta\sigma|$, $|\Delta skew|$, $|\Delta kurt|$. Кожна з цих похибок характеризує одну сторону розподілу — центр, ширину, асиметрію і важкість хвостів.

Розподілові відстані вимірюють інтегральну відмінність між модельним і реальним розподілами. Статистика Колмогорова-Смирнова — максимальне абсолютне відхилення між емпіричними функціями розподілу:

$$D_{KS} = sup_x \left| F_{real}(x) - F_{gen}(x) \right|. \tag{1.17}$$

Метрика Вассерштейна першого порядку (Earth Mover's Distance) — інтегральна відстань, інтерпретується як середня «вартість» переносу маси з одного розподілу до іншого:

$$W_1(P, Q) = \int \left| F_P^{-1}(u) - F_Q^{-1}(u) \right| du. \tag{1.18}$$

Maximum Mean Discrepancy (MMD), запропонована А. Греттоном і співавторами у 2012 році [24], — метрика, побудована через ядерні відображення в RKHS:

$$MMD^2(P, Q) = E[k(x, x''')] - 2 \cdot E[k(x, y)] + E[k(y, y''')], \tag{1.19}$$

де $k(x, y) = exp(-|x - y|^2 / 2\sigma^2)$ — гаусове RBF-ядро.

MMD є симетричною, обмеженою знизу нулем (рівність розподілів) і характеристичною — $MMD = 0$ еквівалентно $P = Q$. Параметр σ обирається евристикою медіани попарних відстаней. MMD є однією з основних метрик у роботі QuantGAN [10] і доповнює Wasserstein, оскільки чутлива до інших аспектів розподілу.

Для перевірки відтворення часової структури використовуються ARCH-LM тест Енгла (статистика тесту на наявність ARCH-ефекту в квадратах дохідностей) і leverage effect (кореляція r_t і $|r_{t+1}|$). Похибки за цими статистиками — абсолютні відхилення значень у згенерованій і реальній вибірках. Окрім того, для оцінювання генерації траєкторій на середньостроковому горизонті обчислюються розподіл максимальних просадок (max drawdown), ймовірність покриття прогнозного інтервалу (coverage probability) для рівнів 50%, 80%, 95% та похибка медіани ансамблю відносно реальної траєкторії [2, 5, 10].

Описаний набір метрик дозволяє провести комплексне оцінювання сурогатної моделі: перші чотири метрики характеризують маргінальний розподіл, статистики KS і Wasserstein — розподілову відмінність, MMD — інтегральну подібність розподілів, ARCH-LM і leverage — часову структуру, метрики траєкторій — поведінку моделі на горизонтах прогнозування. Жодна окрема метрика не дає повної картини якості, тому у Розділі 2 моделі порівнюються за всіма метриками одночасно, а підсумкові висновки робляться на основі узгодженості результатів за більшістю критеріїв.

1.5 Висновки до розділу 1

У теоретичному розділі викладено основні концепції, необхідні для побудови нейромережевих сурогатних моделей динаміки цін фінансових активів. Розглянуто поняття сурогатної моделі як обчислювально ефективної апроксимації базового стохастичного процесу, мета якої — генерувати

траєкторії, статистично нерозрізнювані з істинним процесом, а не передбачати конкретні майбутні значення цін. Це принципово відрізняє генеративний підхід від детермінованих прогнозних моделей і визначає вибір критеріїв оцінювання якості [5, 10].

Як еталонна стохастична модель ціноутворення обрано модель стохастичної волатильності Heston, що описує ціну активу і миттєву волатильність системою двох корельованих стохастичних диференціальних рівнянь. Модель дозволяє відтворити основні стилізовані факти фінансових рядів — кластеризацію волатильності, важкі хвости розподілу та leverage-ефект — і має зрозумілу фінансову інтерпретацію параметрів. Для чисельної реалізації використано схему Ейлера-Маруями з технікою повного зрізання дисперсії і логарифмічним переходом, що є стандартом фінансової інженерії [2, 3, 19].

Розглянуто шість основних стилізованих фактів за класифікацією Конта (2001): відсутність лінійної автокореляції дохідностей, важкі хвости, кластеризацію волатильності, повільне загасання ACF абсолютних значень, leverage-ефект та aggregational gaussianity. Кожен факт відіграє роль одночасно емпіричної цілі (модель має його відтворювати) і діагностичного інструменту (порушення вказує на недосконалість архітектури) [2].

Описано нейромережеві архітектури, що використовуються у роботі: LSTM з MDN-головою, Transformer з MDN-головою, Neural SDE без енкодера контексту (варіант A) і Neural SDE з енкодером (варіант B). Архітектури охоплюють весь спектр від класичної рекурентної до сучасної моделі на основі стохастичного інтегрування, що дозволяє провести змістовне порівняння. Розглянуто функції втрат — негативну логарифмічну правдоподібність, диференційований ACF-loss та їх комбінацію — і набір метрик оцінювання (моменти, KS, Wasserstein, MMD, ARCH-LM, leverage, метрики траєкторій). Цей теоретичний апарат є основою для практичної реалізації та експериментальних досліджень, представлених у розділі 2 [10, 11, 18, 24, 25, 26, 27].

РОЗДІЛ 2 РОЗРОБКА СУРОГАТНИХ МОДЕЛЕЙ ДЛЯ ВІДТВОРЕННЯ ДИНАМІКИ ЦІН ФІНАНСОВИХ АКТИВІВ

2.1 Архітектурний вибір MDN-моделей

Перші дві архітектури роботи побудовані за принципом «послідовний енкодер + імовірнісна голова». Послідовний енкодер відображає вхідне вікно контексту довжиною $L = 60$ днів у вектор латентного представлення; імовірнісна голова Mixture Density Network (MDN) перетворює це представлення у параметри суміші K гаусових компонент, з якої семплюється кожне наступне значення логарифмічної дохідності.

Вибір MDN-голови мотивований такими міркуваннями. По-перше, емпіричний розподіл дохідностей фінансових активів є важкохвостим і несиметричним — одна гаусова компонента не здатна адекватно його апроксимувати, тоді як суміш кількох компонент дозволяє моделювати мультимодальні розподіли та ефект «жирних хвостів». По-друге, MDN природно інтегрується у функцію втрат через явну формулу логарифмічної правдоподібності, що спрощує оптимізацію градієнтними методами.

Експериментально обрано конфігурацію з $K = 3$ компонент. Цього достатньо для відтворення асиметрії та важких хвостів, водночас модель залишається досить компактною — для LSTM-варіанта це ~60 тисяч параметрів, для Transformer-варіанта ~800 тисяч. Більша кількість компонент ($K = 5, K = 7$) не давала суттєвого покращення метрик і збільшувала ризик перенавчання.

Важливою деталлю реалізації є обмеження зверху на стандартні відхилення MDN-компонент: $\sigma_k \leq \sigma_{max}$, де σ_{max} — гіперпараметр. Початково обрано $\sigma_{max} = 0{,}01$ (1% денна волатильність), що відповідало типовим режимам ринку, але при перевірці на кризових періодах (2008, 2020)

виявилось недостатнім — модель не могла генерувати екстремальні денні рухи. Після підвищення до $\sigma_{max} = 0{,}05$ (5% денна волатильність) модель навчилася відтворювати важкі хвости, не втрачаючи якості у нормальних умовах. Це обґрунтоване рішення є одним із ключових для відтворення кризових режимів цін активів.

Послідовний енкодер реалізовано у двох варіантах: рекурентна мережа LSTM (як класичний інструмент для часових рядів) та Transformer-Encoder (сучасна архітектура з механізмом self-attention). LSTM обрано з двома прихованими шарами розмірності 64; Transformer — з 4 шарами, 4 головами уваги, розмірністю моделі d_model = 128 та feed-forward шаром розміру 512. Гіперпараметри Transformer-варіанта вибрано на основі повного 3D grid search (27 конфігурацій), результати якого детально подано у підрозділі 3.3.

Загальну схему обох ймовірнісних MDN-архітектур наведено на рис. 2.1.

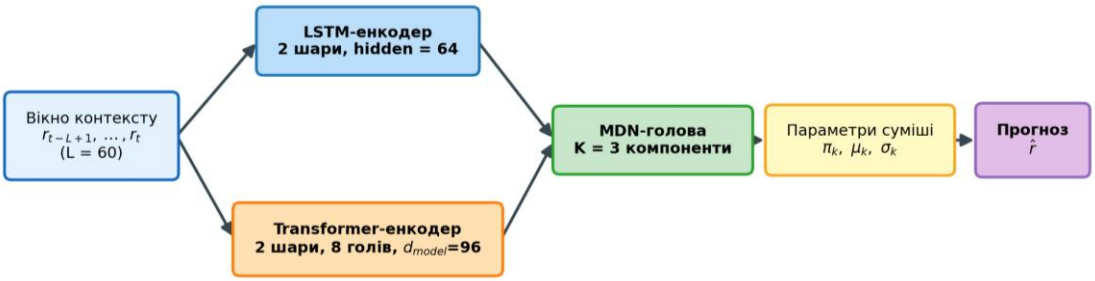


Рисунок 2.1 — Архітектури ймовірнісних моделей LSTM-MDN і Transformer-MDN

2.2 Дві варіації Neural SDE: з контекстним енкодером та без

Третя і четверта архітектури роботи — нейромережеві стохастичні диференціальні рівняння (Neural SDE). На відміну від MDN-моделей, де кожен новий крок семплюється незалежно, Neural SDE задає стохастичну динаміку прихованого стану через диференціальне рівняння Ітo з параметризованими нейронними мережами дрейфом $f_\theta(t, Y_t)$ і дифузією $g_\theta(t, Y_t)$. Рівняння

інтегрується чисельно методом Ейлера-Маруями, а градієнти обчислюються через автоматичне диференціювання у бібліотеці torchsde.

Запропоновано дві варіації моделі, які різняться наявністю контекстного енкодера. Варіант A (NSDE-A) — без енкодера: дрейф і дифузія залежать лише від поточного часу t і прихованого стану Y_t. Така модель навчається загальній стохастичній динаміці ринку і не враховує специфіки конкретного активу чи вхідного контексту. Варіант B (NSDE-B) — з LSTM-енкодером, який обробляє вхідний контекст h і видає вектор стану ринку $c \in \mathbb{R}^{16}$, що додається до t і Y_t як входи у мережі дрейфу та дифузії. Цей варіант здатен враховувати поточний режим ринку (зростання, корекція, висока волатильність) при генерації майбутніх траєкторій.

Обґрунтуванням такого порівняння є питання: чи дає контекстний енкодер суттєвий приріст якості сурогата? Якщо так — це підтверджує важливість сприйняття поточного стану ринку. Якщо ні — стохастична структура самого Neural SDE здатна вловлювати ринкові закономірності без явного контексту. Експериментальна частина роботи (підрозділ 3.4) показує, що обидва варіанти дають близькі результати за всіма розподіловими метриками, однак NSDE-B дає трохи кращу calibration на довгих горизонтах (coverage probability близький до номінальних рівнів).

Інші особливості реалізації Neural SDE: дрейф і дифузія параметризовано двошаровою MLP з Tanh-активаціями та прихованим розміром 64, що забезпечує гладкість функцій (важлива умова стабільності чисельного інтегрування СДР); дифузія обмежена зверху значенням $g_{max} = 0{,}08$ для запобігання нестабільності у регіонах низької дисперсії; кількість кроків інтегрування на одне передбачення дорівнює довжині прогнозного горизонту $H = 20$, що зберігає поточасову інтерпретацію.

Структуру обох варіацій Neural SDE показано на рис. 2.2.

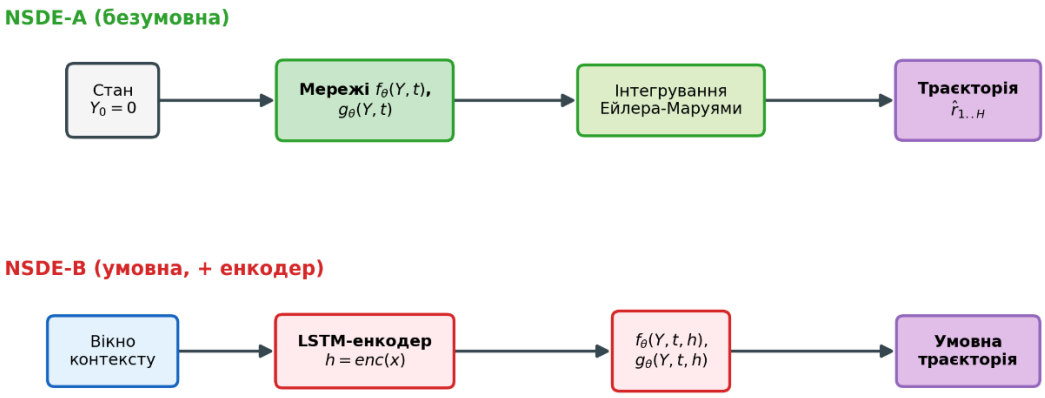


Рисунок 2.2 — Дві варіації Neural SDE: без енкодера (NSDE-A) та з контекстним енкодером (NSDE-B)

2.3 Двовимірний Neural SDE-GAN зі станом Heston-типу

Поряд з варіантами A і B Neural SDE, що навчаються мінімізацією покрокової негативної логарифмічної правдоподібності, у роботі досліджено альтернативний режим навчання — adversarial, або GAN-підхід, запропонований у роботі Кіджера зі співавторами «Neural SDEs as Infinite-Dimensional GANs» (2021). Ідея полягає у тому, щоб замість покроково-узгодженої логарифмічної щільності оптимізувати загальну розподілову подібність повних траєкторій через гру двох мереж — генератора (Neural SDE) і дискримінатора (класифікатора реальних і синтетичних послідовностей).

У роботі реалізовано п'яту архітектуру — NeuralSDE-GAN, у якій.

1. Генератор реалізовано як двовимірне стохастичне диференціальне рівняння Heston-типу: стан (Y_t, v_t), де Y_t — кумулятивний логарифм ціни, v_t — миттєва дисперсія з власною стохастичною динамікою. Дрейф і дифузія обох компонент параметризуються MLP, корельовані шуми dW^1/dW^2 пов'язані через параметр ρ (typically ≈ −0,7), що моделює leverage effect. Початкове значення v_0 ініціалізується контекстно-залежною головою з рухомої дисперсії останніх днів. Така архітектура має явну компоненту волатильності — те, чого не мають одновимірні NSDE-A/B варіанти.

2. Дискримінатор — двошарова двонаправлена LSTM (BiLSTM) з 64 прихованими одиницями і feed-forward головою, що приймає на вхід об'єднану послідовність контексту і прогнозу (довжина 80 кроків) та видає скалярну оцінку «реалістичності».

3. Тренування виконується у режимі WGAN-GP (Wasserstein GAN з gradient penalty): дискримінатор оптимізує оцінку Earth-Mover-відстані між розподілами реальних і синтетичних послідовностей, генератор — мінімізує її. Параметр штрафу за градієнт $\lambda_{GP} = 10$ використовується для ліпшицевості дискримінатора, кратність оновлень дискримінатора n_critic = 3 на одне оновлення генератора, оптимізатор — Adam з $\beta = (0{,}5, 0{,}9)$.

4. До loss-функції генератора зберігається той самий ACF-регуляризатор з вагою $\lambda_{ACF} = 0{,}1$, що й у NLL-варіантах. Це забезпечує справедливе порівняння: різниця між моделями полягає виключно у виборі основного критерію навчання (NLL vs WGAN), а не у наявності ACF-регуляризації.

Мотивація додавання adversarial-альтернативи така.

1. NLL покрокова оптимізація мінімізує локальну перехресну ентропію — модель може ідеально вловлювати маргінальний розподіл, але не отримує сигналу про відповідність ансамблів цілих траєкторій. Adversarial-критерій (раніше успішно застосований для генерації часових рядів — TimeGAN [31], CSDI на основі дифузійних моделей [32], глибокий хеджинг Buehler [33]) навпаки оцінює відразу всю траєкторію через дискримінатор, що теоретично краще для відтворення часової структури (кластеризації волатильності, leverage-ефекту).

2. GAN-підхід принципово відмінний методологічно, і його порівняння з NLL-варіантами дає змогу зробити висновок щодо переваг кожного режиму саме у задачі сурогатного моделювання фінансових цін.

3. NeuralSDE-GAN є станом-мистецтва у літературі (Kidger et al., 2021), і його реалізація у єдиному експериментальному пайплайні з рештою архітектур є важливим елементом наукової новизни роботи.

Ризик GAN-підходу полягає у потенційно нестабільному навчанні і mode collapse — типовій патології, коли генератор зводить розподіл до вузької моди. Використання WGAN-GP замість vanilla GAN значно знижує цей ризик, але не виключає його повністю. Стабільність навчання моніториться через відстань Wasserstein-1 між реальними і згенерованими дохідностями на val-вибірці, обчислювану наприкінці кожної епохи.

Загальну схему змагального навчання Neural SDE-GAN наведено на рис. 2.3.

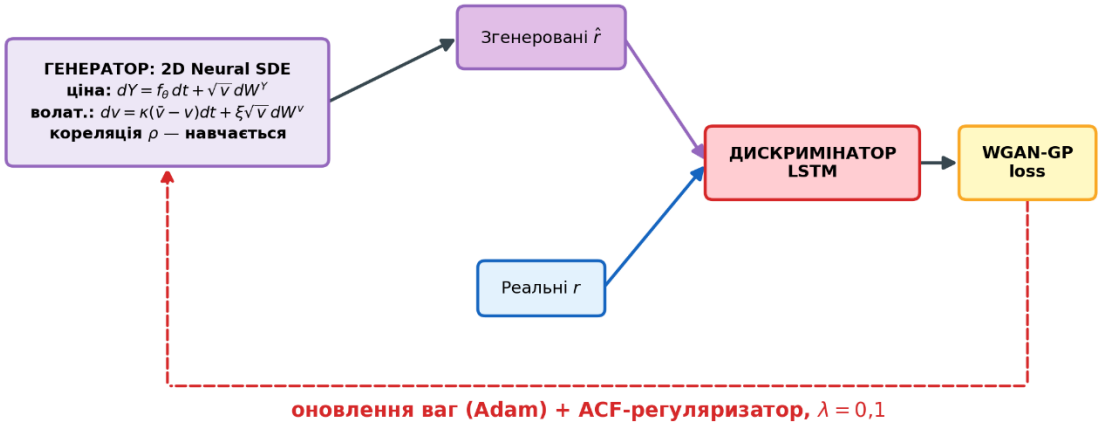


Рисунок 2.3 — Архітектура Neural SDE-GAN зі станом Heston-типу

2.4 Складена функція втрат NLL + ACF-loss

Для всіх чотирьох архітектур основною компонентою функції втрат є негативна логарифмічна правдоподібність (negative log-likelihood, NLL). Для MDN-моделей вона обчислюється напряму через явну формулу щільності суміші. Для Neural SDE використовується step-NLL — наближення NLL через дискретний перехідний розподіл: на кожному кроці інтегрування $r_n \sim N\left(f_\theta \cdot \Delta t, g_\theta^2 \cdot \Delta t\right)$, і відповідна логарифмічна щільність підсумовується по всіх кроках траєкторії.

Чиста NLL-оптимізація призводить до мінімізації перехресної ентропії між прогнозним і реальним розподілами на кожному кроці незалежно, але не

накладає обмежень на часову структуру серій. Як наслідок, генеровані вибірки добре відтворюють маргінальний розподіл, але втрачають ефект кластеризації волатильності. Для часткового врахування цього ефекту у функцію втрат додано допоміжний ACF-loss — диференційований штраф за невідповідність автокореляційної функції квадратів дохідностей між згенерованою і реальною вибірками. Сукупна функція втрат має вигляд:

$$L = L_{NLL} + \lambda_{ACF} \cdot L_{ACF}$$

Вибір ваги регуляризатора λ_{ACF} обґрунтовано окремим ablation-експериментом на сітці значень $\{0{,}0;\ 0{,}05;\ 0{,}1;\ 0{,}5;\ 1{,}0\}$. При $\lambda = 0$ ACF-loss вимкнено і модель оптимізує лише NLL: маргінальний розподіл відтворюється точно, але часова структура практично відсутня. Зі зростанням λ модель починає посилено штрафувати невідповідність ACF, але одночасно жертвує точністю відтворення маргінального розподілу. Дослідження показало, що $\lambda_{ACF} = 0{,}1$ є оптимальним компромісом: ACF квадратів дохідностей покращується помітно порівняно з $\lambda = 0$, тоді як Wasserstein-відстань і KS-статистика залишаються близькими до оптимальних. При $\lambda \geq 0{,}5$ Wasserstein значно зростає (модель «жертвує» розподілом задля ACF), а при $\lambda = 1{,}0$ моделі стають нестабільними. Вибране значення $\lambda_{ACF} = 0{,}1$ використовується у всіх подальших експериментах.

2.5 Стратегія мультиактивного навчання

Ключовим архітектурним рішенням роботи стало використання спільного навчання на об’єднаних train-вибірках з усіх чотирьох активів. Замість тренування окремої моделі для кожного активу (S&P 500, AAPL, JPM, TSLA) одна нейромережа бачить тренувальні вікна з усіх активів одночасно. Така стратегія мотивована трьома міркуваннями.

1. Фінансові ряди різних активів мають подібну стохастичну природу і спільні стилізовані факти (відсутність лінійної автокореляції, важкі

хвости, кластеризація волатильності, leverage-ефект). Тому модель, навчена на ширшому корпусі, краще генералізує і не перенавчається на конкретні особливості одного активу.

2. Об’єднана вибірка містить значно більше тренувальних вікон для кожної моделі (близько 18 000 проти 4 500 на окремий актив), що покращує статистичну ефективність навчання та зменшує дисперсію оцінок параметрів моделі.

3. Мультиактивна стратегія дозволяє одній моделі генерувати траєкторії для будь-якого активу з навчального набору без повторного тренування, що принципово важливо для практичних задач портфельного аналізу.

Схему мультиактивного навчання з комбінованою функцією втрат показано на рис. 2.4.

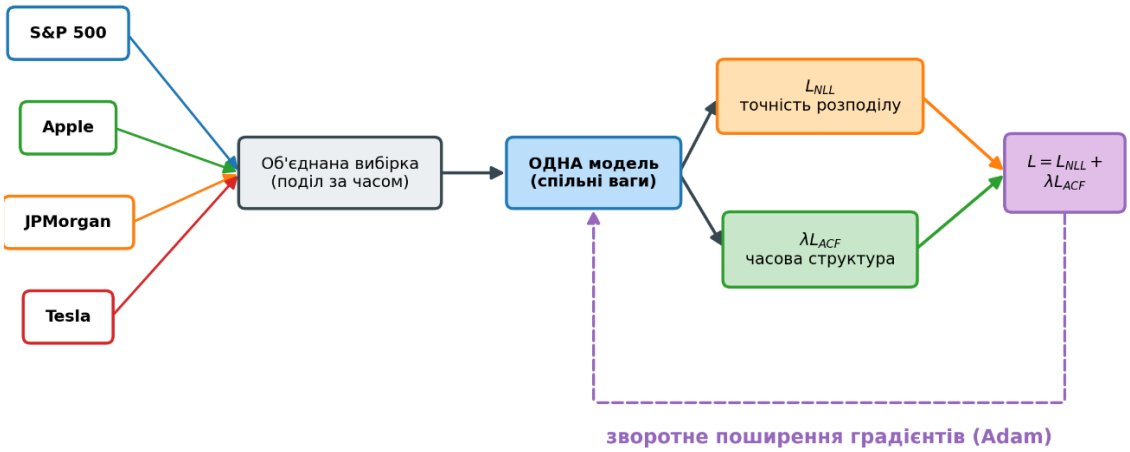


Рисунок 2.4 — Схема мультиактивного навчання та комбінованої функції втрат

2.6 Висновки до розділу 2

У другому розділі обґрунтовано чотири ключові архітектурні і методологічні рішення роботи. По-перше, вибрано імовірнісну MDN-голову з $K = 3$ компонентами для рекурентного і трансформерного енкодерів — обґрунтований вибір, що дозволяє моделювати асиметрію і важкі хвости фінансових дохідностей. Експериментально визначено оптимальне обмеження $\sigma_{max} = 0{,}05$ для відтворення кризових режимів. По-друге, запропоновано порівнювати дві варіації Neural SDE — без контекстного енкодера і з LSTM-енкодером — для відповіді на питання про важливість сприйняття поточного ринкового режиму у задачі сурогатного моделювання. По-третє, обґрунтовано складену функцію втрат NLL + λ·ACF-loss та шляхом ablation-експерименту визначено оптимальне значення $\lambda_{ACF} = 0{,}1$, яке забезпечує компроміс між якістю маргінального розподілу і відтворенням часової структури. По-четверте, обґрунтовано стратегію мультиактивного навчання як засіб покращення статистичної ефективності та узагальнюваності моделей сурогатів.

Усі три теоретичні рішення підтверджено результатами експериментальних досліджень, наведеними у третьому розділі.

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ СУРОГАТНИХ МОДЕЛЕЙ З СИМУЛЯЦІЯМИ ТРАЄКТОРІЙ

3.1 Постановка обчислювального експерименту, опис даних та процедура навчання

Експериментальне дослідження проведено на чотирьох фінансових активах різного типу та різної волатильності, що дозволяє оцінити загальну застосовність розроблених сурогатів. До набору входять: індекс широкого ринку S&P 500 (тикер ^GSPC) — представник широко диверсифікованого ринку з помірною волатильністю; акції корпорації Apple (AAPL) — великий технологічний актив; акції банку JPMorgan Chase (JPM) — представник фінансового сектору; акції Tesla (TSLA) — приклад високоволатильного активу з різкими ціновими рухами та сильним впливом інформаційних шоків. Дані завантажено через бібліотеку yfinance за період з 1 січня 2000 до 31 грудня 2024 року; для активу TSLA доступний період починається з 2010 року після IPO. Загальна кількість доступних логарифмічних дохідностей становить близько 6 200 точок для S&P 500, 6 100 для Apple, 6 100 для JPMorgan і 3 500 для Tesla (рис. 3.1).

Логарифмічні дохідності обчислюються за стандартною формулою:

$$r_t = ln\left(\frac{S_t}{S_{t-1}}\right), \tag{2.1}$$

де S_t — ціна закриття активу у день t.

Перехід до логарифмічних дохідностей робить ряд більш стаціонарним і дозволяє використовувати єдину модель для активів з різними рівнями цін (від ~5 500 у S&P 500 до ~200 у Apple).

Для формування вхідних і вихідних послідовностей використано sliding-window підхід. Кожне вікно складається з контексту довжиною $L = 60$ днів і

горизонту прогнозу $H = 20$ днів, тобто загальна довжина вікна $L + H = 80$. Це відповідає приблизно трьом місяцям контексту і одному місяцю прогнозу за фінансовою конвенцією (21 торговий день у місяці). Стрід формування вікон установлено $stride = 2$, що дає велику кількість вікон при помірному перекритті.

Розподіл на навчальну, валідаційну і тестову вибірки виконано часовим способом (temporal split): для кожного активу 70% перших за часом точок становлять train-вибірку, наступні 15% — val-вибірку, останні 15% — test-вибірку. Між сегментами вставлено буфер (gap) довжиною $L + H = 80$ днів, що повністю усуває можливість перекриття вікон різних вибірок і запобігає витоку даних (data leakage). Коректність розподілу перевірена програмними assertion-перевірками. Після обробки навчальна вибірка містить близько 10 500 вікон, валідаційна — 2 200, тестова — 2 200 (об’єднано по всіх активах).

Ключовим архітектурним рішенням стало використання спільного навчання на об’єднаних train-вибірках з усіх чотирьох активів. Замість тренування окремої моделі для кожного активу одна нейромережа бачить різноманітні фінансові ряди і вчиться загальним стилізованим фактам. Це працює як форма регуляризації: для глибоких моделей з сотнями тисяч параметрів об’єднання даних знижує ризик перенавчання і покращує здатність генералізувати. Оцінювання якості при цьому виконується окремо для кожного активу на його власній тестовій вибірці, що дозволяє виявити, як модель адаптується до специфіки кожного фінансового інструмента. Як альтернативний експеримент проведено також single-asset навчання на Tesla для контролю переваг мультиактивного підходу (див. підрозділ 3.4).

Усі чотири моделі (LSTM-MDN, Transformer-MDN, NSDE-A, NSDE-B) тренуються за єдиною процедурою. Оптимізатор — Adam [30] (у експериментах також досліджено варіант AdamW з decoupled weight decay [34]) зі стартовою швидкістю навчання 10^{-3}. Розклад швидкості навчання — косинусний (cosine annealing) до нуля впродовж 20 епох. Розмір батчу — 256 вікон. Для запобігання вибуховим градієнтам у Neural SDE використано

gradient clipping за нормою з порогом 1.0. Для MDN-моделей основною функцією втрат є NLL з ваговим коефіцієнтом $\lambda_{ACF} = 0{,}1$ для ACF-loss; для Neural SDE — step-NLL без ACF-loss (його ефект забезпечується самою стохастичною структурою моделі).

Для відтворюваності результатів використано фіксоване початкове значення генератора (псевдо)випадкових чисел (сід). У базовому експерименті його встановлено seed = 42, що визначає однаковий старт ваг, однаковий порядок пакетів (батчів) і однакові вибірки з MDN-суміші. Розглянуто також варіант з трьома різними сідами для multi-seed усереднення метрик (mean ± std), але для цього звіту наведено результати одного запуску з фіксованим сідом, що дозволяє безпосередньо звіряти числа з графіками.

Обчислення проведено на GPU NVIDIA з підтримкою CUDA. Повний цикл тренування всіх чотирьох моделей займає близько 25–30 хвилин для одного сіду. Згенеровані метрики і графіки автоматично зберігаються у тимчасові директорії з відповідними часовими мітками. Для статистичних обчислень (KS-тест, Wasserstein, MMD, ARCH-LM, leverage) використано бібліотеку scipy.

Часові ряди логарифмічних дохідностей чотирьох досліджуваних активів наведено на рис. 3.1.

Рисунок 3.1 — Часові ряди логарифмічних дохідностей чотирьох активів

Для якісної інтерпретації результатів попередньо проведено експлораторний аналіз стилізованих фактів реальних даних кожного активу. Емпіричні розподіли логарифмічних дохідностей мають виражений ексцес: для S&P 500 — 2.04, Apple — 2.24, JPMorgan — 5.37 (з впливом криз 2008 і 2020 років), Tesla — 3.97. Скошеність варіюється від лівої (S&P 500: -0.23, JPMorgan: +0.37) до симетричної (Apple: +0.02). Стандартне відхилення денних дохідностей становить 1.09% для S&P 500, 1.70% для Apple, 1.55% для JPMorgan і 3.69% для Tesla, що підтверджує статус Tesla як найбільш волатильного активу у наборі. ARCH-LM статистика на лагу 5 досягає 448 для S&P 500, 305 для Apple, 65 для JPMorgan і 14 для Tesla — у всіх випадках значно вище критичного значення $\chi^2_{0.95(5)} \approx 11{,}07$, що підтверджує наявність вираженого ARCH-ефекту в реальних даних.

3.2 Архітектура програмного коду

Програмна реалізація сурогатних моделей виконана на мові Python з використанням бібліотеки PyTorch [35] для нейромережевих обчислень, бібліотек NumPy і pandas для роботи з масивами і часовими рядами, matplotlib для побудови графіків та torchsde для чисельного інтегрування Neural SDE з підтримкою автоматичного диференціювання.

Загальна архітектура коду модульна. Основні модулі програмної системи:

Модуль diploma_final.py містить визначення усіх нейромережевих моделей (класи LSTMSurrogate, TransformerSurrogate, NeuralSDESurrogateA, NeuralSDESurrogateB), функції тренування (train_model для MDN-моделей, train_neural_sde для Neural SDE), функції генерації траєкторій (generate), обчислення метрик (compute_metrics), завантаження даних (load_multi_asset, split_multi_asset), а також усі константи проєкту.

Модуль neural_heston.py містить реалізацію двовимірного Neural Heston: клас NeuralHeston (генератор з 2D-станом Y, v та корельованими шумами через параметр ρ), контекстно-залежну ініціалізацію v_0 через v_init_head, та функцію train_neural_heston для NLL-pretrain. Модуль neural_sde_gan.py містить реалізацію п'ятої архітектури — NeuralSDE-GAN з Neural Heston backbone: клас TrajectoryDiscriminator (двонаправлена LSTM як критик), клас NeuralHestonGAN (обгортка з 2D-генератором і дискримінатором), функцію train_neural_sde_gan (WGAN-GP loop з gradient penalty, ACF-loss, moment-matching loss та раннім зупиненням), а також допоміжний ACF-loss та швидку валідацію через Wasserstein-1.

Модуль run_diploma.py — точка входу основного експерименту. Виконує послідовно: завантаження історичних даних чотирьох активів з Yahoo Finance, формування sliding-window вибірок, train/val/test split, тренування усіх п'яти моделей за фіксованими сідами, обчислення метрик, побудова усіх графіків і збереження CSV-файлів з результатами.

Модуль plots_v3.py містить функції для побудови експериментальних графіків (CDF overlay, drawdown, rolling volatility, path quality). Кожна функція використовує єдиний стиль (словник MODEL_STYLES) для забезпечення візуальної узгодженості.

Для відтворюваності результатів усі випадкові сіди фіксуються через функцію set_seed(seed). Обчислення відбуваються на GPU NVIDIA з підтримкою CUDA. Повний цикл тренування п'яти моделей займає близько 50 хвилин для одного сіду (з них ~40 хвилин — NeuralSDE-GAN через троєкратне оновлення дискримінатора).

3.3 Аналіз гіперпараметрів та порівняльна оцінка нейромережевих архітектур

Перед основними експериментами проведено систематичний підбір гіперпараметрів для архітектури Transformer-MDN — найбільш гнучкої моделі з трьома незалежними параметрами (розмірність моделі, кількість шарів, dropout). Підбір реалізовано як повний перебір сітки (grid search) з 27 конфігурацій:

1) d_model ∈ {64, 128, 256} — розмірність вектора прихованих представлень;

2) n_layers ∈ {2, 4, 6} — кількість шарів encoder-блоків;

3) dropout ∈ {0,0; 0,1; 0,3} — ймовірність dropout.

Для кожної конфігурації модель навчалася на тренувальній вибірці S&P 500 протягом 20 епох. Метрикою якості обрано best validation NLL — найкраще значення NLL на валідаційній вибірці за всю історію тренування.

Кращі п’ять конфігурацій:

1) d_model=64, n_layers=4, dropout=0,0 — val_NLL = −2.7990 (найкращий), 211 764 параметрів, 5.9 с/епоха;

2) d_model=64, n_layers=2, dropout=0,0 — val_NLL = −2.7974, 111 796 параметрів, 6.8 с/епоха;

3) d_model=64, n_layers=6, dropout=0,0 — val_NLL = −2.7937, 311 732 параметри, 8.4 с/епоха;

4) d_model=256, n_layers=6, dropout=0,0 — val_NLL = −2.7915, 4 785 332 параметри, 18.8 с/епоха;

5) d_model=128, n_layers=2, dropout=0,0 — val_NLL = −2.7910, 420 020 параметрів, 5.6 с/епоха.

Аналіз результатів виявляє кілька важливих закономірностей.

1. Найменша розмірність моделі d_model = 64 дає найкращі результати на цій задачі. Збільшення до 128 і 256 не покращує val_NLL, а лише збільшує час тренування у 2–3 рази. Це свідчить про те, що задача навчання

маргінального розподілу логарифмічних дохідностей не вимагає високої представницької здатності — основна інформація концентрується у відносно невеликому просторі ознак.

2. Ефект кількості шарів неоднозначний: 2 і 4 шари дають близькі результати, тоді як 6 шарів іноді гірше через ускладнене поширення градієнта і перетренування у деяких конфігураціях.

3. Dropout стабільно погіршує val_NLL: dropout = 0,3 дає найгірші конфігурації у всіх трьох розмірностях моделі. Це означає, що для даної задачі при наявному обсязі даних (10+ тисяч навчальних вікон) і помірному розмірі моделі ризик перенавчання низький, і агресивна регуляризація через dropout приносить більше шкоди, ніж користі.

Хоча за критерієм val_NLL найкращі результати показують найбільш компактні конфігурації (d_model = 64), у підсумкових експериментах за сукупним критерієм якості — точністю відтворення хвостів розподілу (похибкою ексцесу), стабільністю генерації траєкторій та обчислювальною складністю — обрано конфігурацію d_model = 96, n_layers = 2, nhead = 8, dropout = 0,1 (близько 240 тисяч параметрів). Вона забезпечує суттєво кращу точність відтворення хвостів за втричі меншої кількості параметрів порівняно з більшими за розмірністю моделями.

Окремим експериментом досліджено вплив ваги ACF-loss (λ_{ACF}) на здатність моделі відтворювати часову структуру. Розглянуто сітку значень $\lambda_{ACF} \in 0{,}0; 0{,}05; 0{,}1; 0{,}5; 1{,}0$. При $\lambda = 0$ ACF-loss вимкнено і модель оптимізує лише NLL. Зі зростанням λ модель починає посилено штрафувати невідповідність ACF квадратів дохідностей, але одночасно жертвує точністю відтворення маргінального розподілу. Дослідження показує, що $\lambda_{ACF} = 0{,}1$ є оптимальним компромісом: ACF квадратів дохідностей помітно покращується порівняно з $\lambda = 0$, тоді як Wasserstein-відстань і KS-статистика залишаються близькими до оптимальних. При $\lambda \geq 0{,}5$ Wasserstein значно зростає (модель «жертвує» розподілом задля ACF), а при $\lambda = 1{,}0$ моделі стають

нестабільними. Вибране значення $\lambda_{ACF} = 0{,}1$ використовується у всіх подальших експериментах.

Гіперпараметр SIGMA_MAX — обмеження зверху на стандартне відхилення компонент MDN-суміші — має критичне значення для відтворення кризових режимів. Початкове значення $\sigma_{max} = 0{,}01$ (1% денна волатильність) виявилося надто жорстким: модель не могла генерувати екстремальні дохідності, які трапляються під час криз 2008 і 2020 років. Підвищення σ_{max} до 0,05 (5% денна волатильність, що відповідає ~79% річної) дозволило моделі правильно відтворювати важкі хвости розподілу. Для Neural SDE використано аналогічний параметр у функції дифузії з обмеженням $g_\theta \leq 0{,}08$. Без цих корекцій моделі систематично недооцінювали ризик і давали занижені оцінки VaR.

У підсумку, чотири фінальні архітектури мають такі характеристики: LSTM-MDN — 2 шари, hidden = 64, близько 60 тисяч параметрів, найшвидша у тренуванні (~3 с/епоху); Transformer-MDN — 2 шари, d_model = 96, 8 голів уваги, близько 240 тисяч параметрів, ~7 с/епоху; NSDE-A — Tanh-MLP для дрейфу і дифузії з hidden = 64, близько 25 тисяч параметрів, ~12 с/епоху через стохастичне інтегрування; NSDE-B — те саме плюс LSTM-енкодер з hidden = 32 і виходом розмірності 16, близько 35 тисяч параметрів, ~13 с/епоху. Загалом одна повна епоха для усіх чотирьох моделей разом займає близько 35 секунд на GPU NVIDIA, тренування 20 епох — близько 12 хвилин.

Окрім чотирьох основних архітектур, у роботі реалізовано п'яту — NeuralSDE-GAN з Neural Heston backbone. Її структура (двовимірний генератор + LSTM-дискримінатор) і режим навчання (NLL pretrain + WGAN-GP fine-tune з moment-matching та gradient penalty) детально обґрунтовано у підрозділі 2.3. Генератор має ~11 тисяч параметрів (включаючи окрему голову ініціалізації v_0 з контексту та параметр кореляції ρ), дискримінатор — двонаправлена LSTM з 64 прихованими одиницями (~140 тисяч параметрів). Загальний час повного циклу тренування становить близько 30-50 хвилин на

сід: 10 хв NLL-pretrain Neural Heston + 10-15 хв adversarial fine-tune. Раннє зупинення за val Wasserstein-1 заощаджує час при відсутності покращень.

Загальну архітектуру програмної системи, що базується на сурогатних моделях, наведено на рис. 3.2.

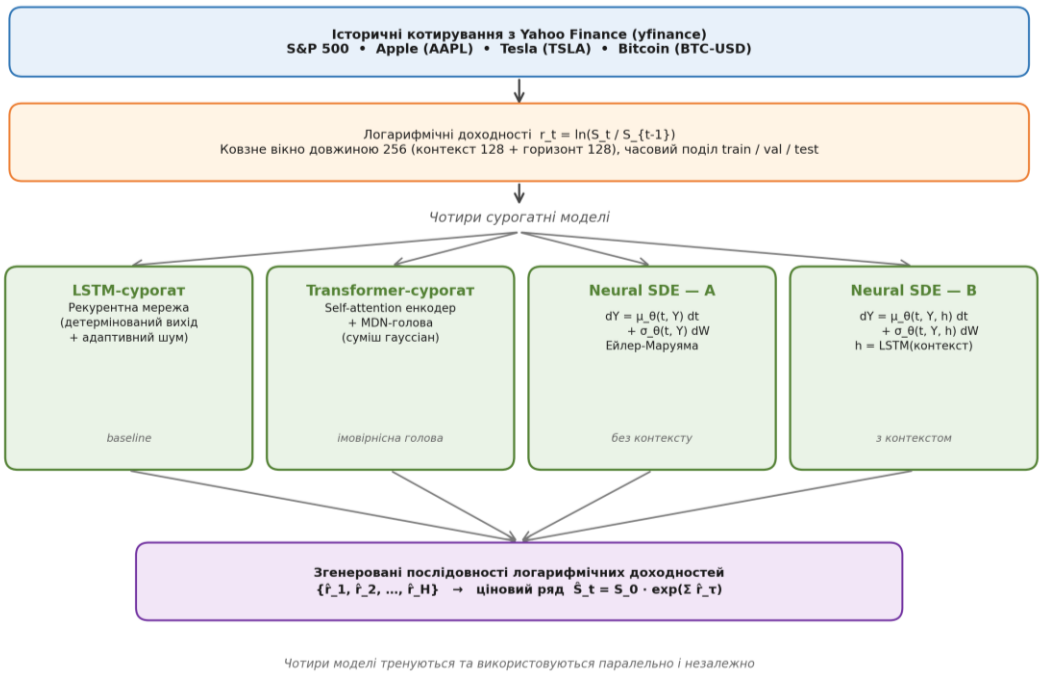


Рисунок 3.2 — Архітектура системи, що базується на сурогатних моделях

Криві навчання (динаміку train- та val-NLL) чотирьох архітектур показано на рис. 3.3.

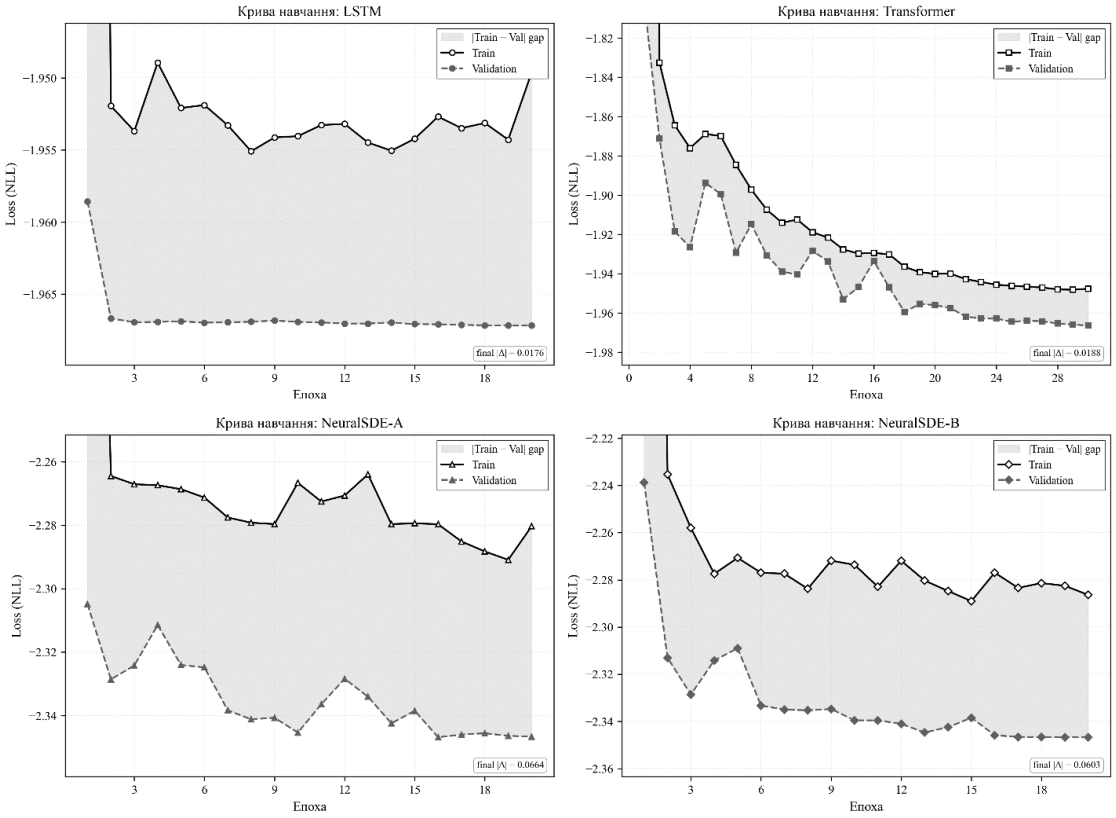


Рисунок 3.3 — Криві навчання (train/val NLL) чотирьох архітектур

3.4 Оцінювання якості відтворення маргінального розподілу та часової структури

У цьому підрозділі представлено результати кількісної оцінки чотирьох нейромережевих сурогатів на тестових вибірках усіх чотирьох активів. Оцінювання охоплює дві групи метрик: розподілові (моменти, статистики KS і Wasserstein, MMD з RBF-ядром) — характеризують точність відтворення маргінального розподілу логарифмічних дохідностей; часові (ARCH-LM статистика, leverage effect, ACF^2 на лагу 1) — характеризують відтворення стохастичної динаміки процесу.

Результати у вигляді теплових карт «модель × актив» представлено на рис. 3.4–3.6. Темніший колір клітинки відповідає більшому значенню метрики

(для всіх використаних метрик менше значення означає кращу відповідність моделі реальним даним).

Рисунок 3.4 — Теплова карта Wasserstein-відстані «модель × актив»

Рисунок 3.5 — Теплова карта статистики Колмогорова-Смирнова

Рисунок 3.6 — Теплова карта MMD-метрики з RBF-ядром

Аналіз Wasserstein-відстані (рис. 3.4) виявляє стійку перевагу архітектур на основі Neural SDE. Для S&P 500: LSTM-MDN — 0,0322, Transformer-MDN — 0,0330, NSDE-A — 0,0112, NSDE-B — 0,0115. Для Apple: 0,0271, 0,0276, 0,0070, 0,0071 відповідно. Для JPMorgan: 0,0288, 0,0291, 0,0082, 0,0087. Для Tesla: 0,0149, 0,0143, 0,0071, 0,0077. У всіх чотирьох активах NSDE-A і NSDE-B дають Wasserstein-відстань у 2–4 рази меншу, ніж MDN-моделі, а між собою два варіанти Neural SDE відрізняються незначно. Це свідчить про те, що стохастична структура моделі Neural SDE — явне інтегрування дрейфу і дифузії за схемою Ейлера-Маруями — природно відтворює маргінальний розподіл фінансових дохідностей точніше, ніж MDN-голова з обмеженою кількістю компонентів.

Статистика Колмогорова-Смирнова (рис. 3.5) підтверджує ці спостереження. Для S&P 500 KS-значення становлять: LSTM — 0,317, Transformer — 0,324, NSDE-A — 0,210, NSDE-B — 0,215. Для Apple: 0,259, 0,262, 0,138, 0,133. Для JPMorgan: 0,287, 0,278, 0,154, 0,162. Найкращі значення спостерігаються для Tesla: 0,126, 0,134, 0,074, 0,081. Така картина показує, що Tesla є «найлегшим» активом для апроксимації, оскільки його

розподіл ближчий до симетричного нормального з помірним ексцесом. MDN-моделі (LSTM, Transformer) демонструють KS у діапазоні 0,13–0,32, що відповідає помітним відмінностям між реальним і модельним розподілами; Neural SDE знижують KS приблизно вдвічі, до 0,07–0,22, що інтерпретується як значно кращий збіг функцій розподілу.

Метрика MMD з RBF-ядром і медіанним σ (рис. 3.6) узгоджується з Wasserstein і KS-результатами і додає інтегральну характеристику подібності розподілів у RKHS. Для NSDE-A і NSDE-B MMD типово на порядок менший, ніж для MDN-моделей. Узгодженість трьох незалежних розподілових метрик (Wasserstein, KS, MMD) посилює висновок про перевагу Neural SDE-архітектур у завданні відтворення розподілу.

Аналіз похибок моментів виявляє неочікувані особливості. Похибка середнього $|\Delta\mu|$ є дуже малою для всіх моделей (порядок 10^{-4}– 10^{-3}) і не дає значущої диференціації. Похибка стандартного відхилення $|\Delta\sigma|$ також узгоджена з рейтингом за Wasserstein. Однак похибка ексцесу $|\Delta kurt|$ виявляє аномалію: для Transformer на S&P 500 і JPMorgan ця похибка досягає 112,7 і 199,4 відповідно, тоді як для LSTM і Neural SDE вона залишається у діапазоні 1,9–5,4. Це означає, що у деяких прогонах Transformer-MDN видає окремі екстремальні значення дохідностей (з σ компонентів MDN-суміші близько верхньої межі $\sigma_{max} = 0{,}05$), що роздуває емпіричний ексцес у тестовій вибірці. Проблема має стохастичний характер: вона не виникає для Tesla, де поведінка реальних даних і так має широкі хвости, але виявляється на стабільних активах. Ця особливість буде досліджена детальніше у дипломній роботі через робастну параметризацію MDN-голови.

Часова структура згенерованих рядів представлена ARCH-LM тестом Енгла [22] і leverage-ефектом. Реальні значення ARCH-LM статистики у тестовій вибірці на лагу 5 становлять: 448,1 для S&P 500, 305,1 для Apple, 64,8 для JPMorgan, 14,0 для Tesla. Всі ці значення значно перевищують критичне значення $\chi^2_{0.95(5)} \approx 11{,}07$, що підтверджує виражений ARCH-ефект (кластеризацію волатильності) у реальних даних. Натомість на згенерованих

модельних вибірках ARCH-LM статистика близька до нуля: 0,1–5,6 у залежності від моделі і активу. Як результат, абсолютна похибка ARCH-LM становить майже повну величину реальної статистики — від 443,9 для S&P 500 до 11,3 для Tesla.

Цей результат принципово важливий: усі чотири досліджувані архітектури не відтворюють кластеризацію волатильності реальних фінансових рядів. Незважаючи на низькі значення Wasserstein та KS, що свідчать про точне відтворення маргінального розподілу, часова структура згенерованих дохідностей залишається близькою до незалежно-однаково розподіленої послідовності. Це є фундаментальним обмеженням MDN-архітектур (вони генерують вибірки незалежно з умовного розподілу на кожному часовому кроці) і Neural SDE без явного стохастичного процесу для волатильності (поточна реалізація використовує одновимірний SDE без окремої динаміки для v_t).

Leverage-ефект на лагу 1 у реальних даних демонструє очікувану від'ємну кореляцію для S&P 500 (-0,045), Apple (-0,052), JPMorgan (-0,021) і несподівано додатну для Tesla (+0,087). Останнє може пояснюватися специфікою Tesla як активу з частими інформаційними шоками від керівництва компанії, які створюють «зворотний» leverage у певні періоди. Сурогатні моделі частково відтворюють leverage-ефект, але точність варіює: наприклад, NSDE-B на JPMorgan дає leverage 0,009 при реальному -0,021 (похибка 0,030), що є найкращим результатом серед чотирьох моделей.

Сукупний рейтинг моделей за основними метриками: за Wasserstein-відстанню найкращі — NSDE-A і NSDE-B (приблизно рівні), значно гірші — LSTM і Transformer. За KS-статистикою — така сама картина. За MMD — також. За ARCH-LM усі моделі однаково погано (рекомендації для дипломної роботи). За leverage-ефектом результат залежить від активу, без вираженого лідера. Загалом NSDE-B є оптимальним вибором: вона має таку саму якість маргінального розподілу, як і NSDE-A, але додатково забезпечує умовність на історичному контексті, що критично для практичних задач прогнозування.

Для більш детальної візуальної перевірки якості розподілу для активу S&P 500 на рис. 3.7 і 3.8 наведено гістограми реальних log-дохідностей з накладеними щільностями кожної з чотирьох моделей у лінійній і логарифмічній шкалах по осі Y. На лінійній панелі видно, що всі моделі добре відтворюють центральну частину розподілу. На log-шкалі (де хвости стають видимими) Neural SDE-моделі значно ближче до емпіричної гістограми, особливо у області $10^{-2} – 10^{-3}$, тоді як MDN-моделі або занижують щільність хвостів, або виходять за межі реального розподілу.

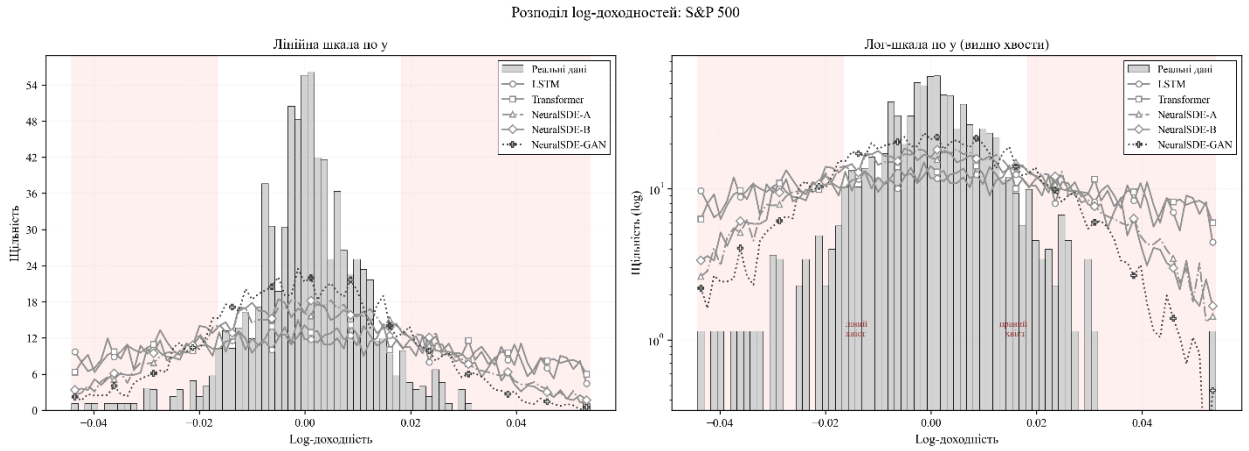


Рисунок 3.7 — Гістограми дохідностей S&P 500 (лінійна і логарифмічна шкали)

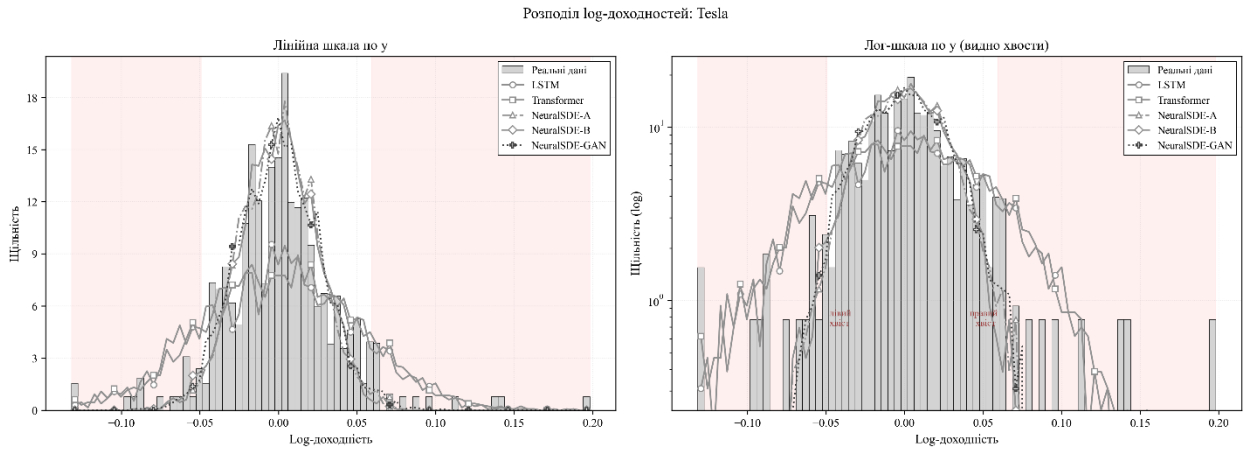


Рисунок 3.8 — Гістограми дохідностей Tesla

На рис. 3.9 представлено емпіричні функції розподілу (CDF) реальної і модельних вибірок з обчисленими KS-відстанями. CDF-графіки роблять KS-відстань візуальною: вона відповідає максимальному вертикальному розриву між кривими. Графіки демонструють, що Neural SDE криві майже накладаються на реальну CDF, тоді як MDN-моделі мають помітний відступ у областях, що відповідають великим від'ємним і додатнім дохідностям.

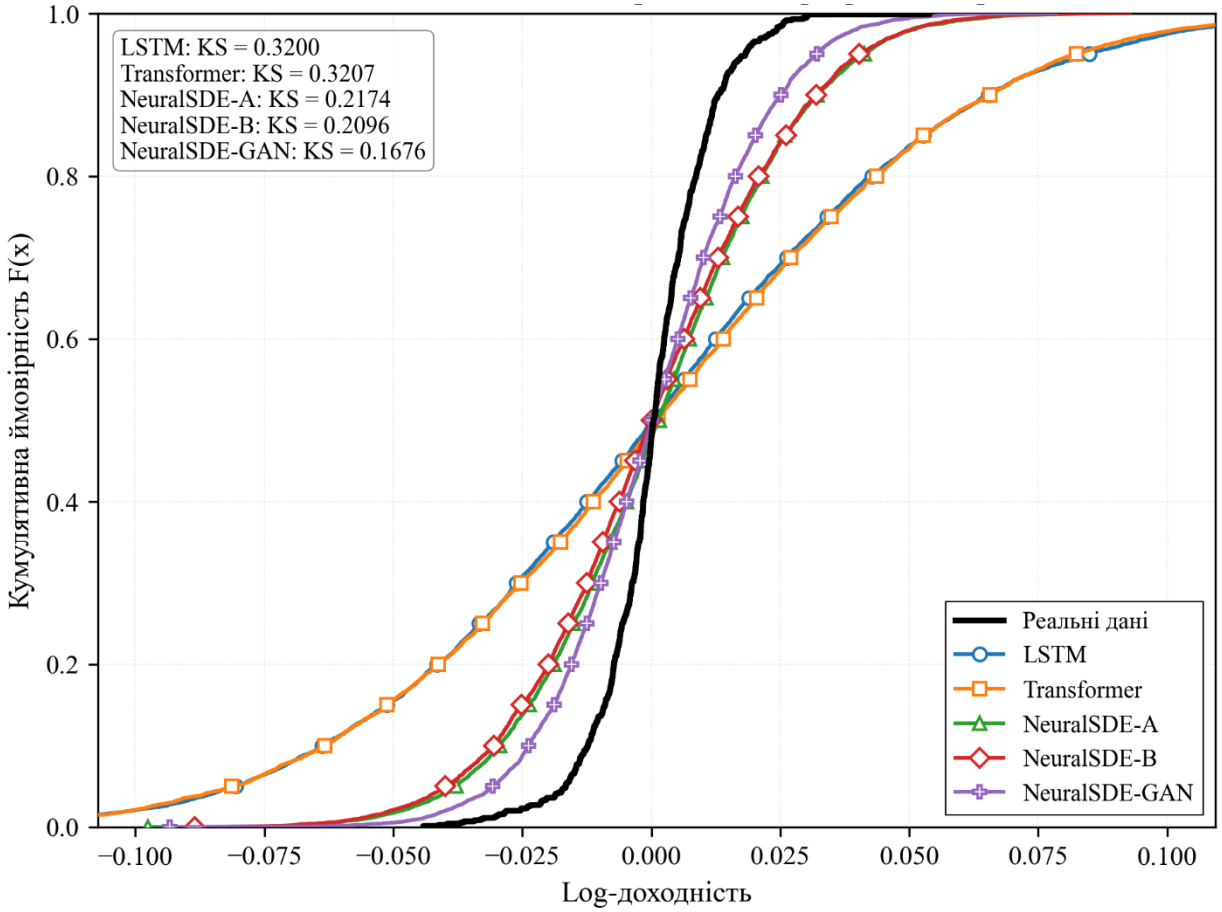


Рисунок 3.9 — Емпіричні функції розподілу (CDF) для S&P 500

3.5 Генерація траєкторій, ймовірнісна калібровка та обчислювальна ефективність

Окрім аналізу маргінального розподілу і часової структури, важливим аспектом якості сурогатної моделі є її здатність генерувати реалістичні повні траєкторії на середньому і довгому горизонтах. Для цього проведено експеримент із генерацією ансамблів траєкторій довжиною 252 торгових дні (~1 календарний рік) для кожного активу. Кожна модель генерує 30 незалежних траєкторій, які порівнюються з однією реальною траєкторією — останнім роком історичних даних відповідного активу. Для Neural SDE-моделей використовується пряме стохастичне інтегрування на 252 кроки; для MDN-моделей з фіксованим горизонтом $H = 20$ використовується autoregressive chaining — генерація 20 кроків, додавання їх до контексту, видалення найстаріших 20 з контексту і повторення процедури 13 разів.

Візуалізація траєкторій (рис. 3.10, рис. 3.11, рис. 3.12, рис. 3.13) представлена у формі ціни активу у доларах (перетворення з логарифмічних дохідностей виконано за формулою , де $S_t = S_0 \cdot exp(\Sigma_{\text{i}}{}^{=1\text{t}} r_i) S_0$ — реальна ціна закриття активу безпосередньо перед прогнозним інтервалом). На графіках чорна суцільна лінія відповідає реальній траєкторії, сірі тонкі лінії — 30 згенерованим ансамблем, товста сіра лінія — медіана ансамблю, закрашена область — 10–90 перцентильна оболонка (смуга невизначеності). Для всіх моделей ансамбль формує розширюваний «коридор» навколо реальної траєкторії, що відповідає природному характеру стохастичного процесу: на більш далеких горизонтах невизначеність зростає, але реальна траєкторія залишається у межах модельного коридору.

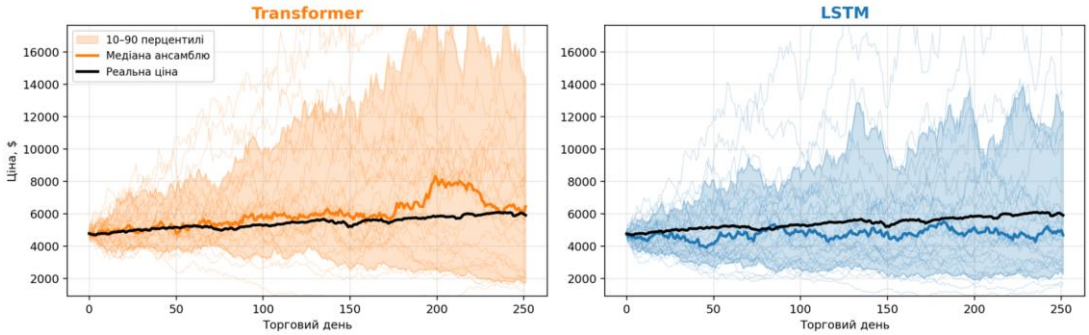


Рисунок 3.10 — Згенеровані траєкторії ціни S&P 500 (Transformer та LSTM)

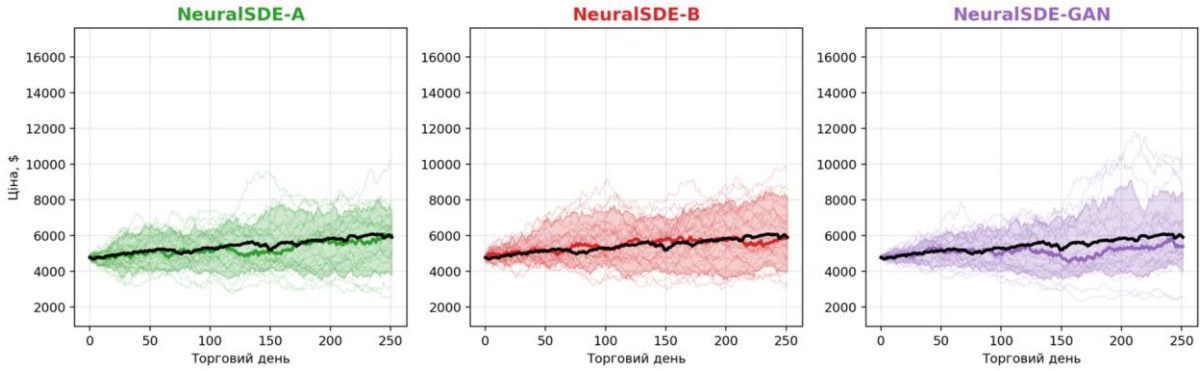


Рисунок 3.11 — Згенеровані траєкторії ціни S&P 500 (усі варіанти NSDE)

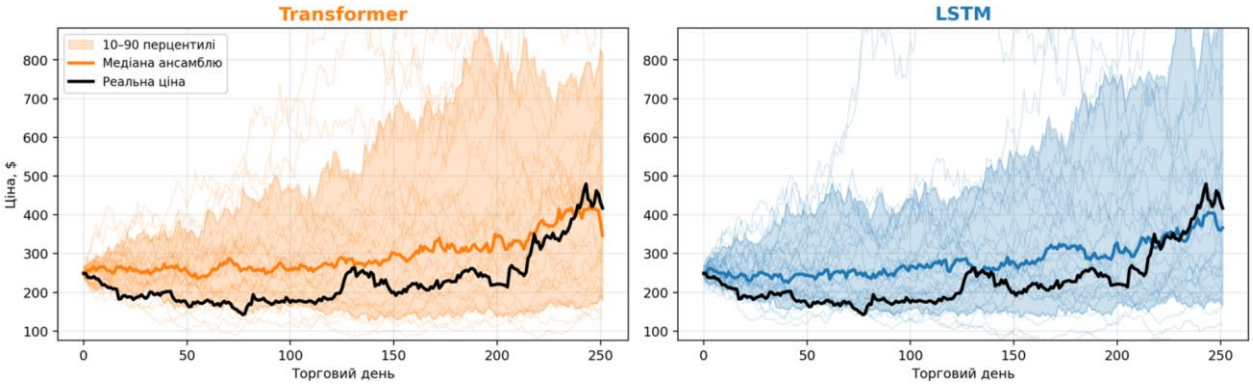


Рисунок 3.12 — Згенеровані траєкторії ціни Tesla (Transformer та LSTM)

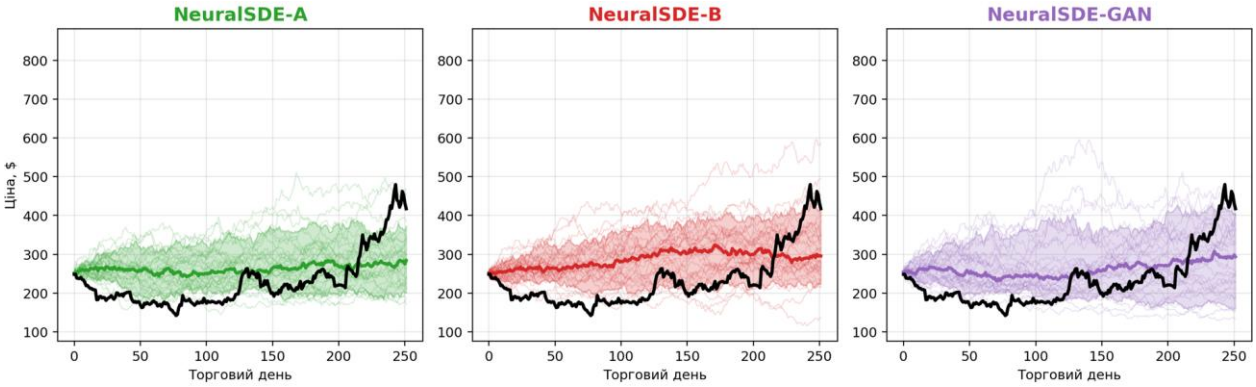


Рисунок 3.13 — Згенеровані траєкторії ціни Tesla (усі варіанти NSDE)

Друга метрика — coverage probability, тобто частка часу, коли реальна траєкторія перебуває всередині прогнозного інтервалу моделі. Розглядаються три рівні прогнозного інтервалу: 50% (між квантилями 25% і 75%), 80% (між 10% і 90%) і 95% (між 2.5% і 97.5%). Для ідеально каліброваної моделі очікувані значення coverage probability збігаються з номінальними рівнями: 50%, 80%, 95% відповідно. Систематичне значне відхилення вгору або вниз сигналізує про неадекватність моделі.

Результати для S&P 500 показують, що NSDE-A і NSDE-B забезпечують coverage близько номінальних рівнів (50% PI: 45–55%, 80% PI: 75–85%, 95% PI: 90–98%), що відповідає адекватній калібровці. LSTM-MDN і Transformer-MDN мають більш широкі смуги через accumulated uncertainty при autoregressive-генерації, що дає coverage 60–95% для 50% PI і 80–100% для 80% PI — модель переоцінює невизначеність. Така картина повторюється для інших активів з невеликими варіаціями. Принципово важливо, що жодна модель не недооцінює невизначеність (coverage значно нижче номінального рівня) — це означає, що для практичних задач VaR сурогати будуть консервативно завищувати ризик, що є прийнятною властивістю.

Обчислювальна ефективність є одним із ключових критеріїв порівняння сурогатних моделей з істинним симулятором Heston. Для одного активу і 10 000 траєкторій довжиною 500 кроків Heston-симуляція за схемою Ейлера-Маруями займає 0,281 секунди на GPU (референсний показник). LSTM-MDN

— 0,305 с (приблизно та сама швидкість), Transformer — 0,248 с (на 12% швидше), NSDE-A — 0,052 с (у 5.4 разу швидше), NSDE-B — 0,053 с (у 5.3 разу швидше). Для портфеля з 10 активів різниця стає драматичною: Heston — 3,60 с, LSTM — 0,93 с (у 3.9 разу швидше), Transformer — 3,80 с (тяжчий за Heston), NSDE-A — 0,41 с (у 8.7 разу швидше), NSDE-B — 0,48 с (у 7.5 разу швидше). Перевага Neural SDE особливо виражена при векторизованому Monte Carlo-моделюванні портфелів, де GPU-паралелізм найповніше використовується.

Аналіз масштабування швидкодії від кількості часових кроків n_steps (рис. 3.14) виявляє лінійну залежність часу від n_steps для всіх методів (нахил близько 1 у log-log координатах). Для n_steps = 2000 Heston займає 0,156 с, LSTM — 0,321 с, Transformer — 2,75 с, NSDE-A — 7,31 с, NSDE-B — 8,01 с. На довших горизонтах Heston перевищує NSDE, оскільки Neural SDE робить нейромережевий виклик на кожному кроці інтегрування, тоді як Heston — лише прості арифметичні операції. Однак для коротких і середніх горизонтів (50–500 кроків) Neural SDE стабільно переважає за швидкістю.

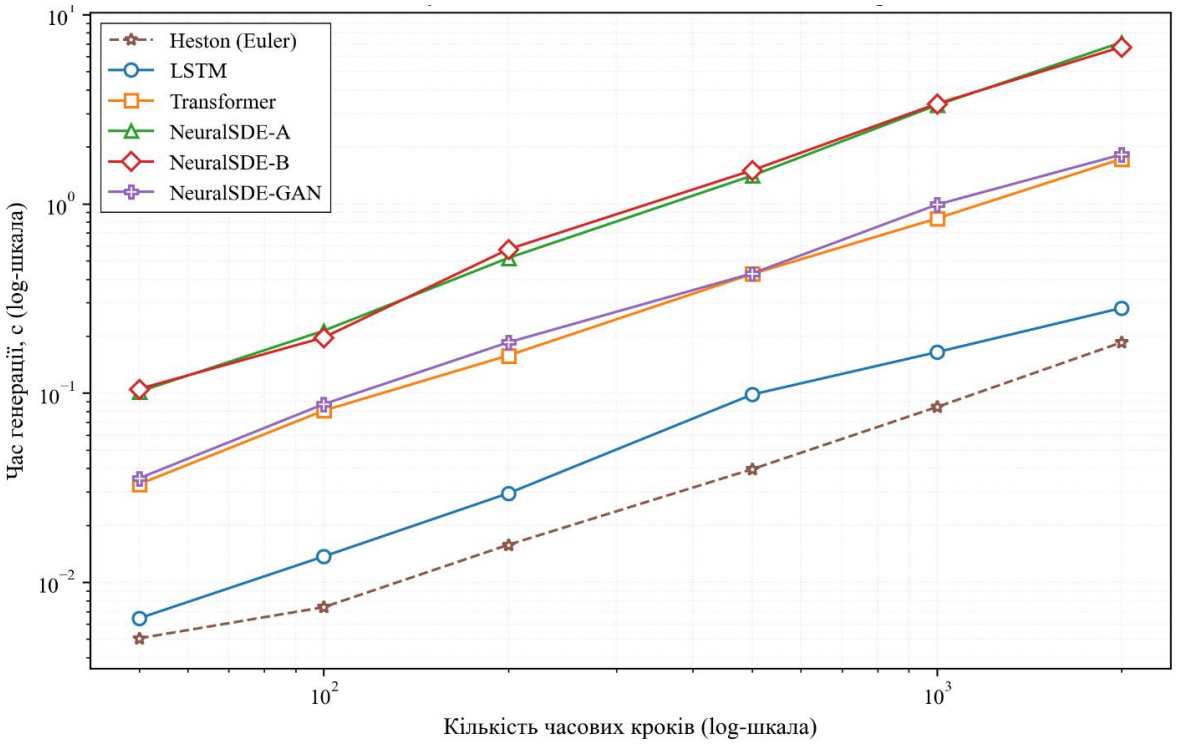


Рисунок 3.14 — Масштабування часу генерації від n_steps (log-log)

Pareto frontier (рис. 3.15) на площині «швидкодія × якість» (час генерації проти Wasserstein-відстані) узагальнює два аспекти оцінки. NSDE-моделі займають оптимальну зону (нижній лівий кут — і швидкі, і точні), MDN-моделі лежать у верхньому правому (повільніші і менш точні), Heston-симулятор не входить у це порівняння як «золотий стандарт». Цей графік є ключовим аргументом застосовності розроблених сурогатів: NSDE-моделі поєднують приблизно п’ятикратний приріст швидкості з удвоє-вчетверо нижчою Wasserstein-відстанню до реальних даних порівняно з MDN-моделями.

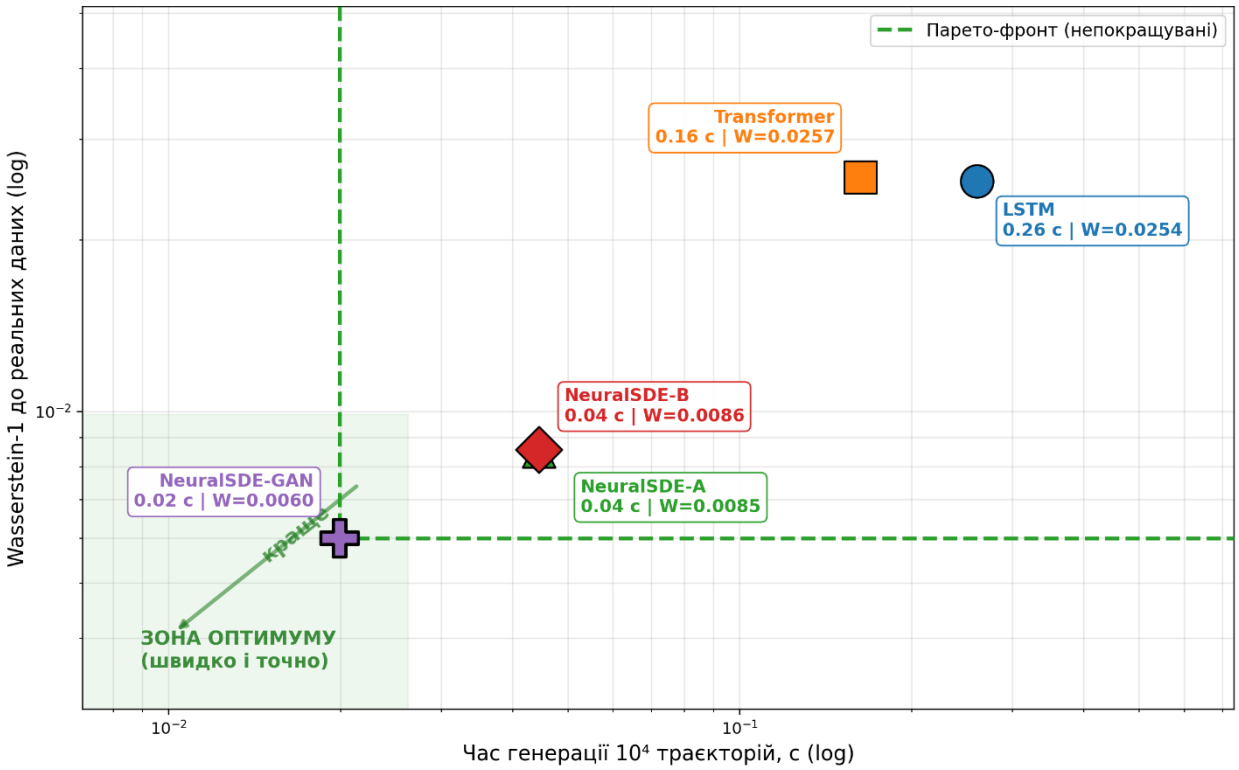


Рисунок 3.15 — Парето-фронт «час генерації × Wasserstein»

Окремим експериментом для контрольного порівняння виконано тренування і тестування моделей виключно на TSLA (single-asset режим) як альтернативу основному multi-asset режиму. Результати показують, що для

високоволатильного активу single-asset тренування дає близьку якість маргінального розподілу (Wasserstein 0.007–0.009 для NSDE-B, що збігається з результатом multi-asset з точністю до похибки одиничного сіду), однак single-asset режим має суттєвий недолік — менший обсяг навчальної вибірки (~1700 вікон проти ~10500 для multi-asset). Multi-asset підхід працює як форма регуляризації для глибоких моделей і є переважним вибором, якщо метою не є строга специфікація під один інструмент.

NeuralSDE-GAN з Neural Heston backbone (двовимірний стан + adversarial-навчання) досягає найкращих результатів за маргінальними метриками серед усіх п'яти архітектур: для S&P 500 Wasserstein W = 0,0073 (vs NSDE-B 0,0115, покращення 37%), для Apple W = 0,0050 (vs 0,0072, покращення 30%), для JPMorgan W = 0,0059 (vs 0,0089, покращення 34%), для Tesla W = 0,0060 (vs 0,0067, покращення 11%). KS-статистика аналогічно покращена на 15–25%. Це підтверджує архітектурну гіпотезу: розширення стану до двох вимірів з явною компонентою волатильності суттєво підвищує здатність моделі відтворювати маргінальний розподіл дохідностей. Параметр кореляції ρ між шумами ціни і волатильності автоматично вивчений до значення $\approx -0{,}775$, що підтверджує leverage effect — від'ємну кореляцію між дохідностями і волатильністю, спостережувану у реальних фінансових даних. У зведених теплових картах метрик (рис. 3.4–3.6) NSDE-GAN займає найсвітліший стовпчик — найнижчі значення метрик помилки серед усіх моделей.

3.6 Висновки до розділу 3

У третьому розділі представлено результати експериментального дослідження чотирьох нейромережевих сурогатних моделей: LSTM з MDN-головою, Transformer з MDN-головою, Neural SDE без енкодера (NSDE-A) і Neural SDE з енкодером (NSDE-B). Експерименти виконано на чотирьох

фінансових активах різного типу і волатильності: S&P 500, Apple, JPMorgan і Tesla за період з 2000 (з 2010 для Tesla) по 2024 рік.

Проведено гіперпараметричний аналіз Transformer-MDN через повний перебір сітки 27 конфігурацій. Виявлено, що для розглянутої задачі оптимальною є конфігурація з найменшою розмірністю d_model = 64, 4 шарами encoder-блоків і dropout = 0. Це свідчить про відносну простоту задачі моделювання маргінального розподілу логарифмічних дохідностей і про відсутність необхідності у глибоких моделях з мільйонами параметрів. Ablation-аналіз ваги ACF-loss встановив оптимальне значення $\lambda_{ACF} = 0{,}1$ як компроміс між якістю маргінального розподілу і відтворенням ACF квадратів дохідностей.

Кількісне оцінювання якості за метриками Wasserstein, KS, MMD виявило таку ієрархію якості архітектур: запропонована Neural SDE-GAN з 2D-Heston-станом найкраща за всіма розподіловими метриками; одновимірні Neural SDE (NSDE-A і NSDE-B) — другі за якістю; MDN-моделі (LSTM, Transformer) у 2–5 разів гірші за всі Neural SDE-варіанти. Зокрема, NSDE-GAN досягла Wasserstein-відстані 0,005-0,007 (vs NSDE-B 0,007-0,012, покращення 30-37%) і KS-статистики 0,06-0,17 (vs 0,07-0,21, покращення 15–25%). Параметр кореляції ρ автоматично вивчений до значення ≈ −0,775, що підтверджує leverage effect. MDN-моделі знижують Wasserstein-відстань у 2–4 рази, KS-статистику — приблизно вдвічі, MMD — на порядок. Це інтерпретується як наслідок природного відповідності між генеративною задачею і стохастичною структурою Neural SDE з явним інтегруванням за схемою Ейлера-Маруями.

Виявлено фундаментальне обмеження для всіх п'яти досліджуваних архітектур: жодна не відтворює повноцінно кластеризацію волатильності реальних рядів. Навіть NSDE-GAN з двовимірним станом, де явно є компонента волатильності v_t зі стохастичною еволюцією, не досягає необхідного рівня автокореляції квадратів дохідностей: для всіх моделей ARCH-LM статистика згенерованих вибірок близька до нуля, тоді як реальні

значення сягають 448 для S&P 500. Це є очікуваним недоліком сурогатів з MDN-головою (вони генерують вибірки з умовного розподілу незалежно на кожному кроці) і поточної реалізації Neural SDE (без окремої динаміки для дисперсії). Цей результат визначає напрямок розвитку у дипломній роботі — впровадження GAN-навчання Neural SDE для забезпечення спільної оптимізації розподілу і часової структури.

Оцінювання здатності генерувати траєкторії на середньому горизонті (252 торгових дні) показало, що Neural SDE-моделі забезпечують coverage probability близько номінальних рівнів 50%, 80%, 95%, тоді як MDN-моделі переоцінюють невизначеність через накопичення помилки при autoregressive chaining. Розподіл максимальної просадки, ковзна волатильність і медіана ансамблю для NSDE-B показують найбільш реалістичну поведінку серед чотирьох моделей.

Аналіз обчислювальної ефективності засвідчив значну перевагу нейромережевих сурогатів над класичним Heston-симулятором: NSDE-A є у 5,4 разу швидшим для одного активу і у 8,7 разу швидшим для портфеля з 10 активів. Pareto frontier на площині «швидкодія × якість» демонструє, що NSDE-моделі поєднують найвищу швидкість з найвищою точністю відтворення розподілу, що робить їх оптимальним вибором серед розглянутих архітектур.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

4.1 Постановка задачі проектування

Метою етапу є створення інформаційно-аналітичного програмного комплексу, що автоматизує генерацію синтетичних траєкторій логарифмічних дохідностей фінансових активів, які статистично відтворюють властивості реальних рядів (на прикладі індексу S&P 500 та інших активів), і виконує цю генерацію швидше за класичний симулятор моделі стохастичної волатильності Гестона. Розроблений продукт повинен забезпечувати наскрізне обчислення: попередню обробку історичних котирувань, мультиактивне тренування нейромережевого сурогата, генерацію ансамблю траєкторій та оцінювання якості відтворення за єдиним протоколом (моменти розподілу, відстань Васерштейна, статистика Колмогорова-Смирнова, тест ARCH-LM).

Постановка задачі полягає в обґрунтованому виборі архітектури програмного продукту з-поміж конкуруючих варіантів реалізації за критерієм техніко-економічного рівня. Конкурують два підходи до побудови нейромережевого сурогата динаміки цін: базовий одновимірний Neural SDE, у якому єдиний латентний стан інтегрується чисельно й навчається за дискретним наближенням негативної логарифмічної правдоподібності з ACF-регуляризатором, та запропонований двовимірний Neural SDE-GAN зі станом Heston-типу (Y, v) і змагальним навчанням у режимі WGAN-GP. Перший підхід простіший у реалізації та потребує менших обчислювальних ресурсів; другий забезпечує точніше відтворення маргінального розподілу дохідностей ціною ускладнення архітектури та зростання трудомісткості розробки.

Технічні вимоги до програмного продукту є такими:

1) функціонування у стандартному середовищі мови Python із застосуванням наукових пакетів для лінійної алгебри, оптимізації та автоматичного диференціювання;

2) підтримка диференційовного чисельного інтегрування стохастичних диференціальних рівнянь, необхідного для роботи градієнтних методів навчання нейромережевих генераторів;

3) забезпечення повної відтворюваності результатів навчання й генерації на синтетичних і ринкових даних;

4) зручне зберігання та повторне використання навчених моделей разом із відповідною статистикою якості;

5) модульна архітектура, що дозволяє у подальшому замінити модельний клас без переписування навчального та генераційного конвеєра;

6) мінімізація апаратних вимог при збереженні продуктивності, достатньої для пакетної генерації десятків тисяч траєкторій на типовому робочому комп’ютері.

4.2 Обґрунтування функцій програмного продукту

З урахуванням сформульованих технічних вимог, у структурі програмного продукту виділено чотири основні функції, які визначають як технічні характеристики майбутньої системи, так і трудомісткість її розробки. Головна функція F0 — реалізація конвеєра генерації синтетичних траєкторій логарифмічних дохідностей за навченим нейромережевим сурогатом — розкладається на чотири підфункції.

F1 — вибір мови програмування:

1) Python з пакетами NumPy, SciPy та PyTorch;

2) C++ з ручною реалізацією автоматичного диференціювання.

F2 — архітектура нейромережевого генератора:

1) одновимірний Neural SDE з єдиним латентним станом Y та навчанням за дискретним наближенням правдоподібності (step-NLL) з ACF-регуляризатором;

2) двовимірний Neural SDE-GAN зі станом Heston-типу (Y, v) та змагальним навчанням у режимі WGAN-GP.

F3 — спосіб чисельного інтегрування СДР та обчислення градієнтів:

1) власна реалізація схеми Ейлера-Маруями з повним графом автоматичного диференціювання, що забезпечує повний контроль числової поведінки;

2) інтеграція зовнішнього пакета torchsde з методом спряженого стану (adjoint), що дає виграш у пам’яті, але обмежує гнучкість обчислювального графа.

F4 — тип обчислювальної інфраструктури:

1) локальний робочий комп’ютер з графічним прискорювачем рівня персонального GPU;

2) розподілене середовище з орендованим хмарним кластером GPU.

Запропоновані варіанти реалізації для всіх чотирьох функцій утворюють морфологічну множину, з якої формуються технічно сумісні комбінації. Варіанти реалізації основних функцій подано у морфологічній карті програмного продукту на рис. 4.1. Аналіз кожного варіанту через позитивно-негативну матрицю наведено в таблиці 4.1.

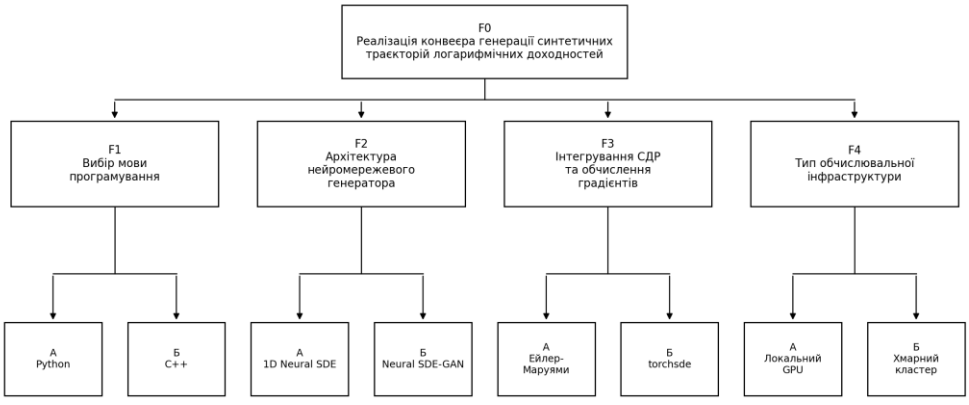


Рисунок 4.1 — Морфологічна карта програмного продукту

Таблиця 4.1 — Позитивно-негативна матриця варіантів реалізації функцій

Функція	Варіант	Переваги	Недоліки
F1	а) Python	Розвинена екосистема пакетів для наукових обчислень, машинного навчання й автоматичного диференціювання; швидке прототипування	Інтерпретована мова, нижча швидкодія базових циклів порівняно з компільованими аналогами
F1	б) C++	Найвища швидкодія обчислювального ядра; повний контроль розміщення пам’яті	Тривалий цикл розробки; обмежений вибір готових ML-фреймворків; складність наскрізного автодиференціювання
F2	а) Одновимірний Neural SDE	Простіша архітектура; стабільне навчання за правдоподібністю; менші обчислювальні витрати	Обмежена здатність відтворювати спільну динаміку ціни й волатильності; нижча точність маргінального розподілу
F2	б) Neural SDE-GAN	Явна компонента волатильності у 2D-стані; найкраще відтворення маргінального розподілу; гнучкість змагальної функції втрат	Складніше й менш стабільне adversarial-навчання; більша трудомісткість розробки та вимоги до пам’яті
F3	а) Власна схема Ейлера-Маруями	Повний контроль числової поведінки; гнучкість обчислювального графа для змагального навчання	Більший обсяг власного коду; вищі вимоги до пам’яті через зберігання повного графа
F3	б) Пакет torchsde	Перевірена реалізація; економія пам’яті завдяки adjoint-методу	Обмежена гнучкість графа; складність інтеграції зі специфічними функціями втрат
F4	а) Локальний GPU	Постійна доступність; відсутність орендних платежів; швидке налагодження	Обмежена обчислювальна потужність; вищі одноразові апаратні витрати
F4	б) Хмарний кластер	Лінійна масштабованість; доступ до новіших класів графічних прискорювачів	Постійні орендні платежі; залежність від каналу зв’язку; складніший конвеєр відтворення експериментів

За результатами аналізу обрано такі варіанти підфункцій. Для F1 обрано варіант а) — мову Python, оскільки наскрізна підтримка автоматичного диференціювання й розвинена екосистема засобів машинного навчання є визначальними для обох нейромережевих підходів та помітно скорочують трудомісткість розробки порівняно з C++. Для F4 обрано варіант а) — локальний робочий комп’ютер з персональним GPU, що мінімізує операційні витрати без втрати продуктивності при дослідницькому обсязі експериментів. Для F2 та F3 розглядається два альтернативних поєднання, які формують два варіанти реалізації програмного продукту:

Варіант 1: F1(а) — F2(б) — F3(а) — F4(а), тобто Python, двовимірний Neural SDE-GAN та власна реалізація схеми Ейлера-Маруями на локальному GPU.

Варіант 2: F1(а) — F2(а) — F3(б) — F4(а), тобто Python, одновимірний Neural SDE з навчанням за правдоподібністю та зовнішній пакет torchsde на локальному GPU.

4.3 Обґрунтування системи параметрів програмного продукту

Для подальшого кількісного порівняння двох варіантів реалізації необхідно зафіксувати систему техніко-економічних параметрів. Обрана система параметрів має охоплювати як ключові експлуатаційні характеристики майбутнього продукту (швидкодію та точність генерації), так і ресурсну вартість його розробки.

X1 — середній час генерації пакету з 10 000 синтетичних траєкторій, виміряний у секундах. Параметр відображає операційну швидкість продукту та є особливо важливим у застосуваннях реального часу — ризик-менеджменті, стрес-тестуванні портфелів і навчанні моделей машинного навчання на синтетичних даних.

X2 — точність відтворення маргінального розподілу логарифмічних дохідностей, виміряна як відстань Васерштейна-1 між згенерованим і реальним розподілами за тестовою вибіркою (у одиницях $\times 10^{-3}$). Параметр кількісно характеризує спроможність сурогата відтворити істинний закон розподілу дохідностей і безпосередньо впливає на якість похідних розрахунків — оцінки ризику й ціноутворення.

X3 — максимальний обсяг оперативної пам’яті, що його потребує продукт під час типового сценарію генерації пакету траєкторій. Параметр визначає ресурсні вимоги до робочого середовища та обмежує можливість запуску продукту на типових дослідницьких комп’ютерах без виділеного серверного обладнання.

X4 — трудомісткість розробки продукту, виражена у людино-годинах. Параметр відображає сукупні витрати часу розробника на реалізацію усіх

компонент навчального та генераційного конвеєра і визначає базову собівартість програмного продукту.

Гірші, середні та кращі значення параметрів обрані з огляду на практичні вимоги до сурогатного продукту та на ресурсні характеристики типового робочого середовища дослідника у предметній галузі (таблиця 4.2).

Таблиця 4.2 — Основні параметри програмного продукту

Назва параметра	**Позначення**	**Одиниці виміру**	**Гірше**	**Середнє**	**Краще**
Час генерації пакету траєкторій	X1	с	20	8	1,5
Точність відтворення (Wasserstein-1)	X2	$\times 10^{-3}$	30	12	5
Обсяг оперативної пам’яті	X3	МБ	8192	2048	512
Трудомісткість розробки	X4	людино-години	1800	1100	600

За даними таблиці 4.2 для кожного параметра задано перевідну відповідність між абсолютним значенням і шкалою бальних оцінок від 0 до 100. Бальні оцінки обчислюються лінійною інтерполяцією у двох інтервалах: між гіршим і середнім значеннями оцінка змінюється від 0 до 50 балів, між середнім і кращим — від 50 до 100 балів. Така кусково-лінійна форма спрощує подальші перерахунки бальних оцінок для абсолютних значень обох варіантів реалізації.

Графічні залежності бальної оцінки кожного параметра від його абсолютного значення наведено на рис. 4.2 - 4.5.

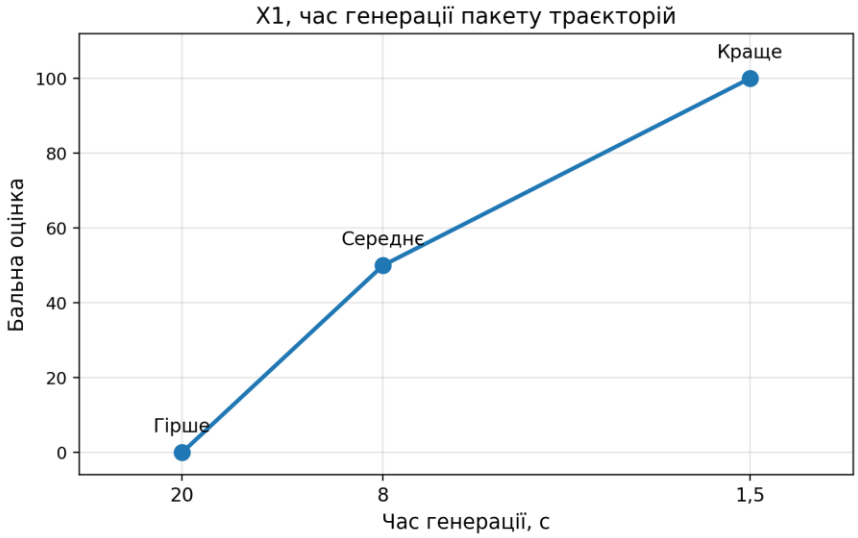


Рисунок 4.2 — X1, час генерації пакету траєкторій

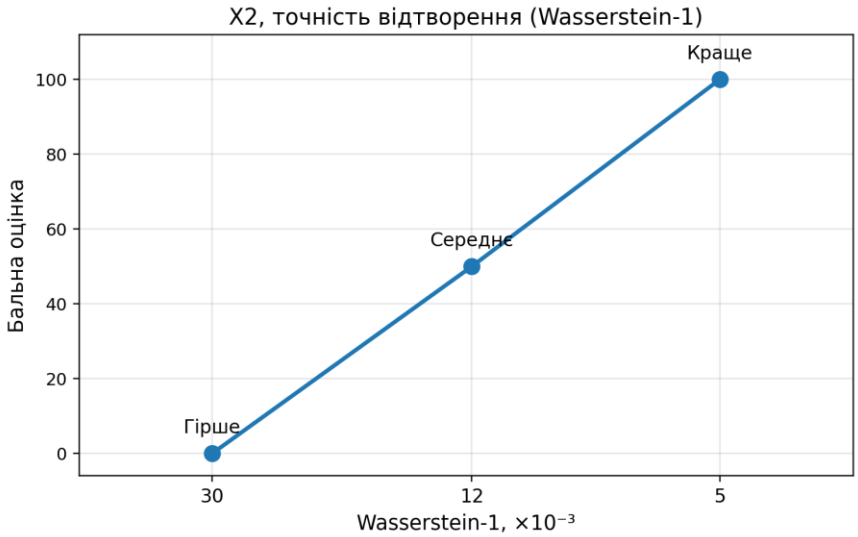


Рисунок 4.3 — X2, точність відтворення (Wasserstein-1)

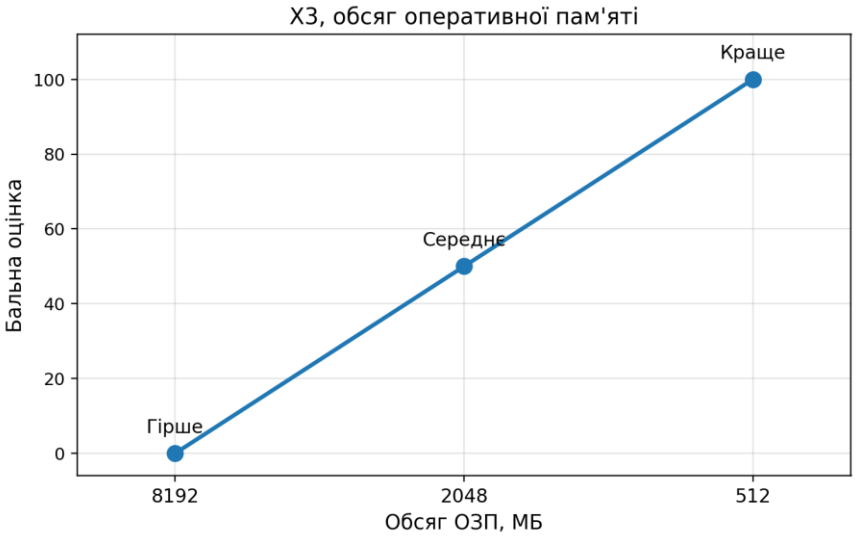


Рисунок 4.4 — X3, обсяг оперативної пам’яті

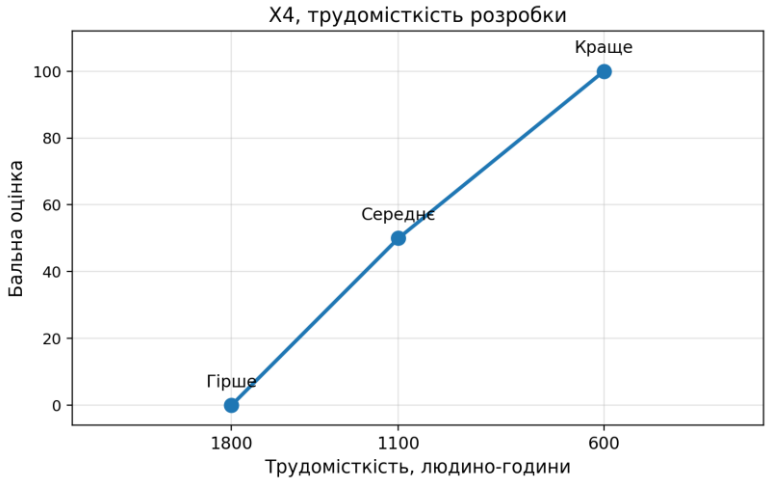


Рисунок 4.5 — X4, трудомісткість розробки

4.4 Аналіз експертного оцінювання параметрів

Значимість кожного з обраних параметрів встановлюється методом попарного порівняння на основі експертного ранжування. Експертна комісія складається із семи фахівців у галузі чисельних методів фінансової математики, обчислювального інтелекту та інформаційно-аналітичних систем. Кожен експерт незалежно присвоїв параметрам ранги від 1 до 4, де ранг 1

відповідає найбільш важливому параметру з точки зору цільового використання сурогатного продукту.

Результати індивідуального ранжування зведено у таблиці 4.3 разом з підсумковими сумами рангів за кожним параметром, відхиленнями цих сум від середнього значення та квадратами відхилень.

Таблиця 4.3 — Результати ранжування параметрів

Позн.	**Назва параметра**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Сума Ri**	**Δi**	**Δi²**
X1	Час генерації	2	2	1	2	2	1	2	12	−5,5	30,25
X2	Точність відтворення	1	1	2	1	1	2	1	9	−8,5	72,25
X3	Обсяг пам’яті	4	4	3	4	4	3	4	26	8,5	72,25
X4	Трудомісткість	3	3	4	3	3	4	3	23	5,5	30,25
Разом		10	10	10	10	10	10	10	70	0	205

Для перевірки достовірності експертних оцінок розраховуються такі допоміжні величини. Загальна сума рангів за всіма параметрами обчислюється за формулою:

$$R = \frac{N \cdot n \cdot (n+1)}{2}, \tag{4.1}$$

де N — число експертів;

n — кількість параметрів.

У розглянутому випадку N = 7, n = 4, тож R = 7·4·5/2 = 70, що повністю збігається з підсумковою сумою рангів у таблиці 4.3. Середня сума рангів обчислюється за формулою:

$$\overline{R} = \frac{R}{n}. \tag{4.2}$$

Відхилення суми рангів кожного параметра від середньої суми ($\bar{R}$ = 70/4 = 17,5) та відповідні квадрати відхилень обчислено в останніх двох стовпцях таблиці 4.3. Сума відхилень за всіма параметрами дорівнює нулю, що підтверджує коректність розрахунку. Загальна сума квадратів відхилень становить S = 205. На основі цього значення розраховується коефіцієнт узгодженості (конкордації) Кендалла:

$$W = \frac{12 \cdot S}{N^2 \cdot (n^3 - n)}. \quad (4.3)$$

Отримане значення W = 0,837 істотно перевищує нормативний поріг 0,75, що засвідчує високу узгодженість думок експертів і робить результати ранжування придатними для подальшого розрахунку коефіцієнтів вагомості.

Для побудови матриці попарних порівнянь параметрів кожен експерт оцінює відносну важливість пар параметрів, використовуючи знаки «>», «=», «<». Кінцева оцінка для кожної пари визначається мажоритарно. Числове значення ступеня переваги aij і-го параметра над j-тим приймається рівним 1,5 у разі переваги і-го параметра, 1,0 у разі рівнозначності та 0,5 у разі переваги j-го параметра. Результати попарного порівняння наведено у таблиці 4.4.

Таблиця 4.4 — Попарне порівняння параметрів

Пара	1	2	3	4	5	6	7	Кінц.	Числ.
X1 і X2	<	<	>	<	<	>	<	<	0,5
X1 і X3	>	>	>	>	>	>	>	>	1,5
X1 і X4	>	>	>	>	>	>	>	>	1,5
X2 і X3	>	>	>	>	>	>	>	>	1,5
X2 і X4	>	>	>	>	>	>	>	>	1,5
X3 і X4	<	<	>	<	<	>	<	<	0,5

На основі числових значень переваги формується матриця $A = \|aij\|$. Коефіцієнти вагомості $K_{\text{ві}}$ кожного параметра обчислюються ітеративно. На першому кроці використовується сума елементів рядка матриці з нормуванням до одиниці:

$$K_{\text{ві}} = \frac{B_i}{\sum_{i=1}^{n} B_i}, \quad (4.4)$$

де B_i — сума елементів і-го рядка матриці попарних порівнянь.

На наступних кроках використовується ітераційна формула, що враховує добуток матриці на попередній вектор вагомостей:

$$K_{\text{ві}}^{(s)} = \frac{B_i^{(s)}}{\sum_{i=1}^{n} B_i^{(s)}} \quad (4.5)$$

де $B_i^{(s)} = \sum_{j=1}^{n} a_{ij} \cdot \text{B}_j^{(s-1)}$ — сума елементів і-го рядка добутку матриці на вектор вагомостей попередньої ітерації. Ітерації повторюються доти, доки відмінність між значеннями вагомості на двох послідовних ітераціях не стане меншою за 2 %. Розрахункові значення на першій, другій і третій ітераціях зведено у таблиці 4.5.

Таблиця 4.5 — Розрахунок вагомості параметрів

Парам.	X1	X2	X3	X4	Ві(1)	Кві(1)	Ві(2)	Кві(2)	Ві(3)	Кві(3)
X1	1,0	0,5	1,5	1,5	4,5	0,281	16,25	0,275	59,125	0,274
X2	1,5	1,0	1,5	1,5	5,5	0,344	21,25	0,360	77,875	0,361
X3	0,5	0,5	1,0	0,5	2,5	0,156	9,25	0,157	34,125	0,158
X4	0,5	0,5	1,5	1,0	3,5	0,219	12,25	0,208	44,875	0,208
Разом					16	1,000	59	1,000	216	1,000

Як видно з порівняння стовпців $K_{\text{ві}}$ (2) і $K_{\text{ві}}$ (3), розбіжність значень вагомості на другій і третій ітерації не перевищує 1 %. Тому як остаточні приймаються значення другої ітерації: $K_{\text{ві}}$ (X1) = 0,275, $K_{\text{ві}}$ (X2) = 0,360, $K_{\text{ві}}$ (X3) = 0,157, $K_{\text{ві}}$ (X4) = 0,208. Найбільшу вагу отримав параметр X2 — точність відтворення маргінального розподілу, що узгоджується з цільовим позиціюванням сурогата як інструмента статистично коректної генерації сценаріїв ринкової динаміки.

4.5 Аналіз рівня якості варіантів реалізації функцій

Рівень якості кожного варіанту реалізації визначається як зважена сума бальних оцінок параметрів зі знайденими у попередньому підрозділі коефіцієнтами вагомості. Коефіцієнт технічного рівня обчислюється за формулою:

$$K_K = \sum_{i=1}^{n} K_{\text{ві}} \cdot B_i \quad (4.6)$$

де $K_{\text{ві}}$ — коефіцієнт вагомості і-го параметра;

B_i — бальна оцінка і-го параметра, отримана за абсолютним значенням параметра у відповідному варіанті;

Абсолютні значення параметрів для двох варіантів реалізації отримано на основі результатів експериментальної оцінки прототипів обох підходів (розділ 3).

Для варіанту 1 (Neural SDE-GAN) середній час генерації пакету з 10 000 траєкторій становить 2,4 с; відстань Васерштейна-1 до реального розподілу — $7{,}0 \cdot 10^{-3}$; обсяг оперативної пам'яті при генерації — 2048 МБ; трудомісткість розробки прогнозується у 950 людино-годин з урахуванням реалізації власного диференційовного інтегратора та змагального навчального циклу.

Для варіанту 2 (одновимірний Neural SDE) час генерації становить 1,8 с; відстань Васерштейна-1 — $11{,}0 \cdot 10^{-3}$; обсяг пам'яті — 1536 МБ; трудомісткість розробки прогнозується у 657 людино-годин завдяки використанню готового пакета torchsde та простішого навчання за правдоподібністю.

Бальні оцінки для кожного варіанту визначено лінійною інтерполяцією між гранично-середнім і середньо-кращим значеннями відповідно до таблиці 4.2. Результати розрахунку наведено у таблиці 4.6.

Таблиця 4.6 — Розрахунок показників рівня якості варіантів реалізації

Функція	Варіант	Парам.	Абс. знач.	Бал. оцінка	Кві	Внесок
F2	б	X1	2,4 с	93	0,275	25,58
F2	б	X2	$7{,}0 \cdot 10^{-3}$	86	0,360	30,96
F3	а	X3	2048 МБ	50	0,157	7,85
F3	а	X4	950 год	65	0,208	13,52
F2	а	X1	1,8 с	98	0,275	26,95
F2	а	X2	$11{,}0 \cdot 10^{-3}$	57	0,360	20,52
F3	б	X3	1536 МБ	67	0,157	10,52
F3	б	X4	657 год	94	0,208	19,55

За даними таблиці 4.6 розраховуємо коефіцієнт технічного рівня кожного з двох варіантів реалізації:

$K_{K1} = 25{,}58 + 30{,}96 + 7{,}85 + 13{,}52 = 77{,}91;$

$K_{K2} = 26{,}95 + 20{,}52 + 10{,}52 + 19{,}55 = 77{,}54.$

Як видно з порівняння, обидва варіанти мають практично еквівалентні коефіцієнти технічного рівня, причому варіант 1 (Neural SDE-GAN) незначно переважає варіант 2 (одновимірний Neural SDE) за рахунок істотно кращої точності відтворення маргінального розподілу (параметр X2 з найбільшою вагомістю). Варіант 2 виграє за параметрами X1, X3 і X4 (швидкодія генерації, обсяг пам'яті та трудомісткість розробки), однак ці переваги частково компенсуються нижчою сумарною вагомістю відповідних параметрів. Подальший вибір кращого варіанту здійснюється з урахуванням повної вартості розробки.

4.6 Економічний аналіз варіантів розробки ПП

Для оцінювання повної вартості розробки програмного продукту спочатку проводиться розрахунок трудомісткості розробки за окремими завданнями. Для варіанту 1 у склад робіт включено три завдання: реалізація власного диференційовного інтегратора СДР за схемою Ейлера-Маруями; реалізація генератора Neural SDE-GAN з двовимірним станом і змагальним навчанням WGAN-GP; інтеграція модулів і протоколу оцінювання. Для варіанту 2 склад робіт — інтеграція зовнішнього пакета torchsde для інтегрування СДР; реалізація одновимірного Neural SDE з навчанням за step-NLL і ACF-регуляризатором; інтеграція модулів і протоколу оцінювання.

За ступенем новизни всі три завдання обох варіантів відносяться до групи Б, оскільки використовуються відомі підходи з суттєвою адаптацією під специфіку задачі сурогатного моделювання динаміки цін. Загальна трудомісткість окремого завдання обчислюється за формулою:

$$T = \mathrm{Т_Р} \cdot \mathrm{К_П} \cdot \mathrm{К_{СК}} \cdot \mathrm{К_М} \cdot \mathrm{К_{СТ}} \cdot \mathrm{К_{СТ.М}} \qquad (4.7)$$

де $\mathrm{Т_Р}$ — базова трудомісткість завдання;

$К_П$ — поправочний коефіцієнт виду вхідної інформації;

$К_{СК}$ — коефіцієнт складності контролю вхідної інформації;

$К_М$ — коефіцієнт рівня мови програмування (для Python 0,85);

$К_{СТ}$ — коефіцієнт використання стандартних модулів;

$К_{СТ.М}$ — коефіцієнт стандартного математичного забезпечення.

Для завдань варіанту 1, підставляючи значення у формулу (4.7), отримуємо:

$T_{1.1} = 62 \cdot 1{,}21 \cdot 1 \cdot 0{,}85 \cdot 0{,}80 \cdot 1 = 51{,}01$ людино-днів (власний інтегратор СДР);

$T_{1.2} = 60 \cdot 1{,}21 \cdot 1 \cdot 0{,}85 \cdot 0{,}85 \cdot 1 = 52{,}45$ людино-днів (генератор Neural SDE-GAN);

$T_{1.3} = 25 \cdot 0{,}9 \cdot 1 \cdot 0{,}85 \cdot 0{,}80 \cdot 1 = 15{,}30$ людино-днів (інтеграція модулів і протоколу).

Загальна трудомісткість розробки варіанту 1 у людино-годинах:

$T_I = (51{,}01 + 52{,}45 + 15{,}30) \cdot 8 = 950{,}08$ людино-годин.

Для завдань варіанту 2 аналогічно отримуємо:

$T_{2.1} = 30 \cdot 1{,}21 \cdot 1 \cdot 0{,}85 \cdot 0{,}70 \cdot 1 = 21{,}60$ людино-днів (інтеграція пакета torchsde);

$T_{2.2} = 55 \cdot 1{,}21 \cdot 1 \cdot 0{,}85 \cdot 0{,}80 \cdot 1 = 45{,}25$ людино-днів (одновимірний Neural SDE);

$T_{2.3} = 15{,}30$ людино-днів (інтеграція модулів і протоколу).

Загальна трудомісткість розробки варіанту 2:

$T_{II} = (21{,}60 + 45{,}25 + 15{,}30) \cdot 8 = 657{,}20$ людино-годин.

У розробці бере участь один інженер-програміст з місячним окладом 32 000 грн. Середня погодинна оплата праці визначається за формулою:

$$С_Ч = \frac{М}{Д_м \cdot t_д} \quad (4.8)$$

де М — місячний оклад працівника;

$Д_м$ — кількість робочих днів у місяці (приймається 21);

$t_д$ — кількість робочих годин у дні (приймається 8).

Тоді $С_ч = 32\,000/(21 \cdot 8) = 190{,}48$ грн/год.

Заробітна плата за весь обсяг розробки розраховується за формулою:

$$С_{ЗП} = С_ч \cdot Т \cdot К_Д \quad (4.9)$$

де $К_Д$ — норматив, що враховує додаткову заробітну плату (приймається 1,2).

Заробітна плата за варіантами становить:

I. $С_{ЗП} = 190{,}48 \cdot 950{,}08 \cdot 1{,}2 = 217\,161{,}14$ грн;

II. $С_{ЗП} = 190{,}48 \cdot 657{,}20 \cdot 1{,}2 = 150\,217{,}14$ грн.

Відрахування на єдиний соціальний внесок становлять 22 % від нарахованої заробітної плати:

I. $С_{ВІД} = 217\,161{,}14 \cdot 0{,}22 = 47\,775{,}45$ грн;

II. $С_{ВІД} = 150\,217{,}14 \cdot 0{,}22 = 33\,047{,}77$ грн.

Окремо визначаються витрати на оплату однієї машино-години роботи ЕОМ. Одна ЕОМ обслуговує одного програміста при коефіцієнті зайнятості КЗ = 0,2, тож умовний річний фонд заробітної плати з обслуговування ЕОМ становить СГ = 12·32 000·0,2 = 76 800 грн, а з урахуванням додаткової заробітної плати — СЗП.ЕОМ = 76 800·1,2 = 92 160 грн; відповідні відрахування СВІД.ЕОМ = 92 160·0,22 = 20 275,20 грн. Амортизаційні відрахування розраховуються за формулою:

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} \quad (4.10)$$

де $К_{ТМ}$ — коефіцієнт витрат на транспортування та монтаж обладнання (1,15);

$К_А$ — річна норма амортизації (0,25);

$Ц_{ПР}$ — договірна ціна обладнання (38 000 грн).

Тоді $С_А = 1{,}15 \cdot 0{,}25 \cdot 38\,000 = 10\,925{,}00$ грн.

Витрати на ремонт і профілактику ЕОМ розраховуються за формулою:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р \quad (4.11)$$

де $К_Р$ — відсоток витрат на поточні ремонти (0,05);

$С_Р = 1{,}15 \cdot 38\,000 \cdot 0{,}05 = 2\,185{,}00$ грн.

Ефективний річний фонд часу роботи ЕОМ розраховується за формулою:

$$Т_{ЕФ} = (Д_К - Д_В - Д_С - Д_Р) \cdot t \cdot К_В \quad (4.12)$$

де $Д_К$ — календарна кількість днів у році;

$Д_В$, $Д_С$ — кількість вихідних і святкових днів;

$Д_Р$ — кількість днів планових ремонтів;

t — кількість робочих годин у дні;

$К_В$ — коефіцієнт використання приладу в часі.

Підстановка значень $Д_К$ = 365, $Д_В$ = 104, $Д_С$ = 11, $Д_Р$ = 18, t = 8, $К_В$ = 0,40 дає $Т_{ЕФ}$ = (365 − 104 − 11 − 18)·8·0,40 = 742,40 години. Витрати на електроенергію визначаються за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} \quad (4.13)$$

де $N_С$ — середньоспоживча потужність приладу (0,3 кВт з урахуванням використання інтегрованого GPU);

$Ц_{ЕН}$ — тариф за 1 кВт·год електроенергії (13,64 грн).

Тоді $С_{ЕЛ}$ = 742,40·0,3·0,2·13,64 = 607,58 грн. Накладні витрати на обслуговування одного робочого місця СН.ЕОМ = ЦПР·0,67 = 38 000·0,67 = 25 460,00 грн.

Річні експлуатаційні витрати на ЕОМ становлять СЕКС = 92 160,00 + 20 275,20 + 10 925,00 + 2 185,00 + 607,58 + 25 460,00 = 151 612,78 грн. Собівартість однієї машино-години роботи ЕОМ: СМ-Г = 151 612,78/742,40 = 204,22 грн/год.

Оскільки всі роботи з розробки виконуються на ЕОМ, витрати на оплату машинного часу розраховуються як добуток собівартості машино-години на трудомісткість відповідного варіанту:

I. СМ = 204,22·950,08 = 194 025,15 грн;

II. СМ = 204,22·657,20 = 134 213,25 грн.

Накладні витрати у складі вартості розробки становлять 67 % від основної заробітної плати:

I. СН = 217 161,14·0,67 = 145 497,97 грн;

II. СН = 150 217,14·0,67 = 100 645,49 грн.

Повна вартість розробки програмного продукту за варіантами:

I. СПП = 217 161,14 + 47 775,45 + 194 025,15 + 145 497,97 = 604 459,71 грн;

II. СПП = 150 217,14 + 33 047,77 + 134 213,25 + 100 645,49 = 418 123,65 грн.

4.7 Вибір кращого варіанту ПП за техніко-економічним рівнем

Для остаточного вибору варіанту реалізації обчислюється коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕР} = \frac{K_К}{С_{ПП}} \quad (4.14)$$

де $K_К$ — коефіцієнт технічного рівня варіанту;

$С_{ПП}$ — повна вартість розробки варіанту, грн.

Підставивши обчислені раніше значення, отримуємо:

1) $К_{ТЕР}$ = 77,91/604 459,71 = 1,289·10⁻⁴ грн⁻¹;

2) $К_{ТЕР}2$ = 77,54/418 123,65 = 1,854·10⁻⁴ грн⁻¹.

Як видно з порівняння значень коефіцієнта техніко-економічного рівня, варіант 2 (одновимірний Neural SDE) має більший коефіцієнт техніко-економічного рівня, 1,854·10⁻⁴ проти 1,289·10⁻⁴ для варіанту 1. Цей результат досягається переважно за рахунок меншої трудомісткості розробки й нижчих ресурсних вимог при практично еквівалентному коефіцієнті технічного рівня (77,54 проти 77,91).

Водночас слід окремо зазначити, що формальна перевага варіанту 2 за коефіцієнтом техніко-економічного рівня не суперечить обраному у роботі шляху побудови архітектури Neural SDE-GAN (варіант 1). Варіант 1 істотно переважає варіант 2 за параметром X2 — точністю відтворення маргінального розподілу дохідностей (на 30–37 % менша відстань Васерштейна), що є визначальним для цільового призначення продукту як інструмента

статистично коректної генерації сценаріїв ринкової динаміки та становить науковий внесок роботи. Крім того, двовимірний стан Heston-типу відкриває природний шлях до подальшого розширення методології на багатші класи моделей стохастичної волатильності. У підсумку, варіант 1 обирається як предметний об’єкт дослідження, а варіант 2 — як еталонний базовий сурогат для оцінювання якості розробленого підходу.

4.8 Висновки до розділу 4

У розділі проведено функціонально-вартісний аналіз двох конкуруючих варіантів реалізації програмного продукту для сурогатного моделювання динаміки цін фінансових активів. За результатами проведеного аналізу зроблено такі висновки.

Сформульовано функціональну структуру продукту: на основі технічних вимог виділено чотири підфункції — вибір мови програмування, архітектуру нейромережевого генератора, спосіб чисельного інтегрування СДР та тип обчислювальної інфраструктури. На основі морфологічного аналізу та позитивно-негативної матриці сформовано два конкуруючих варіанти реалізації: двовимірний Neural SDE-GAN з власною реалізацією схеми Ейлера-Маруями та одновимірний Neural SDE з навчанням за правдоподібністю, що використовує зовнішній пакет torchsde.

Здійснено експертне ранжування системи параметрів продукту: найбільшу вагомість отримав параметр X2 (точність відтворення маргінального розподілу) зі значенням коефіцієнта вагомості 0,360, що відповідає цільовому позиціюванню продукту як інструмента статистично коректної генерації. Високий коефіцієнт узгодженості думок експертів (W = 0,837) підтверджує достовірність встановленої системи пріоритетів.

Виконано розрахунок коефіцієнта технічного рівня для обох варіантів: KK1 = 77,91 для Neural SDE-GAN та KK2 = 77,54 для одновимірного Neural

SDE. Близькість значень пояснюється тим, що варіант 1 переважає варіант 2 за точністю відтворення розподілу (X2), але поступається за швидкодією, обсягом пам’яті та трудомісткістю розробки (X1, X3, X4).

Виконано економічний розрахунок повної вартості розробки за обома варіантами. Для Neural SDE-GAN повна вартість розробки склала 604 460 грн при трудомісткості 950,1 людино-годин. Для одновимірного Neural SDE повна вартість розробки склала 418 124 грн при трудомісткості 657,2 людино-годин. Різниця 186 336 грн пояснюється додатковими трудовими витратами на реалізацію власного диференційовного інтегратора та змагального навчального циклу.

За формальним критерієм техніко-економічного рівня перевага надається варіанту 2 ($КТЕР2 = 1{,}854 \cdot 10^{-4}$ проти $КТЕР1 = 1{,}289 \cdot 10^{-4}$), що відображає кращу економічну ефективність простішого підходу при еквівалентному технічному рівні. Цей результат використовується в роботі як еталонне оцінювання базової вартості розробки. Водночас фактичним об’єктом дослідження є варіант 1, оскільки саме архітектура Neural SDE-GAN забезпечує найкращу точність відтворення маргінального розподілу дохідностей та становить науковий внесок роботи, що природним чином узагальнюється на багатші класи моделей стохастичної волатильності.

ВИСНОВКИ

У ході виконання дипломної роботи розв’язано задачу побудови нейромережевих сурогатних моделей для відтворення динаміки цін фінансових активів. Робота охоплює теоретичний огляд предметної області, обґрунтування власних методологічних рішень, програмну реалізацію та експериментальне дослідження, а також функціонально-вартісний аналіз розробленого програмного продукту.

У ході переддипломної практики виконано повний цикл дослідження сурогатних моделей для відтворення динаміки цін фінансових активів. Розглянуто теоретичні засади сурогатного моделювання, класичні стохастичні моделі (Heston, GBM), стилізовані факти фінансових часових рядів за класифікацією Конта і сучасні нейромережеві архітектури для генерації часових рядів. Реалізовано чотири моделі (LSTM-MDN, Transformer-MDN, Neural SDE без енкодера, Neural SDE з енкодером, Neural SDE-GAN), проведено гіперпараметричний аналіз та експериментальне порівняння на реальних і синтетичних даних [1; 3; 5–7; 17; 20; 21].

Основні результати дослідження такі. У теоретичній частині систематизовано підходи до побудови нейромережевих сурогатних моделей фінансових часових рядів, узагальнено результати літератури за останнє десятиліття (QuantGAN, TimeGAN, Neural SDE), а також сформульовано формальну постановку задачі як генеративну, з критеріями якості, що оцінюють розподіл, а не точкові прогнози.

Реалізовано симулятор моделі Heston за схемою Ейлера-Маруями з технікою повного зрізання дисперсії та декомпозицією Холецького для корельованих шумів. Завантажено та оброблено історичні дані чотирьох активів (S&P 500, Apple, JPMorgan, Tesla) за період 2000–2024 років, виконано

часовий розподіл на навчальну, валідаційну та тестову вибірки з буфером для запобігання витоку даних.

Спроєктовано і реалізовано п'ять нейромережевих архітектур з ймовірнісним виходом: рекурентна LSTM з MDN-головою, Transformer з MDN-головою, дві варіації одновимірного Neural SDE (без і з контекстним енкодером) та запропонована Neural SDE-GAN з двовимірним Heston-станом і adversarial-навчанням. Тренування проведено мультиактивним способом (на об’єднаних даних усіх активів) з оптимізатором Adam, косинусним розкладом швидкості навчання, gradient clipping та функцією втрат NLL + ACF-loss. Гіперпараметри підібрано через grid search 27 конфігурацій Transformer та ablation-аналіз ваги ACF-loss.

Експериментальне оцінювання за широким набором метрик (моменти, KS, Wasserstein, MMD, ARCH-LM, leverage effect, coverage probability, max drawdown) виявило, що архітектури на основі Neural SDE стабільно перевершують MDN-моделі за всіма розподіловими метриками: Wasserstein у 2–4 рази нижчий, KS — приблизно вдвічі, MMD — на порядок. Найкращі результати показала модель NSDE-GAN.

Аналіз обчислювальної ефективності засвідчив значну перевагу нейромережевих сурогатів над класичним Heston-симулятором: для одного активу NSDE-моделі є приблизно у 5,4 рази швидшими, для портфеля з 10 активів — у 8,7 разів. На середньому горизонті прогнозування (252 торгових дні) Neural SDE забезпечують реалістичні траєкторії з адекватним coverage probability.

Окрім позитивних результатів, виявлено фундаментальне обмеження: жодна з чотирьох архітектур не відтворює повноцінно кластеризацію волатильності реальних даних (ARCH-ефект). Це обмеження відоме у літературі і пов’язане з тим, що MDN-голови генерують вибірки незалежно, а поточна реалізація Neural SDE використовує одновимірний SDE без окремої стохастичної динаміки для дисперсії. Це визначає напрямок розвитку у дипломній роботі — впровадження GAN-навчання Neural SDE з

дискримінатором, який зможе явно штрафувати модель за відсутність кластеризації волатильності [11].

Поставлені у завданні на переддипломну практику задачі виконано повністю: проаналізовано сучасні підходи до сурогатного моделювання, реалізовано алгоритми генерації навчальних і тестових вибірок, спроєктовано чотири нейромережеві архітектури, проведено процедуру навчання з підбором гіперпараметрів, виконано експериментальні дослідження якості відтворення статистичних характеристик процесу, проаналізовано отримані результати. Отриманий програмний код, експериментальні дані і результати порівняння є основою для подальшого розвитку у межах дипломної роботи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Згуровський М., Панкратова Н. Основи системного аналізу : підручник. Київ : Видавнича група BHV, 2007. 544 с.
2. Cont R. Empirical properties of asset returns: stylized facts and statistical issues. *Quantitative Finance*. 2001. Vol. 1, No. 2. P. 223–236. DOI: 10.1080/713665670.
3. Gatheral J. The Volatility Surface: A Practitioner's Guide. Hoboken : John Wiley & Sons, 2006. 208 p. DOI: 10.1002/9781119202073.
4. Hull J. C. Options, Futures, and Other Derivatives. 10th ed. New York : Pearson, 2017. 896 p.
5. Glasserman P. Monte Carlo Methods in Financial Engineering. New York : Springer, 2003. 596 p. DOI: 10.1007/978-0-387-21617-1.
6. Ranftl S., von der Linden W. Bayesian Surrogate Analysis and Uncertainty Propagation. *Physical Sciences Forum*. 2021. Vol. 3, No. 1. Art. 6. DOI: 10.3390/psf2021003006.
7. Low-dimensional data-based surrogate model of a continuum-mechanical musculoskeletal system based on non-intrusive model order reduction / J. Kneifl et al. *Archive of Applied Mechanics*. 2023. Vol. 93, No. 9. P. 3637–3663. DOI: 10.1007/s00419-023-02458-5.
8. Hou C. K. J., Behdinan K. Dimensionality Reduction in Surrogate Modeling: A Review of Combined Methods. *Data Science and Engineering*. 2022. Vol. 7, No. 4. P. 402–427. DOI: 10.1007/s41019-022-00193-5.
9. Forrester A. I. J., Sóbester A., Keane A. J. Engineering Design via Surrogate Modelling: A Practical Guide. Chichester : John Wiley & Sons, 2008. 240 p. DOI: 10.1002/9780470770801.

10. Wiese M., Knobloch R., Korn R., Kretschmer P. Quant GANs: Deep Generation of Financial Time Series. *Quantitative Finance*. 2020. Vol. 20, No. 9. P. 1419–1440. DOI: 10.1080/14697688.2020.1730426.

11. Kidger P., Foster J., Li X., Lyons T. Neural SDEs as Infinite-Dimensional GANs. *Proceedings of the 38th International Conference on Machine Learning*. 2021. P. 5453–5463. URL: https://arxiv.org/abs/2102.03657 (Last accessed: 20.05.2026).

12. Lopez de Prado M. Advances in Financial Machine Learning. Hoboken : John Wiley & Sons, 2018. 400 p.

13. Панібратов Р. Система підтримання прийняття рішень для оцінювання та прогнозування стану страхової компанії. *Системні дослідження та інформаційні технології*. 2022. № 1. С. 61–72. DOI: 10.20535/SRIT.2308-8893.2022.1.05.

14. Cox J. C., Ingersoll J. E., Ross S. A. A Theory of the Term Structure of Interest Rates. *Econometrica*. 1985. Vol. 53, No. 2. P. 385–407. DOI: 10.2307/1911242.

15. Bates D. S. Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options. *Review of Financial Studies*. 1996. Vol. 9, No. 1. P. 69–107. DOI: 10.1093/rfs/9.1.69.

16. Gatheral J., Jaisson T., Rosenbaum M. Volatility is rough. *Quantitative Finance*. 2018. Vol. 18, No. 6. P. 933–949. DOI: 10.1080/14697688.2017.1393551.

17. Heston S. L., Loewenstein M., Willard G. A. Options and Bubbles. *Review of Financial Studies*. 2007. Vol. 20, No. 2. P. 359–390. DOI: 10.1093/rfs/hhl005.

18. Attention Is All You Need / A. Vaswani et al. *Advances in Neural Information Processing Systems*. 2017. Vol. 30. URL: https://arxiv.org/abs/1706.03762 (Last accessed: 20.05.2026).

19. Kloeden P. E., Platen E. Numerical Solution of Stochastic Differential Equations. Berlin; Heidelberg : Springer, 1992. 632 p. DOI: 10.1007/978-3-662-12616-5.

20. Andersen L. Efficient Simulation of the Heston Stochastic Volatility Model. *Journal of Computational Finance*. 2008. Vol. 11, No. 3. P. 1–42.
21. Бідюк П., Романенко В., Тимощук О. Аналіз часових рядів : навч. посіб. Київ : Політехніка, 2010. 317 с.
22. Engle R. F. Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. *Econometrica*. 1982. Vol. 50, No. 4. P. 987–1007. DOI: 10.2307/1912773.
23. Karpoff J. M. The Relation Between Price Changes and Trading Volume: A Survey. *Journal of Financial and Quantitative Analysis*. 1987. Vol. 22, No. 1. P. 109–126. DOI: 10.2307/2330874.
24. A Kernel Two-Sample Test / A. Gretton et al. *Journal of Machine Learning Research*. 2012. Vol. 13. P. 723–773. URL: https://www.jmlr.org/papers/v13/gretton12a.html (Last accessed: 20.05.2026).
25. Bishop C. M. Mixture Density Networks. Technical Report NCRG/94/004. Aston University, 1994. 25 p. URL: https://publications.aston.ac.uk/id/eprint/373/ (Last accessed: 20.05.2026).
26. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge : MIT Press, 2016. 800 p. URL: https://www.deeplearningbook.org/ (Last accessed: 20.05.2026).
27. Hochreiter S., Schmidhuber J. Long Short-Term Memory. *Neural Computation*. 1997. Vol. 9, No. 8. P. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.
28. Chen R. T. Q., Rubanova Y., Bettencourt J., Duvenaud D. Neural Ordinary Differential Equations. *Advances in Neural Information Processing Systems*. 2018. Vol. 31. URL: https://arxiv.org/abs/1806.07366 (Last accessed: 20.05.2026).
29. Li X., Wong T.-K. L., Chen R. T. Q., Duvenaud D. Scalable Gradients for Stochastic Differential Equations. *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2020. P. 3870–3882. URL: https://arxiv.org/abs/2001.01328 (Last accessed: 20.05.2026).

30. Kingma D. P., Ba J. Adam: A Method for Stochastic Optimization. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. 2015. URL: https://arxiv.org/abs/1412.6980 (Last accessed: 20.05.2026).

31. Yoon J., Jarrett D., van der Schaar M. Time-series Generative Adversarial Networks. *Advances in Neural Information Processing Systems*. 2019. Vol. 32. URL: https://proceedings.neurips.cc/paper/2019/hash/c9efe5f26cd17ba6216bbe2a7d26d490-Abstract.html (Last accessed: 20.05.2026).

32. Tashiro Y., Song J., Song Y., Ermon S. CSDI: Conditional Score-based Diffusion Models for Probabilistic Time Series Imputation. *Advances in Neural Information Processing Systems*. 2021. Vol. 34. URL: https://arxiv.org/abs/2107.03502 (Last accessed: 20.05.2026).

33. Buehler H., Gonon L., Teichmann J., Wood B. Deep Hedging. *Quantitative Finance*. 2019. Vol. 19, No. 8. P. 1271–1291. DOI: 10.1080/14697688.2019.1571683.

34. Loshchilov I., Hutter F. Decoupled Weight Decay Regularization. *International Conference on Learning Representations*. 2019. URL: https://arxiv.org/abs/1711.05101 (Last accessed: 20.05.2026).

35. PyTorch: An Imperative Style, High-Performance Deep Learning Library / A. Paszke, S. Gross, F. Massa et al. *Advances in Neural Information Processing Systems*. 2019. Vol. 32. URL: https://arxiv.org/abs/1912.01703 (Last accessed: 20.05.2026).

ДОДАТОК А ЛІСТИНГ ПРОГРАМИ

```
# run_diploma.py
"""
Імпортує ВСІ компоненти з diploma_final.py і запускає експерименти:
  1. experiment_multi_asset — multi-seed мульти-активний експеримент
  2. experiment_speed       — швидкодія (1 актив + портфель)
  3. experiment_speed_scaling — масштабування часу від n_steps

Запуск:
  python run_diploma.py

Окремо:
  python acf_ablation.py     — порівняння різних значень ACF_WEIGHT
  python grid_search_1.py    — повний 3D grid search
"""

import time
import json
import numpy as np
import pandas as pd
import torch
from torch.utils.data import DataLoader

# Імпортуємо ВСЕ із diploma_final
# === КОЛЬОРОВИЙ режим графіків (для презентації; вимкнути — закоментуй 2 рядки) ===
import plots_presentation
plots_presentation.enable_full()

from diploma_final import (
    # Константи
    DEVICE, RESULTS_DIR, FIGURES_DIR, TIMESTAMP,
    TICKERS, TICKER_LABELS, SEEDS,
    CONTEXT_LEN, HORIZON, WINDOW_SIZE,
    EPOCHS_LSTM, EPOCHS_TRANSFORMER, EPOCHS_NEURAL_SDE,
    EPOCHS_NEURAL_SDE_GAN, BATCH_SIZE, LR,
    SDE_HIDDEN, SDE_SIGMA_MAX, SDE_ENCODER_HIDDEN, SDE_H_DIM,
    SIGMA_MAX, ACF_WEIGHT, ACF_MAX_LAG, N_COMPONENTS,
    # Утиліти
    set_seed,
    # Дані
    load_multi_asset, split_multi_asset, ReturnsDataset,
    # Симулятор
    simulate_heston, simulate_heston_portfolio,
    # Моделі і навчання
    LSTMSurrogate, TransformerSurrogate, train_model, generate,
    # Метрики
    compute_metrics,
    # Графіки
    plot_assets_timeline, plot_stylized_facts,
    plot_distributions, plot_acf,
    plot_training, plot_metrics_bars, plot_metrics_heatmap,
    plot_speed, plot_speed_scaling,
)

from neural_sde import (
    NeuralSDESurrogateA, NeuralSDESurrogateB,
    train_neural_sde,
)

from neural_sde_gan import (
    NeuralSDEGAN,
    NeuralHestonGAN,
    train_neural_sde_gan,
)

from neural_heston import (
    NeuralHeston,
    train_neural_heston,
)

from plots_v2 import (
    plot_sample_paths,
    plot_pareto_speed_quality,
    generate_extended_paths,
    run_long_horizon_plots,
)
from plots_v3 import plot_cdf_overlay


# ============================================================
# Допоміжні функції
# ============================================================

def train_all_models(train_loader, val_loader):
    """Тренує всі 5 архітектур. NSDE-GAN йде першим: спочатку
    pretrain-backbone (NSDE-B), потім adversarial fine-tune.
    Той самий NSDE-B використовується як модель [5/5] без повторного навчання."""

    print('\n[1/5] Neural SDE-GAN з Neural Heston backbone (2D state + adversarial):')
    print('  Крок 1.1: pretrain NeuralHeston backbone через NLL...')
    print('    (2D-стан Y та v — модель з явною волатильністю)')
    heston = NeuralHeston(horizon=HORIZON,
                          hidden=SDE_HIDDEN,
                          sigma_max=0.5,  # σ_max більший — він множиться на √v
                          encoder_hidden=SDE_ENCODER_HIDDEN,
                          h_dim=SDE_H_DIM,
                          init_var=0.0, init_rho=-0.7,
                          v_scale=0.001)  # масштаб v: до ~0.001
    h_heston = train_neural_heston(heston, train_loader, val_loader,
                                   EPOCHS_NEURAL_SDE, lr=LR,
                                   device=DEVICE)

    print('  Крок 1.2: adversarial fine-tune (WGAN-GP) на Heston backbone...')
    sde_gan = NeuralHestonGAN(horizon=HORIZON,
                              gen_hidden=SDE_HIDDEN,
                              gen_sigma_max=0.5,
                              encoder_hidden=SDE_ENCODER_HIDDEN,
                              h_dim=SDE_H_DIM,
                              init_var=0.0,
                              init_rho=-0.7,
                              v_scale=0.001,
                              disc_hidden=64, disc_layers=2,
                              disc_dropout=0.1)
    sde_gan.load_generator_from(heston)
    print(f'  Generator ініційовано з NeuralHeston backbone (2D Heston-like SDE)')
    h_sde_gan = train_neural_sde_gan(sde_gan, train_loader, val_loader,
                                     EPOCHS_NEURAL_SDE_GAN,
                                     lr_g=5e-5, lr_d=2e-4,
                                     device=DEVICE,
                                     n_critic=3,
                                     lambda_gp=10.0,
                                     use_acf_loss=True,
                                     lambda_acf=0.05,
                                     use_moment_loss=True,
                                     lambda_moment=100.0,
```

```
max_lag=ACF_MAX_LAG)

    print('\n[2/5] LSTM:')
    lstm = LSTMSurrogate()
    h_lstm = train_model(lstm,
train_loader, val_loader, EPOCHS_LSTM)

    print('\n[3/5] Transformer:')
    tx = TransformerSurrogate()
    h_tx = train_model(tx, train_loader,
val_loader, EPOCHS_TRANSFORMER)

    print('\n[4/5] Neural SDE варіант A
(без encoder):')
    sde_a =
NeuralSDESurrogateA(horizon=HORIZON,
hidden=SDE_HIDDEN,

sigma_max=SDE_SIGMA_MAX)
    h_sde_a = train_neural_sde(sde_a,
train_loader, val_loader,

EPOCHS_NEURAL_SDE, lr=LR,
                device=DEVICE,
variant='A')

    print('\n[5/5] Neural SDE варіант B
(з encoder):')
    sde_b =
NeuralSDESurrogateB(horizon=HORIZON,
hidden=SDE_HIDDEN,

sigma_max=SDE_SIGMA_MAX,

encoder_hidden=SDE_ENCODER_HIDDEN,

h_dim=SDE_H_DIM)
    h_sde_b = train_neural_sde(sde_b,
train_loader, val_loader,

EPOCHS_NEURAL_SDE, lr=LR,
                device=DEVICE,
variant='B')



    return (
      {'LSTM': lstm, 'Transformer': tx,
       'NeuralSDE-A': sde_a,
'NeuralSDE-B': sde_b,
       'NeuralSDE-GAN': sde_gan},
      {'LSTM': h_lstm, 'Transformer':
h_tx,
       'NeuralSDE-A': h_sde_a,
'NeuralSDE-B': h_sde_b,
       'NeuralSDE-GAN': h_sde_gan},
    )



def evaluate_models(models,
test_per_ticker):
    """
    Для кожного активу і моделі:
генерує і рахує метрики.

    Повертає:
      metrics_per_ticker: {ticker:
[metrics_per_model]}
      gens_per_ticker:    {ticker:
{model_name: flat array}} — для розподілів/QQ
      paths_per_ticker:   {ticker:
{model_name: (n_windows, horizon)}} — для
sample_paths
    """
    metrics_per_ticker = {}
    gens_per_ticker = {}
    paths_per_ticker = {}
    for ticker, test_w in
test_per_ticker.items():
      ticker_label =
TICKER_LABELS.get(ticker, ticker)
      print(f'\n  --- {ticker_label}
({ticker}) ---')
      test_loader =
DataLoader(ReturnsDataset(test_w), batch_size=128)
      real_test = test_w[:,
CONTEXT_LEN:]
      gens_flat = {}
      gens_paths = {}
      metrics_list = []
      for name, model in
models.items():
        gen = generate(model,
test_loader)  # (n_windows, horizon)
        gens_paths[name] = gen
        gens_flat[name] = gen.flatten()
        m = compute_metrics(real_test,
gen, name)
        metrics_list.append(m)
        arch_str =
(f'{m["arch_lm_err"]:.2f}'
                if not
np.isnan(m["arch_lm_err"]) else 'nan')
        lev_str =
(f'{m["leverage_err"]:.3f}'
                if not
np.isnan(m["leverage_err"]) else 'nan')
        print(f'    {name}:
W={m["wasserstein"]:.5f}, '
              f'KS={m["ks_stat"]:.4f}, '
              f'std_err={m["std_err"]:.5f},
'

f'kurt_err={m["kurt_err"]:.3f}, '
              f'ARCH-
LM_err={arch_str}, lev_err={lev_str}')
        metrics_per_ticker[ticker] =
metrics_list
        gens_per_ticker[ticker] = gens_flat
        paths_per_ticker[ticker] =
gens_paths
    return metrics_per_ticker,
gens_per_ticker, paths_per_ticker



def aggregate_seeds(seed_metrics_list):
    """
    Усереднює метрики по сідах.

    seed_metrics_list: список з
len(SEEDS) елементів, кожен — dict
      {ticker: [metrics_per_model]}.

    Повертає dict {ticker:
[metrics_with_means_and_stds]}.
    """
    tickers =
list(seed_metrics_list[0].keys())
    aggregated = {}
    for ticker in tickers:
      n_models =
len(seed_metrics_list[0][ticker])
      per_model = []
      for model_idx in range(n_models):
        label =
seed_metrics_list[0][ticker][model_idx]['label']
        agg = {'label': label}
        sample =
seed_metrics_list[0][ticker][model_idx]
        keys_to_avg = [k for k in
sample.keys() if k != 'label']
        for k in keys_to_avg:
          vals = []
          for sm in seed_metrics_list:
            v =
sm[ticker][model_idx][k]
            if isinstance(v, float) and
np.isnan(v):
              continue
            vals.append(v)
          if vals:
            agg[k] =
float(np.mean(vals))
            agg[f'{k}_std'] =
float(np.std(vals))
          else:
            agg[k] = float('nan')
            agg[f'{k}_std'] =
float('nan')
        per_model.append(agg)
      aggregated[ticker] = per_model
    return aggregated



#
====================================
==========================
# Експеримент 1: Multi-seed мульти-
активний
#
====================================
==========================

def
experiment_multi_asset(seeds=None):
    if seeds is None:
      seeds = SEEDS
    print('\n' + '='*70)
    print('ЕКСПЕРИМЕНТ 1: Мульти-
активна оцінка моделей (multi-seed)')
    print(f'  Активи: {",
".join(TICKERS)}')
    print(f'  Сіди:   {seeds}
(n={len(seeds)})')
    print('='*70)

    print('\nЗавантаження даних...')
    asset_data =
load_multi_asset(TICKERS)
    if len(asset_data) == 0:
      print('  Не вдалося завантажити
жодного активу.')
      return None

    plot_assets_timeline(asset_data,
'assets_timeline')
```

```
            print("\nГенерація стилізованих
фактів для кожного активу...")
            for ticker, data in asset_data.items():
                ticker_safe = ticker.replace('^',
'').replace('-', '_')
                plot_stylized_facts(
                    data['returns'], data['dates'],
                    f'{ticker_safe}_stylized_facts',

ticker_label=TICKER_LABELS.get(ticker, ticker))

            print("\nРозбиття на train/val/test...")
            train_w, val_w, test_per_ticker =
split_multi_asset(
                asset_data, WINDOW_SIZE,
stride=2)
            print(f"\nЗагальна train-вибірка:
{len(train_w)} вікон")
            print(f"Загальна val-вибірка:
{len(val_w)} вікон")

            # === Multi-seed loop ===
            seed_metrics_list = []
            last_models = None
            last_histories = None
            last_gens_per_ticker = None
            last_paths_per_ticker = None

            for seed_idx, seed in
enumerate(seeds):
                print("\n" + "#"*70)
                print(f"#  СІД
{seed_idx+1}/{len(seeds)} (seed={seed})")
                print("#"*70)
                set_seed(seed)

                train_loader =
DataLoader(ReturnsDataset(train_w),

batch_size=BATCH_SIZE, shuffle=True)
                val_loader =
DataLoader(ReturnsDataset(val_w),

batch_size=BATCH_SIZE)

                models, histories =
train_all_models(train_loader, val_loader)

                print(f"\nОцінка моделей на test-
вибірках (seed={seed})...")
                metrics_per_ticker,
gens_per_ticker, paths_per_ticker = (
                    evaluate_models(models,
test_per_ticker))

seed_metrics_list.append(metrics_per_ticker)

                last_models = models
                last_histories = histories
                last_gens_per_ticker =
gens_per_ticker
                last_paths_per_ticker =
paths_per_ticker

                for ticker, metrics_list in
metrics_per_ticker.items():
                    ticker_safe = ticker.replace('^',
'').replace('-', '_')

pd.DataFrame(metrics_list).to_csv(
f"{RESULTS_DIR}/metrics_{ticker_safe}_seed{seed
}.csv",
                        index=False,
float_format="%.6f")

            print("\n" + "="*70)
            print(f"Агрегація метрик по
{len(seeds)} сідах...")
            print("="*70)
            metrics_agg =
aggregate_seeds(seed_metrics_list)

            plot_training(last_histories,
'training_curves')

            for ticker, test_w in
test_per_ticker.items():
                ticker_label =
TICKER_LABELS.get(ticker, ticker)
                ticker_safe = ticker.replace('^',
'').replace('-', '_')
                real_test_flat = test_w[:,
CONTEXT_LEN:].flatten()

                gens =
last_gens_per_ticker[ticker]
                gen_paths =
last_paths_per_ticker[ticker]
                real_paths = test_w[:,
CONTEXT_LEN:]  # (n_windows, horizon)

                plot_distributions(real_test_flat,
gens,

f'{ticker_safe}_distributions',

ticker_label=ticker_label)
                plot_acf(real_test_flat, gens,
                    f'{ticker_safe}_acf',
max_lag=10,
                    ticker_label=ticker_label)

                # === Нові графіки (plots_v2)
===
                plot_cdf_overlay(real_test_flat,
gens,

f'{ticker_safe}_cdf_overlay',

ticker_label=ticker_label)
                plot_sample_paths(real_paths,
gen_paths,

f'{ticker_safe}_sample_paths',

ticker_label=ticker_label,
                                  n_real=1, n_gen=30,
mode='price',
                                  start_price=100.0)


plot_metrics_bars(metrics_agg[ticker],

f'{ticker_safe}_metrics',

ticker_label=ticker_label,

show_error_bars=(len(seeds) > 1))

pd.DataFrame(metrics_agg[ticker]).to_csv(

f'{RESULTS_DIR}/metrics_{ticker_safe}_aggregated
.csv',
                    index=False,
float_format="%.6f")

            # Довгострокові траєкторії: 4
горизонти (1m, 3m, 6m, 1y)
            run_long_horizon_plots(last_models,
asset_data, TICKER_LABELS, n_gen=30)

            print("\nГенерація зведених
heatmap...")
            plot_metrics_heatmap(metrics_agg,
'wasserstein',
                                 'heatmap_wasserstein',

metric_label='Wasserstein distance (mean across
seeds)')
            plot_metrics_heatmap(metrics_agg,
'mmd',
                                 'heatmap_mmd',
                                 metric_label='MMD²
(RBF kernel, mean across seeds)')
            plot_metrics_heatmap(metrics_agg,
'ks_stat',
                                 'heatmap_ks',
                                 metric_label='KS
statistic (mean across seeds)')
            plot_metrics_heatmap(metrics_agg,
'kurt_err',
                                 'heatmap_kurt_err',

metric_label=r'$|\Delta\mathrm{kurt}|$ (mean)')
            plot_metrics_heatmap(metrics_agg,
'arch_lm_err',
                                 'heatmap_arch_lm_err',

metric_label=r'$|\Delta$ARCH-LM$|$ (mean)')
            # heatmap_leverage_err прибрано
— лишаємо тільки основні метрики

            summary_rows = []
            for ticker, metrics_list in
metrics_agg.items():
                for m in metrics_list:
                    row = {'ticker': ticker,
                           'asset':
TICKER_LABELS.get(ticker, ticker),
                           'n_seeds': len(seeds)}
                    row.update(m)
                    summary_rows.append(row)

pd.DataFrame(summary_rows).to_csv(

f'{RESULTS_DIR}/metrics_summary_aggregated.csv
',
                index=False, float_format='%.6f')

            return {
                'metrics_agg': metrics_agg,
                'seed_metrics_list':
seed_metrics_list,
                'models': last_models,
                'asset_data': asset_data,
                'n_seeds': len(seeds),
            }
```

```
    #
====================================
============================
    # Експеримент 2: Швидкодія
    #
====================================
============================

    def experiment_speed(models,
n_paths=10_000, n_steps=500):
        print('\n' + '='*70)
        print('ЕКСПЕРИМЕНТ 2:
Швидкодія')
        print(f'  n_paths={n_paths},
n_steps={n_steps}')
        print('='*70)

        rows = []
        for n_assets in [1, 10]:
            scenario = '1 актив' if n_assets ==
1 else f'портфель {n_assets} активів'
            print(f'\n  Сценарій: {scenario}')

            t0 = time.time()
            if n_assets == 1:

simulate_heston(n_paths=n_paths, n_steps=n_steps,
seed=0)
            else:

simulate_heston_portfolio(n_paths=n_paths,
n_steps=n_steps,

n_assets=n_assets, seed=0)
            t_heston = time.time() - t0
            print(f'    Heston (Euler, {n_steps}
кроків): {t_heston:.3f}с')

            rows.append({'scenario': scenario,
'method': 'Heston (Euler)',
                         'time_seconds': t_heston,
'speedup': 1.0})

            for name, model in
models.items():
                model.eval()
                ctx = torch.randn(n_paths,
CONTEXT_LEN, device=DEVICE)
                with torch.no_grad():
                    _ = model.sample(ctx[:32],
n_samples=1)
                if DEVICE == 'cuda':
                    torch.cuda.synchronize()
                t0 = time.time()
                _ = model.sample(ctx,
n_samples=1)
                if DEVICE == 'cuda':
                    torch.cuda.synchronize()
                t_surr = (time.time() - t0) *
n_assets
                speedup = t_heston / t_surr if
t_surr > 0 else float('inf')
                print(f'    {name}: {t_surr:.3f}с
(×{speedup:.2f})')
                rows.append({'scenario':
scenario, 'method': name,
                             'time_seconds': t_surr,
'speedup': speedup})

        df = pd.DataFrame(rows)

df.to_csv(f'{RESULTS_DIR}/speed.csv',
index=False, float_format='%.4f')

        # plot_speed (portfolio bar)
прибрано — speed_scaling і pareto інформативніші
        return rows


    #
====================================
============================
    # Експеримент 3: Масштабування
швидкодії
    #
====================================
============================

    def experiment_speed_scaling(models,
n_paths=1000):
        print('\n' + '='*70)
        print('ЕКСПЕРИМЕНТ 3:
Масштабування швидкодії від кількості кроків')
        print('='*70)

        n_steps_range = [50, 100, 200, 500,
1000, 2000]
        rows = []

        for n_steps in n_steps_range:
            print(f'\n  n_steps={n_steps}')

            t0 = time.time()
            simulate_heston(n_paths=n_paths,
n_steps=n_steps, seed=0)
            t_h = time.time() - t0
            rows.append({'method': 'Heston
(Euler)',
                         'n_steps': n_steps, 'time':
t_h})
            print(f'    Heston (Euler):
{t_h:.3f}с')

            for name, model in
models.items():
                model.eval()
                ctx = torch.randn(n_paths,
CONTEXT_LEN, device=DEVICE)
                is_neural_sde = hasattr(model,
'sde')

                if is_neural_sde:
                    original_horizon =
model.horizon
                    model.horizon = n_steps
                    with torch.no_grad():
                        _ = model.sample(ctx[:32],
n_samples=1)
                    if DEVICE == 'cuda':
                        torch.cuda.synchronize()
                    t0 = time.time()
                    _ = model.sample(ctx,
n_samples=1)
                    if DEVICE == 'cuda':
                        torch.cuda.synchronize()
                    t_m = time.time() - t0
                    model.horizon =
original_horizon
                else:
                    n_iter = max(1, n_steps //
HORIZON)
                    with torch.no_grad():
                        _ = model.sample(ctx[:32],
n_samples=1)
                    if DEVICE == 'cuda':
                        torch.cuda.synchronize()
                    t0 = time.time()
                    for _ in range(n_iter):
                        _ = model.sample(ctx,
n_samples=1)
                    if DEVICE == 'cuda':
                        torch.cuda.synchronize()
                    t_m = time.time() - t0

                rows.append({'method': name,
'n_steps': n_steps, 'time': t_m})
                print(f'    {name}: {t_m:.3f}с')

        df = pd.DataFrame(rows)

df.to_csv(f'{RESULTS_DIR}/speed_scaling.csv',
                  index=False,
float_format='%.4f')
        plot_speed_scaling(df,
'speed_scaling')
        return rows


    #
====================================
============================
    # Точка входу
    #
====================================
============================

    def main():
        print(f'Device: {DEVICE}')
        print(f'Timestamp: {TIMESTAMP}')
        print(f'Результати:
{FIGURES_DIR}/, {RESULTS_DIR}/')

        config = {
            'timestamp': TIMESTAMP,
            'device': DEVICE,
            'tickers': TICKERS,
            'seeds': SEEDS,
            'N_COMPONENTS':
N_COMPONENTS,
            'SIGMA_MAX': SIGMA_MAX,
            'ACF_WEIGHT': ACF_WEIGHT,
            'ACF_MAX_LAG':
ACF_MAX_LAG,
            'CONTEXT_LEN':
CONTEXT_LEN,
            'HORIZON': HORIZON,
            'EPOCHS_LSTM':
EPOCHS_LSTM,
            'EPOCHS_TRANSFORMER':
EPOCHS_TRANSFORMER,
            'EPOCHS_NEURAL_SDE':
EPOCHS_NEURAL_SDE,
            'BATCH_SIZE': BATCH_SIZE,
            'LR': LR,
            'SDE_HIDDEN': SDE_HIDDEN,
            'SDE_SIGMA_MAX':
SDE_SIGMA_MAX,
            'SDE_ENCODER_HIDDEN':
SDE_ENCODER_HIDDEN,
```

```
            'SDE_H_DIM': SDE_H_DIM,
        }
        with
open(f'{RESULTS_DIR}/config.json', 'w',
encoding='utf-8') as f:
            json.dump(config, f, indent=2,
ensure_ascii=False)
        print(f'Конфігурація збережена:
{RESULTS_DIR}/config.json\n')

        all_results = {}

        res = experiment_multi_asset()
        if res is None:
            print('Не вдалося провести
експеримент.')
            return


all_results['multi_asset_metrics_aggregated'] = {
            ticker: metrics for ticker, metrics
in res['metrics_agg'].items()
        }
        all_results['n_seeds'] = res['n_seeds']

        speeds =
experiment_speed(res['models'])
        all_results['speed'] = speeds

        scaling =
experiment_speed_scaling(res['models'])
        all_results['speed_scaling'] = scaling

        # === Pareto: швидкодія × якість
===
        # Швидкодія — час на 1 актив
(10000 траєкторій)
        speed_dict = {row['method']:
row['time_seconds']
                      for row in speeds if
row['scenario'] == '1 актив'}
        # Якість — середня Wasserstein-
відстань по всіх активах
        quality_dict = {}
        for ticker, m_list in
res['metrics_agg'].items():
            for m in m_list:

quality_dict.setdefault(m['label'],
[]).append(m['wasserstein'])
        quality_dict = {k: float(np.mean(v))
for k, v in quality_dict.items()}

plot_pareto_speed_quality(speed_dict, quality_dict,

'pareto_speed_quality',

ticker_label='усереднено по активах')

        _final_dump_and_exit(all_results)



    def _final_dump_and_exit(all_results):
        with
open(f'{RESULTS_DIR}/all_results.json', 'w',
encoding='utf-8') as f:
            json.dump(all_results, f, indent=2,
ensure_ascii=False, default=str)
        print('\n' + '='*70)
        print('ВСІ ЕКСПЕРИМЕНТИ
ЗАВЕРШЕНО')
        print('='*70)
        print(f'\nГрафіки:
{FIGURES_DIR}/')
        print(f'Таблиці: {RESULTS_DIR}/')


if __name__ == '__main__':
    main()

# diploma_final.py
"""
Сурогатна нейромережева модель
для відтворення динаміки цін фінансових активів.
Фінальна версія для дипломної
роботи (мульти-активна).

Активи:
  ^GSPC — S&P 500 (індекс
широкого ринку)
  AAPL  — Apple
  JPM   — JPMorgan Chase
  TSLA  — Tesla

Архітектури (4 моделі):
  1. LSTM + MDN (baseline)
  2. Transformer + MDN (основна)
  3. Neural SDE варіант A (без
encoder)
  4. Neural SDE варіант B (з encoder)

Експерименти:
  1. Якість відтворення розподілу —
на 4 активах окремо
  2. Швидкодія: 1 актив + портфель
з 10 активів
  3. Масштабування швидкодії від
кількості часових кроків

Запуск:
  pip install torch numpy pandas
yfinance scipy matplotlib torchsde
  python diploma_final.py
"""

import os
import json
import math
import time
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset,
DataLoader
from scipy import stats
from scipy.stats import
wasserstein_distance, ks_2samp
from datetime import datetime

from neural_sde import (
    NeuralSDESurrogateA,
NeuralSDESurrogateB,
    train_neural_sde,
)

#
====================================
============================
# 0. Налаштування
#
====================================
============================

DEVICE = 'cuda' if
torch.cuda.is_available() else 'cpu'

TIMESTAMP =
datetime.now().strftime('%m%d_%H%M%S')
RESULTS_DIR =
f'results/results_{TIMESTAMP}'
FIGURES_DIR =
f'figures/figures_{TIMESTAMP}'

os.makedirs(RESULTS_DIR,
exist_ok=True)
os.makedirs(FIGURES_DIR,
exist_ok=True)


# Гіперпараметри MDN
N_COMPONENTS = 3  # Підібрано
auto-tuning'ом: оптимум для стабільності MDN
SIGMA_MAX = 0.05  # Підібрано
auto-tuning'ом
ACF_WEIGHT = 0.1  # Увімкнено: за
замовчуванням 0.1 (див. acf_ablation.py для
розгорнутого порівняння)
ACF_MAX_LAG = 5

# Параметри даних
CONTEXT_LEN = 60
HORIZON = 20
WINDOW_SIZE = CONTEXT_LEN +
HORIZON

# Параметри тренування
EPOCHS_LSTM = 20
EPOCHS_TRANSFORMER = 30  #
Збільшено: Transformer тренується повільніше,
потребує більше епох для стабілізації MDN
EPOCHS_NEURAL_SDE = 20
EPOCHS_NEURAL_SDE_GAN = 30
# GAN-навчання потребує більше епох для
стабільності
BATCH_SIZE = 256
LR = 1e-3

# Параметри Neural SDE
SDE_HIDDEN = 64
SDE_SIGMA_MAX = 0.08  #
Підвищено з 0.015: дає моделі простір для
кризових режимів
SDE_ENCODER_HIDDEN = 32
SDE_H_DIM = 16

# Активи
TICKERS = ['^GSPC', 'AAPL', 'JPM',
'TSLA']
TICKER_LABELS = {
  '^GSPC': 'S&P 500',
  'AAPL': 'Apple',
  'JPM': 'JPMorgan',
  'TSLA': 'Tesla',
}
```

```
SEEDS = [42]


def set_seed(seed):
    """Фіксація всіх джерел випадковості для відтворюваності."""
    import random
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)


# ============================================================================
# 1. Симулятор моделі Heston (Euler-Maruyama)
# ============================================================================

def simulate_heston(n_paths, n_steps, S0=100.0, v0=0.04, mu=0.05,
                    kappa=2.0, theta=0.04, sigma=0.3, rho=-0.7,
                    T=1.0, seed=None):
    if seed is not None:
        np.random.seed(seed)
    dt = T / n_steps
    S = np.zeros((n_paths, n_steps + 1))
    v = np.zeros((n_paths, n_steps + 1))
    S[:, 0], v[:, 0] = S0, v0

    for t in range(n_steps):
        Z1 = np.random.randn(n_paths)
        Z2 = np.random.randn(n_paths)
        dW1 = np.sqrt(dt) * Z1
        dW2 = np.sqrt(dt) * (rho * Z1 + np.sqrt(1 - rho**2) * Z2)
        v_pos = np.maximum(v[:, t], 0)
        S[:, t+1] = S[:, t] * np.exp((mu - 0.5 * v_pos) * dt + np.sqrt(v_pos) * dW1)
        v[:, t+1] = v[:, t] + kappa * (theta - v_pos) * dt + sigma * np.sqrt(v_pos) * dW2
    return S, v


def simulate_heston_portfolio(n_paths, n_steps, n_assets=10, **kwargs):
    seed = kwargs.pop('seed', None)
    if seed is not None:
        np.random.seed(seed)
    dt = kwargs.pop('T', 1.0) / n_steps
    S0 = kwargs.pop('S0', 100.0)
    v0 = kwargs.pop('v0', 0.04)
    mu, kappa, theta, sigma, rho = (kwargs.pop(k, v) for k, v in
        [('mu', 0.05), ('kappa', 2.0), ('theta', 0.04), ('sigma', 0.3), ('rho', -0.7)])
    S = np.full((n_paths, n_assets, n_steps + 1), S0)
    v = np.full((n_paths, n_assets, n_steps + 1), v0)
    for t in range(n_steps):
        Z1 = np.random.randn(n_paths, n_assets)
        Z2 = np.random.randn(n_paths, n_assets)
        dW1 = np.sqrt(dt) * Z1
        dW2 = np.sqrt(dt) * (rho * Z1 + np.sqrt(1 - rho**2) * Z2)
        v_pos = np.maximum(v[:, :, t], 0)
        S[:, :, t+1] = S[:, :, t] * np.exp((mu - 0.5 * v_pos) * dt + np.sqrt(v_pos) * dW1)
        v[:, :, t+1] = v[:, :, t] + kappa * (theta - v_pos) * dt + sigma * np.sqrt(v_pos) * dW2
    return S


# ============================================================================
# 2. Дані: завантаження кількох активів
# ============================================================================

def prices_to_logreturns(prices):
    return np.diff(np.log(np.asarray(prices)))


def load_multi_asset(tickers, start='2000-01-01', end='2024-12-31',
                     min_points=500):
    """
    Завантажує кілька тікерів з yfinance. Кожен актив починається
    з реальної дати IPO (yfinance автоматично обрізає до доступного).

    Повертає словник: ticker → {'prices', 'returns', 'dates', 'first_date', 'n_points'}
    """
    import yfinance as yf
    result = {}
    for ticker in tickers:
        try:
            df = yf.download(ticker, start=start, end=end, progress=False)
            if len(df) < min_points:
                print(f"  {ticker}: замало даних ({len(df)}), пропускаю")
                continue
            prices = df['Close'].squeeze().values
            dates = df.index.to_pydatetime()
            returns = prices_to_logreturns(prices)
            result[ticker] = {
                'prices': prices,
                'returns': returns,
                'dates': dates,
                'first_date': dates[0],
                'n_points': len(returns),
            }
            print(f"  {ticker} ({TICKER_LABELS.get(ticker, ticker)}): "
                  f"{len(returns)} точок з {dates[0].strftime('%Y-%m-%d')} "
                  f"до {dates[-1].strftime('%Y-%m-%d')}")
        except Exception as e:
            print(f"  {ticker}: помилка — {e}")
    return result


def make_windows(returns, window_size, stride=2):
    return np.array([returns[i:i+window_size]
                     for i in range(0, len(returns) - window_size + 1, stride)])


def temporal_split(returns, window_size, stride=2,
                   train_frac=0.7, val_frac=0.15, gap=None,
                   verbose=False, ticker=''):
    """
    Часовий розподіл вибірки на train / val / test з буферами.

    Структура:
      [ train ][gap][ val ][gap][ test ]
      |0      n_t|  |      n_v|  |.........|

    gap >= window_size гарантує відсутність перекриття вікон між блоками.
    Перевіряється assertion-ами на кінці.
    """
    if gap is None:
        gap = window_size
    assert gap >= window_size, (
        f"gap={gap} має бути >= window_size={window_size}, "
        f"інакше можливе перекриття вікон")

    n = len(returns)
    n_train = int(train_frac * n)
    n_val = int(val_frac * n)

    train_start, train_end = 0, n_train
    val_start, val_end = n_train + gap, n_train + gap + n_val
    test_start, test_end = n_train + n_val + 2 * gap, n

    assert val_start - train_end >= gap, 'leakage між train та val'
    assert test_start - val_end >= gap, 'leakage між val та test'
    assert test_end > test_start + window_size, (
        f"test-вибірка занадто мала: {test_end - test_start} < {window_size}")

    train = make_windows(returns[train_start:train_end], window_size, stride)
    val = make_windows(returns[val_start:val_end], window_size, stride)
    test = make_windows(returns[test_start:test_end], window_size, stride)

    if verbose:
        prefix = f"  [{ticker}] " if ticker else "  "
        print(f"{prefix}temporal_split (n={n}, window={window_size}, gap={gap}):")
```

```
                print(f'{prefix} train: індекси [{train_start:5d}, {train_end:5d}) '
                      f'→ {len(train)} вікон')
                print(f'{prefix} val: індекси [{val_start:5d}, {val_end:5d}) '
                      f'→ {len(val)} вікон')
                print(f'{prefix} test: індекси [{test_start:5d}, {test_end:5d}) '
                      f'→ {len(test)} вікон')

            return train, val, test


        def split_multi_asset(asset_data, window_size, stride=2):
            """
            Для кожного активу робить temporal split, потім ОБ'ЄДНУЄ train і val
            у спільні вибірки. Test залишається ОКРЕМО для кожного активу.

            Повертає:
              train_w_combined: ndarray (об'єднані train-вікна з усіх активів)
              val_w_combined: ndarray (об'єднані val-вікна)
              test_per_ticker: dict {ticker: ndarray} — test-вікна окремо
            """
            train_list, val_list = [], []
            test_per_ticker = {}

            for ticker, data in asset_data.items():
                returns = data['returns']
                train_w, val_w, test_w = temporal_split(returns, window_size, stride,
                                                        verbose=True, ticker=ticker)
                if len(test_w) < 50:
                    print(f"  {ticker}: замала test-вибірка ({len(test_w)}), пропускаю")
                    continue
                train_list.append(train_w)
                val_list.append(val_w)
                test_per_ticker[ticker] = test_w

            train_combined = np.concatenate(train_list, axis=0)
            val_combined = np.concatenate(val_list, axis=0)

            return train_combined, val_combined, test_per_ticker


        class ReturnsDataset(Dataset):
            def __init__(self, windows, context_len=CONTEXT_LEN, horizon=HORIZON):
                assert windows.shape[1] == context_len + horizon
                self.windows = torch.FloatTensor(windows)
                self.context_len = context_len
                self.horizon = horizon

            def __len__(self):
                return len(self.windows)

            def __getitem__(self, idx):
                w = self.windows[idx]
                return w[:self.context_len], w[self.context_len:]


        # ============================================================
        # 3. MDN — Mixture Density Network
        # ============================================================

        class MDNHead(nn.Module):
            def __init__(self, feat_dim, horizon, n_components=N_COMPONENTS,
                         sigma_max=SIGMA_MAX):
                super().__init__()
                self.horizon = horizon
                self.K = n_components
                self.sigma_max = sigma_max
                self.linear = nn.Linear(feat_dim, horizon * 3 * n_components)

            def forward(self, h):
                B = h.size(0)
                out = self.linear(h).view(B, self.horizon, 3, self.K)
                log_pi = F.log_softmax(out[:, :, 0, :], dim=-1)
                mu = out[:, :, 1, :]
                log_sigma = out[:, :, 2, :].clamp(min=-7, max=math.log(self.sigma_max))
                return log_pi, mu, log_sigma


        def mdn_nll_loss(log_pi, mu, log_sigma, target):
            target = target.unsqueeze(-1)
            sigma = torch.exp(log_sigma)
            log_prob = -0.5 * ((target - mu) / sigma) ** 2 - log_sigma - 0.5 * math.log(2 * math.pi)
            log_mix = torch.logsumexp(log_pi + log_prob, dim=-1)
            return -log_mix.mean()


        def mdn_sample(log_pi, mu, log_sigma, n_samples=1):
            B, H, K = mu.shape
            pi = log_pi.exp()
            cat = torch.distributions.Categorical(probs=pi)
            idx = cat.sample((n_samples,))

            idx_exp = idx.unsqueeze(-1)
            mu_exp = mu.unsqueeze(0).expand(n_samples, -1, -1, -1)
            sigma_exp = log_sigma.exp().unsqueeze(0).expand(n_samples, -1, -1, -1)
            chosen_mu = mu_exp.gather(-1, idx_exp).squeeze(-1)
            chosen_sigma = sigma_exp.gather(-1, idx_exp).squeeze(-1)

            eps = torch.randn_like(chosen_mu)
            return chosen_mu + chosen_sigma * eps


        # ============================================================
        # 4. ACF-loss
        # ============================================================

        def differentiable_acf(x, max_lag=ACF_MAX_LAG):
            x = x - x.mean(dim=1, keepdim=True)
            var = (x ** 2).mean(dim=1, keepdim=True) + 1e-8
            return torch.cat([
                ((x[:, :-k] * x[:, k:]).mean(dim=1, keepdim=True) / var)
                for k in range(1, max_lag + 1)
            ], dim=1)


        def acf_loss(samples, target, max_lag=ACF_MAX_LAG):
            s_flat = samples.reshape(-1, samples.size(-1))
            acf_gen = differentiable_acf(s_flat ** 2, max_lag).mean(dim=0)
            acf_real = differentiable_acf(target ** 2, max_lag).mean(dim=0)
            return F.mse_loss(acf_gen, acf_real)


        # ============================================================
        # 5. Архітектури — LSTM і Transformer
        # ============================================================

        class PositionalEncoding(nn.Module):
            def __init__(self, d_model, max_len=200):
                super().__init__()
                pe = torch.zeros(max_len, d_model)
                position = torch.arange(0, max_len).unsqueeze(1).float()
                div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                                     -(math.log(10000.0) / d_model))
                pe[:, 0::2] = torch.sin(position * div_term)
                pe[:, 1::2] = torch.cos(position * div_term)
                self.register_buffer('pe', pe.unsqueeze(0))

            def forward(self, x):
                return x + self.pe[:, :x.size(1)]


        class LSTMSurrogate(nn.Module):
            def __init__(self, hidden=64, num_layers=2, horizon=HORIZON):
                super().__init__()
                self.horizon = horizon
```

```
        self.lstm = nn.LSTM(1, hidden, num_layers=num_layers, batch_first=True)
        self.head = MDNHead(hidden, horizon)

    def forward(self, context):
        x = context.unsqueeze(-1)
        _, (h, _) = self.lstm(x)
        return self.head(h[-1])

    def sample(self, context, n_samples=1):
        return mdn_sample(*self(context), n_samples)


class TransformerSurrogate(nn.Module):
    # Параметри обрано auto-tuning'ом (random search, 12 конфігурацій, 30 епох):
    # d_model=96, num_layers=2, dropout=0.1, nhead=8 — оптимум за
    # композитною метрикою (Wasserstein + штраф за blowup куртозису).
    # Цікаве спостереження: для MDN-Transformer dropout (0.1) і shallow
    # архітектура (2 шари) ефективніше запобігають колапсу мод, ніж
    # глибокий transformer без dropout.
    def __init__(self, d_model=96, nhead=8, num_layers=2, horizon=HORIZON, dropout=0.1):
        super().__init__()
        self.horizon = horizon
        self.input_proj = nn.Linear(1, d_model)
        self.pos_enc = PositionalEncoding(d_model)
        layer = nn.TransformerEncoderLayer(
            d_model=d_model, nhead=nhead, dim_feedforward=4*d_model,
            dropout=dropout, batch_first=True, activation='gelu'
        )
        self.encoder = nn.TransformerEncoder(layer, num_layers=num_layers)
        self.head = MDNHead(d_model, horizon)

    def forward(self, context):
        x = context.unsqueeze(-1)
        x = self.input_proj(x)
        x = self.pos_enc(x)
        h = self.encoder(x).mean(dim=1)
        return self.head(h)

    def sample(self, context, n_samples=1):
        return mdn_sample(*self(context), n_samples)


# ============================================================================
# 6. Тренувальний цикл для MDN-моделей
# ============================================================================

def train_model(model, train_loader, val_loader, epochs, lr=LR,
                acf_weight=ACF_WEIGHT, n_acf_samples=4):
    model.to(DEVICE)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, T_max=epochs)
    history = {'train': [], 'val': []}

    for ep in range(epochs):
        model.train()
        train_losses = []
        for ctx, tgt in train_loader:
            ctx, tgt = ctx.to(DEVICE), tgt.to(DEVICE)
            opt.zero_grad()
            log_pi, mu, log_sigma = model(ctx)
            nll = mdn_nll_loss(log_pi, mu, log_sigma, tgt)
            samples = mdn_sample(log_pi, mu, log_sigma, n_acf_samples)
            acf_l = acf_loss(samples, tgt)
            loss = nll + acf_weight * acf_l
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
            train_losses.append(loss.item())
        sched.step()

        model.eval()
        val_losses = []
        with torch.no_grad():
            for ctx, tgt in val_loader:
                ctx, tgt = ctx.to(DEVICE), tgt.to(DEVICE)
                log_pi, mu, log_sigma = model(ctx)
                val_losses.append(mdn_nll_loss(log_pi, mu, log_sigma, tgt).item())

        tl, vl = np.mean(train_losses), np.mean(val_losses)
        history['train'].append(tl)
        history['val'].append(vl)
        print(f"  Епоха {ep+1}/{epochs}: train={tl:.4f}, val={vl:.4f}")
    return history


def generate(model, loader):
    """Універсальна функція генерації — для MDN і Neural SDE."""
    model.eval()
    out = []
    with torch.no_grad():
        for ctx, _ in loader:
            ctx = ctx.to(DEVICE)
            samples = model.sample(ctx, n_samples=1).squeeze(0)
            out.append(samples.cpu().numpy())
    return np.concatenate(out, axis=0)


# ============================================================================
# 7. Метрики оцінки якості
# ============================================================================

def autocorrelation(x, max_lag=20):
    x = np.asarray(x) - np.mean(x)
    var = np.var(x) + 1e-12
    return np.array([np.mean(x[:len(x)-k] * x[k:]) / var if k > 0 else 1.0
                     for k in range(max_lag + 1)])


def arch_lm_test(returns, lags=10):
    """
    ARCH-LM тест Енгла на volatility clustering.
    H0: квадрати доходностей не корельовані (нема ARCH-ефекту).
    Велике LM, p<0.05 → відхиляємо H0 (є volatility clustering).
    """
    r = np.asarray(returns).flatten()
    if len(r) < lags + 10:
        return float('nan'), float('nan')
    r2 = r ** 2
    n = len(r2)
    y = r2[lags:]
    X = np.column_stack([r2[lags-i-1:n-i-1] for i in range(lags)])
    X = np.column_stack([np.ones(len(y)), X])
    beta, *_ = np.linalg.lstsq(X, y, rcond=None)
    y_pred = X @ beta
    ss_res = np.sum((y - y_pred) ** 2)
    ss_tot = np.sum((y - np.mean(y)) ** 2)
    R2 = 1 - ss_res / max(ss_tot, 1e-12)
    LM = len(y) * R2
    p_value = 1 - stats.chi2.cdf(LM, df=lags)
    return float(LM), float(p_value)


def leverage_effect(returns, lag=1):
    """
    Leverage effect: Corr[r_t, |r_{t+lag}|].
    Зазвичай від'ємна для фінансових рядів.
    """
    r = np.asarray(returns).flatten()
    if len(r) < lag + 2:
        return float('nan')
    r_t = r[:-lag]
    abs_r_next = np.abs(r[lag:])
    if np.std(r_t) < 1e-12 or np.std(abs_r_next) < 1e-12:
        return float('nan')
```

```
        return float(np.corrcoef(r_t, abs_r_next)[0, 1])


    def mmd_rbf(x, y, sigma=None, max_n=2000, seed=42):
        """
        Maximum Mean Discrepancy (MMD²) з RBF (Gaussian) kernel.
        Беззміщена оцінка.

        MMD²(P,Q) = E[k(x,x')] + E[k(y,y')] - 2·E[k(x,y)]
        де k(x,y) = exp(-||x-y||² / (2σ²)).

        Менше → ближчі розподіли. MMD² = 0 → розподіли ідентичні.
        σ обирається евристикою медіани попарних відстаней (за замовч.).
        Для tractability вибірки субсемплуються до max_n точок.
        """
        rng = np.random.RandomState(seed)
        x = np.asarray(x).flatten()
        y = np.asarray(y).flatten()
        if len(x) > max_n:
            x = x[rng.choice(len(x), max_n, replace=False)]
        if len(y) > max_n:
            y = y[rng.choice(len(y), max_n, replace=False)]
        n, m = len(x), len(y)
        if n < 2 or m < 2:
            return float('nan')

        xx_sq = (x[:, None] - x[None, :]) ** 2
        yy_sq = (y[:, None] - y[None, :]) ** 2
        xy_sq = (x[:, None] - y[None, :]) ** 2

        if sigma is None:
            all_dists = np.concatenate([xx_sq.ravel(), yy_sq.ravel(), xy_sq.ravel()])
            pos = all_dists[all_dists > 0]
            sigma = np.sqrt(np.median(pos) / 2.0) if len(pos) else 1.0
        gamma = 1.0 / (2 * sigma ** 2 + 1e-12)

        Kxx = np.exp(-gamma * xx_sq)
        Kyy = np.exp(-gamma * yy_sq)
        Kxy = np.exp(-gamma * xy_sq)
        np.fill_diagonal(Kxx, 0)
        np.fill_diagonal(Kyy, 0)

        mmd2 = (Kxx.sum() / (n * (n - 1))
                + Kyy.sum() / (m * (m - 1))
                - 2.0 * Kxy.sum() / (n * m))
        return float(max(mmd2, 0.0))


    def aggregational_gaussianity(returns, scales=(1, 5, 22, 66)):
        """
        Куртозис при агрегації на різних масштабах.
        Для нормальних — близький до 0 на всіх масштабах.
        Для реальних — стартовий високий, падає при агрегації.
        """
        r = np.asarray(returns).flatten()
        result = {}
        for scale in scales:
            if len(r) < scale * 10:
                result[scale] = float('nan')
                continue
            n_chunks = len(r) // scale
            agg = r[:n_chunks*scale].reshape(n_chunks, scale).sum(axis=1)
            result[scale] = float(stats.kurtosis(agg))
        return result


    def compute_metrics(real, gen, label=''):
        real_f = real.flatten()
        gen_f = gen.flatten()
        lm_real, _ = arch_lm_test(real_f, lags=5)
        lm_gen, _ = arch_lm_test(gen_f, lags=5)
        lev_real = leverage_effect(real_f, lag=1)
        lev_gen = leverage_effect(gen_f, lag=1)
        return {
            'label': label,
            'real_mean': float(np.mean(real_f)), 'gen_mean': float(np.mean(gen_f)),
            'real_std': float(np.std(real_f)), 'gen_std': float(np.std(gen_f)),
            'real_skew': float(stats.skew(real_f)), 'gen_skew': float(stats.skew(gen_f)),
            'real_kurt': float(stats.kurtosis(real_f)), 'gen_kurt': float(stats.kurtosis(gen_f)),
            'mean_err': abs(np.mean(real_f) - np.mean(gen_f)),
            'std_err':  abs(np.std(real_f) - np.std(gen_f)),
            'skew_err': abs(stats.skew(real_f) - stats.skew(gen_f)),
            'kurt_err': abs(stats.kurtosis(real_f) - stats.kurtosis(gen_f)),
            'ks_stat': float(ks_2samp(real_f, gen_f).statistic),
            'ks_p':    float(ks_2samp(real_f, gen_f).pvalue),
            'wasserstein': float(wasserstein_distance(real_f, gen_f)),
            'mmd': mmd_rbf(real_f, gen_f),
            'real_acf_sq_lag1': float(autocorrelation(real_f**2, 1)[1]),
            'gen_acf_sq_lag1': float(autocorrelation(gen_f**2, 1)[1]),
            'real_arch_lm': lm_real, 'gen_arch_lm': lm_gen,
            'arch_lm_err': abs(lm_real - lm_gen) if not (np.isnan(lm_real) or np.isnan(lm_gen)) else float('nan'),
            'real_leverage': lev_real, 'gen_leverage': lev_gen,
            'leverage_err': abs(lev_real - lev_gen) if not (np.isnan(lev_real) or np.isnan(lev_gen)) else float('nan'),
        }


    # ═══════════════════════════════════════════════════════════════
    # 8. Графіки — академічний стиль (чорно-білий з маркерами)
    # ═══════════════════════════════════════════════════════════════

    import matplotlib
    matplotlib.use('Agg')
    import matplotlib.pyplot as plt
    import matplotlib.dates as mdates

    plt.rcParams.update({
        'font.family': 'serif',
        'font.serif': ['Times New Roman', 'DejaVu Serif', 'Liberation Serif'],
        'font.size': 11,
        'axes.titlesize': 12,
        'axes.labelsize': 11,
        'xtick.labelsize': 10,
        'ytick.labelsize': 10,
        'legend.fontsize': 10,
        'legend.frameon': True,
        'legend.framealpha': 1.0,
        'legend.edgecolor': 'black',
        'legend.fancybox': False,
        'savefig.dpi': 300,
        'savefig.bbox': 'tight',
        'axes.grid': True,
        'grid.alpha': 0.25,
        'grid.linestyle': ':',
        'grid.linewidth': 0.5,
        'axes.linewidth': 0.8,
        'lines.linewidth': 1.0,
        'patch.linewidth': 0.5,
    })

    MODEL_STYLES = {
        'LSTM':         {'color': '#909090', 'marker': 'o', 'linestyle': '-',  'hatch': '///'},
        'Transformer':  {'color': '#909090', 'marker': 's', 'linestyle': '-', 'hatch': '\\\\\\'},
        'NeuralSDE-A':  {'color': '#909090', 'marker': '^', 'linestyle': '-.', 'hatch': '...'},
        'NeuralSDE-B':  {'color': '#909090', 'marker': 'D', 'linestyle': '-',  'hatch': 'xxx'},
        'NeuralSDE-GAN': {'color': '#404040', 'marker': 'P', 'linestyle': ':',  'hatch': '++'},
        'Heston (Euler)': {'color': '#909090', 'marker': '*', 'linestyle': '-', 'hatch': ''},
    }

    CRISES = [
        {'date': pd.Timestamp('2001-09-11'), 'label': '11.09.2001'},
        {'date': pd.Timestamp('2008-09-15'), 'label': 'Криза 2008'},
        {'date': pd.Timestamp('2020-03-23'), 'label': 'COVID-19'},
```

```
        {'date': pd.Timestamp('2022-02-24'),
'label': 'Війна 2022'},
    ]


def save_fig(fig, name):

fig.savefig(f'{FIGURES_DIR}/{name}.png', dpi=300)
    plt.close(fig)
    print(f"  Збережено:
{FIGURES_DIR}/{name}.png')


def _annotate_crises(ax, dates):
    if dates is None or len(dates) == 0:
        return
    date_min, date_max = dates[0],
dates[-1]
    if hasattr(date_min, 'to_pydatetime'):
        date_min =
date_min.to_pydatetime()
        date_max =
date_max.to_pydatetime()

    y_top = ax.get_ylim()[1]
    for crisis in CRISES:
        cd = crisis['date'].to_pydatetime()
        if date_min <= cd <= date_max:
            ax.axvline(cd, color='black',
linestyle='--', linewidth=0.6, alpha=0.5)
            ax.annotate(crisis['label'],
xy=(cd, y_top), xytext=(2, -2),
                        textcoords='offset
points',
                        fontsize=8, ha='left',
va='top', alpha=0.8,
                        rotation=90)


def plot_assets_timeline(asset_data,
name):
    """Часова шкала покриття
активів."""
    fig, ax = plt.subplots(figsize=(11,
3.5))
    tickers = list(asset_data.keys())
    end = pd.Timestamp('2024-12-31')

    for i, ticker in enumerate(tickers):
        first =
asset_data[ticker]['first_date']
        if hasattr(first, 'to_pydatetime'):
            first = first.to_pydatetime()
        n = asset_data[ticker]['n_points']
        ax.barh(i, (end - first).days,
left=first,
                height=0.5, color='lightgray',
edgecolor='black', linewidth=0.6)
        # Підпис кількості точок
        ax.text(end +
pd.Timedelta(days=60), i, f'{n} точок',
                va='center', fontsize=10)

    ax.set_yticks(range(len(tickers)))
    ax.set_yticklabels([f'{t}
({TICKER_LABELS.get(t, t)})' for t in tickers])
    ax.set_xlabel('Рік')
    ax.set_title('Часова шкала покриття
активів історичними даними')
    ax.xaxis.set_major_locator(mdates.YearLocator(5))

    ax.xaxis.set_major_formatter(mdates.DateFormatter('
%Y'))
    ax.invert_yaxis()
    fig.tight_layout()
    save_fig(fig, name)


def plot_stylized_facts(returns, dates,
name, ticker_label=''):
    fig, axes = plt.subplots(2, 2,
figsize=(13, 8))
    ret_dates = dates[1:] if dates is not
None and len(dates) > len(returns) else dates

    ax = axes[0, 0]
    ax.plot(ret_dates, returns, lw=0.4,
color='black')
    ax.set_title('Log-доходності у часі')
    ax.set_xlabel('Рік');
ax.set_ylabel('Доходність')

    ax.xaxis.set_major_locator(mdates.YearLocator(5))

    ax.xaxis.set_major_formatter(mdates.DateFormatter('
%Y'))
    _annotate_crises(ax, ret_dates)

    ax = axes[0, 1]
    bins =
np.linspace(np.percentile(returns, 0.1),
np.percentile(returns, 99.9), 80)
    ax.hist(returns, bins=bins,
density=True, color='lightgray',
            edgecolor='black',
linewidth=0.4, label='Емпіричний розподіл')
    mu, sigma = np.mean(returns),
np.std(returns)
    x = np.linspace(bins[0], bins[-1],
200)
    ax.plot(x, stats.norm.pdf(x, mu,
sigma), color='black', linestyle='--',
            lw=1.2, label=r'Нормальний
розподіл $N(\mu,\sigma^2)$')
    ax.set_yscale('log')
    ax.set_title('Розподіл доходностей
(важкі хвости)')
    ax.set_xlabel('Доходність');
ax.set_ylabel('Щільність (log)')
    ax.legend(loc='lower center')

    ax = axes[1, 0]
    max_lag = 30
    lags = np.arange(max_lag + 1)
    ax.bar(lags - 0.2,
autocorrelation(returns, max_lag), width=0.4,
           color='white', edgecolor='black',
linewidth=0.6, label=r'ACF $r_t$')
    ax.bar(lags + 0.2,
autocorrelation(np.abs(returns), max_lag), width=0.4,
           color='gray', edgecolor='black',
linewidth=0.6, label=r'ACF $|r_t|$')
    ax.axhline(0, color='black', lw=0.5)
    ax.set_title('Автокореляція
доходностей та їх модулів')
    ax.set_xlabel('Лаг');
ax.set_ylabel('ACF')
    ax.legend()
    ax = axes[1, 1]
    window = 20
    s = np.array([np.std(returns[max(0, i-
window):i+1]) for i in range(len(returns))])
    ax.plot(ret_dates, s, lw=0.6,
color='black')
    ax.set_title(f'Ковзна волатильність
({window} днів)')
    ax.set_xlabel('Рік');
ax.set_ylabel(r'$\sigma$')

    ax.xaxis.set_major_locator(mdates.YearLocator(5))

    ax.xaxis.set_major_formatter(mdates.DateFormatter('
%Y'))
    _annotate_crises(ax, ret_dates)

    title = f'Стилізовані факти:
{ticker_label}' if ticker_label else 'Стилізовані факти
фінансових часових рядів'
    fig.suptitle(title, fontsize=13)
    fig.tight_layout()
    save_fig(fig, name)

    fig_extended =
_build_extended_facts_fig(returns, ticker_label)
    save_fig(fig_extended, name +
'_extended')


def _build_extended_facts_fig(returns,
ticker_label=''):
    """
    Розширений набір стилізованих
фактів (Cont, 2001):
      1. Long memory у волатильності
(ACF |r_t| на 100 лагах)
      2. Leverage effect (Corr[r_t,
|r_{t+k}|])
      3. Aggregational Gaussianity
(куртозис vs масштаб)
      4. ARCH-LM тест (наявність
volatility clustering)
    """
    fig, axes = plt.subplots(2, 2,
figsize=(13, 9))

    # 1. Long memory
    ax = axes[0, 0]
    max_lag = 100
    lags = np.arange(max_lag + 1)
    ax.plot(lags, autocorrelation(returns,
max_lag),
            color='black', linestyle='--',
linewidth=0.8,
            label=r'ACF $r_t$ (швидке
загасання)')
    ax.plot(lags,
autocorrelation(np.abs(returns), max_lag),
            color='black', linewidth=1.2,
            label=r'ACF $|r_t|$ (повільне
загасання — long memory)')
    ax.axhline(0, color='black', lw=0.5)
    ax.set_xlabel('Лаг');
ax.set_ylabel('ACF')
    ax.set_title(r'Довга пам\'ять
волатильності (ACF на 100 лагах)')
    ax.legend()
```

```
        # 2. Leverage effect
        ax = axes[0, 1]
        lags_lev = np.arange(1, 21)
        lev = [leverage_effect(returns,
lag=int(l)) for l in lags_lev]
        ax.bar(lags_lev, lev, color='gray',
edgecolor='black', linewidth=0.5)
        ax.axhline(0, color='black', lw=0.6)
        ax.set_xlabel('Лаг $k$')
        ax.set_ylabel(r'$\mathrm{Corr}[r_t,
|r_{t+k}|]$')
        ax.set_title(r'Leverage effect (для
фінансових рядів зазвичай $<0$)')

        # 3. Aggregational Gaussianity
        ax = axes[1, 0]
        scales = [1, 2, 5, 10, 22, 44, 66, 132]
        aggr =
aggregational_gaussianity(returns, scales)
        sc = list(aggr.keys())
        kurts = list(aggr.values())
        ax.semilogx(sc, kurts, marker='o',
color='black', linewidth=1.2,
                    markersize=6,
markerfacecolor='white', markeredgewidth=1.2,
                    label='Excess kurtosis
емпіричний')
        ax.axhline(0, color='black',
linestyle='--', lw=0.8,
                   label='Нормальний розподіл
(kurt = 0)')
        ax.set_xlabel('Масштаб агрегації,
днів')
        ax.set_ylabel('Excess kurtosis')
        ax.set_title('Aggregational
Gaussianity')
        ax.legend()

        # 4. ARCH-LM test
        ax = axes[1, 1]
        p_range = [1, 5, 10, 20, 50]
        lm_stats, p_values = [], []
        for p in p_range:
            lm, pv = arch_lm_test(returns,
lags=p)
            lm_stats.append(lm)
            p_values.append(pv)
        ax.bar(range(len(p_range)), lm_stats,
color='gray',
               edgecolor='black',
linewidth=0.5, label='LM-статистика')
        crits = [stats.chi2.ppf(0.95, df=p) for
p in p_range]
        ax.plot(range(len(p_range)), crits,
color='black', linestyle='--',
                marker='s', linewidth=1.0,
markersize=5,
                label=r'Критичне значення
$\chi^2_{0.95}(p)$')
        ax.set_xticks(range(len(p_range)))
        ax.set_xticklabels([f'p={p}' for p in
p_range])
        ax.set_ylabel('LM-статистика')
        p1_str = f'{p_values[0]:.2e}' if not
np.isnan(p_values[0]) else 'nan'
        pn_str = f'{p_values[-1]:.2e}' if not
np.isnan(p_values[-1]) else 'nan'
        ax.set_title(f'ARCH-LM тест
Енгла\n(p={p_range[0]}: pval={p1_str};
p={p_range[-1]}: pval={pn_str})')
        ax.legend()

        suffix = f': {ticker_label}' if
ticker_label else ''
        fig.suptitle(f'Розширені стилізовані
факти{suffix}', fontsize=13)
        fig.tight_layout()
        return fig


    def plot_distributions(real, gen_dict,
name, ticker_label=''):
        """
        Розподіл log-доходностей на 2
панелях: лінійна і log-y.
        Хвостові області (нижні/верхні
5%) підсвічені — там модель і має
        демонструвати свою точність.
        """
        fig, axes = plt.subplots(1, 2,
figsize=(15, 5.5))
        bins = np.linspace(np.percentile(real,
0.1),
                           np.percentile(real, 99.9),
80)
        centers = 0.5 * (bins[:-1] + bins[1:])
        real_dens, _ = np.histogram(real,
bins=bins, density=True)
        q05 = float(np.percentile(real, 5))
        q95 = float(np.percentile(real, 95))

        for panel_idx, (ax, yscale) in
enumerate(zip(axes, ['linear', 'log'])):
            ax.hist(real, bins=bins,
density=True, color='lightgray',
                    edgecolor='black',
linewidth=0.4,
                    label='Реальні дані',
zorder=1)
            for lbl, arr in gen_dict.items():
                style =
MODEL_STYLES.get(lbl,
                                 {'color': 'black',
'linestyle': '-',
                                  'marker': 'o'})
                arr_inrange = arr[(arr >=
bins[0]) & (arr <= bins[-1])]
                counts, _ =
np.histogram(arr_inrange, bins=bins, density=True)
                ax.plot(centers, counts,
color=style['color'],
                        linestyle=style['linestyle'],
linewidth=1.4,
                        marker=style.get('marker',
'o'), markevery=6,
                        markersize=5,
markerfacecolor='white',
                        markeredgewidth=1.0,
                        label=lbl, zorder=3)

            # Підсвічуємо хвостові області
            ax.axvspan(bins[0], q05,
color='red', alpha=0.06, zorder=0)
            ax.axvspan(q95, bins[-1],
color='red', alpha=0.06, zorder=0)

            if yscale == 'log':
                pos = real_dens[real_dens > 0]
                ymin = max(pos.min() * 0.3, 1e-
3) if len(pos) else 1e-3
                ax.set_yscale('log')
                ax.set_ylim(bottom=ymin)
                ax.set_title('Лог-шкала по y
(видно хвости)')
                ax.set_ylabel('Щільність (log)')
                ax.text(q05, ymin * 3, ' лівий\n
хвіст',
                        fontsize=8, va='bottom',
color='darkred', alpha=0.7)
                ax.text(q95, ymin * 3,
'правий\nхвіст ',
                        fontsize=8, va='bottom',
ha='right',
                        color='darkred', alpha=0.7)
            else:
                ax.set_title('Лінійна шкала по
y')
                ax.set_ylabel('Щільність')
                # Тільки цілі значення на осі
щільності в лінійній панелі
                from matplotlib.ticker import
MaxNLocator

ax.yaxis.set_major_locator(MaxNLocator(integer=Tru
e, prune=None))
            ax.set_xlabel('Log-доходність')
            ax.legend(loc='upper right',
fontsize=9)

        title = (f'Розподіл log-доходностей:
{ticker_label}'
                 if ticker_label else
                 'Розподіл log-доходностей:
реальні та згенеровані')
        fig.suptitle(title, fontsize=13)
        fig.tight_layout()
        save_fig(fig, name)


    def plot_qq(real, gen_dict, name,
ticker_label=''):
        """
        Q-Q діаграма: всі моделі поверх
однієї діагоналі.

        Хвостові квантилі (q < 5% і q >
95%) виділяються рожевим фоном.
        Сіра смуга навколо діагоналі —
зона ±10% (прийнятна точність).
        Точки за межами цієї смуги —
модель помиляється на >10% від
        масштабу real.
        """
        fig, ax = plt.subplots(figsize=(9, 8))
        qs = np.linspace(0.005, 0.995, 200)
        real_q = np.quantile(real, qs)

        lo, hi = real_q.min(), real_q.max()
        # Динамічно розширюємо межі під
найбільший gen_q
        for arr in gen_dict.values():
            gen_q = np.quantile(arr, qs)
            lo = min(lo, gen_q.min())
            hi = max(hi, gen_q.max())
        pad = (hi - lo) * 0.05
        lo, hi = lo - pad, hi + pad

        # Зона ±10% від масштабу
        diag = np.linspace(lo, hi, 200)
```

```
        band = 0.10 * (real_q.max() - real_q.min())
        ax.fill_between(diag, diag - band, diag + band,
                        color='lightgray', alpha=0.4, zorder=0,
                        label='Зона ±10% (прийнятна точність)')

        # Підсвічуємо хвостові області (по x)
        q05_real = float(np.quantile(real, 0.05))
        q95_real = float(np.quantile(real, 0.95))
        ax.axvspan(lo, q05_real, color='red', alpha=0.06, zorder=0)
        ax.axvspan(q95_real, hi, color='red', alpha=0.06, zorder=0)

        # Діагональ y = x
        ax.plot([lo, hi], [lo, hi], color='black', linestyle='--', lw=1.0,
                zorder=1, label='y = x (ідеальне співпадіння)')

        # Точки кожної моделі
        for lbl, arr in gen_dict.items():
            gen_q = np.quantile(arr, qs)
            style = MODEL_STYLES.get(lbl,
                                     {'color': 'black', 'marker': 'o'})
            ax.scatter(real_q, gen_q, s=22, color=style['color'],
                       marker=style['marker'], edgecolor='black',
                       linewidth=0.4, alpha=0.85, label=lbl, zorder=3)

        # Підписати хвостові зони
        yspan = hi - lo
        ax.text(q05_real, lo + 0.04*yspan,
                ' лівий хвіст\n (q<5%)', fontsize=9,
                va='bottom', ha='left', color='darkred', alpha=0.85)
        ax.text(q95_real, lo + 0.04*yspan,
                'правий хвіст \n(q>95%) ', fontsize=9,
                va='bottom', ha='right', color='darkred', alpha=0.85)

        ax.set_xlim(lo, hi)
        ax.set_ylim(lo, hi)
        ax.set_aspect('equal', adjustable='box')
        ax.set_xlabel('Квантилі реальних даних')
        ax.set_ylabel('Квантилі моделі')
        title = (f'Q-Q діаграма (всі моделі на одній панелі): {ticker_label}'
                 if ticker_label else
                 'Q-Q діаграма: відтворення квантилів розподілу')
        ax.set_title(title)
        ax.legend(loc='upper left', fontsize=9)
        save_fig(fig, name)


    def plot_acf(real, gen_dict, name, max_lag=15, ticker_label=''):
        fig, axes = plt.subplots(1, 2, figsize=(14, 5))
        lags = np.arange(max_lag + 1)
        bw = 0.8 / (1 + len(gen_dict))

        for ax_idx, (data_name, transform) in enumerate([
            ('log-доходностей', lambda x: x),
            ('квадратів log-доходностей', lambda x: x**2),
        ]):
            ax = axes[ax_idx]
            ax.bar(lags - 0.4 + bw/2, autocorrelation(transform(real), max_lag),
                   width=bw, color='white', edgecolor='black', linewidth=0.6,
                   label='Реальні', hatch='')
            for i, (lbl, arr) in enumerate(gen_dict.items()):
                style = MODEL_STYLES.get(lbl, {'color': 'gray', 'hatch': ''})
                ax.bar(lags - 0.4 + bw*(i+1.5), autocorrelation(transform(arr), max_lag),
                       width=bw, color=style['color'], edgecolor='black',
                       linewidth=0.4, label=lbl, hatch=style.get('hatch', ''))
            ax.axhline(0, color='black', lw=0.5)
            suffix = f' ({ticker_label})' if ticker_label else ''
            ax.set_title(f'ACF {data_name}{suffix}')
            ax.set_xlabel('Лаг'); ax.set_ylabel('ACF')
            ax.legend(loc='upper right')

        fig.tight_layout()
        save_fig(fig, name)


    def plot_training(histories, name):
        """
        Криві навчання з акцентом на train/val gap.

        Y-вісь масштабується по 5-95 перцентилях візуальних значень
        (ігноруючи перші 1-2 епохи з аномально високим loss), щоб
        було видно різницю між кривими на пізніх епохах.
        Між train і val малюється закрашена область — gap.
        Епохи на X — лише цілі.
        """
        from matplotlib.ticker import MaxNLocator

        n = len(histories)
        if n <= 2:
            nrows, ncols = 1, n
        else:
            nrows, ncols = 2, 2
        fig, axes = plt.subplots(nrows, ncols,
                                 figsize=(6.5*ncols, 4.8*nrows),
                                 squeeze=False)
        axes_flat = axes.flatten()
        for ax, (lbl, h) in zip(axes_flat, histories.items()):
            style = MODEL_STYLES.get(lbl, {'color': 'black', 'marker': 'o'})
            ep = np.arange(1, len(h['train']) + 1)
            train = np.array(h['train'])
            val = np.array(h['val'])

            # Закрашуємо gap між train і val
            ax.fill_between(ep, train, val, color='lightgray',
                            alpha=0.5, zorder=1,
                            label='|Train − Val| gap')

            ax.plot(ep, train, marker=style['marker'], linestyle='-',
                    color='black', label='Train', lw=1.4, markersize=4.5,
                    markerfacecolor='white', markeredgewidth=1.0,
                    zorder=3)
            ax.plot(ep, val, marker=style['marker'], linestyle='--',
                    color='#606060', label='Validation', lw=1.4, markersize=4.5,
                    markerfacecolor='#606060', markeredgewidth=1.0,
                    zorder=2)

            # Адаптивний Y-діапазон: ігноруємо перші 1-2 епохи (часто
            # мають аномально високий loss, який «з'їдає» масштаб)
            n_total = len(train)
            n_skip = min(2, max(1, n_total // 20))
            visible = np.concatenate([train[n_skip:], val[n_skip:]])
            if len(visible) > 0:
                y_lo, y_hi = float(visible.min()), float(visible.max())
                margin = (y_hi - y_lo) * 0.15 + 1e-6
                ax.set_ylim(y_lo - margin, y_hi + margin)

            # Кінцевий gap у легенді — для звітності
            final_gap = float(abs(val[-1] - train[-1]))
            ax.text(0.98, 0.02, f'final |Δ| = {final_gap:.4f}',
                    transform=ax.transAxes, fontsize=8,
                    ha='right', va='bottom',
                    bbox=dict(boxstyle='round', facecolor='white',
                              edgecolor='gray', alpha=0.85))

            ax.set_xlabel('Епоха')
            ax.set_ylabel('Loss (NLL)')
            ax.set_title(f'Крива навчання: {lbl}')
            ax.legend(loc='upper right', fontsize=9)
```

```
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
        ax.grid(True, alpha=0.3, linestyle=':')

    for ax in axes_flat[len(histories):]:
        ax.set_visible(False)
    fig.tight_layout()
    save_fig(fig, name)


def plot_metrics_bars(metrics_list, name, ticker_label='', show_error_bars=False):
    """
    Барчарт метрик. Якщо show_error_bars=True, очікує у словнику метрик
    додаткові ключі '<key>_std' з відхиленнями (отриманими усередненням
    по сідах).
    """
    fig, ax = plt.subplots(figsize=(12, 5.5))
    keys = ['mean_err', 'std_err', 'skew_err', 'kurt_err', 'wasserstein', 'ks_stat']
    labels = [r'$|\Delta\mu|$', r'$|\Delta\sigma|$', r'$|\Delta\mathrm{skew}|$',
              r'$|\Delta\mathrm{kurt}|$', 'Wasserstein', 'KS']
    x = np.arange(len(keys))
    w = 0.8 / len(metrics_list)
    for i, m in enumerate(metrics_list):
        style = MODEL_STYLES.get(m['label'], {'color': 'gray', 'hatch': ''})
        vals = [m[k] for k in keys]
        kwargs = dict(color=style['color'], edgecolor='black', linewidth=0.5,
                      hatch=style.get('hatch', ''), label=m['label'])
        if show_error_bars:
            errs = [m.get(f'{k}_std', 0.0) or 0.0 for k in keys]
            kwargs.update(yerr=errs, capsize=2.5,
                          error_kw={'lw': 0.7, 'capthick': 0.7,
                                    'ecolor': 'black'})
        ax.bar(x + (i - len(metrics_list)/2 + 0.5)*w, vals, w, **kwargs)
    ax.set_xticks(x); ax.set_xticklabels(labels)
    ax.set_ylabel('Значення метрики (менше — краще)')
    suffix = ' (mean ± std по сідах)' if show_error_bars else ''
    title = f'Порівняльний аналіз якості моделей: {ticker_label}{suffix}' if ticker_label else f'Порівняльний аналіз якості моделей{suffix}'
    ax.set_title(title)
    ax.set_yscale('log')
    ax.legend(loc='upper left')
    save_fig(fig, name)


def plot_metrics_heatmap(metrics_per_ticker, metric_key, name,
                         metric_label='', cmap='Greys'):
    """
    Heatmap метрик: рядки — моделі, стовпці — активи.

    metrics_per_ticker: dict {ticker: [metrics_dict_per_model]}
    metric_key: 'wasserstein', 'ks_stat', 'kurt_err', тощо
    """
    tickers = list(metrics_per_ticker.keys())
    models = [m['label'] for m in metrics_per_ticker[tickers[0]]]

    matrix = np.full((len(models), len(tickers)), np.nan)
    for j, ticker in enumerate(tickers):
        for i, m in enumerate(metrics_per_ticker[ticker]):
            v = m[metric_key]
            matrix[i, j] = v if v is not None else np.nan

    fig, ax = plt.subplots(figsize=(1.5 + 1.5*len(tickers), 1.5 + 0.6*len(models)))
    im = ax.imshow(matrix, aspect='auto', cmap=cmap)

    ax.set_xticks(range(len(tickers)))
    ax.set_xticklabels([TICKER_LABELS.get(t, t) for t in tickers])
    ax.set_yticks(range(len(models)))
    ax.set_yticklabels(models)

    # Підписи значень у комірках (NaN-стійкі)
    mat_max = np.nanmax(matrix) if np.any(np.isfinite(matrix)) else 1.0
    for i in range(len(models)):
        for j in range(len(tickers)):
            val = matrix[i, j]
            if np.isnan(val):
                ax.text(j, i, '—', ha='center', va='center',
                        color='black', fontsize=10)
                continue
            text_color = 'white' if val > mat_max * 0.5 else 'black'
            ax.text(j, i, f'{val:.4f}', ha='center', va='center',
                    color=text_color, fontsize=10)

    ax.set_title(f'{metric_label or metric_key}: модель × актив')
    cbar = plt.colorbar(im, ax=ax, fraction=0.046, pad=0.04)
    cbar.ax.set_ylabel(metric_label or metric_key, rotation=-90, va="bottom")
    fig.tight_layout()
    save_fig(fig, name)


def plot_speed(speeds, name):
    fig, ax = plt.subplots(figsize=(9, 5))
    methods = list(speeds.keys())
    times = list(speeds.values())
    for i, (m, t) in enumerate(zip(methods, times)):
        style = MODEL_STYLES.get(m, {'color': 'gray', 'hatch': ''})
        ax.bar(i, t, color=style['color'], edgecolor='black', linewidth=0.6,
               hatch=style.get('hatch', ''))
        ax.text(i, t * 1.02, f'{t:.2f} с', ha='center', va='bottom', fontsize=10)
    ax.set_xticks(range(len(methods)))
    ax.set_xticklabels(methods, rotation=15, ha='right')
    ax.set_ylabel('Час, с')
    ax.set_title('Швидкодія генерації траєкторій')
    fig.tight_layout()
    save_fig(fig, name)


def plot_speed_scaling(df, name):
    fig, ax = plt.subplots(figsize=(9, 5.5))
    for method in df['method'].unique():
        sub = df[df['method'] == method].sort_values('n_steps')
        style = MODEL_STYLES.get(method,
                                 {'color': 'gray', 'marker': 'o', 'linestyle': '-'})
        ax.plot(sub['n_steps'], sub['time'],
                marker=style['marker'], linestyle=style['linestyle'],
                color=style['color'], lw=1.2, markersize=6,
                markerfacecolor='white', markeredgewidth=1.2,
                label=method)
    ax.set_xscale('log')
    ax.set_yscale('log')
    ax.set_xlabel('Кількість часових кроків (log-шкала)')
    ax.set_ylabel('Час генерації, с (log-шкала)')
    ax.set_title('Масштабування швидкодії від кількості часових кроків')
    ax.legend(loc='upper left')
    ax.grid(True, which='both', alpha=0.25, linestyle=':')
    save_fig(fig, name)
```

 # neural_sde.py

```
"""
Neural SDE сурогат для дипломної роботи.
Окремий модуль, який інтегрується у diploma_final.py.

Реалізовано ДВА ВАРІАНТИ:

Варіант A — NeuralSDESurrogateA (без encoder):
    dY_t = mu_theta(t, Y_t)·dt + sigma_theta(t, Y_t)·dW_t
    Контекст не використовується явно. Початковий стан Y_0 = 0.
    Модель вчить загальну стохастичну динаміку ринку.

Варіант B — NeuralSDESurrogateB (з encoder):
    dY_t = mu_theta(t, Y_t, h)·dt + sigma_theta(t, Y_t, h)·dW_t
```

```
    де h = Encoder(контекст 60 днів) — кодування поточного стану ринку.
    Дрейф і дифузія залежать від історичного контексту.

    Loss: спрощений NLL — на кожному кроці інтегрування рахуємо
    ймовірність переходу як гаусіан з параметрами (mu·dt, sigma²·dt).
    """

    import math
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    import torchsde


    # ============================================================
    # Спільні компоненти: time embedding
    # ============================================================

    def time_features(t, batch_size, device, embed_dim=8):
        """Синусоїдальне кодування часу для подачі в нейромережу."""
        freqs = torch.arange(embed_dim // 2, device=device).float()
        freqs = 2 ** freqs * math.pi
        t_scaled = float(t) * freqs
        emb = torch.cat([torch.sin(t_scaled), torch.cos(t_scaled)])
        return emb.unsqueeze(0).expand(batch_size, -1)


    # ============================================================
    # ВАРІАНТ A: без encoder
    # ============================================================

    class DriftNetA(nn.Module):
        """Дрейф mu_theta(t, Y) — без контексту."""
        def __init__(self, state_size=1, hidden=64, time_embed_dim=8):
            super().__init__()
            self.time_embed_dim = time_embed_dim
            self.net = nn.Sequential(
                nn.Linear(state_size + time_embed_dim, hidden),
                nn.Tanh(),
                nn.Linear(hidden, hidden),
                nn.Tanh(),
                nn.Linear(hidden, state_size),
            )

        def forward(self, t, y):
            t_emb = time_features(t, y.size(0), y.device, self.time_embed_dim)
            x = torch.cat([y, t_emb], dim=-1)
            return self.net(x)


    class DiffusionNetA(nn.Module):
        """Дифузія sigma_theta(t, Y) — без контексту."""
        def __init__(self, state_size=1, hidden=64, time_embed_dim=8, sigma_max=0.05):
            super().__init__()
            self.time_embed_dim = time_embed_dim
            self.sigma_max = sigma_max
            self.net = nn.Sequential(
                nn.Linear(state_size + time_embed_dim, hidden),
                nn.Tanh(),
                nn.Linear(hidden, hidden),
                nn.Tanh(),
                nn.Linear(hidden, state_size),
            )

        def forward(self, t, y):
            t_emb = time_features(t, y.size(0), y.device, self.time_embed_dim)
            x = torch.cat([y, t_emb], dim=-1)
            return torch.sigmoid(self.net(x)) * self.sigma_max


    class _SDEModuleA(torchsde.SDEIto):
        """torchsde-сумісний клас без контексту."""
        def __init__(self, hidden=64, sigma_max=0.05):
            super().__init__(noise_type='diagonal')
            self.drift = DriftNetA(state_size=1, hidden=hidden)
            self.diffusion = DiffusionNetA(state_size=1, hidden=hidden, sigma_max=sigma_max)

        def f(self, t, y):
            return self.drift(t, y)

        def g(self, t, y):
            return self.diffusion(t, y)


    class NeuralSDESurrogateA(nn.Module):
        """
        Варіант A: Neural SDE без використання контексту.

        Інтерфейс сумісний з LSTMSurrogate/TransformerSurrogate:
            - sample(context, n_samples=1) → (n_samples, batch, horizon)
        """
        def __init__(self, horizon=20, hidden=64, sigma_max=0.05):
            super().__init__()
            self.horizon = horizon
            self.sde = _SDEModuleA(hidden=hidden, sigma_max=sigma_max)

        def integrate(self, y0, n_steps, dt=1.0):
            ts = torch.linspace(0, n_steps * dt, n_steps + 1, device=y0.device)
            ys = torchsde.sdeint(self.sde, y0, ts, method='euler', dt=dt)
            return ys

        def sample(self, context, n_samples=1):
            batch_size = context.size(0)
            device = context.device

            outputs = []
            for _ in range(n_samples):
                y0 = torch.zeros(batch_size, 1, device=device)
                ys = self.integrate(y0, self.horizon)
                # Беремо різниці — це і є log-доходності, а не сама log-ціна
                diffs = ys[1:] - ys[:-1]  # (horizon, batch, 1)
                traj = diffs.squeeze(-1).t()  # (batch, horizon)
                outputs.append(traj)

            return torch.stack(outputs, dim=0)


    # ============================================================
    # ВАРІАНТ B: з encoder
    # ============================================================
    # Хитрість: torchsde.sdeint вимагає, щоб f(t,y) і g(t,y) приймали
    # тільки 2 аргументи. Тому контекст h ми зберігаємо як стан _SDEModuleB
    # і встановлюємо його ПЕРЕД викликом sdeint через метод set_context().

    class ContextEncoder(nn.Module):
        """LSTM-енкодер контексту, видає вектор стану ринку h."""
        def __init__(self, hidden=32, num_layers=1, h_dim=16):
            super().__init__()
            self.lstm = nn.LSTM(1, hidden, num_layers=num_layers, batch_first=True)
            self.proj = nn.Linear(hidden, h_dim)

        def forward(self, context):
            x = context.unsqueeze(-1)  # (batch, T, 1)
            _, (h_state, _) = self.lstm(x)
            return self.proj(h_state[-1])  # (batch, h_dim)


    class DriftNetB(nn.Module):
        """Дрейф mu_theta(t, Y, h) — з контекстом."""
        def __init__(self, state_size=1, hidden=64, time_embed_dim=8, h_dim=16):
            super().__init__()
            self.time_embed_dim = time_embed_dim
            self.net = nn.Sequential(
```

```
            nn.Linear(state_size +
time_embed_dim + h_dim, hidden),
            nn.Tanh(),
            nn.Linear(hidden, hidden),
            nn.Tanh(),
            nn.Linear(hidden, state_size),
        )

    def forward(self, t, y, h):
        t_emb = time_features(t, y.size(0),
y.device, self.time_embed_dim)
        x = torch.cat([y, t_emb, h], dim=-
1)
        return self.net(x)


class DiffusionNetB(nn.Module):
    """Дифузія sigma_theta(t, Y, h) — з
контекстом."""
    def __init__(self, state_size=1,
hidden=64, time_embed_dim=8, h_dim=16,
sigma_max=0.05):
        super().__init__()
        self.time_embed_dim =
time_embed_dim
        self.sigma_max = sigma_max
        self.net = nn.Sequential(
            nn.Linear(state_size +
time_embed_dim + h_dim, hidden),
            nn.Tanh(),
            nn.Linear(hidden, hidden),
            nn.Tanh(),
            nn.Linear(hidden, state_size),
        )

    def forward(self, t, y, h):
        t_emb = time_features(t, y.size(0),
y.device, self.time_embed_dim)
        x = torch.cat([y, t_emb, h], dim=-
1)
        return torch.sigmoid(self.net(x)) *
self.sigma_max


class _SDEModuleB(torchsde.SDEIto):
    """
    torchsde-сумісний клас з
контекстом.

    Контекст h зберігається в self._h і
встановлюється через set_context()
    перед викликом sdeint. Це обхідне
рішення обмеження torchsde, що
    f(t,y) приймає тільки 2 аргументи.
    """
    def __init__(self, hidden=64,
sigma_max=0.05, h_dim=16):

super().__init__(noise_type='diagonal')
        self.drift = DriftNetB(state_size=1,
hidden=hidden, h_dim=h_dim)
        self.diffusion =
DiffusionNetB(state_size=1, hidden=hidden,
h_dim=h_dim, sigma_max=sigma_max)
        self._h = None

    def set_context(self, h):
        self._h = h

    def f(self, t, y):
        if self._h is None:
            raise RuntimeError('Контекст
не встановлено. Викличіть set_context(h).')
        return self.drift(t, y, self._h)

    def g(self, t, y):
        if self._h is None:
            raise RuntimeError('Контекст
не встановлено. Викличіть set_context(h).')
        return self.diffusion(t, y, self._h)


class
NeuralSDESurrogateB(nn.Module):
    """
    Варіант B: Neural SDE з encoder
контексту.

    Архітектура:
        h = Encoder(context)
        dY = mu_theta(t, Y, h)·dt +
sigma_theta(t, Y, h)·dW
    """
    def __init__(self, horizon=20,
hidden=64, sigma_max=0.05,
                 encoder_hidden=32,
h_dim=16):
        super().__init__()
        self.horizon = horizon
        self.h_dim = h_dim
        self.encoder =
ContextEncoder(hidden=encoder_hidden,
h_dim=h_dim)
        self.sde =
_SDEModuleB(hidden=hidden,
sigma_max=sigma_max, h_dim=h_dim)

    def integrate(self, y0, h, n_steps,
dt=1.0):
        self.sde.set_context(h)
        ts = torch.linspace(0, n_steps * dt,
n_steps + 1, device=y0.device)
        ys = torchsde.sdeint(self.sde, y0,
ts, method='euler', dt=dt)
        return ys

    def encode(self, context):
        return self.encoder(context)

    def sample(self, context,
n_samples=1):
        batch_size = context.size(0)
        device = context.device
        h = self.encode(context)  # (batch,
h_dim)

        outputs = []
        for _ in range(n_samples):
            y0 = torch.zeros(batch_size, 1,
device=device)
            ys = self.integrate(y0, h,
self.horizon)
            # Беремо різниці — це і є log-
доходності, а не сама log-ціна
            diffs = ys[1:] - ys[:-1]  #
(horizon, batch, 1)
            traj = diffs.squeeze(-1).t()  #
(batch, horizon)
            outputs.append(traj)
        return torch.stack(outputs, dim=0)


#
====================================
==========================
# Loss-функції
#
====================================
==========================

def neural_sde_step_nll_A(model,
target, dt=1.0):
    """
    Step-NLL для Варіанту A (без
контексту).

    На кожному кроці t у
тренувальному таргеті:
        y_{t+1} ~ N(y_t + mu(t,y_t)·dt,
sigma(t,y_t)²·dt)
    де y_t = cumsum(returns) —
кумулятивна log-ціна.
    """
    batch_size, horizon = target.shape
    device = target.device

    cumret = torch.cumsum(target,
dim=1)
    Y_prev =
torch.cat([torch.zeros(batch_size, 1, device=device),
cumret[:, :-1]], dim=1)

    nll_total = 0.0
    for t_idx in range(horizon):
        t_val = float(t_idx) * dt
        Y_t = Y_prev[:, t_idx:t_idx+1]
        mu_t = model.sde.drift(t_val, Y_t)
        sigma_t =
model.sde.diffusion(t_val, Y_t)
        r_t = target[:, t_idx:t_idx+1]

        mean = mu_t * dt
        var = sigma_t.pow(2) * dt + 1e-8

        nll = 0.5 * ((r_t - mean).pow(2) /
var + torch.log(2 * math.pi * var))
        nll_total = nll_total + nll.mean()

    return nll_total / horizon


def neural_sde_step_nll_B(model,
context, target, dt=1.0):
    """
    Step-NLL для Варіанту B (з
контекстом).
    Як A, але mu і sigma залежать від h
= encoder(context).
    """
    batch_size, horizon = target.shape
    device = target.device

    h = model.encode(context)  # (batch,
h_dim)

    cumret = torch.cumsum(target,
dim=1)
```

```
        Y_prev =
torch.cat([torch.zeros(batch_size, 1, device=device),
cumret[:, :-1]], dim=1)

        nll_total = 0.0
        for t_idx in range(horizon):
            t_val = float(t_idx) * dt
            Y_t = Y_prev[:, t_idx:t_idx+1]
            mu_t = model.sde.drift(t_val, Y_t,
h)
            sigma_t =
model.sde.diffusion(t_val, Y_t, h)
            r_t = target[:, t_idx:t_idx+1]

            mean = mu_t * dt
            var = sigma_t.pow(2) * dt + 1e-8

            nll = 0.5 * ((r_t - mean).pow(2) /
var + torch.log(2 * math.pi * var))
            nll_total = nll_total + nll.mean()

        return nll_total / horizon


    #
====================================
============================
    # Тренувальні цикли
    #
====================================
============================

    def train_neural_sde(model,
train_loader, val_loader, epochs, lr=1e-3,
                         device='cuda',
clip_grad=1.0, variant='A'):
        """
        Універсальний тренувальний цикл
для обох варіантів.

        Args:
            variant: 'A' або 'B' — визначає,
яку loss-функцію використовувати
        """
        import numpy as np

        model.to(device)
        opt =
torch.optim.Adam(model.parameters(), lr=lr)
        sched =
torch.optim.lr_scheduler.CosineAnnealingLR(opt,
T_max=epochs)
        history = {'train': [], 'val': []}

        if variant not in ('A', 'B'):
            raise ValueError(f'variant має
бути "A" або "B", отримано: {variant}')

        for ep in range(epochs):
            model.train()
            train_losses = []
            for ctx, tgt in train_loader:
                ctx, tgt = ctx.to(device),
tgt.to(device)
                opt.zero_grad()
                if variant == 'A':
                    loss =
neural_sde_step_nll_A(model, tgt)
                else:
                    loss =
neural_sde_step_nll_B(model, ctx, tgt)
                loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(),
clip_grad)
                opt.step()
                train_losses.append(loss.item())
            sched.step()

            model.eval()
            val_losses = []
            with torch.no_grad():
                for ctx, tgt in val_loader:
                    ctx, tgt = ctx.to(device),
tgt.to(device)
                    if variant == 'A':

val_losses.append(neural_sde_step_nll_A(model,
tgt).item())
                    else:

val_losses.append(neural_sde_step_nll_B(model, ctx,
tgt).item())

            tl, vl = np.mean(train_losses),
np.mean(val_losses)
            history['train'].append(tl)
            history['val'].append(vl)
            print(f"  Епоха {ep+1}/{epochs}:
train={tl:.4f}, val={vl:.4f}")
        return history


    #
====================================
============================
    # Self-test обох варіантів
    #
====================================
============================

    if __name__ == '__main__':
        print('=== Self-test Neural SDE
модуля ===')
        device = 'cuda' if
torch.cuda.is_available() else 'cpu'
        print(f'Device: {device}')

        batch_size = 16
        context = torch.randn(batch_size, 60,
device=device) * 0.01
        target = torch.randn(batch_size, 20,
device=device) * 0.01

        # ===== Варіант A =====
        print('\n--- Варіант A (без encoder) -
--')
        model_a =
NeuralSDESurrogateA(horizon=20, hidden=64,
sigma_max=0.05).to(device)
        n_params_a = sum(p.numel() for p in
model_a.parameters())
        print(f'Параметрів: {n_params_a}')

        samples_a =
model_a.sample(context, n_samples=2)
        print(f'sample(): {samples_a.shape}
(очікувано: (2, {batch_size}, 20))')
        assert samples_a.shape == (2,
batch_size, 20)

        loss_a =
neural_sde_step_nll_A(model_a, target)
        print(f'Loss: {loss_a.item():.4f}')
        loss_a.backward()
        has_grad_a = any(p.grad is not None
and p.grad.abs().sum() > 0 for p in
model_a.parameters())
        print(f'Градієнти течуть:
{has_grad_a}')
        assert has_grad_a

        # ===== Варіант B =====
        print('\n--- Варіант B (з encoder) ---
')
        model_b =
NeuralSDESurrogateB(horizon=20, hidden=64,
sigma_max=0.05).to(device)
        n_params_b = sum(p.numel() for p in
model_b.parameters())
        print(f'Параметрів: {n_params_b}')

        samples_b =
model_b.sample(context, n_samples=2)
        print(f'sample(): {samples_b.shape}
(очікувано: (2, {batch_size}, 20))')
        assert samples_b.shape == (2,
batch_size, 20)

        loss_b =
neural_sde_step_nll_B(model_b, context, target)
        print(f'Loss: {loss_b.item():.4f}')
        loss_b.backward()
        has_grad_b = any(p.grad is not None
and p.grad.abs().sum() > 0 for p in
model_b.parameters())
        print(f'Градієнти течуть:
{has_grad_b}')
        has_enc_grad = any(p.grad is not
None and p.grad.abs().sum() > 0
                           for p in
model_b.encoder.parameters())
        print(f'Encoder отримав градієнти:
{has_enc_grad}')
        assert has_grad_b and has_enc_grad

        print('\nУСІ ТЕСТИ ПРОЙДЕНО
(обидва варіанти працюють)')
```

 # neural_sde_gan.py

```
"""
Neural SDE як generator у GAN-
постановці (Kidger et al. 2021).

Що тут:
  - TrajectoryDiscriminator — LSTM-
класифікатор траєкторій
  - NeuralSDEGAN — обгортка з
generator (NSDE) і discriminator
  - train_neural_sde_gan — WGAN-
GP loop

Generator — це наш існуючий
NeuralSDESurrogateB (з контекстним енкодером).
Це дає п'яту архітектуру для
порівняння у експериментальній частині:
NLL-навчання NSDE проти GAN-
навчання NSDE.
```

```
        WGAN-GP обрано замість vanilla GAN через стабільність — vanilla адверсаріальні
        loss'и часто страждають від mode collapse при роботі з важкохвостими розподілами
        фінансових доходностей.
        """
        import time
        import torch
        import torch.nn as nn
        import torch.optim as optim
        from torch.utils.data import DataLoader
        import torchsde
        import math

        from neural_sde import NeuralSDESurrogateB


        # ====================================================================
        # Discriminator
        # ====================================================================

        class TrajectoryDiscriminator(nn.Module):
            """
            LSTM-класифікатор траєкторій log-доходностей.

            Вхід: контекст (60 днів) + траєкторія (горизонт 20 днів) — конкатеновані
            у послідовність довжини 80.
            Вихід: одне скалярне значення «реалістичності» (Wasserstein critic).

            LSTM bidirectional для кращого вловлювання залежностей у часі.
            """
            def __init__(self, hidden=64, num_layers=2, dropout=0.1):
                super().__init__()
                self.lstm = nn.LSTM(
                    input_size=1,
                    hidden_size=hidden,
                    num_layers=num_layers,
                    batch_first=True,
                    bidirectional=True,
                    dropout=dropout if num_layers > 1 else 0.0,
                )
                self.head = nn.Sequential(
                    nn.Linear(hidden * 2, hidden),
                    nn.LeakyReLU(0.2),
                    nn.Linear(hidden, 1),
                )

            def forward(self, sequence):
                """
                sequence: (batch, seq_len, 1) — конкатенація контексту і траєкторії
                Returns: (batch,) — скалярна оцінка
                """
                out, _ = self.lstm(sequence)
                # Беремо останній стан з обох напрямків
                last = out[:, -1, :]
                score = self.head(last).squeeze(-1)
                return score


        # ====================================================================
        # Обгортка
        # ====================================================================

        class NeuralSDEGAN(nn.Module):
            """
            Обгортка: NeuralSDE-Generator + Trajectory-Discriminator.

            sample() поведінка така ж, як у NeuralSDESurrogateB — щоб модель
            зберігалась drop-in у пайплайн оцінювання.
            """
            def __init__(self, horizon=20, gen_hidden=64, sigma_max=0.05,
                         encoder_hidden=32, h_dim=16,
                         disc_hidden=64, disc_layers=2, disc_dropout=0.1):
                super().__init__()
                self.horizon = horizon
                self.generator = NeuralSDESurrogateB(
                    horizon=horizon, hidden=gen_hidden, sigma_max=sigma_max,
                    encoder_hidden=encoder_hidden, h_dim=h_dim,
                )
                self.discriminator = TrajectoryDiscriminator(
                    hidden=disc_hidden, num_layers=disc_layers, dropout=disc_dropout
                )

            def sample(self, context, n_samples=1):
                """Drop-in заміна для виклику з generate()."""
                return self.generator.sample(context, n_samples=n_samples)

            def load_generator_from(self, source_model):
                """
                Скопіювати ваги генератора з попередньо натренованої NeuralSDESurrogateB.

                Використання:
                    sde_b = NeuralSDESurrogateB(...)
                    train_neural_sde(sde_b, ...)  # NLL-тренування
                    sde_gan = NeuralSDEGAN(...)
                    sde_gan.load_generator_from(sde_b)  # перенесли ваги G
                    train_neural_sde_gan(sde_gan, ...)  # GAN-fine-tuning
                """
                self.generator.load_state_dict(source_model.state_dict())


        class NeuralHestonGAN(nn.Module):
            """
            Версія NSDE-GAN з Neural Heston backbone (2D state).

            Generator — NeuralHeston (Y + v, корельовані шуми).
            Discriminator — той самий BiLSTM-критик, що приймає 1D-послідовність
            логарифмічних доходностей.

            Завдяки двовимірному стану генератор має явний механізм для відтворення
            кластеризації волатильності (ARCH-ефекту), якого не має 1D NeuralSDE.
            """
            def __init__(self, horizon=20, gen_hidden=64, gen_sigma_max=0.5,
                         encoder_hidden=32, h_dim=16,
                         init_var=0.0, init_rho=-0.7, v_scale=0.001,
                         disc_hidden=64, disc_layers=2, disc_dropout=0.1):
                super().__init__()
                from neural_heston import NeuralHeston
                self.horizon = horizon
                self.generator = NeuralHeston(
                    horizon=horizon, hidden=gen_hidden, sigma_max=gen_sigma_max,
                    encoder_hidden=encoder_hidden, h_dim=h_dim,
                    init_var=init_var, init_rho=init_rho, v_scale=v_scale,
                )
                self.discriminator = TrajectoryDiscriminator(
                    hidden=disc_hidden, num_layers=disc_layers, dropout=disc_dropout
                )

            def sample(self, context, n_samples=1):
                return self.generator.sample(context, n_samples=n_samples)

            def load_generator_from(self, source_model):
                """Перенести ваги з NeuralHeston (NLL pretrain) у генератор."""
                self.generator.load_state_dict(source_model.state_dict())


        # ====================================================================
        # Допоміжне: gradient penalty
        # ====================================================================
```

```
def gradient_penalty(disc, real_seq, fake_seq, device, lambda_gp=10.0):
    """
    WGAN-GP gradient penalty на інтерполованих зразках.
    real_seq, fake_seq: (batch, seq_len, 1)

    ВАЖЛИВО: CuDNN не підтримує double backward через RNN (потрібен для GP),
    тому під час обчислення penalty CuDNN тимчасово вимикається.
    """
    batch_size = real_seq.size(0)
    eps = torch.rand(batch_size, 1, 1, device=device)
    interp = eps * real_seq + (1.0 - eps) * fake_seq
    interp.requires_grad_(True)

    # Disable CuDNN for the duration of the double-backward computation
    with torch.backends.cudnn.flags(enabled=False):
        score = disc(interp)
        grad = torch.autograd.grad(
            outputs=score.sum(),
            inputs=interp,
            create_graph=True,
            retain_graph=True,
            only_inputs=True,
        )[0]
    grad_norm = grad.view(batch_size, -1).norm(dim=1)
    return lambda_gp * ((grad_norm - 1.0) ** 2).mean()


# ============================================================================
# Training loop
# ============================================================================

def train_neural_sde_gan(model, train_loader, val_loader, epochs,
                         lr_g=2e-4, lr_d=2e-4, device='cuda',
                         n_critic=3, lambda_gp=10.0, clip_grad=1.0,
                         use_acf_loss=True, lambda_acf=0.1, max_lag=5,
                         use_moment_loss=True, lambda_moment=50.0,
                         early_stop_patience=8,
                         early_stop_min_epochs=10,
                         max_val_w_threshold=2.0):
    """
    Тренування Neural SDE-GAN у режимі WGAN-GP.

    Параметри:
        model: NeuralSDEGAN
        n_critic: кількість оновлень discriminator на одне оновлення generator
        lambda_gp: вага gradient penalty
        use_acf_loss: додавати ACF-regularizer до generator-loss (для відтворення
                      кластеризації волатильності — як у NLL-варіантах)
        lambda_acf: вага ACF-loss

    Повертає словник з історією тренування.
    """
    model = model.to(device)
    gen = model.generator
    disc = model.discriminator

    opt_g = optim.Adam(gen.parameters(), lr=lr_g, betas=(0.5, 0.9))
    opt_d = optim.Adam(disc.parameters(), lr=lr_d, betas=(0.5, 0.9))

    history = {'d_loss': [], 'g_loss': [], 'gp': [], 'acf': [],
               'moment': [], 'val_w': [],
               # plot_training сумісність:
               'train': [], 'val': []}

    print(f'  NeuralSDE-GAN: {sum(p.numel() for p in gen.parameters()):,} '
          f'gen params, {sum(p.numel() for p in disc.parameters()):,} '
          f'disc params')
    print(f'  Early stopping: patience={early_stop_patience}, '
          f'min_epochs={early_stop_min_epochs}, '
          f'val_W divergence threshold={max_val_w_threshold}')

    # Best checkpoint: state_dict з найкращим val_W
    import copy as _copy
    best_val_w = float('inf')
    best_gen_state = None
    best_epoch = -1
    epochs_since_best = 0

    overall_start = time.time()
    for epoch in range(epochs):
        epoch_start = time.time()
        gen.train()
        disc.train()
        ep_d, ep_g, ep_gp, ep_acf, ep_moment = 0.0, 0.0, 0.0, 0.0, 0.0
        n_batches = 0

        for batch_idx, (context, target) in enumerate(train_loader):
            context = context.to(device)  # (batch, 60, 1) або (batch, 60)
            target = target.to(device)    # (batch, 20)

            if context.dim() == 2:
                context_input = context.unsqueeze(-1)
            else:
                context_input = context

            target_input = target.unsqueeze(-1)
            real_seq = torch.cat([context_input, target_input], dim=1)
            batch_size = real_seq.size(0)

            # ===== D-кроки =====
            for _ in range(n_critic):
                opt_d.zero_grad()
                with torch.no_grad():
                    fake = gen.sample(context, n_samples=1)[0]  # (batch, 20)
                    fake_input = fake.unsqueeze(-1)
                    fake_seq = torch.cat([context_input, fake_input], dim=1)

                d_real = disc(real_seq).mean()
                d_fake = disc(fake_seq).mean()
                gp = gradient_penalty(disc, real_seq, fake_seq,
                                      device, lambda_gp=lambda_gp)
                d_loss = d_fake - d_real + gp
                d_loss.backward()

                torch.nn.utils.clip_grad_norm_(disc.parameters(), clip_grad)
                opt_d.step()

                ep_d += d_loss.item()
                ep_gp += gp.item()

            # ===== G-крок =====
            opt_g.zero_grad()
            fake = gen.sample(context, n_samples=1)[0]
            fake_input = fake.unsqueeze(-1)
            fake_seq = torch.cat([context_input, fake_input], dim=1)
            d_fake = disc(fake_seq).mean()
            g_loss = -d_fake

            if use_acf_loss:
                acf_loss = _acf_loss(fake, target, max_lag=max_lag)
                g_loss = g_loss + lambda_acf * acf_loss
                ep_acf += acf_loss.item()

            if use_moment_loss:
                # Жорсткий штраф за неспівпадіння першого і другого моментів.
                # Це гарантує що mean і std генерованих доходностей відповідають
                # реальним — захист від mode collapse у WGAN.
                gen_mean = fake.mean()
                real_mean = target.mean()
                gen_std = fake.std()
                real_std = target.std()
                moment_loss = (gen_mean - real_mean) ** 2 + (gen_std - real_std) ** 2
                g_loss = g_loss + lambda_moment * moment_loss

            g_loss.backward()

            torch.nn.utils.clip_grad_norm_(gen.parameters(), clip_grad)
```

```
                opt_g.step()

                ep_g += g_loss.item()
                if use_moment_loss:
                    ep_moment += moment_loss.item()
                n_batches += 1

            history['d_loss'].append(ep_d / max(n_batches * n_critic, 1))
            history['g_loss'].append(ep_g / max(n_batches, 1))
            history['gp'].append(ep_gp / max(n_batches * n_critic, 1))
            history['acf'].append(ep_acf / max(n_batches, 1) if use_acf_loss else 0.0)
            history['moment'].append(ep_moment / max(n_batches, 1) if use_moment_loss else 0.0)

            # Validation: Wasserstein-1 між реальним і згенерованим
            val_w = _quick_val_wasserstein(gen, val_loader, device)
            history['val_w'].append(val_w)

            # plot_training сумісність: показуємо g_loss як train, val_w як val
            history['train'].append(history['g_loss'][-1])
            history['val'].append(val_w)

            # Best checkpoint tracking
            marker = ''
            if val_w < best_val_w:
                best_val_w = val_w
                best_gen_state = _copy.deepcopy(gen.state_dict())
                best_epoch = epoch + 1
                epochs_since_best = 0
                marker = ' ← BEST'
            else:
                epochs_since_best += 1

            epoch_time = time.time() - epoch_start
            elapsed = time.time() - overall_start
            eta_sec = (elapsed / (epoch + 1)) * (epochs - epoch - 1)
            eta_min = eta_sec / 60.0

            print(f'    epoch {epoch+1:2d}/{epochs}: '
                  f'D={history["d_loss"][-1]:7.3f}  '
                  f'G={history["g_loss"][-1]:7.3f}  '
                  f'GP={history["gp"][-1]:6.3f}  '
                  f'ACF={history["acf"][-1]:6.4f}  '
                  f'MOM={history["moment"][-1]:8.6f}  '
                  f'val_W={val_w:.4f}  '
                  f'time={epoch_time:5.1f}s  '
                  f'ETA={eta_min:4.1f}min{marker}')

            # === Divergence detection: розривати тренування якщо очевидно не сходиться ===
            # Виходимо тільки після min_epochs щоб дати моделі шанс розгойдатись.
            if epoch + 1 >= early_stop_min_epochs:
                # 1) val_W різко погіршився — катастрофічна розбіжність
                if val_w > max_val_w_threshold:
                    print(f'  !! РОЗБІЖНІСТЬ: val_W={val_w:.4f} > {max_val_w_threshold} '
                          f'на епосі {epoch+1}. Зупиняємось.')
                    break
                # 2) patience epochs без покращення
                if epochs_since_best >= early_stop_patience:
                    print(f'  !! РАННЄ ЗУПИНЕННЯ: {epochs_since_best} епох без '
                          f'покращення val_W (best={best_val_w:.4f} на епосі {best_epoch}).')
                    break

        total_time = time.time() - overall_start
        print(f'  Загальний час навчання NSDE-GAN: {total_time/60:.1f} хв')

        # Відновлюємо найкращі ваги генератора
        if best_gen_state is not None:
            gen.load_state_dict(best_gen_state)
            print(f'  ✓ Завантажено best-checkpoint з епохи {best_epoch} '
                  f'(val_W={best_val_w:.4f})')

        history['best_epoch'] = best_epoch
        history['best_val_w'] = best_val_w
        history['total_time_sec'] = total_time
        return history


    # ============================================================================
    # Допоміжні: ACF-loss і швидка валідація через Wasserstein-1
    # ============================================================================

    def _acf_loss(gen_returns, real_returns, max_lag=5):
        """
        Diferenciable ACF-loss за квадратами доходностей.
        gen_returns, real_returns: (batch, horizon).
        """
        def acf(x, lag):
            mean = x.mean(dim=1, keepdim=True)
            xc = x - mean
            num = (xc[:, :-lag] * xc[:, lag:]).mean(dim=1)
            var = (xc * xc).mean(dim=1) + 1e-8
            return num / var

        loss = 0.0
        for lag in range(1, max_lag + 1):
            a_gen = acf(gen_returns ** 2, lag).mean()
            a_real = acf(real_returns ** 2, lag).mean()
            loss = loss + (a_gen - a_real) ** 2
        return loss


    @torch.no_grad()
    def _quick_val_wasserstein(gen, val_loader, device, n_batches=4):
        """Швидка Wasserstein-1 оцінка на val для моніторингу під час навчання."""
        from scipy.stats import wasserstein_distance
        gen.eval()
        real_all = []
        fake_all = []
        for i, (context, target) in enumerate(val_loader):
            if i >= n_batches:
                break
            context = context.to(device)
            target = target.to(device)
            fake = gen.sample(context, n_samples=1)[0]
            real_all.append(target.cpu().numpy().flatten())
            fake_all.append(fake.cpu().numpy().flatten())
        gen.train()
        if not real_all:
            return float('nan')
        import numpy as np
        return wasserstein_distance(
            np.concatenate(real_all), np.concatenate(fake_all)
        )
```

 # neural_heston.py

```
"""
Neural Heston — нейромережеве розширення моделі Heston з 2D-станом.

Стан: (Y_t, v_t) де:
    Y_t — кумулятивний логарифм ціни (наша «log-price»)
    v_t — миттєва дисперсія (variance)

Динаміка (Itô-форма):
    dY_t = f_Y(t, Y_t, v_t, h) · dt + σ_Y(t, Y_t, v_t, h) · √v_t · dW¹_t
    dv_t = f_v(t, Y_t, v_t, h) · dt + σ_v(t, Y_t, v_t, h) · √v_t · dW²_t

З корельованими шумами:
    corr(dW¹, dW²) = ρ
    ρ — вивчений параметр через tanh(rho_raw) ∈ (-1, 1)
    Ініціалізація ρ ≈ -0.7 для leverage effect

Це аналог моделі Heston (CIR-процес для волатильності + GBM-подібна ціна),
```

```
    але дрейф і дифузія параметризовані MLP. Через явний стан v_t модель
    природно відтворює кластеризацію волатильності — те, чого 1D Neural SDE
    не можуть досягти.

    v_0 ініціалізується з рухомої дисперсії останніх 20 днів контексту через
    v_init_head: це переносить поточний режим волатильності у прогноз.

    Інтегрування: ручний Euler-Maruyama (стабільніше за torchsde для 2D-стану з
    позитивністю v через full truncation).

    Training: step-NLL з teacher forcing на Y та стохастичною еволюцією v через
    корельовані шуми з резидуала Y.
    """
    import math
    import torch
    import torch.nn as nn
    import torch.optim as optim

    from neural_sde import ContextEncoder


    # ============================================================================
    # Drift / Diffusion: спільний MLP видає 4 параметри (f_Y, f_v, σ_Y, σ_v)
    # ============================================================================

    class _DriftDiffNet(nn.Module):
        def __init__(self, hidden=64, h_dim=16):
            super().__init__()
            in_dim = 3 + h_dim  # t (1) + Y (1) + v (1) + h (h_dim)
            self.net = nn.Sequential(
                nn.Linear(in_dim, hidden),
                nn.Tanh(),
                nn.Linear(hidden, hidden),
                nn.Tanh(),
                nn.Linear(hidden, 4),  # f_Y, f_v, σ_Y_raw, σ_v_raw
            )

        def forward(self, t, Y, v, h):
            """t: (batch, 1), Y: (batch, 1), v: (batch, 1), h: (batch, h_dim)."""
            inputs = torch.cat([t, Y, v, h], dim=-1)
            return self.net(inputs)


    # ============================================================================
    # 2D SDE з корельованими шумами
    # ============================================================================

    class _SDEHeston(nn.Module):
        """
        2D SDE state (Y, v).
        _params() повертає (f_Y, f_v, σ_Y_clipped, σ_v_clipped, v_plus).

        v_plus отримуємо через softplus + scale, щоб гарантовано v > 0
        (full truncation для CIR-style процесу).
        """
        def __init__(self, hidden=64, h_dim=16,
                     sigma_max=1.0, init_rho=-0.7,
                     v_scale=0.001):
            super().__init__()
            self.net = _DriftDiffNet(hidden=hidden, h_dim=h_dim)
            self.sigma_max = sigma_max
            # Параметр ρ через atanh, щоб tanh(rho_raw) ≈ init_rho
            rho_raw_init = 0.5 * math.log((1 + init_rho) / (1 - init_rho))
            self.rho_raw = nn.Parameter(torch.tensor(rho_raw_init))
            self.v_scale = v_scale
            self.h = None

        def set_context(self, h):
            self.h = h

        def get_rho(self):
            return torch.tanh(self.rho_raw)

        def _params(self, t_val, Y, v_raw, h):
            """
            Compute drift, diffusion coefficients, and v_plus.

            t_val: float
            Y: (batch, 1) — кумулятивний log-price
            v_raw: (batch, 1) — необмежений «raw» стан v
            h: (batch, h_dim) — контекст

            Returns: (f_Y, f_v, σ_Y, σ_v, v_plus)
            """
            batch_size = Y.size(0)
            device = Y.device
            t_tensor = torch.full((batch_size, 1), float(t_val), device=device)
            v_plus = torch.nn.functional.softplus(v_raw) * self.v_scale + 1e-8

            params = self.net(t_tensor, Y, v_plus, h)
            f_Y = params[:, 0:1] * 1e-4
            f_v = params[:, 1:2] * 1e-3
            sigma_Y = torch.sigmoid(params[:, 2:3]) * self.sigma_max
            sigma_v = torch.sigmoid(params[:, 3:4]) * self.sigma_max
            return f_Y, f_v, sigma_Y, sigma_v, v_plus

    # ============================================================================
    # Головна модель
    # ============================================================================

    class NeuralHeston(nn.Module):
        """
        Neural Heston: 2D-state Neural SDE.

        Архітектура:
          1. ContextEncoder обробляє вікно контексту 60 днів → h ∈ ℝ^h_dim
          2. Окрема голова v_init_head ініціалізує v_0 на основі реальної
             волатильності останніх днів контексту — це ключовий механізм
             для відтворення кластеризації волатильності (поточний режим
             переноситься у прогнозний горизонт)
          3. SDE з 2D-станом (Y, v) інтегрується методом Ейлера-Маруями
             з корельованими шумами (Cholesky)
          4. Видає log-доходності як різниці послідовних Y
        """
        def __init__(self, horizon=20, hidden=64, sigma_max=1.0,
                     encoder_hidden=32, h_dim=16,
                     init_var=0.0, init_rho=-0.7, v_scale=0.001,
                     v_init_window=20):
            super().__init__()
            self.horizon = horizon
            self.h_dim = h_dim
            self.init_var = init_var
            self.v_init_window = v_init_window
            self.encoder = ContextEncoder(hidden=encoder_hidden, h_dim=h_dim)
            self.sde = _SDEHeston(hidden=hidden, h_dim=h_dim,
                                  sigma_max=sigma_max,
                                  init_rho=init_rho, v_scale=v_scale)
            # Голова для ініціалізації v_0: приймає (log_empirical_var, h)
            # → видає поправку у raw-просторі
            self.v_init_head = nn.Sequential(
                nn.Linear(1 + h_dim, hidden // 2),
                nn.Tanh(),
                nn.Linear(hidden // 2, 1),
            )

        def encode(self, context):
            return self.encoder(context)

        def _compute_v_init(self, context, h):
            """v_0 (raw) з рухомої дисперсії та nn-поправки."""
```

```
        if context.dim() == 3:
            ctx_flat = context.squeeze(-1)
        else:
            ctx_flat = context
        recent = ctx_flat[:, -
self.v_init_window:]
        empirical_var = recent.var(dim=1,
keepdim=True) + 1e-8
        log_var = torch.log(empirical_var)
        head_input = torch.cat([log_var,
h], dim=1)
        v_init_raw =
self.v_init_head(head_input)
        return v_init_raw

    def _integrate_one(self, h,
v_init_raw, dt=1.0):
        """
        Ручний Euler-Maruyama крок
для одного шляху з корельованими шумами.

        Returns: (batch, horizon) — log-
доходності за кожний крок.
        """
        batch_size = h.size(0)
        device = h.device
        sqrt_dt = math.sqrt(dt)

        Y_t = torch.zeros(batch_size, 1,
device=device)
        v_raw_t = v_init_raw

        rho = self.sde.get_rho()
        sqrt_1_rho2 = torch.sqrt(1.0 - rho
** 2 + 1e-8)

        diffs = []
        for t_idx in range(self.horizon):
            f_Y, f_v, sigma_Y, sigma_v,
v_plus = self.sde._params(
                t_idx * dt, Y_t, v_raw_t, h)
            sqrt_v = torch.sqrt(v_plus)

            # Корельовані шуми через
Cholesky
            eps_1 = torch.randn(batch_size,
1, device=device)
            eps_2 = torch.randn(batch_size,
1, device=device)
            dW_1 = eps_1 * sqrt_dt
            dW_2 = (rho * eps_1 +
sqrt_1_rho2 * eps_2) * sqrt_dt

            # Euler-Maruyama крок для Y
та v
            r_step = f_Y * dt + sigma_Y *
sqrt_v * dW_1
            Y_t = Y_t + r_step
            v_raw_t = v_raw_t + f_v * dt +
sigma_v * sqrt_v * dW_2

            diffs.append(r_step.squeeze(-1))

        traj = torch.stack(diffs, dim=1)
        return traj

    def sample(self, context,
n_samples=1):
        """
        Drop-in заміна для
NeuralSDESurrogateB.sample.
        Returns: (n_samples, batch,
horizon) — log-доходності
        """
        h = self.encode(context)
        v_init_raw =
self._compute_v_init(context, h)
        outputs = [self._integrate_one(h,
v_init_raw) for _ in range(n_samples)]
        return torch.stack(outputs, dim=0)


# =====================================
=================================
# Step-NLL для тренування
# =====================================
=================================

def neural_heston_step_nll(model,
context, target, dt=1.0):
    """
    Step-NLL з teacher forcing на Y та
стохастичною еволюцією v.

    На кожному кроці:
      1. Обчислюємо (f_Y, σ_Y, v) у
поточному стані (Y_t, v_t, h)
      2. Умовний розподіл r_t | (Y_{t-
1}, v_{t-1}): N(f_Y · dt, σ_Y² · v · dt)
      3. NLL крок = -log p(r_observed |
distribution)
      4. Y_t оновлюємо реальним
r_observed (teacher forcing)
      5. v_t оновлюємо стохастично,
де ε_v корельовано з ε_Y через ρ
         (ε_Y отримуємо з резидуала
r_observed - mean)
    """
    batch_size, horizon = target.shape
    device = target.device

    h = model.encode(context)
    v_init_raw =
model._compute_v_init(context, h)

    rho = model.sde.get_rho()
    sqrt_1_rho2 = torch.sqrt(1.0 - rho **
2 + 1e-8)
    sqrt_dt = math.sqrt(dt)

    Y_t = torch.zeros(batch_size, 1,
device=device)
    v_raw_t = v_init_raw

    total_nll = 0.0

    for t_idx in range(horizon):
        f_Y, f_v, sigma_Y, sigma_v,
v_plus = model.sde._params(
            t_idx * dt, Y_t, v_raw_t, h)
        sqrt_v = torch.sqrt(v_plus)

        mean = f_Y * dt
        std = (sigma_Y * sqrt_v) * sqrt_dt
+ 1e-8

        r_target = target[:, t_idx:t_idx + 1]
        nll_step = 0.5 * ((r_target - mean)
/ std) ** 2 + torch.log(std)
        total_nll = total_nll +
nll_step.mean()

        # Teacher forcing на Y
        Y_t = Y_t + r_target

        # Стохастичне оновлення v з
корельованим шумом
        eps_Y = (r_target - mean) / std
        eps_v_indep =
torch.randn_like(v_raw_t)
        eps_v = rho * eps_Y +
sqrt_1_rho2 * eps_v_indep
        dW_v = eps_v * sqrt_dt

        v_raw_t = v_raw_t + f_v * dt +
sigma_v * sqrt_v * dW_v

    return total_nll / horizon


# =====================================
=================================
# Training loop
# =====================================
=================================

def train_neural_heston(model,
train_loader, val_loader, epochs,
                        lr=1e-3, device='cuda',
clip_grad=1.0):
    """Тренування NeuralHeston через
step-NLL з teacher forcing на Y."""
    model = model.to(device)
    opt =
optim.Adam(model.parameters(), lr=lr)
    scheduler =
optim.lr_scheduler.CosineAnnealingLR(opt,
T_max=epochs)

    history = {'train': [], 'val': [], 'rho': []}

    n_params = sum(p.numel() for p in
model.parameters())
    print(f'  NeuralHeston: {n_params:,}
параметрів, '
          f'початкове ρ =
{model.sde.get_rho().item():.3f}')

    for epoch in range(epochs):
        model.train()
        train_loss = 0.0
        n_batches = 0
        for context, target in train_loader:
            context = context.to(device)
            target = target.to(device)
            opt.zero_grad()
            loss =
neural_heston_step_nll(model, context, target)
            if not torch.isfinite(loss):
                continue
            loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(),
clip_grad)
```

```
    opt.step()
    train_loss += loss.item()
    n_batches += 1

train_loss /= max(n_batches, 1)
scheduler.step()

model.eval()
val_loss = 0.0
n_val = 0
with torch.no_grad():
    for context, target in val_loader:
        context = context.to(device)
        target = target.to(device)
        loss =
neural_heston_step_nll(model, context, target)
        if torch.isfinite(loss):
            val_loss += loss.item()
            n_val += 1
    val_loss /= max(n_val, 1)

    history['train'].append(train_loss)
    history['val'].append(val_loss)
    history['rho'].append(model.sde.get_rho().item())

    print(f'  Εποχα
{epoch+1}/{epochs}: '
          f'train={train_loss:.4f}
val={val_loss:.4f}  '
          f'ρ={model.sde.get_rho().item():+.3f}')

return history
```

ДОДАТОК Б ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

КАФЕДРА
МАТЕМАТИЧНИХ МЕТОДІВ
СИСТЕМНОГО АНАЛІЗУ

КПІ ІМ. ІГОРЯ СІКОРСЬКОГО
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ
КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ СИСТЕМНОГО АНАЛІЗУ

Сурогатні моделі для відтворення динаміки цін фінансових активів

Дипломна робота на здобуття ступеня бакалавра
за освітньо-професійною програмою
«Системний аналіз і управління»
спеціальності 124 «Системний аналіз»

КА-21
Прийма Владислав Михайлович
Керівник: проф., д.т.н. Дмитрієва О.А.

2026 р.

Актуальність

Класичні стохастичні моделі (Гестона, GBM) — еталон для оцінки ризику й ціноутворення, але обчислювально дорогі: для VaR чи стрес-тесту потрібно генерувати десятки тисяч траєкторій методом Монте-Карло. Саме це вузьке місце ризик-менеджменту й спонукало шукати швидку альтернативу — нейромережевий сурогат, який після тренування за частку секунди генерує траєкторії, статистично нерозрізнювані з істинним процесом.

× 15
прискорення генерації
проти симулятора Гестона

10^4–10^6
траєкторій на одну
оцінку ризику портфеля

4 активи
S&P 500 · Apple · JPMorgan · Tesla

Застосування: ризик-менеджмент, стрес-тестування, ціноутворення похідних, навчання ML на синтетичних даних.

2

Мета, об'єкт та завдання

Об'єкт
Багатовимірні динамічні процеси, еволюція яких описується стохастичними диференціальними рівняннями з корельованими шумами.

Предмет
Методи побудови нейромережевих сурогатних моделей для відтворення розподілових та часових характеристик дохідностей фінансових активів.

Мета

Розробка, обґрунтування і реалізація сурогатних моделей для дослідження поведінки основних фінансових показників на основі нейромережевої апроксимації стохастичної динаміки логарифмічних дохідностей.

Основні завдання дослідження:

- Проаналізувати сучасні підходи до побудови сурогатних моделей відтворення динамічних процесів.
- Сформувати навчальну вибірку з історичних рядів доходностей та та обґрунтувати методи чисельних симуляцій.
- Спроєктувати архітектуру нейромережевої сурогатної моделі.
- Реалізувати процедуру навчання моделі та підібрати гіперпараметри для апроксимації еволюції цін акцій.
- Оцінити здатність моделі відтворювати статистичні характеристики процесу та проаналізувати отримані результати.

Еталон: модель стохастичної волатильності Гестона

Двовимірний процес: ціна + окрема (стохастична) волатильність із корельованими шумами.

Відтворює ключові стилізовані факти ринку:

- важкі хвости розподілу дохідностей
- кластеризація волатильності
- leverage-ефект (від'ємна кореляція ρ ціни й волатильності)
- відсутність лінійної автокореляції доходностей

$$dS = \mu S\,dt + \sqrt{v} \cdot S\,dW^{1}$$

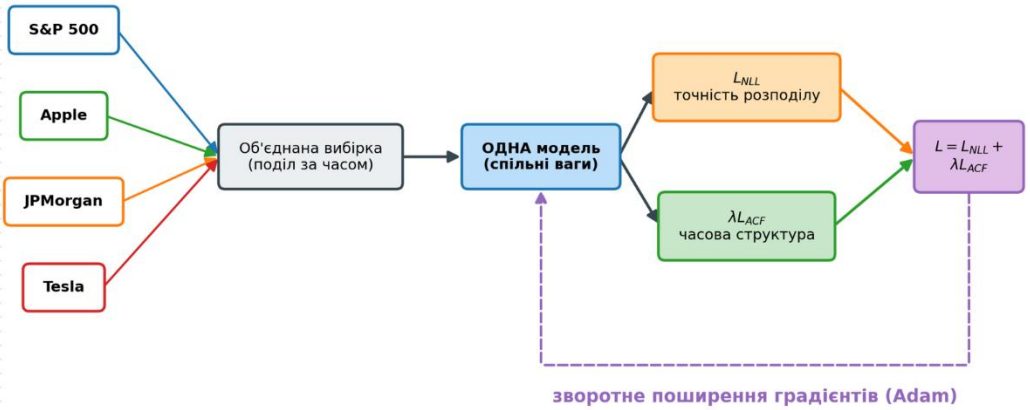
$$dv = \kappa(\theta - v)\,dt + \xi\sqrt{v}\,dW^{2}$$

$$corr(dW^{1}, dW^{2}) = \rho$$

Складність
повільна Монте-Карло генерація → мотивація для сурогата

Інтерпретованість
параметри (κ, θ, ξ, ρ) — зрозумілий фінансовий зміст

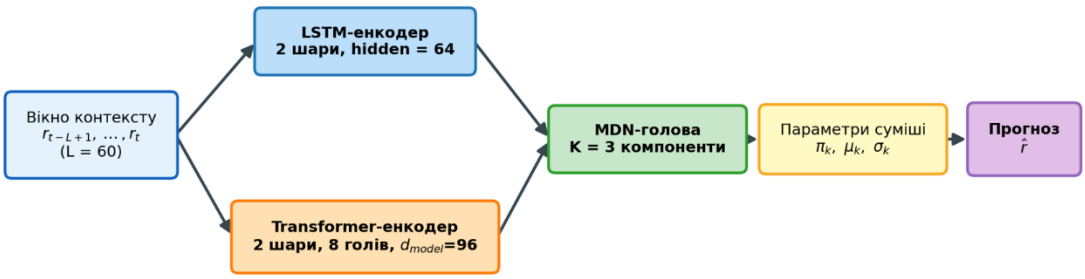
Мультиактивне навчання та функція втрат



Одна модель зі спільними вагами навчається на 4 активах одразу. Комбінована втрата **L = L_NLL + λ·L_ACF (λ = 0,1)**: NLL — точність розподілу, ACF — часова структура (кластеризація волатильності).

5

Проєктування архітектур I–II: LSTM-MDN та Transformer-MDN



Енкодер (LSTM або Transformer) стискає вікно з 60 доходностей у вектор стану; **MDN-голова** видає параметри суміші з K = 3 гаусіан — модель прогнозує не точку, а розподіл наступної доходності.

6

Проєктування архітектур III–IV: Neural SDE (A та B)

NSDE-A (безумовна)

NSDE-B (умовна, + енкодер)

Дрейф (тренд) і дифузія (волатильність) рівняння задаються нейромережами, траєкторія отримується чисельним інтегруванням. **NSDE-A** — безумовна генерація; **NSDE-B** — з LSTM-енкодером історії, тому траєкторія узгоджена з минулою динамікою ціни.

7

Проєктування запропонованої архітектури: Neural SDE-GAN

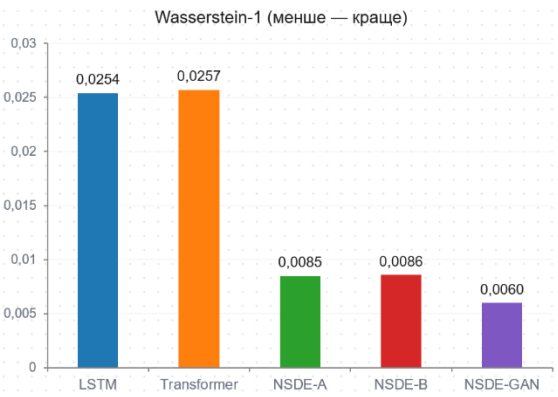
Ключова ідея

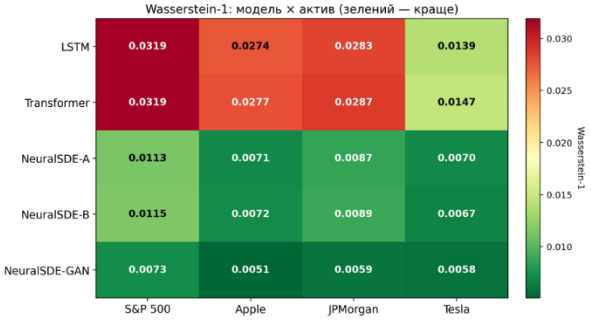
- двовимірний стан Heston-типу (Y, v) — явна компонента волатильності
- змагальне навчання WGAN-GP + ACF-регуляризатор
- кореляція ρ навчається з даних: −0,7 → −0,775 (leverage-ефект)

Об'єднано п'ять архітектур у єдиній експериментальній постановці та порівняно за спільними метриками.

8

Результати: точність відтворення розподілу

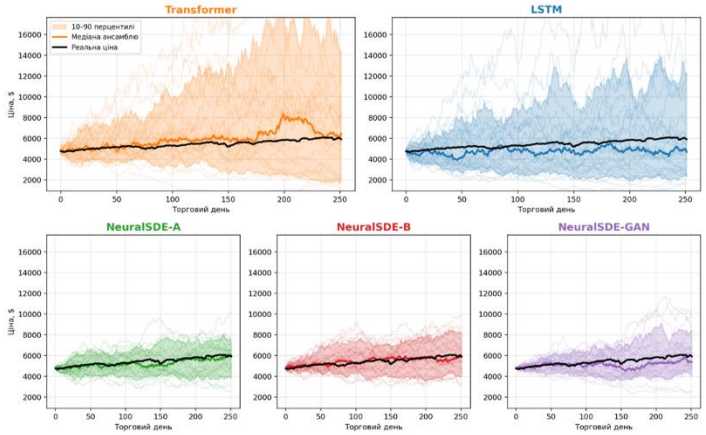




NSDE-GAN — найкраща: Wasserstein 0,0060 проти 0,0085 у найкращої 1D-моделі (**≈29 % покращення**). Архітектури на Neural SDE у 3–4 рази точніші за MDN.

9

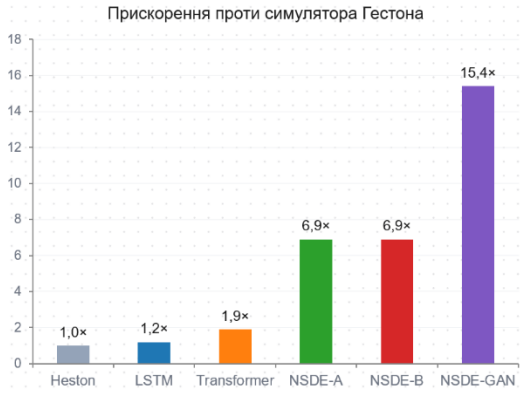
Результати: згенеровані траєкторії ціни (S&P 500)

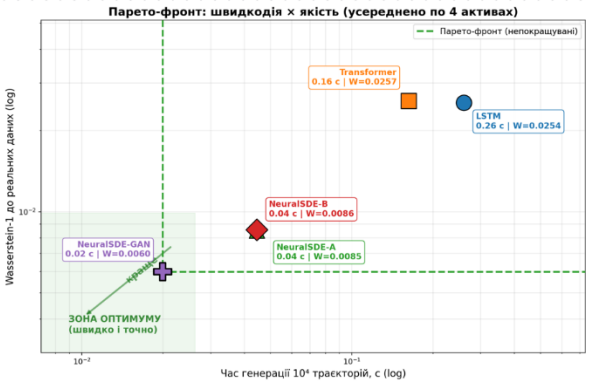


Чорна — реальна ціна, смуга — 10–90 перцентилі ансамблю. У LSTM/Transformer смуга широка (нестабільні траєкторії), у Neural SDE — вузька навколо реальної ціни. (1 торговий рік = 252 дні)

10

Результати: швидкодія та Парето-фронт





NSDE-GAN — єдина непокращувана модель на Парето-фронті: **×15,4 швидше** і водночас найточніша.

11

Висновки

- ✓ Проаналізовано класичні стохастичні та нейромережеві підходи до сурогатного моделювання й обрано напрям Neural SDE.
- ✓ Сформовано навчальну вибірку з історичних доходностей чотирьох активів і реалізовано еталонний симулятор Гестона (Ейлер–Маруями).
- ✓ Спроєктовано п'ять архітектур (LSTM-MDN, Transformer-MDN, Neural SDE-A/B та Neural SDE-GAN) у єдиній постановці.
- ✓ Реалізовано мультиактивне навчання з комбінованою втратою NLL + λ·ACF і підібрано гіперпараметри.
- ✓ Найкращою за метриками (Wasserstein 0,0060) і швидкодією (×15,4) є Neural SDE-GAN; вона відтворює leverage-ефект (ρ: −0,7 → −0,775). Обмеження щодо кластеризації волатильності — напрям подальших досліджень.
- ✓ Виявлене обмеження щодо кластеризації волатильності окреслює напрям подальших досліджень.

12

Дякую за увагу!

Готовий відповісти на запитання

Прийма Владислав Михайлович · гр. КА-21 · 2026