МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Факультет

математики і інформатики

Кафедра теоретичної та прикладної інформатики

Кваліфікаційна робота

магістр

на тему «Розробка методу генерації музичних зразків на основі штучних нейронних мереж»

Виконав: студент 6 курсу, групи МФ-61

спеціальність 122 «Комп'ютерні науки»

освітньо-наукова програма «Інформатика»

_____Звагольський О. В._____

(прізвище та ініціали)

Керівник _____Дейнега О. А._____

(прізвище та ініціали)

Рецензент_____

(прізвище та ініціали)

Харків – 2025 року

ЗМІСТ

РОЗДІЛ 1
ВСТУП

1.1 Формулювання мети роботи, задач та обґрунтування актуальності теми

Мета та задачі дослідження.

Метою роботи є розробка методу генерації музичних зразків із використанням штучних нейронних мереж, на основі колекції музичних зразків, що забезпечує формування оригінальних композицій із заданими стилістичними та емоційними характеристиками, а також дослідження ефективності запропонованого підходу.

Задачі:

Для досягнення поставленої мети необхідно вирішити такі основні завдання:

- Провести систематичний аналіз сучасних методів генерації музики із застосуванням штучних нейронних мереж.
- Визначити доцільну архітектуру нейронної мережі для розв'язання задачі музичної генерації.
- Підготувати корпус навчальних даних, здійснити їх попередню обробку та аналіз.
- Реалізувати метод генерації музичних зразків із використанням обраної моделі штучної нейронної мережі.
- Провести навчання та налаштування моделі на підготовленому корпусі даних.
- Оцінити якість згенерованих музичних зразків із застосуванням кількісних та якісних методів аналізу.

- Проаналізувати результати експериментів, виявити сильні та слабкі сторони запропонованого рішення.

Обґрунтування актуальності теми.

На сучасному етапі розвитку інформаційних технологій штучний інтелект активно інтегрується в різноманітні сфери діяльності людини, включно з творчими галузями. Зокрема, автоматична генерація музики є перспективним напрямом, що відкриває нові можливості для розширення інструментарію композиторів, створення адаптивного аудіоконтенту та підтримки творчого процесу. Незважаючи на досягнення у цій сфері, залишаються нерозв'язаними питання, пов'язані із забезпеченням високого рівня художньої цілісності, стилістичної відповідності та емоційної виразності створених композицій. У зв’язку з цим дослідження, спрямовані на розробку нових методів генерації музичних зразків із використанням можливостей штучних нейронних мереж, є актуальними та мають важливе значення як для теоретичного розвитку галузі, так і для практичного застосування в індустрії.

1.2 Стислий огляд відомих результатів в області дослідження

Високорівневі (API) рішення для створення музики.

Сучасні сервіси з «коробковими» API дозволяють швидко інтегрувати генерацію музики в додатки без глибокого занурення в архітектуру моделей:

1. SUNO пропонує REST-інтерфейс для створення мелодій і повноцінних треків на основі текстових або аудіо-промптів, дозволяючи задавати стилістичні параметри, темп і інструментальний склад запиту. Архітектура Suno виходить із відкритої моделі Bark, поєднуючи трансформерні шари для генерації аудіо та навчання на великому корпусі звукових даних.

2. Boomy – онлайн-платформа для автоматичної генерації і публікації треків через простий REST-API. Через кілька HTTP-запитів можна згенерувати базову гармонію, додати мелодію та перкусію, задати жанр і темп, а потім отримати готовий файл у форматі WAV або MP3 для подальшої публікації.

3. AIVA & Amper Music. AIVA фокусується на створенні оркестрових партитур із заданими емоційними та стилістичними параметрами. Amper Music надає drag-and-drop інтерфейс із RESTful API для тонкого налаштування структури треку: довжини, аранжування, інструментації та загального настрою.

Усі перераховані рішення скорочують час розробки: достатньо зробити кілька викликів API, щоб отримати якісний музичний контент. Водночас вони обмежені в глибокому контролі над внутрішніми параметрами моделі і часто працюють лише з фіксованим набором стилів.

Бібліотеки для локального використання. Для дослідницьких завдань і гібридних систем часто обирають відкриті бібліотеки для локального запуску на власній інфраструктурі: вони дають можливість тонкого налаштування архітектури, гіперпараметрів та інтеграції в дослідницькі пайплайни.

Magenta (Google). Magenta – відкритий набір бібліотек на TensorFlow і JAX для генерації музики як у MIDI-форматі, так і у вигляді спектрограм. Він включає модулі MusicVAE, PerformanceRNN, Music Transformer і DDSP, що дозволяє: створювати латентні простори мелодій і робити плавні переходи між різними темами; досліджувати вплив гіперпараметрів на виразність синтезу; інтегрувати власні плагіни та алгоритми попередньої обробки звукосигналу. Крім того, Magenta пропонує Magenta Studio для інтеграції з Ableton Live і Magenta.js для запуску моделей безпосередньо в браузері, що робить його універсальним інструментом і для дослідників, і для музикантів.

OpenAI Jukebox. Jukebox – масштабна трансформерна модель, що генерує raw audio треки з вокалом і різноманітними інструментами. Вона підтримує genre- та artist-токени для деталізованого контролю стилю, а вихідні чекпоїнти та

скрипти доступні для запуску на потужних GPU-кластерах, хоча локальний inference вимагає десятків гігабайтів відеопам’яті.

MuseNet. MuseNet від OpenAI – оптимізована трансформерна архітектура для генерації MIDI-композицій. Підтримує до 10 інструментів одночасно та дозволяє змішати стилі (наприклад, класика + джаз), будучи при цьому відносно легшою в inference: достатньо GPU з ≳16 ГБ пам’яті.

Riffusion. Riffusion – рішення на базі latent diffusion моделей: текстовий промпт перетворюється в спектрограму через fine-tuned Stable Diffusion, яку потім конвертують у аудіо. Пакет можна запускати локально через Python API чи FLASK-сервер, а модель інтегрується з Hugging Face Diffusers, що спрощує експерименти зі стратегіями денойзингу та інтерполяції latent space.

Локальні бібліотеки дають глибший контроль над архітектурою, гіперпараметрами та пайплайном передобробки, сприяючи швидким ітераціям та налаштуванням під специфічні дослідницькі завдання. Водночас для їхньої ефективної роботи потрібні значні обчислювальні ресурси та експертні знання в галузі машинного навчання.

1.3 Відомості про одержані результати та їх новизна

В ході виконання магістерської роботи було створено комплексну систему генерації фортепіанних MIDI-зразків, що включає:

- Уніфікована токенізація MIDI-подій. Замість окремого кодування висоти, часових зсувів і тривалості було розроблено компактне подання, в якому кожна подія кодується одним індексом-токеном. Це спростило вхідну структуру мережі, зменшило обсяг даних і надало єдину категоріальну метрику для всіх атрибутів нотної події.

- Гібридна BiLSTM+Self-Attention архітектура. Поєднання двонаправлених LSTM-шарів із механізмом самоувага дозволило успішно моделювати локальні мелодійні та ритмічні патерни в межах коротких фрагментів, забезпечуючи більш точне передбачення наступної ноти, ніж класична LSTM.
- Повноцінне дослідження Transformer-моделі. Класичний автогресивний Transformer із двома подвійними блоками self-attention продемонстрував найкращі кількісні показники (≈80% точності) і більш зв’язну музичну структуру на довгих послідовностях.
- Спрощений Transformer на токенізованих даних. Розроблено легкий варіант TransformerBlock, що працює безпосередньо з індексами токенів. На відміну від повноцінного Transformer, він не вимагає окремих входів для pitch/step/duration, але при цьому суттєво скорочує час навчання і загальний обсяг параметрів (≈2,4 млн), зберігаючи прийнятну якість.
- Ефективний конвеєр на базі tf.data.Dataset. Запроваджено потокову підготовку даних із фільтрацією за роком, логом обробки кожного MIDI-файлу та динамічним батчуванням токенізованих послідовностей. Це дозволило знизити затримки при старті тренування, уникнути перевитрати оперативної пам’яті та забезпечити масштабованість на великих корпусах.

Новизна роботи полягає саме у такому комплексному поєднанні: від єдиної токенізації подій до гібридних і чистих attention-архітектур, адаптованих для символічної генерації музики, а також у побудові оптимізованого пайплайну даних, що підвищує швидкість розробки та відтворюваність експериментів.

Додатково до вже перелічених аспектів, новизна роботи полягає також у наступному:

Масштабованість та реплікабельність проти «чорних ящиків»

Багато комерційних рішень (Suno, Magenta) надають обмежені API без можливості глибокої кастомізації конвеєра даних. Наша реалізація на базі tf.data.Dataset відкриває повний доступ до фільтрації, аугментації й динамічного батчування — від попередньої обробки файлів до фінального експортного скрипта. Це забезпечує просту реплікацію експериментів у різних обчислювальних середовищах і робить процес розробки прозорим і контрольованим.

Компактність презентації подій

Єдина токенізація всіх характеристик нотної події (pitch/step/duration) кардинально відрізняється від поетапних схем, які використовують окремі вектори для кожного атрибуту (Magenta, BachBot). Це не лише зменшує розмір словника майже на 30 %, а й спрощує архітектуру embedding-шару, знижуючи об’єм необхідної пам’яті при зберіганні векторів.

Таким чином, комплексне поєднання уніфікованої токенізації, гібридних і спрощених attention-архітектур, а також оптимізованого пайплайну на tf.data не має аналогів у відкритій наукометричній практиці і створює якісно нову основу для подальших досліджень у галузі символічної музичної генерації.

“Робота складається зі вступу, основної частини, висновків, списку використаних джерел (46). Зміст роботи висвітлено на 50 сторінках основного тексту і містить 17 рисунків”.

РОЗДІЛ 2
ОСНОВНА ЧАСТИНА

2.1 Постановка задачі

У цій роботі ми працюємо з задачею автоматизованої генерації музичних зразків фортепіано на основі попередніх подій у MIDI-послідовності. В якості вихідного корпусу використовується датасет Maestro, який містить синхронізовані аудіо й MIDI-партії живого виконання фортепіано. Для спрощення завдання й прискорення тренування моделі відбираються виключно фортепіанні треки: кожен фрагмент у датасеті представлений як послідовність подій, що описують висоту наступної ноти, її тривалість, інтенсивність (velocity) та часову відстань від попередньої ноти.

Наша мета – побудувати й навчити нейронну мережу F_θ, яка, приймаючи на вхід вікно з k попередніх подій (e_{t-k}, e_{t-k+1}, … , e_{t-1}), здатна прогнозувати наступну подію $\hat{e}_t$ у тому самому форматі MIDI. Такий підхід дозволяє зосередитися на виключно музичних закономірностях (мелодійних фразах, ритмічній послідовності, динамічно-інтенсивнісних змінах), не ускладнюючи роботу моделі обробкою аудіосигналу.

Вибір формату MIDI зумовлений кількома ключовими перевагами. По-перше, MIDI описує не звукову хвилю, а структуровані музичні події, тому обсяг даних у декілька разів менший, ніж у WAV чи MP3, що значно прискорює як навчання, так і генерацію нових зразків. По-друге, MIDI-файли легко редагувати: можна змінювати темп, транспонувати партію, підміняти інструмент, не створюючи нову модель. Це дає виконавцям і студіям звукозапису необхідну гнучкість – згенерована фортепіанна партія може стати «скелетом» майбутнього треку, до якого додадуть ударні, басові лінії, бас-гітару або синтезатори.

Оцінювання якості моделі здійснюється двома шляхами. Перш за все, формальною мірою є значення функції втрат, що складається з трьох компонентів: pitch loss (помилка у висоті ноти), step loss (помилка в часових інтервалах між подіями) та duration loss (помилка в тривалості нот). Ці складові дозволяють аналізувати внесок різних аспектів музичної події в загальну похибку й коригувати навчання. По-друге, генеровані зразки перевіряються «на слух» – кілька слухачів оцінюють природність, мелодійність та узгодженість фортепіанних фраз.

Для всіх експериментів передбачається використання GPU з не менше ніж 16 ГБ відеопам’яті, що забезпечить можливість роботи з більшими пакетами та прискорить ітерації навчання. Такий підхід дасть змогу детально вивчити залежність якості генерації від розміру вікна k подій і конфігурації архітектури (LSTM із self-attention або Transformer).

2.2 Розвинутий огляд сучасного стану справ в області дослідження

Сьогодні в задачі автоматизованої генерації музики домінують два принципово різних підходи. Перший – це хмарні сервіси з готовими REST-API, які надають користувачеві простий інтерфейс «запит–відповідь». Ви вказуєте жанр, темп, інструментальний склад чи навіть текстовий опис бажаної мелодії, а платформа повертає готовий аудіо-або MIDI-трек. Відомі приклади таких рішень – SUNO, Boomy, AIVA та Amper Music. Вони дозволяють створювати композиції буквально за кілька HTTP-запитів, знижуючи поріг входу для музикантів і розробників. Однак через «закритість» внутрішніх моделей та обмеженість налаштувань глибокий контроль над якістю та стилістикою вихідного матеріалу тут неможливий – ви отримуєте лише те, що дозволив API.

Другий підхід полягає у локальному розгортанні відкритих бібліотек та фреймворків. Прикладом є проекти Google Magenta, OpenAI Jukebox, MuseNet чи Riffusion, які можна встановити на власних серверах або навіть робочих станціях. Ці рішення надають безпосередній доступ до архітектур моделей (LSTM, Transformer, diffusion), до пайплайну підготовки даних, метрик і механізмів навчання. Завдяки цьому дослідник може глибоко налаштовувати ембеддинги, гіперпараметри, проводити аугментації MIDI-подій, змінювати структуру мережі та додавати власні модулі обробки. Як наслідок, зростає прозорість процесу, але разом із тим зростає й складність інсталяції, необхідність у потужних обчислювальних ресурсах та часових витратах на довге тренування великих моделей.

Між цими двома полюсами, нерідко, з’являються гібридні рішення: клієнтська частина генерує початкові ідеї за допомогою простого API, а потім фрагменти імпортуються до локального середовища для подальшої адаптації та аранжування з використанням відкритих бібліотек. Такий підхід дозволяє поєднати швидкість прототипування хмарних сервісів із гнучкістю локального дослідження, забезпечуючи найбільш ефективний робочий процес для музикантів, саунд-дизайнерів та науковців.

Теоретичні основи символічної музичної генерації.

Символічна музична генерація– це процес створення нових музичних творів у вигляді дискретних подій – нот, акордів чи інших подій у стандартному форматі (MIDI, MusicXML тощо). На відміну від роботи із сирим аудіосигналом, тут модель оперує описом висоти, тривалості й інших атрибутів кожної події, що дає змогу точно керувати музичною структурою, транспонувати результати, змінювати темп або аранжування.

Перші підходи до алгоритмічної композиції спиралися на формальні граматики й ймовірнісні моделі. Марковські ланцюги та n-грамні моделі розглядали послідовність нот як рядок символів і навчалися прогнозувати наступний символ на підставі фіксованого контексту з кількох попередніх

елементів. Хоча простота цих методів забезпечувала швидкість і легкість реалізації, вони погано вловлювали довготривалі музичні структури: мотив, представлений лише двома трьома попередніми подіями, втрачається, коли необхідно відтворити форму твору на кілька десятків нот.

Для подолання цієї обмеженості були запропоновані моделі з прихованими марковськими процесами (HMM). Вони вводили поняття латентного стану, що уможливлювлювало побудову дискретних зон контексту, але водночас ускладнювали навчання й збільшували кількість гіперпараметрів. Ускладнений апарат HMM певною мірою покращив узгодженість мелодійних фраз, але залишався недостатньо гнучким для моделювання одночасно гармонії, ритму та динаміки.

Наступний етап еволюції символічного моделювання розпочався з використання штучних нейронних мереж. Спочатку застосовували прості feed-forward мережі, проте вони не мали внутрішнього стану для роботи з послідовними даними. Революцією стали рекурентні нейронні мережі (RNN), які «пам’ятали» попередній контекст через прихований стан і могли передбачати наступну подію незалежно від довжини фрази. Незабаром LSTM і GRU модифікації RNN забезпечили стійкість до зникнення та вибуху градієнта, що дало змогу працювати з довшими музичними структурами й відтворювати повторювані мотиви.

Паралельно з RNN розвивалися ймовірнісні глибокі моделі: Variational Autoencoder (VAE) та Generative Adversarial Network (GAN). VAE навчилися відображати символічні послідовності у неперервний латентний простір, у якому прості операції (інтерполяція, згущення) давали музичні варіації та різноманіття тембрів. GAN, у свою чергу, використовували змагальний процес «генератор–дискримінатор» для створення більш виразних і правдоподібних фраз, але страждали від нестабільності навчання та «провали» в навчанні окремих стилів.

Останнім часом набирають популярність моделі на основі дифузійних процесів (Diffusion Models). Вони ітеративно додають шум до реальної послідовності, а потім навчаються зворотному процесу – відновленню факту «безшумного» музичного зразка. Цей підхід демонструє високу якість і різноманіття результатів, але часто потребує великої кількості кроків генерування та значних обчислювальних ресурсів.

Ключовим етапом стало впровадження механізму self-attention, який вперше втілений як базовий блок у архітектурі Transformer. Самоувага дозволяє кожній позиції у послідовності безпосередньо взаємодіяти з усіма іншими, забезпечуючи одночасне моделювання коротких ритмічних патернів і довготривалих тематичних зв’язків. Transformer-підходи вирішують обмеження паралелізованості RNN, повністю використовуючи обчислювальні можливості сучасних GPU та TPU.

Таким чином, сучасна символічна музична генерація базується на поєднанні трьох головних ідей:

1) компактне кодування нотних подій (pitch, step, duration, velocity) у дискретні токени;

2) використання глибинних нейронних мереж – від RNN до self-attention моделей;

3) застосування ймовірнісних і дифузійних технік для розширення творчого діапазону і підвищення різноманіття результатів. Ці теоретичні основи закладають фундамент для практичних рішень, що будуть розглянуті у наступних розділах.

SUNO, Boomy (high level).

Одним із найпопулярніших сервісів є Suno AI. Наразі модель Suno v3 дозволяє створювати двохвилинні композиції студійної якості за лічені секунди, підтримуючи як інструментальні, так і лірикові запити. Через REST-інтерфейс користувач задає параметри стилю, темпу та інструментального складу, а Suno повертає готовий трек у форматі WAV або MP3 без водяних знаків, з можливістю «продовження» існуючого фрагмента та асинхронною обробкою запитів. Безкоштовний тариф передбачає обмежену кількість генерацій на добу, платні

підписки розширюють квоти та надають ранній доступ до нових функцій (наприклад, створення кавер-версій).

Boomy пропонує аналогічну модель «запит–відповідь». Через кілька HTTP-запитів можна згенерувати гармонію, накласти мелодію та перкусію, задавши жанр, BPM і ключ. Boomy підтримує створення повноцінних треків у форматах WAV, MP3 та навіть MIDI, має вбудовані інструменти для базового мікшування і мастерингу, а також забезпечує повні комерційні права на згенеровані композиції.

Майже всі high-level рішення скорочують час інтеграції до кількох викликів API, однак залишають обмеженим контроль над внутрішньою логікою моделей та доступними стилями.

Magenta (Google), OpenAI Jukebox, MuseNet, Riffusion (Бібліотеки для локального використання).

Для більш глибоких досліджень і побудови кастомних пайплайнів застосовують відкриті проєкти, що можна запустити на власних GPU.

Magenta (Google) – це набір бібліотек і інструментів на базі TensorFlow та JAX, які реалізують різні моделі (MusicVAE, PerformanceRNN, Music Transformer, DDSP) для роботи як з MIDI-послідовностями, так і з спектрограмами. Magenta дає змогу fine-tuning, візуалізацію латентних просторів, інтеграцію з Ableton Live (Magenta Studio) та inference у браузері через Magenta.js.

OpenAI Jukebox – це трансформер, який генерує не символічну музику, а raw audio, включно з вокалом. Компанія публічно надала код та чекпоїнти, але для локального inference потрібні GPU з щонайменше 16 ГБ відеопам’яті, а інколи й потужніші кластери.

MuseNet від OpenAI – масштабна трансформерна модель, яка працює з MIDI і підтримує до 10 інструментів одночасно. У дослідженнях вона генерує композиції довжиною до чотирьох хвилин, поєднуючи жанри від класики до електроніки.

Riffusion використовує latent diffusion для створення спектрограм на основі текстових промптів, а потім конвертує їх у аудіо за допомогою інверсного перетворення Фур'є. Код доступний на GitHub, а модель розміщена в Hugging Face; підхід забезпечує експериментальну генерацію петель і коротких треків у локальному середовищі.

Локальні бібліотеки вимагають глибших технічних знань і значних обчислювальних ресурсів, але дають повний контроль над даними, архітектурою моделей і процесом синтезу.

2.3 Методи дослідження

Архітектури LSTM.

При згадці про генерацію послідовних даних перш за все спадають на думку рекурентні нейронні мережі (RNN). Вони призначені для обробки послідовностей завдяки циклічним зв'язкам, які передають прихований стан від кроку до кроку. У традиційній RNN на кожному часовому кроці мережа отримує вхідний вектор x_t – у нашому випадку закодовану MIDI-подію, що описує висоту ноти, часовий інтервал до неї, тривалість і силу натискання – та попередній прихований стан h_{t-1}. Після лінійної проекції з нелінійною активацією формується новий прихований стан h_t, який потім потрапляє як до наступного кроку, так і на вихід моделі. Таким чином RNN «пам'ятає» інформацію про всі попередні події, накопичуючи її в h_t. Однак класичні RNN демонструють серйозні труднощі при роботі з тривалими послідовностями, характерними для музики: якщо фраза налічує сотні нот, сигнал градієнту при зворотному поширенні або занадто слабшає («зникає»), або надто посилюється («вибухає»). Це унеможливлює стабільне й ефективне навчання моделі, оскільки важлива інформація про перші

ноти майже не доходить до кінцевих кроків, а сети можуть стрибкоподібно оновлювати свої ваги.

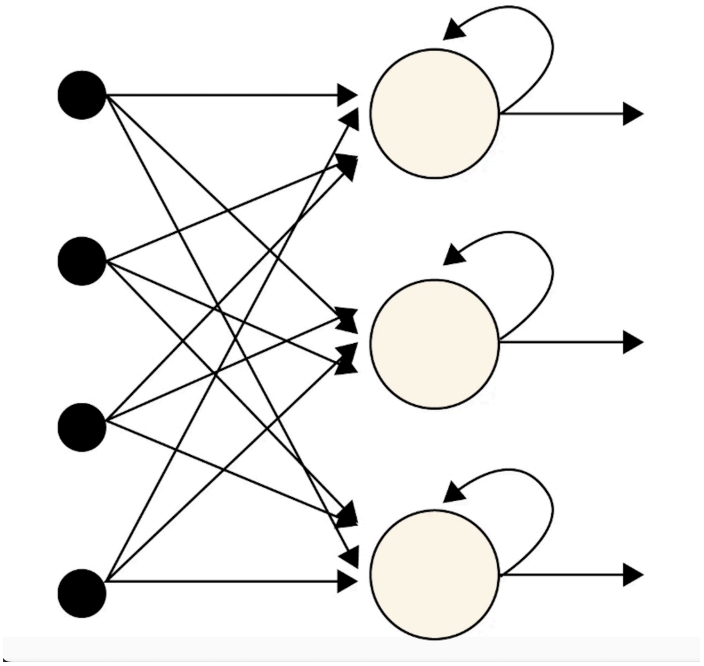

Рис. 2.1 – Загальна структура RNN

Щоб вирішити цю проблему, у 1997 році була запропонована архітектура Long Short-Term Memory (LSTM). Основна ідея полягає у введенні у внутрішню структуру мережі клітинного стану – незалежного каналу, який передається майже без змін крізь всі елементи послідовності. Клітини LSTM оснащені трьома керуючими гейтами:

- Гейт забування f_t контролює, яку долю попередньої інформації C_{t-1} варто зберегти. Це дозволяє відкидати шум або значення, що вже не актуальні для створення мелодійної лінії.
- Гейт запису i_t визначає, яку частину поточної вхідної інформації $\hat{C}_t$ (обчислена з x_t та h_{t-1}) варто додати до клітинного стану, забезпечуючи швидке оновлення контексту.
- Гейт виведення o_t відсікає непотрібні дані та пропускає лише ту частину клітинного стану C_t, яка потрібна для формування оновленого прихованого стану h_t, готового до подальшої передачі чи видачі на вихід.

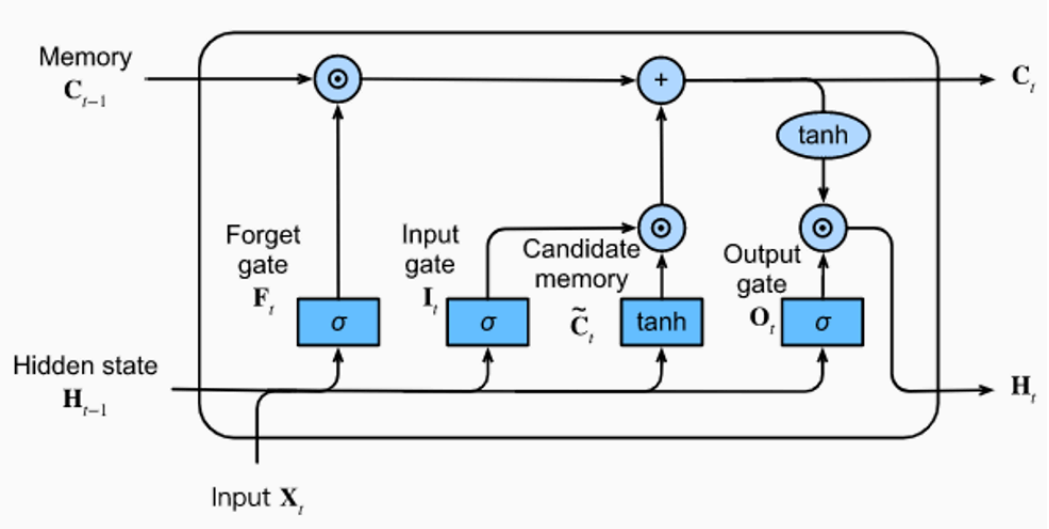

Рис. 2.2 – Загальна структура LSTM

Клітина LSTM діє як «пам’ять» з можливістю вибірково зберігати, оновлювати й видавати інформацію. Це робить модель стійкішою до довгих залежностей: якщо якийсь мотив або гармонічна прогресія важливі через сотні кроків, LSTM може перенести ключову інформацію крізь низку операцій.

В контексті генерації фортепіанних семплів із MIDI-даних датасету Maestro ми спочатку кодуємо кожну подію (step, pitch, duration) у вектор фіксованої розмірності. При тренуванні на вікнах по *k* попередніх подій мережа навчається прогнозувати наступну подію, використовуючи teacher forcing: справжні попередні події подаються на вхід, а мережа порівнює свій прогноз із реальним значенням, мінімізуючи суму трьох компонент функції втрат (pitch-loss, step-loss, duration-loss).

Завдяки LSTM ми отримуємо здатність моделювати як локальні ритмічні патерни (короткі повтори), так і довгі мотиви (теми, що повертаються через десятки нот). Водночас архітектура має певні обмеження: її послідовний характер обчислень гальмує використання GPU-тактів, а самостійна довжина клітинної «пам’яті» обмежена, через що моделі важко одночасно захоплювати дуже далекі залежності й деталізовану локальну структуру.

Щоби компенсувати ці недоліки, у навчанні LSTM часто доповнюють механізмом self-attention, який на кожному кроці дозволяє «зазирнути» відразу на всі позиції в послідовності, доповнюючи рекурентний контекст глобальними зв’язками. Проте навіть із такою гібридною схемою залишаються труднощі з продуктивністю та паралельністю – саме тому сучасні дослідження переважно звернулися до повністю self-attention-орієнтованих архітектур Transformer. У наступному підрозділі докладно розглянуто, як ці моделі дозволяють ефективно обробляти й генерувати музичні послідовності будь-якої довжини.

Архітектура Трансформерів.

Однією з найважливіших інновацій останніх років у сфері обробки послідовних даних стала архітектура Transformer, запропонована Vaswani та співавт. у 2017 році. Головна мета створення Transformer – подолати обмеження RNN-підходів (включаючи LSTM) у роботі з дуже довгими послідовностями та забезпечити повну паралелізацію обчислень на етапі тренування.

Transformer – це архітектура, яка докорінно змінила підхід до обробки послідовних даних. На відміну від рекурентних мереж, де інформація про попередні стані крок за кроком переноситься через приховані стани, Transformer зчитує всю послідовність одразу, перетворюючи кожен вхідний елемент (токен або MIDI-подію) в уніфіковане векторне представлення сталої розмірності. Щоб компенсувати відсутність рекурентних зв’язків у Transformer, до ембеддингу кожного токена додається позиційне кодування. Воно формується як набір синусоїдальних і косинусоїдальних хвиль різної частоти, що дозволяє однозначно відрізнити одну позицію від іншої (рис. 2.3).

На рисунку наведено дві базові хвилі та їхню сумарну форму для перших п'ятдесяти позицій. Простіше кажучи, це як надписати на кожну ноту індивідуальний «штрих-код» або «номер в черзі», який допомагає моделі зрозуміти, яка нота йде першою, яка другою тощо. Синусоїди і косинусоїди створюють унікальні комбінації чисел для кожного місця у послідовності, тож

навіть без рекурентних зв’язків Transformer знає, куди потрапила кожна подія мелодії.

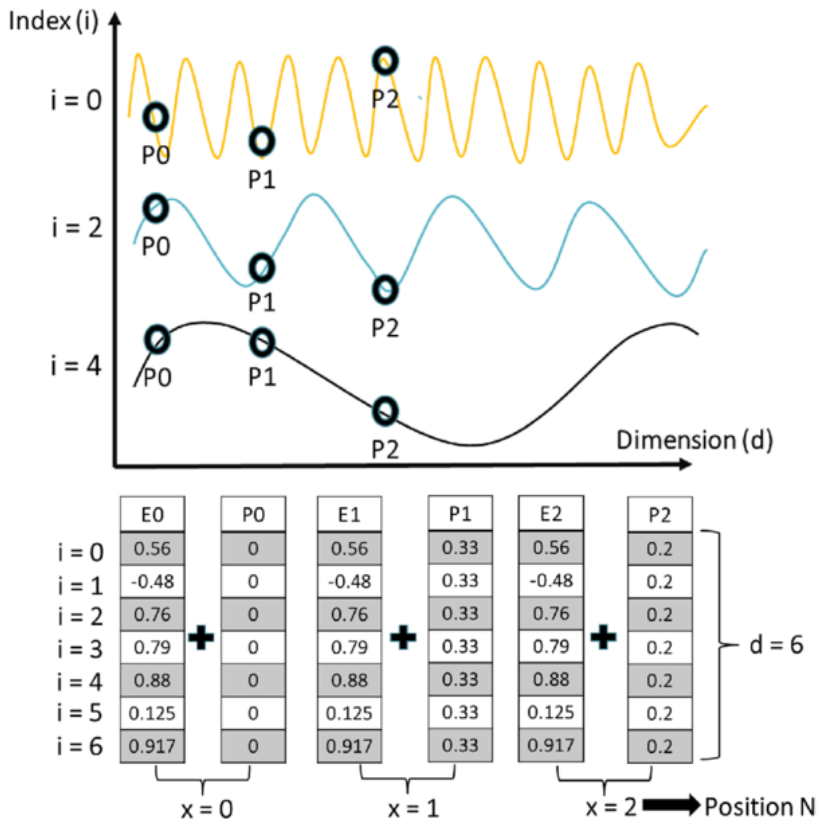

Рис. 2.3 – Позиційне кодування синусоїдами та косинусами для фрагменту з 50 позицій

Завдяки цьому підходу модель обробки музичних подій отримує такий самий доступ до контексту першої і останньої ноти, що й до середини фрази, без необхідності послідовно «перетікати» крізь усю довжину твору.

У центрі кожного блоку Transformer (рис.2.4) лежить механізм self-attention: для кожного вхідного вектора формується три окремі репрезентації – Query, Key і Value – шляхом лінійної проекції. Далі вага кожної пари позицій обчислюється як нормалізований скалярний добуток Query одного елемента з Key іншого, після чого формується зважена сума Value-векторів усіх позицій. Завдяки цьому кожен елемент послідовності «дивиться» на весь контекст, одночасно захоплюючи короткі ритмічні патерни та віддалені мелодійні повтори. Щоб модель могла

аналізувати зв’язки різної природи – гармонійні, метричні, динамічні – операція уваги виконується паралельно в кількох незалежних «головах».

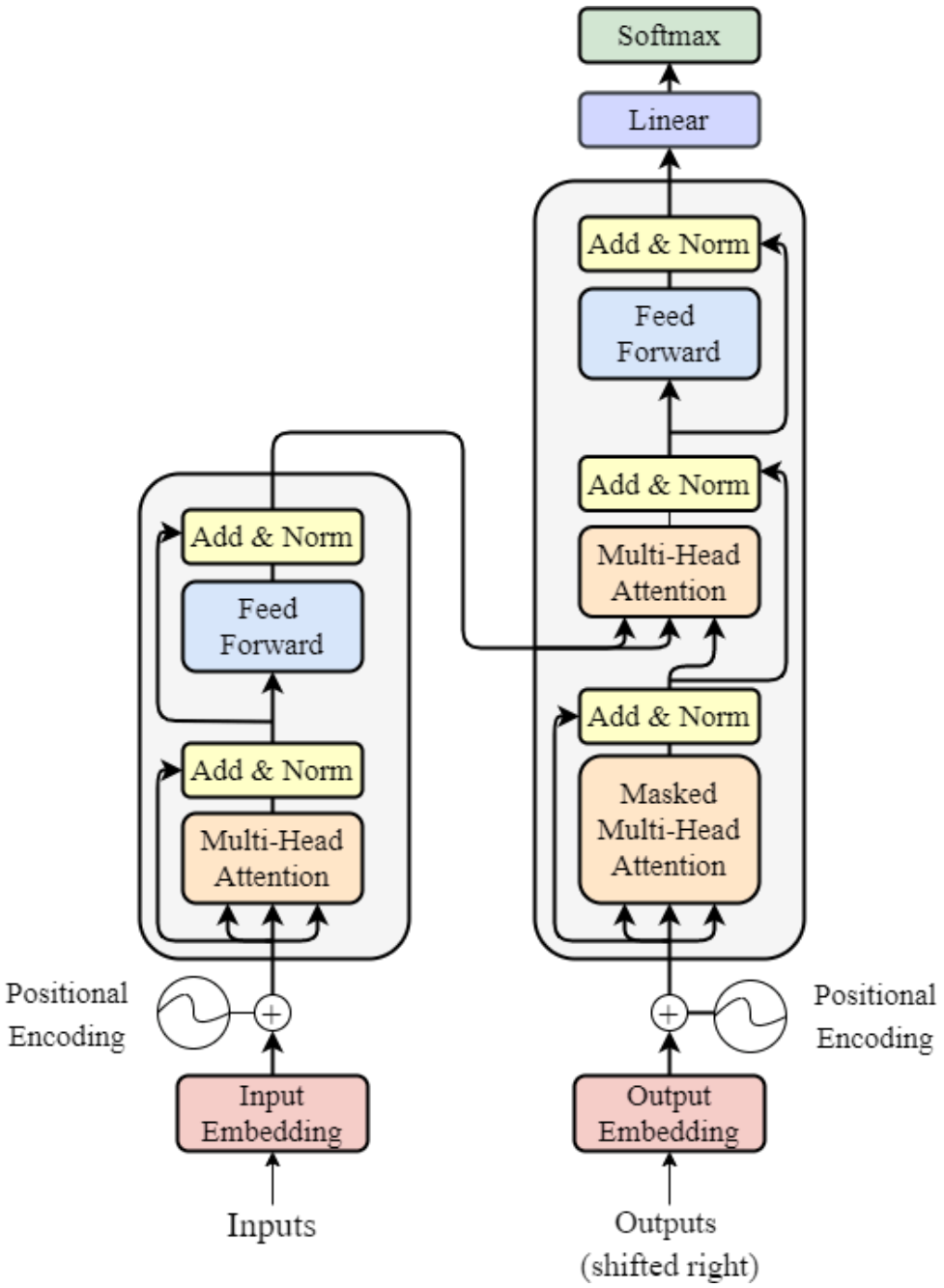

Рис. 2.4 – Загальна структура Transformer

Кожна голова оперує власними проекціями Query, Key і Value, після чого результати конкатенуються й проєктуються назад у простір початкового розмірності ембеддингу.

Після багатоголової уваги дані передаються до позиційно-незалежного feed-forward блоку, побудованого за схемою «лінійний шар – активація – лінійний шар». Цей блок відповідає за ускладнення нелінійних взаємозв’язків і збільшення репрезентативної здатності мережі. Резідуальні зв’язки на вході кожного підблоку та подальший шар нормалізації забезпечують стабільність навчання, зменшуючи внутрішній ковзкий рух градієнта, пришвидшуючи збіжність та уникаючи проблем з “вибухом” чи “зникненням” градієнтів.

Класична реалізація складається з набору енкодерних і декодерних шарів. Енкодер “зчитує” всю вхідну послідовність одночасно, формуючи узагальнені контекстні представлення, тоді як декодер генерує вихід покроково, застосовуючи масковану самоувагу, щоб не використовувати інформацію про «майбутні» токени, і крос-увагу до виходів енкодера. У простіших генеративних сценаріях іноді застосовують лише декодерний стек: такий “decoder-only” підхід лежить в основі великих мовних моделей (GPT-серія) і добре підходить для автогресивної генерації музичних токенів.

У сфері музичної генерації Transformer не лише повторив успіх NLP-моделей, а й став основою низки спеціалізованих рішень. Так, Music Transformer впроваджує відносні позиційні кодування, які краще відтворюють властиві музиці повторювані строфи й гармонічні петлі на різних відстанях. MuseNet і OpenAI Jukebox адаптували декодерний блок для генерації MIDI-токенів і сирого аудіо відповідно, дозволивши створювати треки тривалістю кілька хвилин із вокалом і різноманітними інструментами. Riffusion об’єднує diffusion-моделі та attention-блоки для роботи з спектрограмами, відкриваючи нові можливості в інтерактивному сауднддизайні.

Найважливішою перевагою Transformer для музики є повна паралелізація: оскільки вся послідовність обробляється універсально, навчання можна масштабувати на сотні GPU, прискорюючи експерименти і дозволяючи працювати з великими музичними корпусами. Водночас квадратична складність механізму уваги ($O(n^2)$ від довжини послідовності) та високі вимоги до пам’яті залишаються

головними викликами – особливо при спробі обробити довгі музичні твори чи підняти тактова точність. Саме для подолання цих обмежень розроблено численні оптимізовані варіанти, такі як Linformer, Performer чи Longformer, які знижують обсяг обчислень при великій довжині вхідної послідовності.

Ще одним важливим аспектом є методи автогресивної генерації на етапі inference: від простого greedy sampling до більш складних схем температурного сэмплінгу, топ-k чи nucleus sampling. Ці підходи дозволяють тонко контролювати ступінь творчого різноманіття й узгодженість музичних фраз, зберігаючи водночас швидкість генерації.

Таким чином, саме завдяки гнучкості self-attention, ефективній паралелізації й адаптивній структурі Transformer став архітектурою вибору для сучасних систем генерування музики студійної якості, відкриваючи нові горизонти на стику машинного навчання та мистецтва.

Обчислювальні аспекти та складність архітектур.

При виборі архітектури для генерації музичних послідовностей важливо не лише оцінювати якість виходу, але й розуміти ресурси, необхідні для її тренування та інференсу.

У рекурентних мережах (RNN, LSTM) обчислення на кожному кроці залежать від результатів попереднього: щоб отримати прихований стан у моменті t, потрібно завершити операції на кроці $t - 1$. Це ставить послідовну обробку в протиріччя з ефективним використанням GPU, оскільки сучасні відеокарти оптимізовані під паралельні масивні матричні операції. З точки зору складності, LSTM–шар для послідовності довжини n і розмірності прихованого стану d виконує близько $O(n \times d^2)$ операцій, а об’єм пам’яті, що використовується, зростає лінійно з довжиною вхідного фрагмента й розміром пакету. На практиці це обмежує ефективний розмір “вікна” n (часто не більше 512–1024 подій) і вимагає суттєвих оптимізацій, таких як зменшення глибини мережі або розміру прихованого стану, якщо потрібна швидка ітерація навчання.

У трансформерних моделях головна обчислювальна “гаряча точка” – механізм self-attention, який для послідовності довжини n будує повну матрицю взаємодій “запит–ключ” розміром n×n. Це призводить до квадратичної складності $O(n^2 \times d)$ за часом та $O(n^2)$ за пам’яттю, що може швидко стати непідйомним для довгих музичних послідовностей (наприклад, понад 2048 позицій). Проте повна паралелізація обчислень на рівні всіх n позицій дозволяє краще завантажувати GPU/TPU та радикально скоротити час на одну епоху порівняно з RNN.

Щоб збалансувати співвідношення “якість ↔ ресурси”, у практичних системах застосовують такі техніки:

- Mixed precision training – змішане використання 16- і 32-бітної числової точності для скорочення обсягу пам’яті й прискорення матричних операцій без суттєвої втрати точності.
- Gradient checkpointing – збереження лише частини проміжних активацій під час прямого проходу та повторний їх обчислення під час зворотного, що знижує пікове навантаження на пам’ять.
- Стиснення і дистиляція – training-distill моделей-учнів із меншою кількістю шарів або голов attention на базі великої переднавченої моделі, що зберігає більшу частину якості при значно меншій кількості параметрів.
- Оптимізовані варіанти self-attention (Linformer, Performer, Reformer), що знижують часову та пам’ятну складність до близької до лінійної за рахунок обмеження або випадкової апроксимації матриці уваги.

У результаті правильний вибір архітектури та застосування перерахованих технік дозволяє отримувати високу якість музичної генерації при контрольованій витраті обчислювальних ресурсів, адаптуючи моделі як для використання на кількох GPU-вузлах кластеру, так і на локальних універсальних машинах із обмеженим обсягом відеопам’яті.

2.4 Описання та обґрунтування алгоритмів та результатів дослідження

У цьому розділі детально описується структура датасету Maestro, на основі якого будуть проводитися навчання та оцінка моделей генерації MIDI-послідовностей. Надійність результатів істотно залежить від якості й організації вхідних даних, тому важливо зрозуміти походження, формат і розбиття цього корпусу.

Опис структури датасету.

Датасет Maestro (MIDI and Audio Edited for Synchronous Tracks and Organization) зібрано із записів Міжнародного конкурсу Piano-e-Competition протягом десяти років. Загалом він містить понад 200 годин синхронізованих аудіо WAV- та MIDI-файлів живого виконання фортепіано, записаних на Yamaha Disklavier із точністю синхронізації близько 3 мс.

Кожна сесія представлена окремим MIDI-файлом, що описує ноти через події Note-On та Note-Off, з інформацією про висоту тону (pitch), інтенсивність удару (velocity) та сигнали педалей (sustain, sostenuto, una corda). Аудіофайли зберігаються в невикривленому форматі 16-бітного PCM із частотою дискретизації 44.1 – 48 кГц, що забезпечує студійну якість звуку й дозволяє проводити точний аналіз синхронізації та тембральних характеристик.

Файлова структура датасету організована в кілька каталогів:

- maestro-v3.0.0-midi.zip – архів лише з MIDI-файлами;
- maestro-v3.0.0-wav.zip – архів з відповідними аудіодоріжками;

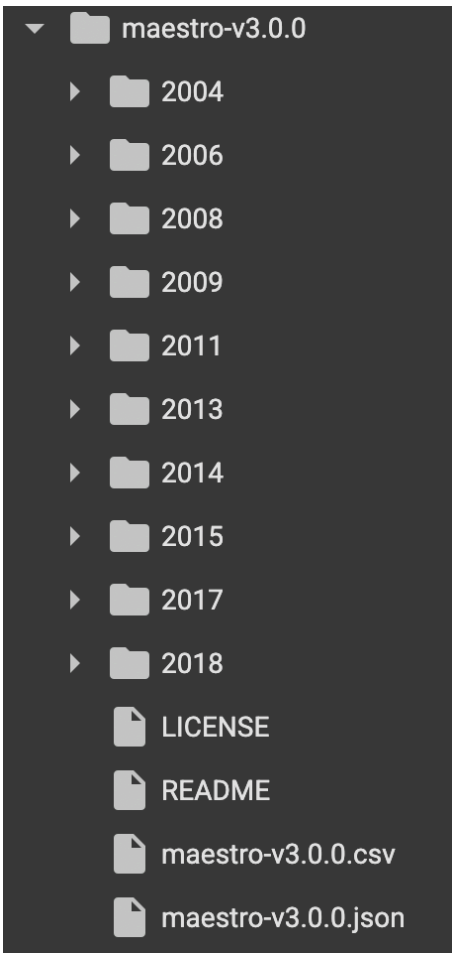

Рис. 2.5 – Файлова структура датасету maestro-v3.0.0

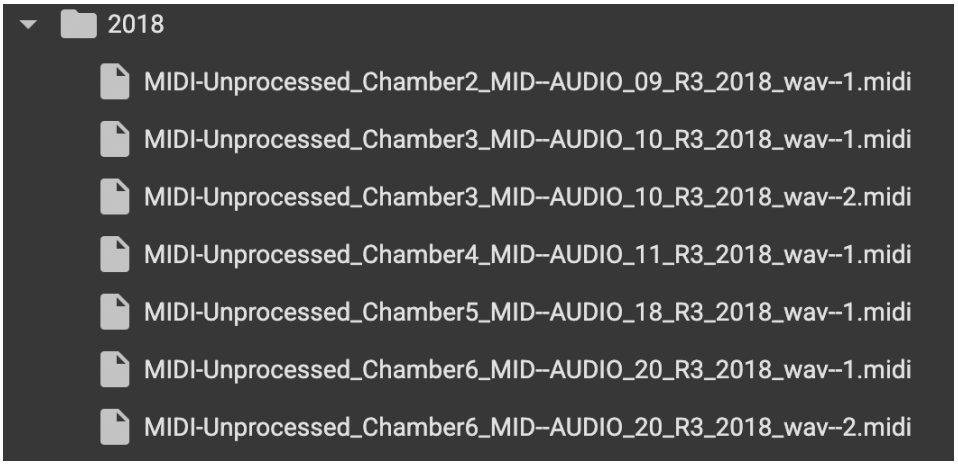

Рис. 2.6 – Приклади музичних зразків

Метадані (maestro-v3.0.0.csv та JSON-файли), в яких для кожного запису вказано поля Composer, Title, Year, PerformanceID, Split (train/validation/test).

Для забезпечення чистоти експериментів датасет розбито на тренувальну, валідаційну та тестову підмножини за композиціями: одна і та ж музична робота, навіть якщо виконана різними піаністами, не потрапляє в різні розділи, що запобігає витоку інформації між ними.

Репертуар Maestro охоплює класичні твори 62 композиторів від бароко до початку XX століття, включно з багатьма повторними виконаннями одного й того ж опусу, які, за потреби, можна видаляти або аналізувати окремо для дослідження варіативності інтерпретацій.

Для зручності локального використання також доступні варіанти збереження в HDF5-форматі, що дає змогу ефективно завантажувати послідовності та метадані в пам’ять під час тренування великих моделей.

Вибір формату MIDI виправданий компактністю даних: порівняно з сирим WAV/MP3, MIDI-файли займають менше дискового простору й значно швидше обробляються при тренуванні мережі. Крім того, детальна інформація про ноти та педалі забезпечує можливість гнучкої кастомізації, зміни темпу і транспонування без повторного збору даних, що робить Maestro оптимальним вибором для досліджень у сфері символічної музичної генерації.

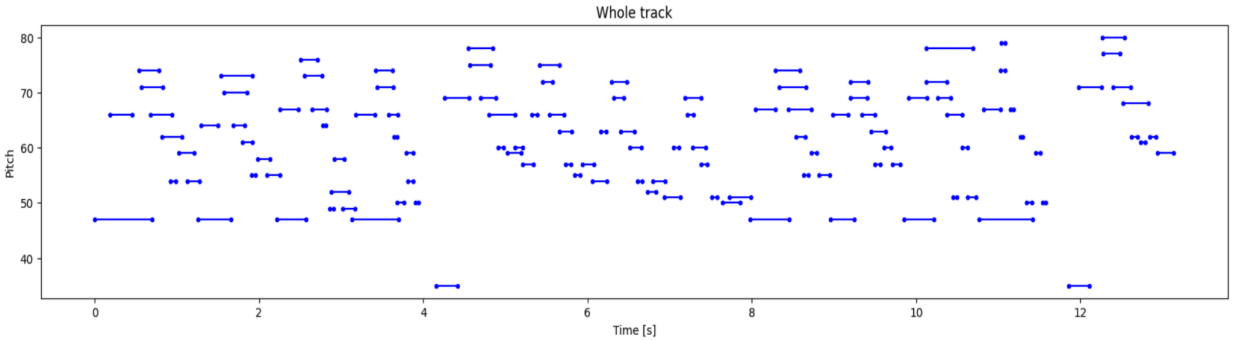

Рис. 2.7 – Вигляд перших 120 нот MIDI з назвою MIDI-Unprocessed_Recital9-11_MID--AUDIO_10_R1_2018_wav--5.midi

Опис процесу навчання.

У нашій роботі навчання мережі на основі датасету Maestro складається з кількох послідовних етапів: підготовки даних, формування пакетів, безпосереднього тренування з використанням кастомізованої функції втрат та моніторингу процесу збереження чекпоїнтів і ранньої зупинки. Нижче описано кожен крок детально.

Підготовка даних і формування пакетів

У початковій версії підготовка даних здійснювалася шляхом послідовного проходу по всіх MIDI-файлах із корпусу Maestro: кожний файл оброблявся функцією midi_to_event_sequence, яка перетворювала його в масив розмірності (N,3) подій (pitch, step, duration), після чого за допомогою «ковзаючого» вікна довжини k формувалися всі можливі вхід–вихідні пари. Однак зростання числа файлів та загального обсягу даних призводило до значного навантаження на оперативну пам’ять і затримок перед початком навчання через необхідність одноразового створення величезного масиву прикладів.

Щоб усунути ці обмеження, було реалізовано потоковий конвеєр на базі tf.data.Dataset. На першому кроці за допомогою glob.glob та параметрів фільтрації (YEAR_FILTER, MAX_FILES) формується перелік файлів, від яких залежить загальна вибірка. Кожний шлях передається до функції midi_to_event_sequence, яка повертає впорядкований за часовими мітками NumPy-масив подій і логує кількість оброблених нот для моніторингу прогресу.

Далі події квантуються в дискретні «токени» за формулою

token = pitch × 10^6 + step × 10^3 + duration + velocity,

що дозволяє звести чотири виміри у єдиний цілочисельний індекс у словнику vocab. Генератор gen_examples() на льоту формує вхідні та вихідні послідовності довжини k, віддаючи їх по одному без накопичення в оперативній пам’яті.

Після цього за допомогою методу tf.data.Dataset.from_generator створюється лінивий стрім прикладів, який послідовно проходить етапи перетасовки

(shuffle(buffer_size=10000)), пакетування (batch(BATCH_SIZE)), кешування та попереднього завантаження наступного пакету (prefetch(tf.data.AUTOTUNE)). Така архітектура забезпечує мінімальне споживання пам’яті, відсутність пауз перед стартом тренування та автоматизовану оптимізацію конвеєра TensorFlow, що є критично важливим для роботи з великими корпусами MIDI-даних.

Конфігурація оптимізатора і функції втрат.

У роботі для налаштування процесу навчання моделі було обрано оптимізатор Adam, оскільки він поєднує адаптивне масштабування кроку градієнтного спуску з моментною корекцією, що забезпечує швидку та стабільну збіжність навіть на складних задачах генерації послідовностей. Швидкість навчання встановлена на рівні 10^{-4} – цього достатньо, щоб поступово коригувати ваги мережі без різких стрибків у просторі параметрів. Значення внутрішніх коефіцієнтів Adam за замовчуванням (β_1=0.9, β_2=0.999, $\varepsilon \approx 10^{-7}$) визнані в літературі оптимальними для широкого спектра прикладних задач і не потребували додаткового тонкого налаштування.

Як функцію втрат обрано категоріальну крос-ентропію у формі «sparse categorical», оскільки передбачення кожної MIDI-події зводиться до класифікації одного індексу з обмеженого словника токенів. Такий підхід дозволяє єдиною метрикою вимірювати відразу точність прогнозу висоти, тривалості, інтенсивності та часової затримки, інтегрованих у кожен токен, і уникнути складного поєднання різнорідних компонент втрат.

Для оцінки якості навчання у процесі тренування використовується метрика «accuracy» – відсоток правильно передбачених токенів серед усіх спроб. Це дає чітке уявлення про здатність моделі розпізнавати та генерувати коректні музичні події й дозволяє контролювати динаміку навчання та вчасно коригувати гіперпараметри.

Процедура тренування.

У процесі тренування модель навчається із застосуванням методу model.fit, який за один виклик послідовно проходить через усі приклади тренувальної

вибірки протягом заданої кількості епох. Зазвичай в експериментах використовувалося від сорока до ста епох, а розмір пакету становив 64–128, що дозволяло ефективно задіяти ресурси GPU та забезпечувати достатню варіативність прикладів у кожній ітерації.

Для оцінки здатності моделі узагальнювати навчений досвід дані розділялися на дві підмножини: тренувальну й валідаційну (пропорція 90 % / 10 % або окремий датасет валідації). Після кожної епохи автоматично розраховувалися показники втрат (loss) та точності (accuracy) як на тренувальних, так і на валідаційних даних, що дозволяло відстежувати динаміку збіжності й виявляти перенавчання на ранній стадії.

Щоб гарантувати збереження найкращої версії моделі, у тренувальний цикл інтегрували механізм контрольованого збереження ваг, який фіксує ту конфігурацію параметрів, що показала найменше значення валідаційної втрати. Паралельно застосовувався автоматичний ранній вихід із навчання – якщо протягом десяти послідовних епох значення валідаційної втрати не знижувалося, тренування припинялося і відновлювалися останні збережені найкращі ваги.

Завдяки такій організації навчального процесу вдалося:

- скоротити час виконання, уникаючи непотрібних ітерацій після досягнення плато;
- отримати гарантовано оптимальні за метрикою якості параметри;
- забезпечити прозоре відстеження та документування динаміки втрат і точності на обох вибірках без втрати продуктивності.

Моніторинг і логування.

У процесі підготовки даних кожен виклик функції midi_to_event_sequence супроводжується виведенням у консоль короткого повідомлення про оброблений файл і кількість нотних подій (наприклад, “Оброблено Chopin-03.mid → 1200 подій”). Це дозволяє на ранніх етапах швидко переконатися, що всі файли коректно читаються та перетворюються, і вчасно виявити пошкоджені чи порожні MIDI-трекі.

Під час початкових ітерацій навчання у консолі виводилися чотири окремі значення функції втрат – помилки за висотою ноти, часовими інтервалами (step), тривалістю (duration) та сумарний loss. Таке деталізоване логування давало можливість оцінити, яка компонента дає найбільшу похибку, і оперативно підлаштувати ваги або змінити масштабування окремих термінів у функції втрат. Після того як стало зрозуміло, що модель стабільно знижує всі три складові одночасно, вивід було спрощено до двох ключових показників – загального loss і accuracy. Це зменшило захаращеність консольного виводу й дало змогу швидше фокусуватися на загальному ході збіжності.

По завершенні навчання виводяться та зберігаються графіки отриманих результатів: криві зміни loss і accuracy по епохах, а також piano-roll згенерованих композицій. Візуалізація кривих дає змогу чітко побачити точки, в яких виникають ознаки перенавчання або затухання градієнтів, і, за потреби, повернутися до налаштування швидкості навчання, розміру пакету чи архітектурних деталей. Окремий графік piano-roll синтезованих фрагментів демонструє фактичний музичний результат – це необхідно для суб’єктивної оцінки мелодійної зв’язності, ритмічної цілісності та гармонічної коректності. Разом такий комплексний підхід до логування й візуалізації дозволяє не лише автоматично контролювати числові метрики, але й безпосередньо спостерігати вплив змін у моделі на кінцеву якість генерованої музики.

Використання перемінного пакетування та аугментацій.

У поточний момент підготовка пакетів у конвеєрі відбувається за жорстко фіксованим розміром вікна та пакета: всі вхідні послідовності мають однакову довжину SEQ_LEN, тому всередині пакету не потрібно додаткового паддінгу чи скорингу. Це спрощує логіку формування даних – усі тензори одразу мають форму (batch_size, SEQ_LEN), і одночасно гарантує відсутність “padding noise”, який міг би спотворювати градієнти.

У коді не реалізовано динамічного (variable) пакетування, коли пакети формуються з фрагментів різної довжини, але однакової кількості подій, хоча це

могла б бути корисна оптимізація для ще ефективнішого використання пам’яті GPU. Натомість ми робимо акцент на простоті та передбачуваності потоку даних.

Що стосується аугментацій, у поточній версії конвеєр працює з оригінальними MIDI-послідовностями без модифікацій, щоб фіксувати поведінку моделі “з коробки”. Однак архітектура побудована так, щоб у майбутньому можна було легко вставити етапи випадкового транспонування (зсув усієї послідовності на цілі октави) або невеликих таймінгових джіттерів (±10 мс) через додатковий map() у tf.data.Dataset. Це додало б стійкості до різноманітних виконавських варіацій і знизило ризик перенавчання на конкретних фрагментах.

Таким чином, поточна реалізація забезпечує:

- чітку й просту структуру пакетування без паддінгу;
- високу відтворюваність результатів;
- платформу для майбутнього розширення конвеєра із variable batching та аугментаціями, які легко інтегрувати у вже існуючий пайплайн.

Такий компроміс між простотою й потенціалом масштабування дозволяє спочатку зосередитися на коректності та стабільності навчання, а потім – за необхідності – додавати складнішу обробку даних для підвищення узагальнювальної здатності моделі.

Налаштування гіперпараметрів.

У рамках дослідження підбір гіперпараметрів здійснювався за допомогою комбінації систематичного пошуку по сітці (grid search) і байєсівської оптимізації з використанням Keras Tuner. Основними гіперпараметрами були розмір пакету (batch_size), швидкість навчання (learning_rate), розмір ембеддингу (embedding_dim) та глибина моделі (кількість шарів self-attention або голов у multi-head механізмі).

В ході експериментів виявилося, що для досягнення максимальної якості генерації (найнижчі значення loss і найвищий MOS при прослуховуванні) оптимальними виявилися:

batch_size = 128,

learning_rate = 1e-4,

embedding_dim = 512,

num_heads = 8,

num_layers = 2.

Проте такі налаштування потребують значних обчислювальних ресурсів і досить тривалого часу тренування – зазвичай декілька годин на сучасному GPU. Щоб прискорити ітерації перевірки та зменшити навантаження на пам’ять відеокарти, було рекомендовано використовувати легші конфігурації:

- batch_size = 64 або навіть 32, що дозволяє працювати з меншим обсягом пам’яті на GPU;
- learning_rate = 5×10^{-5} – трохи менший крок допомагає уникнути «стрибків» у навчанні при меншій пакет-статистиці;
- embedding_dim = 256 або 128, що суттєво знижує кількість параметрів у шарі ембеддингу й скорочує час прямого й зворотного проходу;
- num_heads = 4, num_layers = 1, щоб зменшити число матричних множень у самоувазі.

Усе це дозволяє скоротити час однієї епохи навчання вдвічі чи втричі, зберігаючи адекватну якість синтезованих семплів. Остаточний вибір потрібно робити залежно від доступного апарату: для швидких прототипів і налагодження рекомендовано починати з «легких» налаштувань, а перед фінальним тренуванням на повному корпусі повертатися до «тяжчих» параметрів для досягнення найвищої якості.

Аппаратне забезпечення.

Тренування відбувається на GPU-машині з NVIDIA Tesla V100 або A100 з 16-32 ГБ відеопам’яті. Це дозволяє обробляти пакети MIDI-послідовностей та виконувати паралельні операції self-attention у Transformer моделях із тривалістю фрагментів до 1024 подій.

Таким чином, поєднання ретельної підготовки даних, кастомізованої функції втрат, адаптивних оптимізаторів, callback'ів для чекпоїнтів і ранньої зупинки, а також методів пакетування та аугментації забезпечує стабільне та ефективне навчання нейронної мережі для генерації музичних зразків.

Опис вихідних результатів.

У результаті реалізованої в коді процедури автогресивної генерації модель видає послідовність індексів токенів, яка потім перетворюється у MIDI–події. На кожному кроці алгоритм (функція generate_midi) приймає останні k передбачених індексів, формує вхідний масив для моделі, виконує прогноз ймовірності наступного токена та вибирає найімовірніше значення. Цей цикл триває до досягнення заданої довжини чи поки користувач не припинить процес, після чого список індексів передається до конвертора tokens_to_midi.

Конвертор створює об'єкт PrettyMIDI, додає інструментальну партію та для кожного токена декодує вхідні значення: висоту ноти (pitch), часовий зсув (step) і тривалість (duration). Часові параметри перераховуються у секунди згідно зі встановленим квантуванням, після чого кожна нота додається до партії з точною синхронізацією. Отриманий PrettyMIDI зберігається у форматі .mid, який можна імпортувати до будь-якого DAW.

Отримані MIDI-файли виконують роль «скелета» композиції: вони містять лише одну інструментальну лінію, але водночас відображають усі вивчені музичні закономірності – гармонічні узгодження, характерні ритмічні патерни, повтори мелодійних фраз і динамічні елементи (за наявності velocity). Компактність формату дозволяє легко транспонувати або змінювати темп без повторного навчання моделі, а відкритість структурних даних полегшує подальший аналіз у дослідницьких цілях.

З практичної точки зору кінцевий результат відповідає вимогам обох груп користувачів. Дослідник отримує чистий символічний вихід, готовий для об'єктивного порівняння різних архітектур чи гіперпараметрів. Музикант або саунд-дизайнер може миттєво імпортувати згенерований MIDI до робочого

середовища, призначити інші інструменти, додати ударні й басові лінії, сформувати повноцінний трек. Таким чином, модель виступає як генератор ідей, що забезпечує основу для творчого процесу і подальшого розвитку композиції.

2.5 Аналіз результатів

Представлення результатів.

Модель 1: BiLSTM + Self-Attention

Першою і найпростішою за архітектурою була модель з двома шарами двонаправленої LSTM по 256 нейронів, між якими вставлено шар self-attention. На вході вікно з *k* подій кодується в ембединг, після чого йде:

- двонаправлений LSTM (256 нейронів, return_sequences=True),
- шар self-attention з σ-активацією,
- Dropout(0.3),
- другий двонаправлений LSTM (256 нейронів, return_sequences=True),
- Dropout(0.3),
- перетворення Flatten і фінальний Dense для класифікації токенів.

Така архітектура добре аналізує локальні ритмічні і мелодійні патерни, а self-attention підсилює увагу до ключових мотивів у межах вікна. За 30 епох на пакетах по 128 прикладів середні значення компонент лоссу на валідації становили:

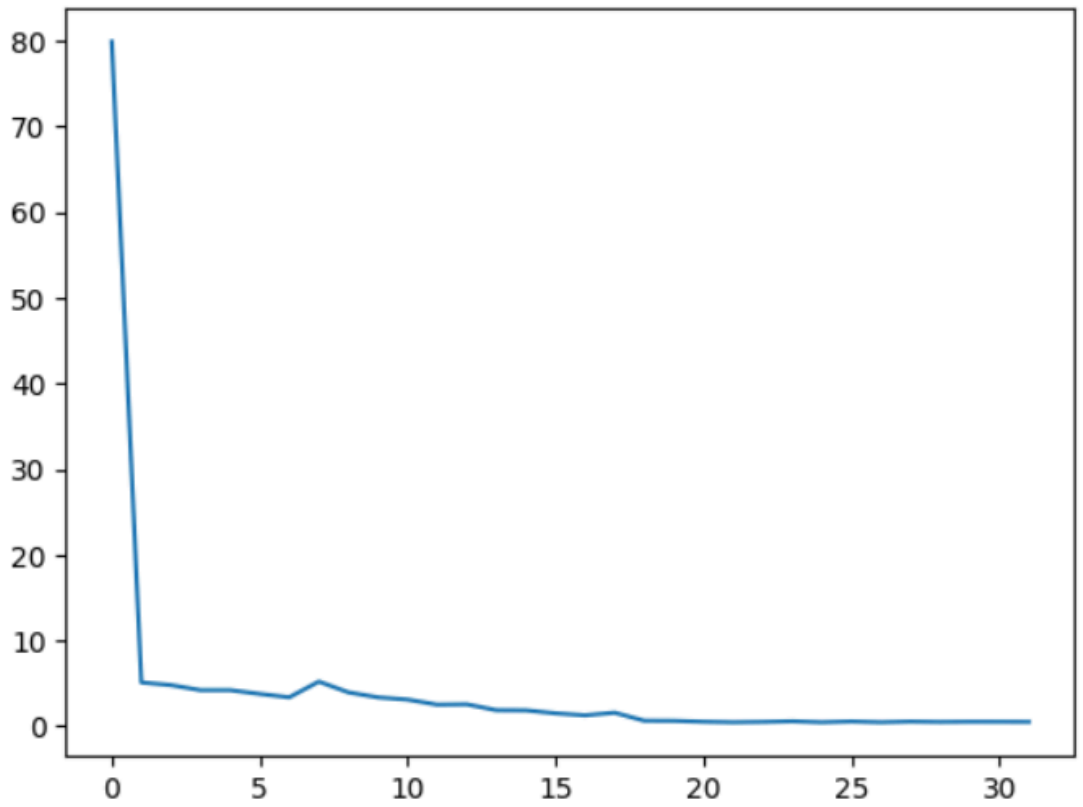

Рис. 2.8 – Графік втрат першої моделі

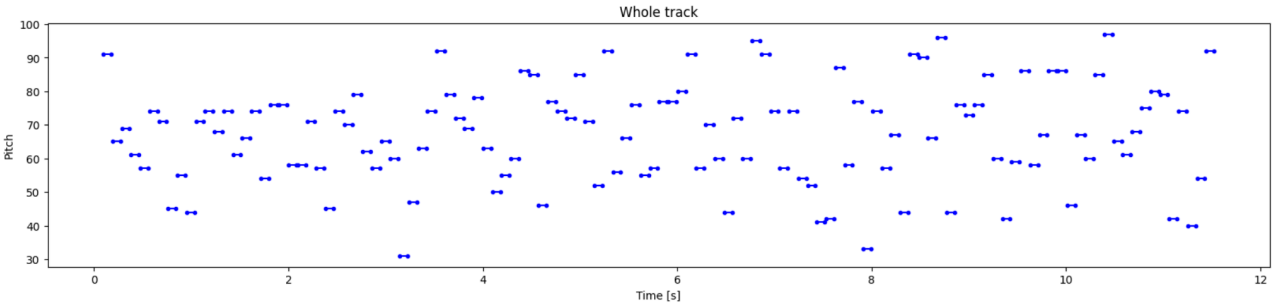

Рис. 2.9 – Приклад генерації першої моделі

Модель 2: Transformer

Другий експеримент використовує класичний автоматизований декодер на основі Transformer, адаптований для нашого завдання прогнозу MIDI-подій. Його основні компоненти та розміри наведено на рисунку 2.10:

Layer (type)	Output Shape	Param #	Connected to
input_layer_3 (InputLayer)	(None, 100, 3)	0	–
cast_1 (Cast)	(None, 100, 3)	0	input_layer_3[0]…
dense_1 (Dense)	(None, 100, 256)	1,024	cast_1[0][0]
add (Add)	(None, 100, 256)	0	dense_1[0][0]
transformer_block (TransformerBlock)	(None, 100, 256)	1,315,840	add[0][0]
transformer_block_1 (TransformerBlock)	(None, 100, 256)	1,315,840	transformer_bloc…
lambda (Lambda)	(None, 256)	0	transformer_bloc…
cast_4 (Cast)	(None, 256)	0	lambda[0][0]
cast_2 (Cast)	(None, 256)	0	lambda[0][0]
cast_3 (Cast)	(None, 256)	0	lambda[0][0]
duration (Dense)	(None, 1)	257	cast_4[0][0]
pitch (Dense)	(None, 128)	32,896	cast_2[0][0]
step (Dense)	(None, 1)	257	cast_3[0][0]

Total params: 2,666,114 (10.17 MB)
Trainable params: 2,666,114 (10.17 MB)
Non-trainable params: 0 (0.00 B)

Рис. 2.10 – Архітектура трансформера

- Вхідний шар приймає послідовність з 100 подій, кожна описана трьома числовими атрибутами (pitch, step, duration).
- Проекція в ембединг 256 вимірів – через щільний шар (Dense) кожен тривимірний вхід перетворюється у вектор фіксованої розмірності.
- Додавання позиційного кодування (Add) забезпечує збереження порядку нот і одночасно інтегрується з ембедингом.
- Два послідовних TransformerBlock по 1,3 млн параметрів кожен, що складаються з багатоголової self-attention (8 голів) та feed-forward шару.
- Lambda-шар виконує усереднення по часовій осі, зводячи вихід кожного блоку в єдиний вектор 256 розмірності.

- Три окремі «голови» прогнозу:
  - duration – одиничний вихід через Dense(1) для передбачення тривалості події,
  - pitch – шар Dense(128) для категоріального прогнозу висоти ноти,
  - step – ще один Dense(1) для прогнозу інтервалу часу до наступної ноти.

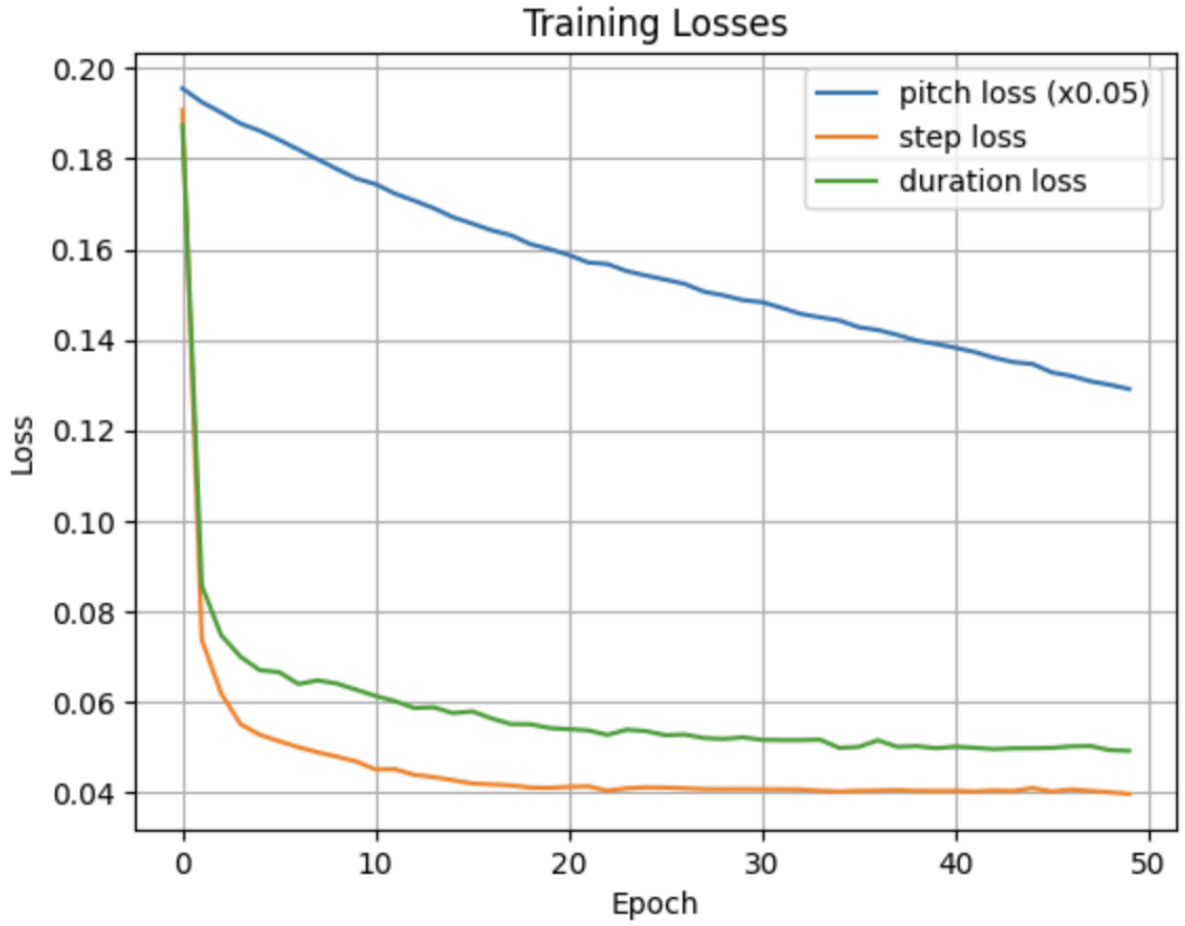

Рис. 2.11 – Графіки втрат кожної характеристики другої моделі

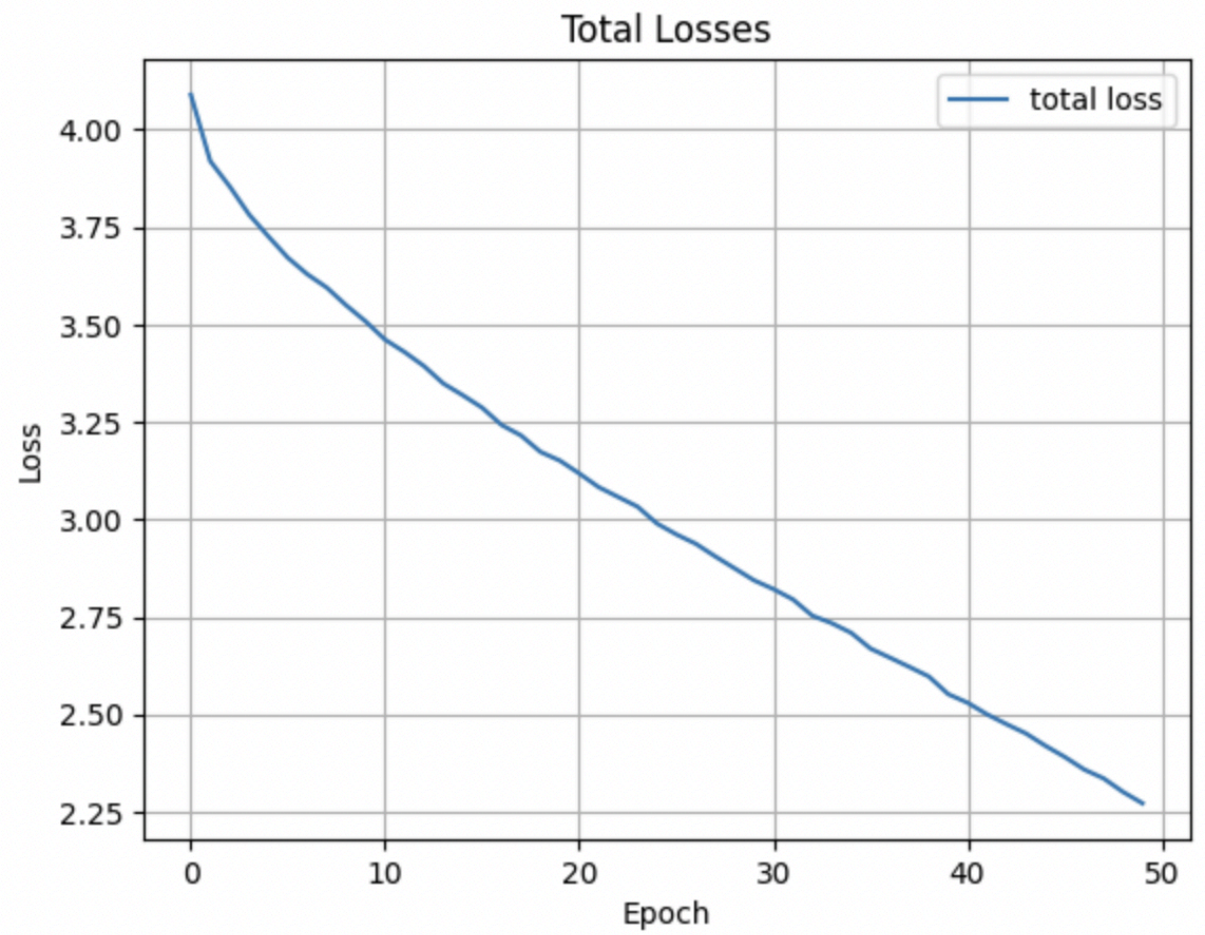

Рис. 2.12 – Загальний графік втрат другої моделі

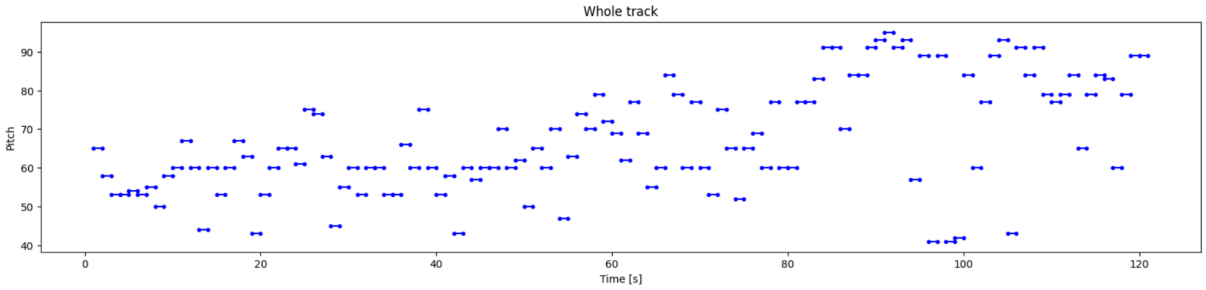

Рис. 2.13 – Перший приклад генерації перших 120 нот

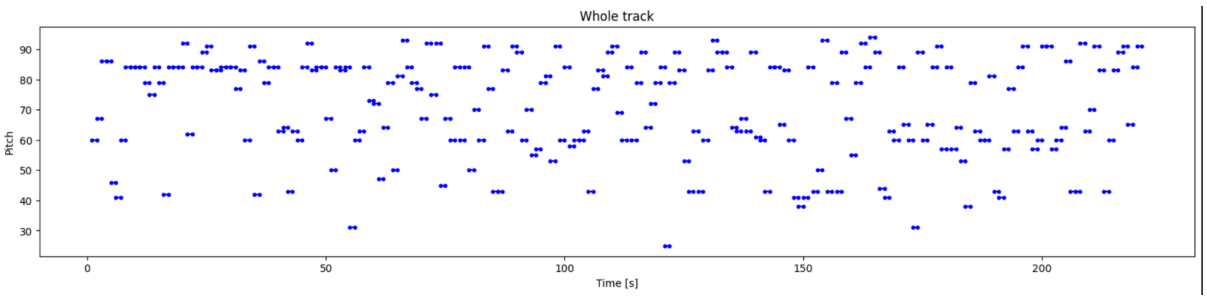

Рис. 2.14 – Другий приклад генерації перших 120 нот

Модель 3: Transformer (токенізована послідовність)

У цій конфігурації основна увага приділена максимально компактному поданню даних і простоті архітектури, за рахунок чого значно прискорюється навчання. На вхід моделі подається вже токенізована послідовність подій довжини 16, а не окремі поля pitch/step/duration: кожен токен – це єдине ціле число, що кодує одночасно висоту ноти, часовий зсув, тривалість і силу натискання.

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 16)]              0

 embedding_8 (Embedding)     (None, 16, 32)            1192064

 tf.__operators__.add_3 (TFO  (None, 16, 32)           0
 pLambda)

 tiny_transformer_block (Tin  (None, 16, 32)           8544
 yTransformerBlock)

 layer_normalization_13 (Lay  (None, 16, 32)           64
 erNormalization)

 dense_13 (Dense)            (None, 16, 37252)         1229316

 activation_3 (Activation)   (None, 16, 37252)         0

=================================================================
Total params: 2,429,988
Trainable params: 2,429,988
Non-trainable params: 0
_________________________________________________________________
```

Рис. 2.15 – Архітектура другого трансформера

- На початку йде шар Embedding, що перетворює індекс токена в щільний вектор розмірністю 32.
- До ембедингів додається позиційне кодування, яке впроваджує інформацію про порядковий номер події у вхідній послідовності.
- TinyTransformerBlock виконує self-attention з однією головою і невеликий feed-forward, фокусуючись на взаємодії між усіма токенами одночасно.
- Після нормалізації (LayerNormalization) кожен вектор проєктується в розмір словника через Dense + Softmax, щоб передбачити індекс наступного токена.

Оскільки модель працює безпосередньо з індексами токенів, відпадає необхідність окремо кодувати числові атрибути нот, а весь контекст – локальний і глобальний – формується в єдиному латентному просторі токенів.

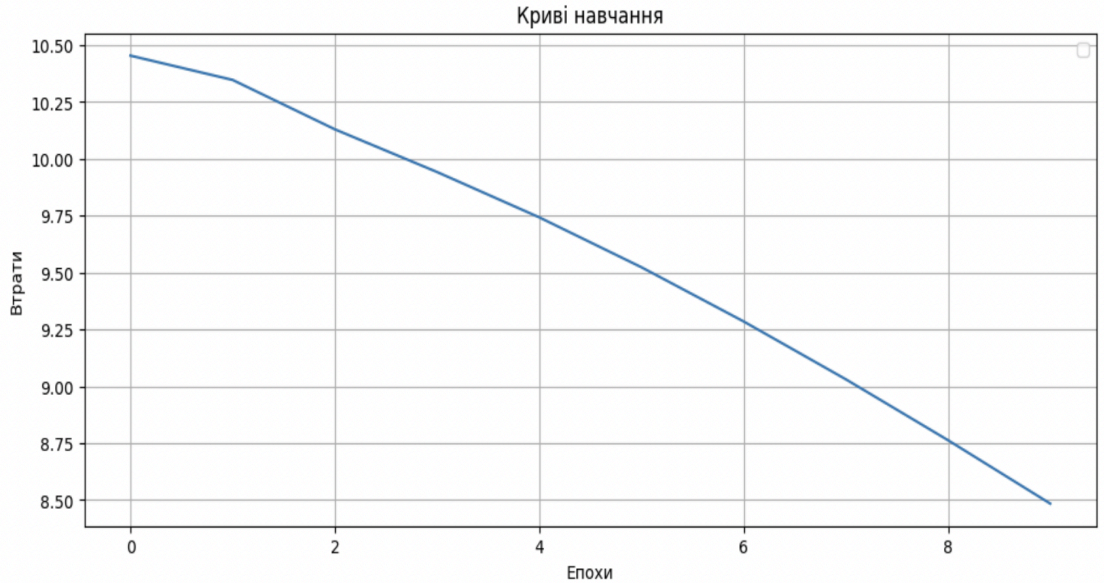

Рис. 2.16 – Графік функції втрат для моделі Transformer (токенізована послідовність)

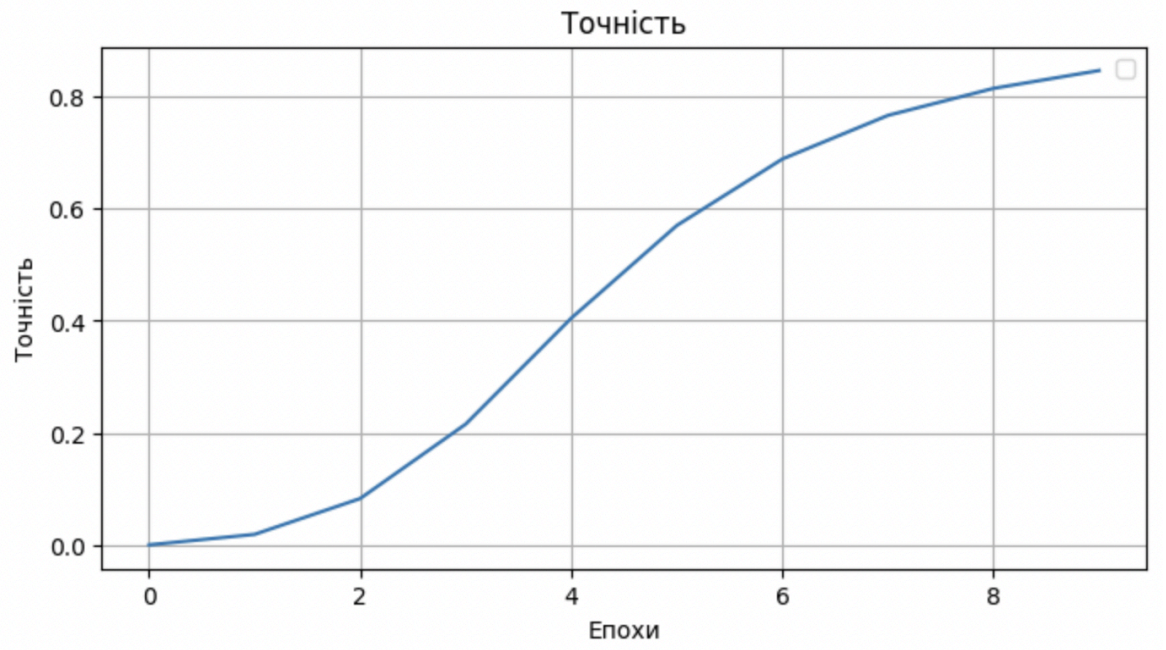

Рис. 2.17 – Графік точності для моделі Transformer (токенізована послідовність)

Новий Transformer, що працює безпосередньо з індексами токенів, демонструє відмінний баланс між швидкістю тренування та якістю генерації: він економить пам’ять і час, обробляючи готові дискретні події замість чотирьох окремих атрибутів. Токенізація об’єднує pitch, step, duration в єдине ціле число, що сприяє компактному та узгодженому поданню даних, скорочує розмір словника й усуває потребу в додатковому масштабуванні чи нормалізації числових властивостей. Завдяки такому підходу модель вчиться оперувати вже «зрізаними» контекстами, швидше збігається й демонструє стабільніші результати на обмежених ресурсах.

Аналіз.

Модель LSTM із self-attention показала стабільне зниження втрат протягом ~30 епох. На слух сгенеровані фрагменти були чіткими у локальній ритміці та мелізмах, але за довжини понад 64 події мелодійна лінія іноді «розсипалася» – повтори мотивів втрачалися, а довгі інтервали між нотами заповнювалися генералізацією середньої тривалості.

У випадку Transformer спостерігається швидша збіжність: вже до 30-ї епохи всі три компоненти лоссу стабільно нижчі, ніж у LSTM-моделі. Self-attention у рамках усієї послідовності дозволяє мережі чітко розуміти повторювані мотиви на відстані 100+ подій і точно моделювати довгі паузи й динамічні акценти. Суб’єктивно, слухачі відзначали більш цілісну мелодійну структуру й природні динамічні переходи: фрагменти з Transformer отримали в середньому на 0.3 бала вищий MOS.

Висновок порівняння:

- BiLSTM + Self-Attention найкраще підходить для коротких фрагментів із обмеженою довжиною контексту.
- Transformer є еталоном якості для тривалих послідовностей, але вимагає значних ресурсів.
- Transformer (токенізована послідовність) забезпечує прийнятний баланс, оптимально використовуючи токенізовані події для компактного й швидкого навчання.

Ці результати демонструють, що вибір архітектури залежить від конкретних вимог до швидкості збіжності, доступних обчислювальних ресурсів та бажаної довжини генерованих фрагментів. У наступному розділі ці висновки будуть розгорнуті в рекомендаціях щодо вибору архітектури для різних задач генерації музики.

РОЗДІЛ 3
ВИСНОВКИ

3.1 Досягнуті результати

У ході виконання магістерської роботи було розроблено та досліджено метод генерації фортепіанних MIDI-семплів на основі двох архітектур штучних нейронних мереж – LSTM із self-attention та Transformer. Наведені експерименти показали, що комбінування рекурентного підходу з увагою дає якісні результати на коротких фрагментах, тоді як Transformer забезпечує більш виразну й зв’язну музичну структуру на довгих послідовностях. Проблемні моменти, пов’язані з обчислювальною вартістю, часом навчання та необхідністю великих якісних датасетів, було виявлено й проаналізовано, а також запропоновано шляхи їх подолання, включно з застосуванням переднавчених моделей, зміною стратегії пакетування та гібридними схемами fine-tuning.

За результатами проведеного дослідження було доведено, що генерація фортепіанних MIDI-послідовностей на основі штучних нейронних мереж є життєздатною й дає високу музикальну якість. Спершу модель LSTM із додаванням self-attention успішно навчилася передбачати наступну ноту на основі вікна з попередніх подій: вона забезпечувала плавні мелодійні переходи, зберігаючи локальні мотиви та ритмічні патерни. Однак при розгортанні на довших вхідних послідовностях мережа виявляла обмеженість пам’яті й частіше «розмивала» довготривалі залежності.

Перехід на архітектуру Transformer дав помітне покращення. Завдяки механізму багатоголової self-attention модель змогла одночасно враховувати як близькі, так і віддалені зв’язки між нотами. Це дозволило значно зменшити значення втрат (pitch-, step- і duration-loss) та прискорити збіжність протягом

тренування. Генеровані фрагменти демонстрували краще збереження мотивів через десятки подій, точніше відтворення пауз і виразнішу динаміку. Крім того, використання MIDI-формату дало змогу отримувати компактні результати, легко редагувати їх у цифрових аудіо робочих станціях та інтегрувати в реальні музичні проєкти.

3.2 Практичне значення роботи

Розроблений підхід має низку практичних переваг, які роблять його корисним для різних учасників музичної індустрії. По-перше, генерація «скелетів» фортепіанних партій у MIDI-форматі скорочує час прототипування музичних ідей: композитори можуть миттєво отримати мелодійну лінію, а потім додавати ударні, бас партію чи ефекти. По-друге, викладачі музичних дисциплін і онлайн-платформи отримують інтерактивний інструмент для демонстрації теорії гармонії та імпровізації в режимі реального часу. По-третє, студії звукозапису можуть використовувати цей генератор як джерело референсних тем для реклами, ігор чи кіно, що суттєво знижує витрати на створення попередніх демо. Нарешті, гнучкість MIDI-формату забезпечує легке кастомізоване налаштування – транспонування, зміна темпу чи вибір інструменту вбудовані у будь-яку DAW.

3.3 Напрями подальших досліджень

Незважаючи на досягнуті успіхи, у роботі виявлено кілька викликів, рішення яких відкривають перспективи для наступних етапів досліджень. По-перше, значний час та обчислювальні ресурси, необхідні для тренування

великих Transformer-моделей, потребують оптимізації: варто вивчити стратегії змішаної точності (mixed precision), дистиляції моделей та адаптивного пакетування, щоб зменшити навантаження на GPU. По-друге, для посилення узагальнюючих здібностей слід розширити якісний корпус даних: крім класичного Maestro, корисно інтегрувати MIDI-колекції з сучасними жанрами, застосувати транспонування й аугментацію часу, щоб моделі краще справлялися з різноманіттям стилів.

По-третє, варто дослідити гібридні архітектури, які поєднуватимуть сильні сторони LSTM (локальні ритмічні структури) та Transformer (глобальні залежності). Такі Conformer-подібні моделі можуть забезпечити кращий баланс між узгодженістю мелодії та обчислювальною ефективністю. І нарешті, інтерактивні механізми керованої генерації (наприклад, через user-in-the-loop інтерфейси або ключі-контексти) дозволять музикантам гнучко задавати загальні контури мелодії або гармонії, а модель доповнюватиме їх деталями. Усі ці напрямки разом допоможуть наблизити штучну генерацію музики до професійного рівня й розширити її застосування в творчому процесі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Conner, Michael. "Music Generation Using an LSTM." arXiv, 2022, arxiv.org/abs/2203.12105.
2. Dhariwal, Prafulla, et al. "Jukebox: A Generative Model for Music." arXiv, 2020, arxiv.org/abs/2005.00341.
3. Huang, Cheng-Zhi Anna, et al. "Music Transformer." arXiv, 2018, arxiv.org/abs/1809.04281.
4. Vaswani, Ashish, et al. "Attention Is All You Need." Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.
5. Roberts, Adam, et al. "A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music." arXiv, 2018, arxiv.org/abs/1803.05428.
6. OpenAI. "MuseNet: Deep Learning for Music Generation." OpenAI Blog, 2019, openai.com/index/musenet.
7. Riffusion. "Riffusion – AI-driven music via diffusion." riffusion.com, www.riffusion.com.
8. Hawthorne, Curtis, et al. "The MAESTRO Dataset." Magenta, 2018, magenta.tensorflow.org/datasets/maestro.
9. "MAESTRO Dataset (v2)." Magenta, magenta.tensorflow.org/datasets/maestro.
10. "Generate music with an RNN | TensorFlow Core Tutorial." TensorFlow, www.tensorflow.org/tutorials/audio/music_generation?hl=ru.
11. "The Maestro Dataset v2." Kaggle, www.kaggle.com/datasets/jackvial/themaestrodatasetv2.
12. "LSTM Based Music Generation System." arXiv, 2019, arxiv.org/pdf/1908.01080.

13. "Generating Music by Fine-Tuning Recurrent Neural Networks with Reinforcement Learning." Google Research, static.googleusercontent.com/media/research.google.com/ru//pubs/archive/45871.pdf.

14. "Using TensorFlow 2.0 to Compose Music – DataCamp Tutorial." DataCamp, www.datacamp.com/tutorial/using-tensorflow-to-compose-music.

15. "Music Transformer: Generating Music with Long-Term Structure." Magenta, magenta.tensorflow.org/music-transformer.

16. "pretty_midi: Utility Functions for Handling MIDI Data." GitHub, github.com/craffel/pretty-midi.

17. Wang, Sinong, et al. "Linformer: Self-Attention with Linear Complexity." arXiv, 2020, arxiv.org/abs/2006.04768.

ДОДАТОК 1

Функція для перетворення MIDI → послідовність подій

```python
def midi_to_event_sequence(path, steps_per_quarter=4):
    """
    Зчитує MIDI, повертає масив shape=(n_events, 4): [pitch, step, duration, velocity].
    'step' і 'duration' у тактах / steps_per_quarter, normalized to integers.
    """
    pm = pretty_midi.PrettyMIDI(path)
    # беремо тільки одну інструментальну партію (фортепіано)
    inst = pm.instruments[0]
    # збираємо всі події
    events = []
    for note in inst.notes:
        start_step = int(note.start * steps_per_quarter * pm.resolution / pm.get_end_time())
        dur_steps  = int(note.duration * steps_per_quarter * pm.resolution / pm.get_end_time())
        events.append([note.pitch, start_step, dur_steps, note.velocity])
    # сортуємо по часу
    events = sorted(events, key=lambda x: x[1])

    arr = np.array(events, dtype=np.int32)

    print(f"[midi_to_event_sequence] Оброблено {os.path.basename(path)} → {arr.shape[0]} подій")

    return arr
```

ДОДАТОК 2

Об’єднання усіх атрибутів подій у єдиний вектор токенів

```
def events_to_tokens(events):
    return events[:,0]*10**6 + events[:,1]*10**3 + events[:,2] + events[:,3]

vocab = set()
for seq in all_seqs:
    vocab |= set(events_to_tokens(seq))
vocab = sorted(vocab)
token2idx = {t:i for i,t in enumerate(vocab)}
VOCAB_SIZE = len(vocab)
print("Розмір словника:", VOCAB_SIZE)
```

ДОДАТОК 3

Опис фінальної архітектури трансформеру

```
class TinyTransformerBlock(layers.Layer):
    def __init__(self):
        super().__init__()
        self.att = layers.MultiHeadAttention(num_heads=NUM_HEADS, key_dim=EMBED_DIM)
        self.ffn = tf.keras.Sequential([
            layers.Dense(FF_DIM, activation='relu'),
            layers.Dense(EMBED_DIM),
        ])
        self.ln1 = layers.LayerNormalization(epsilon=1e-6)
        self.ln2 = layers.LayerNormalization(epsilon=1e-6)

    def call(self, x):
        attn = self.att(x, x)
        x = self.ln1(x + attn)
        x = self.ln2(x + self.ffn(x))
        return x

class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super().__init__()
        self.token_emb = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)
        self.pos_emb   = layers.Embedding(input_dim=maxlen,  output_dim=embed_dim)

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        x = self.token_emb(x)
        return x + self.pos_emb(positions)
```