ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Автоматизація процесу мастерингу аудіо з використанням алгоритмів машинного навчання»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

_____ Євген ЛАБУН
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-61
Євген ЛАБУН

Керівник: Оксана ЗОЛОТУХІНА
к.т.н., доцент

Рецензент: _____
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Лабуну Євгену Миколайовичу

1. Тема кваліфікаційної роботи: «Автоматизація процесу мастерингу аудіо з використанням алгоритмів машинного навчання»

керівник кваліфікаційної роботи Оксана ЗОЛОТУХІНА, к.т.н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, вихідний код програм, налаштування гіперпараметрів, приклади спектрограм оброблених та необроблених аудіотреків.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз предметної області.
2. Алгоритми машинного навчання для обробки аудіо.
3. Розробка та впровадження системи.

5. Перелік ілюстративного матеріалу: *презентація*

1. Тема, мета і предмет дослідження.
2. Актуальність роботи та огляд обраних моделей нейромереж.
3. Недоліки обраних моделей нейромереж.
4. Принцип роботи LSTM
5. Принцип роботи U-Net
6. Способи представлення звуку.
7. Адаптація U-Net (Wave-U-Net) для роботи з часовими проміжками.
8. Проблеми створення датасету.
9. Вплив налаштування гіперпараметрів на фінальний результат та споживання ресурсів.
10. Практичний результат та порівняння.
11. Висновки.

6. Дата видачі завдання«16» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10-20.10.2024	
2	Аналіз наявних програмних рішень мастерингу аудіо	21.10-25.10.2024	
3	Вибір підходів та моделей нейронних мереж для створення системи мастерингу аудіо	26.10-03.11.2024	
4	Розробка системи автоматизації мастерингу аудіо	04.11-10.11.2024	
5	Створення датасету для навчання моделей	25.10-09.11.2024	
6	Навчання моделей	11.11-16.11.2024	
7	Тестування та оцінка результатів	17.11-22.11.2024	
8	Оформлення роботи: вступ, висновки, реферат	12.11-21.11.2024	
9	Розробка демонстраційних матеріалів	23.11-24.11.2024	
10	Попередній захист роботи	25.11.2024	

Здобувач вищої освіти _____ Євген ЛАБУН
(підпис)

Керівник
кваліфікаційної роботи _____ Оксана ЗОЛОТУХІНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 75 стор., 2 табл., 26 рис., 48 джерел.

Мета роботи – спрощення та автоматизація процесу мастерингу аудіо треків за допомогою натренованої моделі

Об'єкт дослідження – процес покращення якості аудіо

Предмет дослідження – методи покращення якості аудіо за допомогою машинного навчання.

У роботі розглянуто сучасні підходи до автоматизації мастерингу аудіо треків із використанням алгоритмів машинного навчання. Проведено аналіз існуючих рішень, таких як SoundCloud AI Mastering, Landr, Masterchannel та Izotope Ozone, визначено їх основні переваги та недоліки. Огляд методів включає згорткові нейронні мережі (CNN), рекурентні нейронні мережі (RNN), довготривалу короткочасну пам'ять (LSTM), архітектуру U-Net та її адаптацію для роботи з аудіосигналами (Wave-U-Net).

Особливу увагу приділено процесу створення датасету для навчання моделей. Ідеальний датасет повинен складатися з великої кількості пар аудіо – в хорошій і поганій якості. Важливим моментом є те, що треки поганої якості повинні бути створені вручну – кожен трек високої якості має бути оброблений певним чином для досягнення ефекту поганого мастерингу. Проте, якщо пар треків недостатньо, датасет допускається просто доповнити треками у високій якості до яких при обробці будуть застосовані аугментації. Особливо важливим етапом формування датасету є процес консолідації треків – приведення їх до одного формату, частоти дискретизації, канальності, бітності та нарізання на сегменти однієї довжини.

Також досліджено вплив гіперпараметрів на якість мастерингу. Проведено порівняння результатів роботи прототипів із існуючими комерційними

рішеннями. Експериментально підтверджено, що розроблені системи забезпечують високу якість обробки аудіо та можуть бути використані для покращення звучання треків різних жанрів. Результати роботи демонструють практичну цінність розроблених рішень для аудіоінженерів, музикантів та користувачів цифрових аудіо станцій (DAW). Розроблені прототипи дозволяють автоматизувати рутинні процеси мастерингу, підвищуючи якість звуку та знижуючи витрати часу.

КЛЮЧОВІ СЛОВА: АУДІО, МУЗИКА, ГЛИБОКЕ НАВЧАННЯ, МАСТЕРИНГ, НЕЙРОННІ МЕРЕЖІ, ОБРОБКА АУДІО, LSTM, U-NET, АУГМЕНТАЦІЯ, МАШИННЕ НАВЧАННЯ, СПЕКТРОГРАММА, DAW

ABSTRACT

Text part of the master's qualification work: 75 pages, 26 pictures, 2 tables, 48 sources.

The purpose of the work – to simplify and automate the process of mastering audio tracks using a trained model.

Object of research – the process of improving audio quality.

Subject of research – methods of improving audio quality using machine learning.

The paper considers modern approaches to automating audio track mastering using machine learning algorithms. Existing solutions, such as SoundCloud AI Mastering, Landr, Masterchannel, and Izotope Ozone, are analyzed, and their main advantages and disadvantages are identified. The review of methods includes convolutional neural networks (CNN), recurrent neural networks (RNN), long-term short memory (LSTM), U-Net architecture and its adaptation to work with audio signals (Wave-U-Net).

Special attention is paid to the process of creating a dataset for model training. An ideal dataset should consist of a large number of audio pairs - in good and bad quality. The important point is that the poor quality tracks should be created manually - each high quality track should be processed in a certain way to achieve the effect of poor mastering. However, if there are not enough pairs of tracks, the dataset can be simply supplemented with high quality tracks to which augmentations will be applied during processing. A particularly important stage in the formation of a dataset is the process of consolidating tracks - bringing them to the same format, sampling rate, channel, bit rate, and cutting them into segments of the same length.

The influence of hyperparameters on mastering quality is also investigated. The results of the prototypes were compared with existing commercial solutions. It has been experimentally confirmed that the developed systems provide high quality audio

processing and can be used to improve the sound of tracks of various genres. The results of the work demonstrate the practical value of the developed solutions for audio engineers, musicians, and users of digital audio workstations (DAWs). The developed prototypes allow automating routine mastering processes, improving sound quality and reducing time costs.

KEYWORDS: AUDIO, MUSIC, DEEP LEARNING, MASTERING, NEURAL NETWORKS, AUDIO PROCESSING, LSTM, U-NET, AUGMENTATION, MACHINE LEARNING, SPECTROGRAM, DAW

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

1. AI – Artificial Intelligence (Штучний інтелект)
2. CNN – Convolutional Neural Networks (Згорткові нейронні мережі)
3. RNN – Recurrent Neural Networks (Рекурентні нейронні мережі)
4. LSTM – Long Short-Term Memory (Довготривала короткочасна пам'ять)
5. VST – Virtual Studio Technology (віртуальна студійна технологія, формат плагінів)
6. SNR – Signal-to-Noise Ratio (співвідношення сигнал/шум)
7. STFT – Short-Time Fourier Transform (короткочасне перетворення Фур'є)
8. DAW – Digital Audio Workstation (цифрова аудіо станція)
9. GAN – Generative Adversarial Networks (генеративні змагальні мережі)
10. SEGAN – Speech Enhancement GAN (покращення мовлення за допомогою GAN)
11. VAE – Variational Autoencoder (варіаційний автоенкодер)
12. Спектрограма – графічне представлення частотного спектру аудіосигналу в часі.
13. Аугментація – метод штучного збільшення датасету шляхом перетворення існуючих даних.
14. Skip Connections – зв'язки, які об'єднують шари в архітектурі нейронних мереж, наприклад в U-Net.
15. Batch Normalization – метод нормалізації даних для стабілізації та прискорення навчання моделей.
16. ReLU – Rectified Linear Unit, функція активації, яка часто використовується в нейронних мережах.
17. Wave-U-Net – модифікація U-Net для роботи з аудіосигналами.

ВСТУП

Розвиток сучасних технологій та стрімкий прогрес у галузі машинного навчання відкривають нові можливості для автоматизації процесів, які раніше потребували значного залучення людських ресурсів. Однією з таких сфер є обробка аудіо, зокрема процес мастерингу треків, завдяки якому музика набуває того самого «професійного» звучання, якого зазвичай здатні досягти лише спеціалісти сфери саунддизайну. Закономірно, що мастеринговий процес вимагав значних технічних знань, використання дорогого обладнання та великої кількості часу, що обмежувало його доступність для широкого кола користувачів.

Актуальність роботи полягає у зростаючому попиті на інноваційні рішення для обробки аудіо, що відповідають сучасним вимогам до якості звуку, а також тим, що наявні засоби автоматизації мастерингу або дають недостатній рівень якості обробки, або дуже складні для опанування звичайним користувачем. Використання машинного навчання дозволяє вирішувати ці задачі, відкриваючи нові горизонти у музичній індустрії та надаючи користувачам інструменти для створення професійного звучання без значних витрат.

Об’єктом дослідження є процес покращення якості аудіо.

Предмет дослідження – методи покращення якості аудіо за допомогою машинного навчання.

Мета роботи полягає у спрощенні та автоматизації процесу мастерингу аудіо треків за допомогою натренованої моделі.

Для досягнення поставленої мети поставлено та виконано наступні завдання:

1) Огляд та аналіз предметної області та підходів до автоматизації обробки звуку з використанням нейронних мереж;

2) Аналіз наявних алгоритмічних та програмних рішень мастерингу аудіо треків;

3) Вибір оптимальних підходів для створення системи, яка дозволить навчати моделі нейронних мереж та використовувати їх для обробки аудіо з метою покращення якості звучання;

4) Розробка системи автоматизації мастерингу аудіо треків із використанням сучасних алгоритмів машинного навчання;

5) Тестування та оцінка результатів роботи системи автоматизованого мастерингу аудіо треків.

У роботі розглянуто актуальні підходи до автоматизації обробки звуку, включаючи архітектури нейронних мереж, такі як CNN, RNN, LSTM та U-Net. Особливу увагу приділено створенню якісного датасету та адаптації моделей до специфіки аудіо сигналів. Результати дослідження демонструють перспективи використання розроблених прототипів у музичній індустрії для автоматизації рутинних завдань, економії часу та ресурсів.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Типи нейромереж та принципи їх роботи

Завдяки унікальним властивостям, нейронні мережі видів CNN, RNN та LSTM є найкращими для обробки аудіо [1, 2]. Більш того, їх властивості доповнюють одна одну, тому при обробці аудіо їх можна комбінувати [9].

1.1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) [1, 2, 8] (Рис. 1.1) зазвичай використовуються для обробки зображень, але вони також можуть бути застосовані для аналізу спектрограм аудіо. Спектрограма [3] — це візуальне представлення частотного спектру аудіосигналу в часі, де горизонтальна вісь представляє час, вертикальна вісь — частоту, а інтенсивність кольору або яскравість вказують на амплітуду або енергію сигналу на певній частоті. Згорткові нейромережі ефективно виявляють локальні патерни і частотні ознаки у спектрограмі, дозволяючи точно визначати та обробляти гармоніки, тембри та інші звукові характеристики. Вони також добре справляються із різними варіаціями у звуках завдяки їх здатності до інваріантності щодо трансформацій.

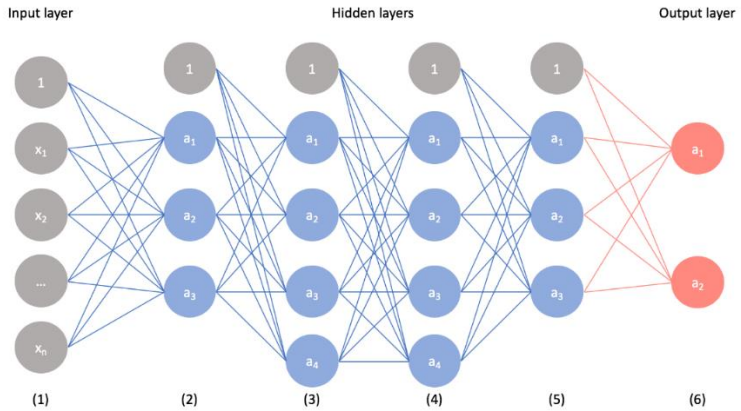


Рис. 1.1 Архітектура згорткової нейронної мережі

Основні компоненти CNN [1, 2, 8]:

- Convolutional Layer (Згортковий шар):

Цей шар застосовує фільтри (ядра згортки) до вхідних даних, щоб виділити локальні ознаки. У випадку аудіо, на цьому етапі спектрограма подається на вхід мережі, і через конволюційні шари проходять згортки (фільтри), які виділяють локальні ознаки, такі як гармоніки. Наприклад, один фільтр може виділяти вертикальні лінії, які відповідають певним частотам, що постійно присутні в аудіо.

- Pooling Layer (Шар підвибірки):

Цей шар зменшує розмірність вихідних даних після згортки, зменшуючи кількість параметрів і обчислювальних ресурсів. Найчастіше використовуються max-pooling або average-pooling.

- Fully Connected Layer (Повнозв'язний шар):

Після кількох шарів згорток та підвибірок вихідні дані передаються на повнозв'язний шар, який інтегрує всі виділені ознаки і проводить остаточну класифікацію або обробку.

Основна операція в CNN - це згортка (convolution), яка застосовує фільтр до вхідного сигналу (спектрограми аудіо) для виділення локальних ознак. Після згорткової операції часто застосовується функція активації ReLU (Rectified Linear Unit) [1, 2, 8].

1.1.2 Recurrent Neural Networks

Recurrent Neural Networks (RNN) [1, 2, 8] (Рис. 1.2) завдяки своїм зворотнім зв'язкам, дозволяють зберігати контекст попередніх кроків [9]. Ця властивість ідеально підходить для задач з тимчасовими залежностями, такими як розпізнавання мови, генерація музики та синтез мовлення. Однак стандартні RNN мають проблему зникання градієнта, що ускладнює обробку довготривалих послідовностей.

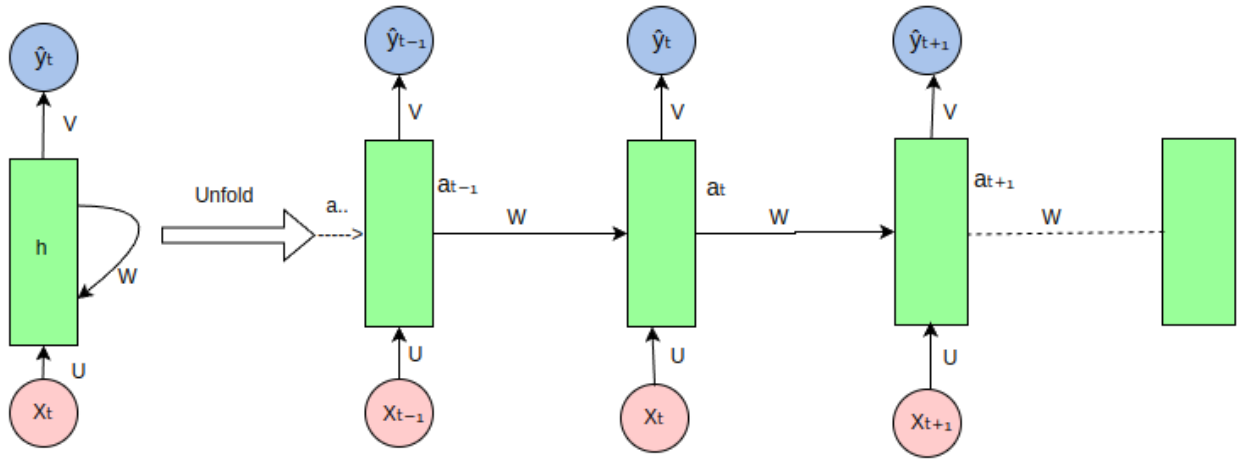


Рис. 1.2 Архітектура рекурентної нейронної мережі

Основні компоненти RNN [1, 2, 7, 8]:

- Hidden State (Прихований стан):

Це стан, який зберігає інформацію про попередні обчислення і оновлюється на кожному кроці послідовності.

- Recurrent Connections (Зворотні зв'язки):

Ці зв'язки дозволяють передавати інформацію крізь час, забезпечуючи збереження послідовності в даних. Наприклад, при обробці музичного треку RNN може зберігати інформацію про ритм та мелодію.

1.1.3 Long Short-Term Memory (LSTM)

Проблему зі зниканням градієнта вирішують Long Short-Term Memory (LSTM) [1, 2, 7] (Рис. 1.3), які мають спеціальні механізми для збереження та використання інформації протягом певного часу, що робить їх дуже ефективними для обробки тривалих аудіосигналів [7, 8]. LSTM можуть зберігати та обробляти інформацію про музичні структури, такі як ритм, мелодія та гармонія, протягом усього треку [9].

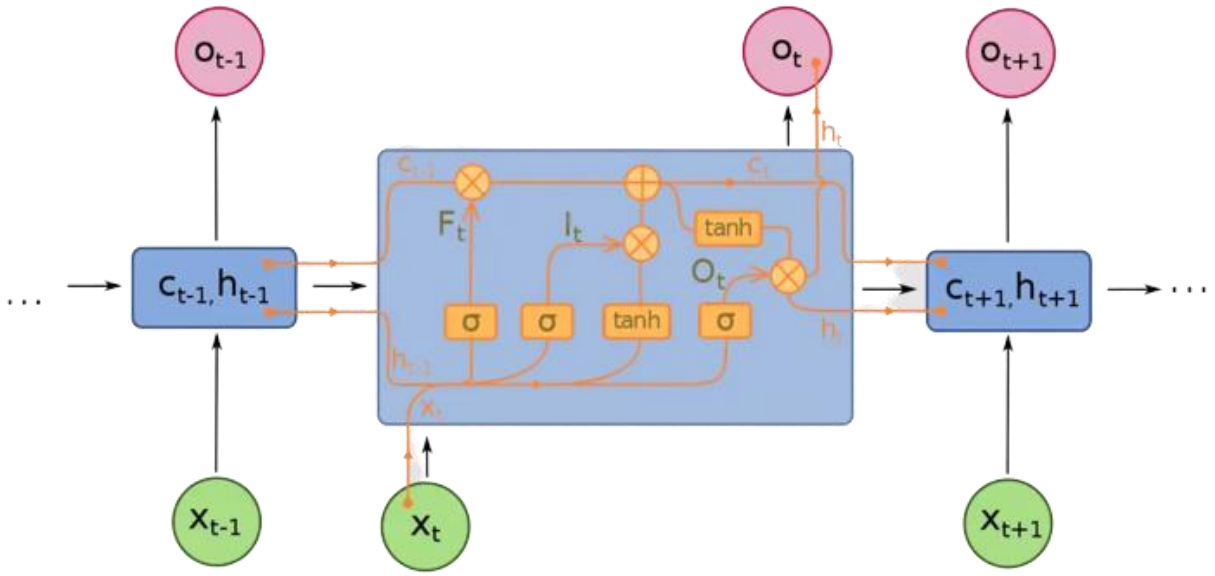


Рис. 1.3 Архітектура довгої короткочасної пам’яті

Основні компоненти LSTM [1, 2]:

- Cell State (Комірка пам’яті):

LSTM має комірку пам'яті, яка дозволяє зберігати і передавати інформацію крізь довгий час. Це дозволяє моделі зберігати важливу інформацію протягом тривалих послідовностей аудіо.

- Forget Gate, Input Gate, Output Gate (Ворота забування, вводу, виводу):

Ці ворота контролюють потік інформації через комірку пам'яті, вирішуючи, яку інформацію зберігати, яку оновлювати, а яку видаляти.

1.2 Огляд наявних рішень для AI мастерингу

На сьогоднішній день існує декілька онлайн сервісів для автоматичного AI мастерингу аудіо [9]. Насправді, їх дещо більше, проте, більшість з них базуються на використанні застосунку (VST плагіна) [5] Izotope Ozone [6]. Відібрані сервіси відрізняються за швидкодією і результатом в залежності від жанру музики. Так, сервіси Landr та Masterchannel добре справляються з мастерингом переважно електронної музики, року, хіп-хопу та поп. Відмінності

цих двох сервісів полягають у тому, що Landr краще обробляє такі жанри як джаз та фолк, а Masterchannel більш гнучкий та має можливість інтеграції з різними музичними платформами. SoundCloud AI mastering найкраще справляється з обробкою музики тих жанрів, які популярні саме на платформі SoundCloud – EDM, хіп-хоп, поп та інді. Фаворитом же виступає професійний інструмент для мастерингу Izotope Ozone [6], що використовується багатьма звукоінженерами для обробки різних музичних жанрів. Він особливо ефективний для мастерингу складних та насичених композицій, таких як рок, класична музика, джаз та електронна музика. Ozone надає широкий спектр інструментів та налаштувань, що дозволяє точно налаштовувати звук відповідно до вимог кожного жанру.

В якості прикладу використовуватимуться власний трек, написаний в жанрі EDM Synthwave [12]. Для аналізу спектрограм була розроблена утиліта-візуалізатор для візуального представлення аудіо.

1.2.1 SoundCloud AI mastering

Для того щоб почати роботу з сервісом необхідно обрати відрізок треку, за яким можна би було оцінити загальну якість мастерингу і, при задовільному результаті, виконати мастеринг всього треку (Рис. 1.4).

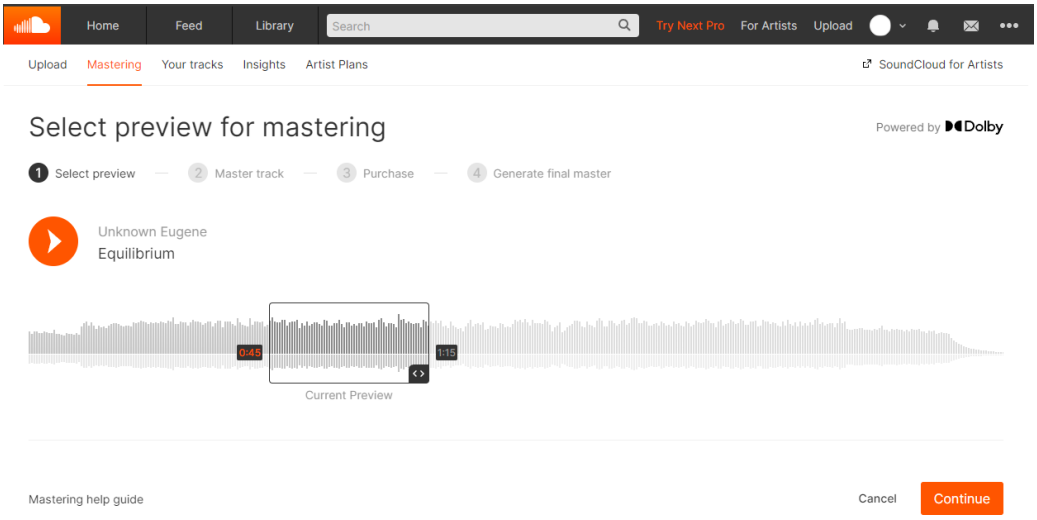


Рис. 1.4 Інтерфейс сервісу SoundCloud AI Mastering

Після натискання на кнопку Continue починається процес аналізу і мастерингу треку (Рис. 1.5). Цей процес може тривати певний час, і по його завершенню користувачеві буде представлено оброблений фрагмент аудіо з можливістю налаштування виду обробки, відсотку її впливу та прослуховування (Рис. 1.6).

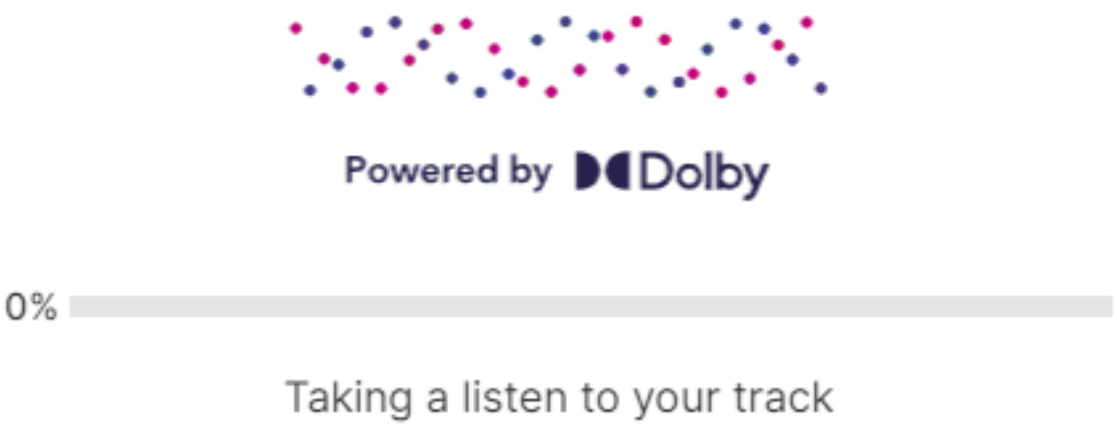


Рис. 1.5 Візуалізація процесу мастерингу

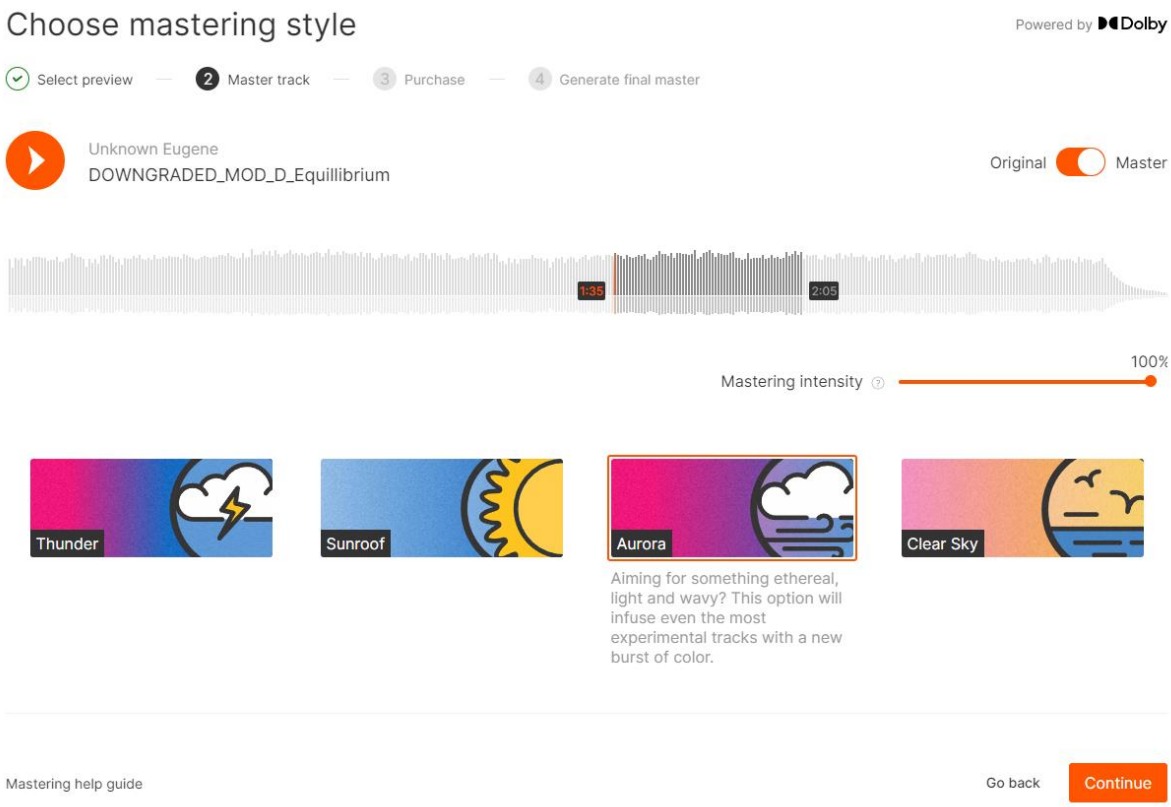


Рис. 1.6 Прослуховування фрагменту аудіо після мастерингу

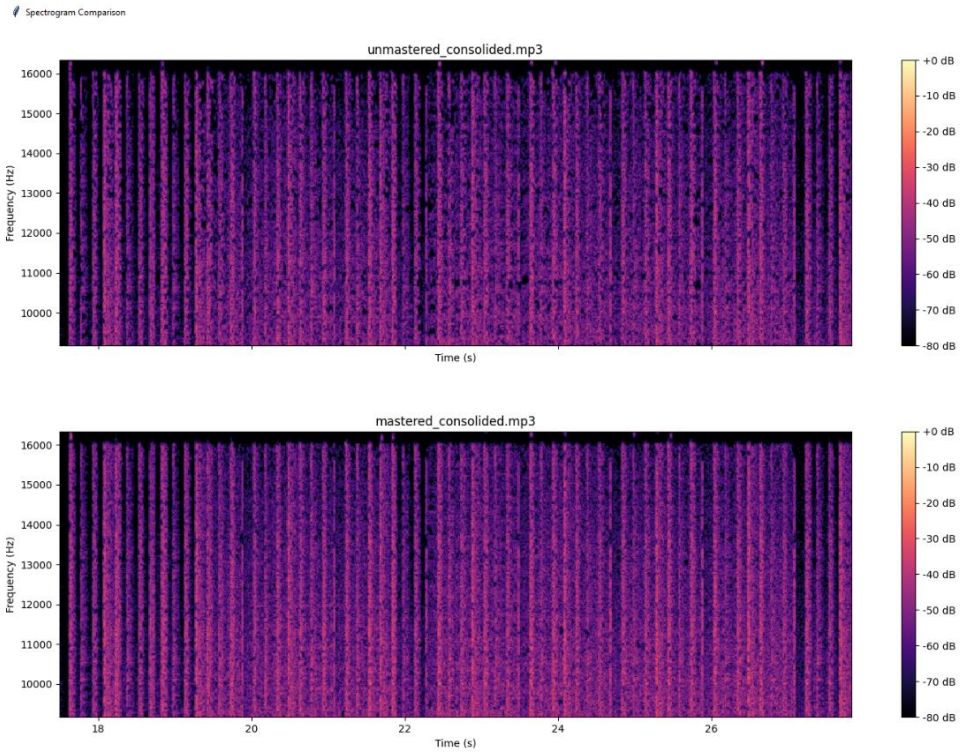


Рис. 1.7 Спектрограми треку до та після автоматичного мастерингу сервісу SoundCloud (Середньо-високі частоти)

На рисунку 1.7 зображено порівняння двох спектрограм [3, 4] середньо-високих частот одного і того самого фрагменту треку (зверху трек до мастерингу, знизу – після), який був оброблений сервісом SoundCloud. Основна відмінність цих двох представлень полягає у тому, що на першій спектрограмі у зображенні високих частот присутні розриви, нерівномірності та певна змазаність гармонік [4]. Однак на зображенні після мастерингу видно що нейромережа виправила більшу частину цих недоліків, зробивши картинку більш чіткою, а звук – приємнішим.

1.2.2 Landr

На відміну від SoundCloud цей сервіс дозволяє завантажити трек повністю для подальшої обробки (Рис. 1.8)

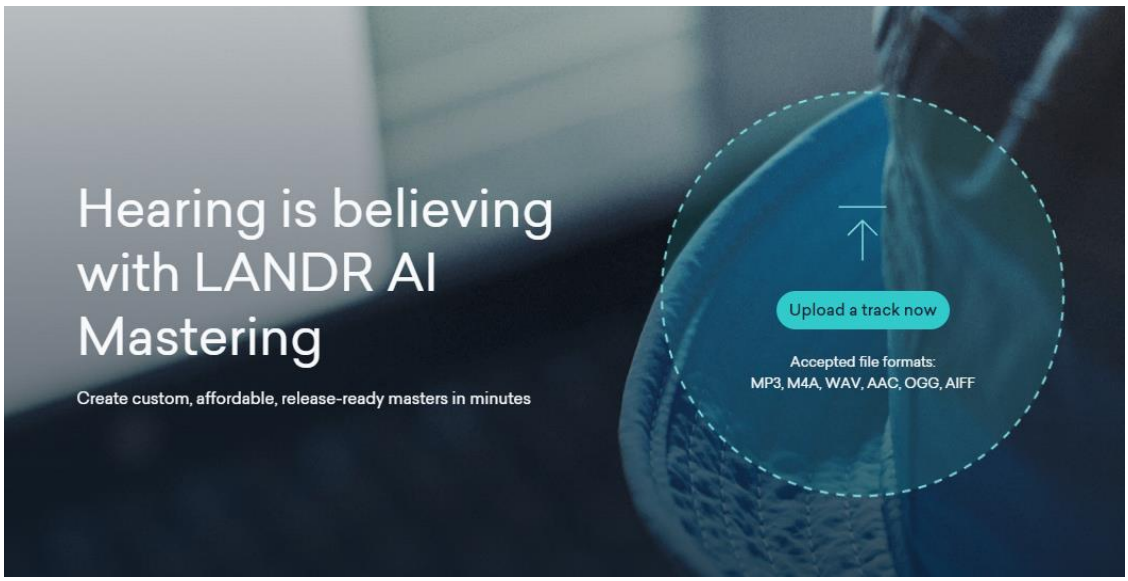


Рис. 1.8 Інтерфейс головної сторінки мастерингу Landr

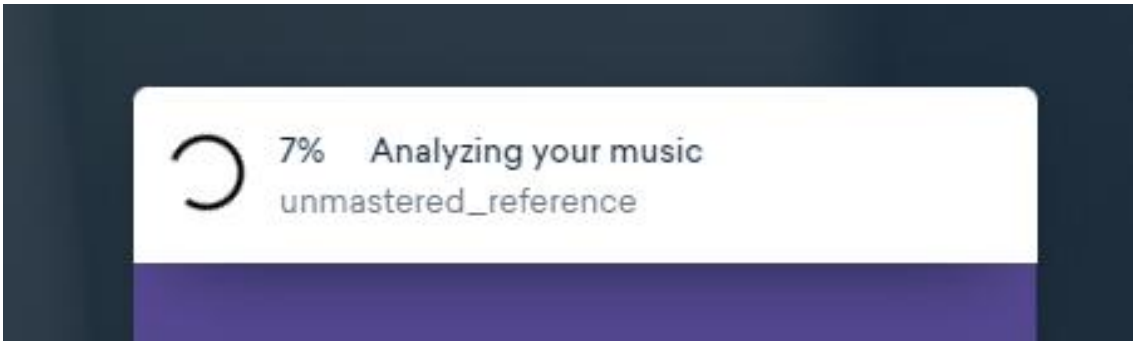


Рис. 1.9 Процес обробки аудіо сервісом Landr

Після завантаження файлу сервіс починає процес обробки, що може тривати кілька хвилин (Рис. 1.9)

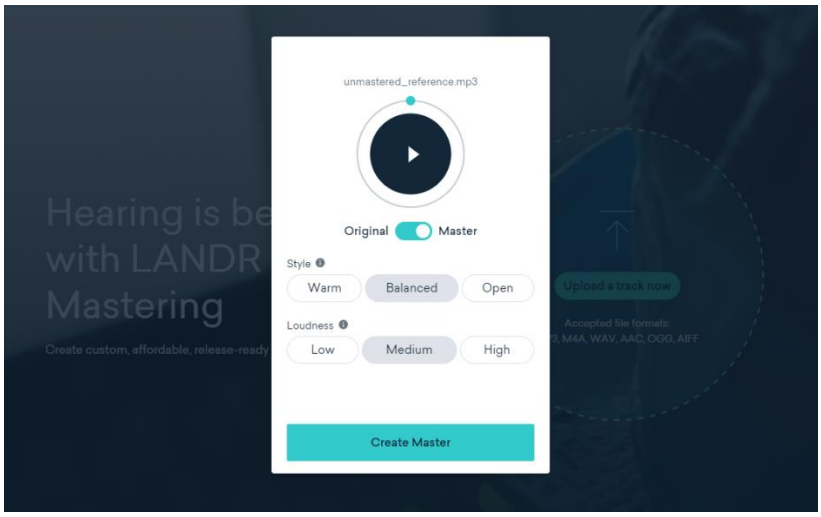


Рис. 1.10 Вікно прослуховування і налаштування

Після закінчення обробки сервіс пропонує прослухати фрагмент обробленого треку та налаштувати параметри мастерингу (Рис. 1.10).

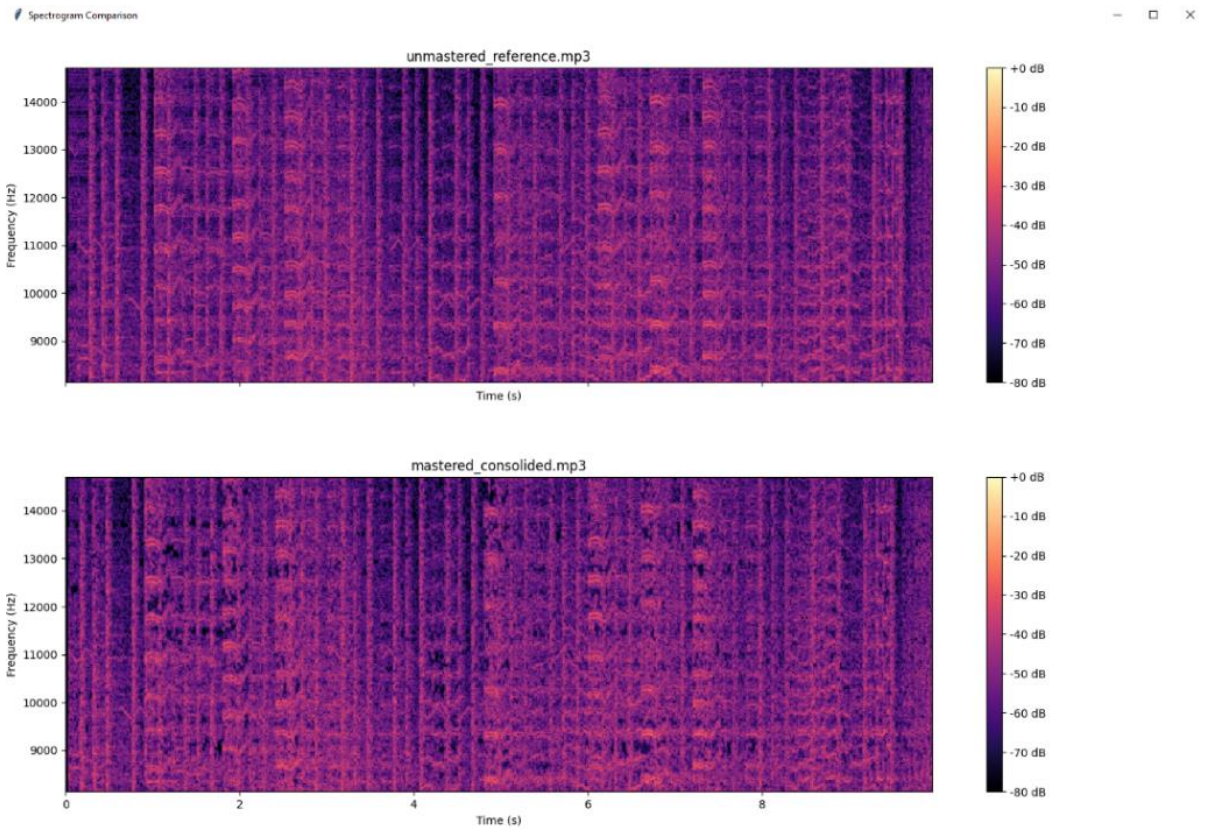


Рис. 1.11 Спектрограми треку до та після автоматичного мастерингу сервісу Landr (Середньо-високі частоти)

На зображенні порівняння двох спектрограм [3, 4] до та після мастерингу (Рис. 1.11) можна помітити дещо інший підхід, відмінний від SoundCloud. Нейромережа прибрала артефакти звуку, пом’якшила різкі переходи і виділила частину гармонік, зробивши їх яскравішими.

1.2.3 Masterchannel

Інтерфейс головної сторінки сервісу містить поле для завантаження аудіотреків, кнопки для аналізу, завантаження обробленого треку та перемикач звучання з оригінального на оброблене для порівняння результату (Рис. 1.12)

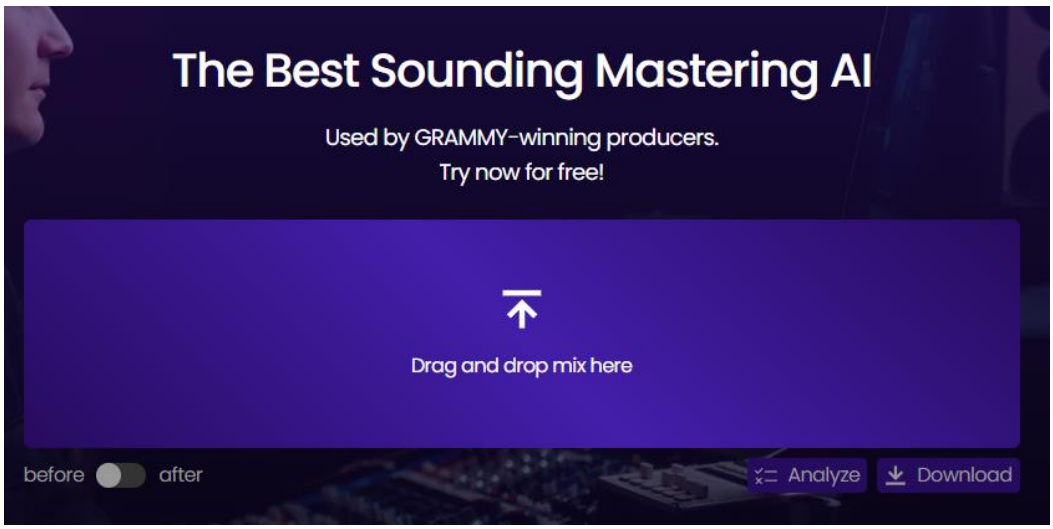


Рис. 1.12 Інтерфейс головної сторінки сервісу Masterchannel

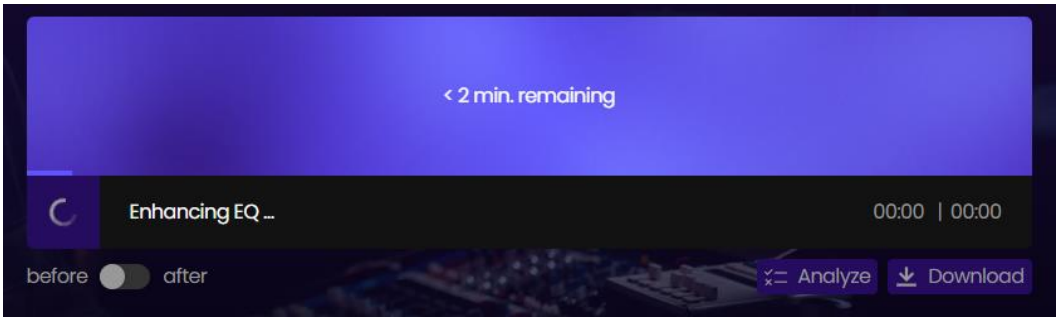


Рис. 1.13 Процес обробки аудіо після завантаження

Як і в попередніх сервісах, після завантаження аудіо починає відбуватися його аналіз і обробка (Рис. 1.13).

Рис. 1.14 Результат обробки аудіо

Masterchannel відрізняється від інших мінімалістичним набором налаштувань після обробки (Рис. 1.14)

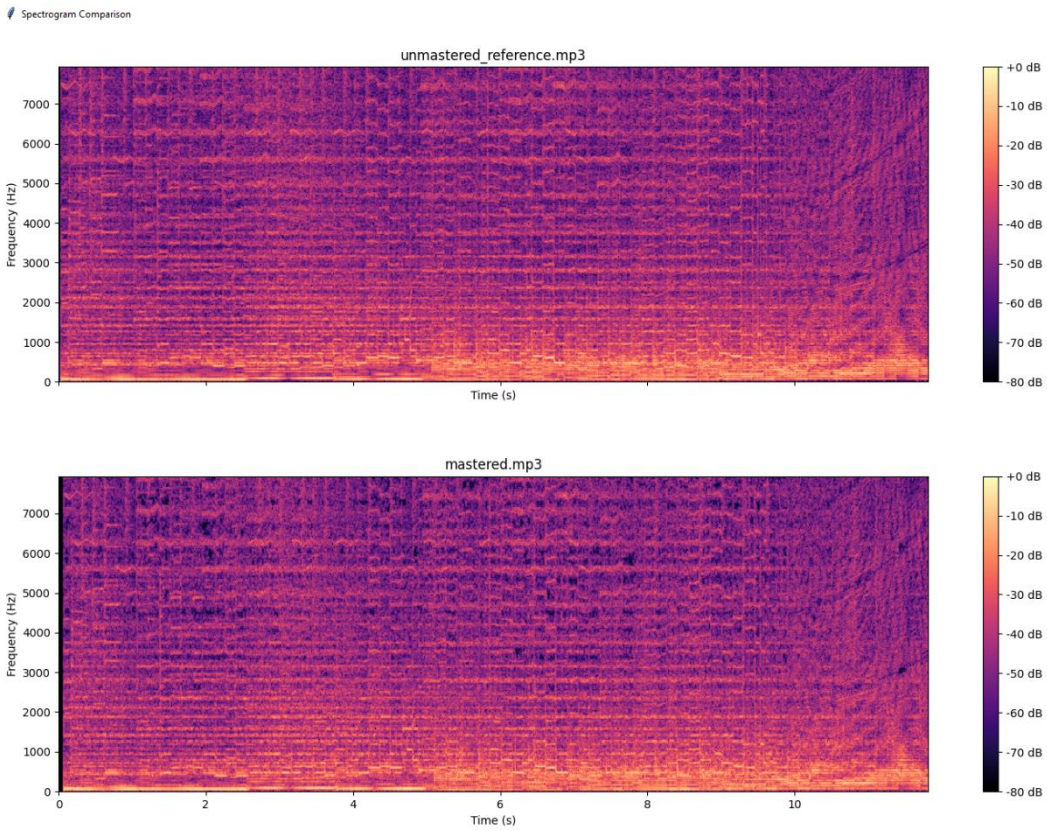


Рис. 1.15 Спектрограми треку до та після автоматичного мастерингу сервісу Landr (Середньо-низькі частоти)

На рисунку 1.15 зображено дві спектрограми [3, 4] до та після мастерингу за допомогою сервісу Masterchannel. Як можна побачити, нейромережа вирівняла загальну гучність треку, зробила тихішими високі частоти і виразила деякі гармоніки [4]. Також, як і у двох попередніх прикладах, була зменшена кількість артефактів та різких переходів що суттєво покращило якість звучання треку.

1.2.4 Izotope Ozone

Цей інструмент суттєво відрізняється від своїх онлайн аналогів передусім тим, що це повноцінний VST плагін [5], який можна встановити та підключити до майже будь-якої DAW. Це професійний інструмент, що включає в себе безліч різноманітних налаштувань та опцій, і дозволяє обробляти звук в режимі реального часу. Інтерфейс Izotope Ozone [6] можна побачити на рисунку 1.16.

Рис. 1.16 Інтерфейс плагіну Izotope Ozone

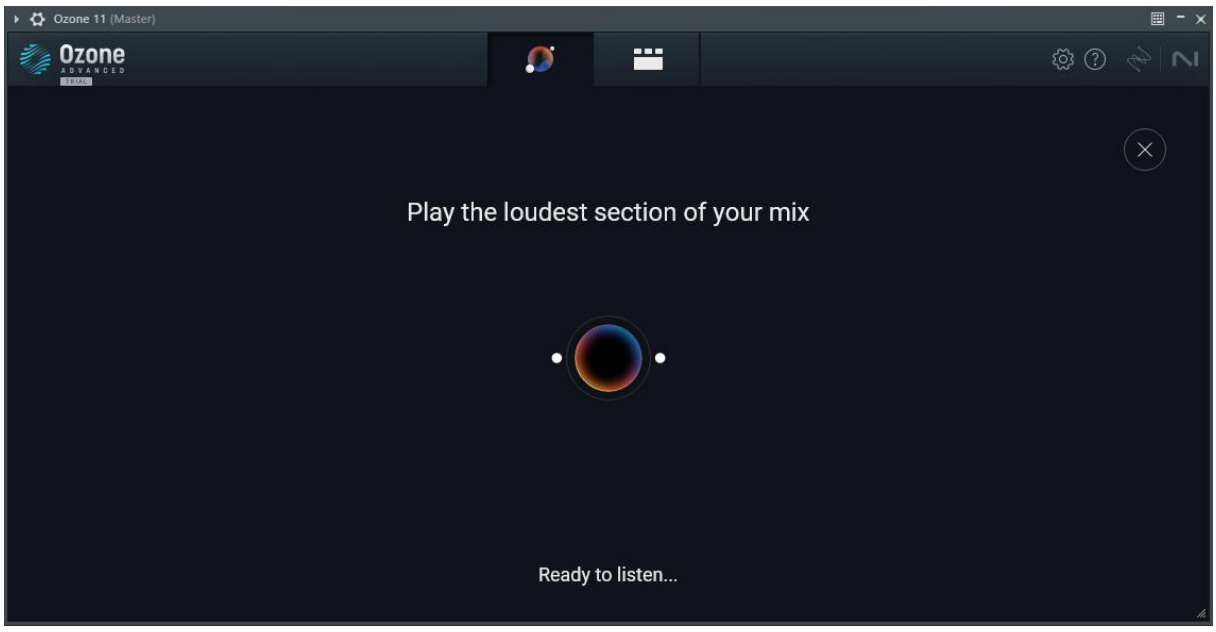


Рис. 1.17 Запуск аналізу звуку з використанням нейромережі.

Рис. 1.18 Інтерфейс налаштування автоматичного мастерингу

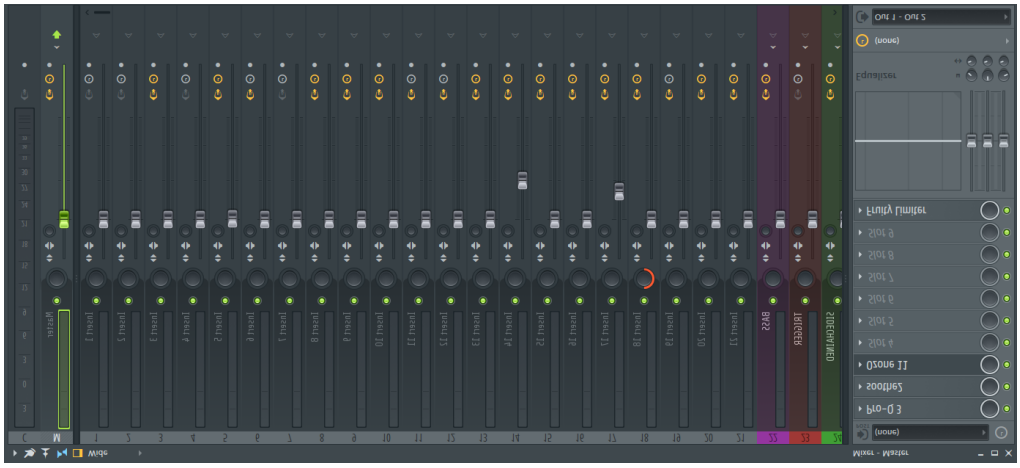


Рис. 1.19 Плагін Ozone підключений на головний канал мікшера

Для того щоб почати роботу з плагіном його треба підключити на головний канал мікшера [5] (Рис. 1.19). Для запуску нейромережевого аналізу треба перейти у відповідний режим та програти найгучніший уривок треку (Рис. 1.17). Через деякий час аналізу, нейромережа налаштує необхідні параметри мастерингу і відкриє інтерфейс користувацьких налаштувань результатів (Рис. 1.18). Як можна побачити з рисунку 1.18, плагін коректно визначив жанр треку. Після закінчення аналізу достатньо просто прослухати трек – у більшості випадків додаткових налаштувань робити не потрібно. Далі представлені спектрограми до обробки та після обробки плагіном Izotope Ozone [6]

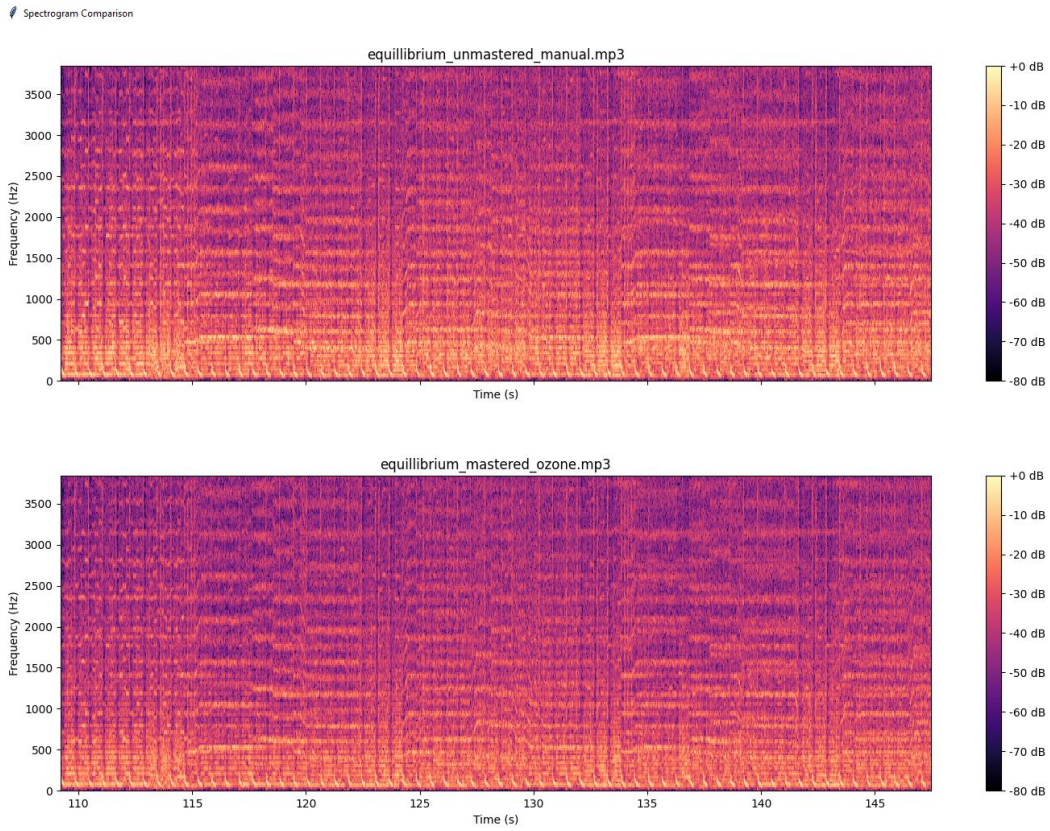


Рис. 1.20 Спектрограми треку до та після автоматичного мастерингу плагіну Izotope Ozone (Середньо-низькі частоти)

На рисунку 1.20 можна побачити як плагін привів гучності частот до одного рівня та зробив низькі частоти менш гучними [4]. Також він зменшив кількість частотних резонансів і додав втрачені високі частоти до мелодії, тим самим зробивши її яскравішою.

1.3 Порівняння результатів роботи готових рішень

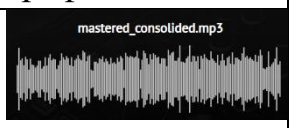
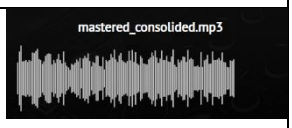
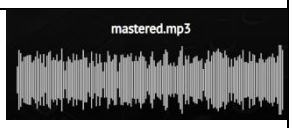
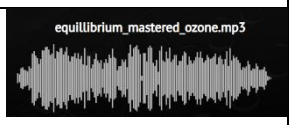
Оскільки досвід сприйняття людиною музики суто суб’єктивний і залежить від великої кількості факторів, то для об’єктивного порівняння наведених сервісів, окрім їх переваг і недоліків, аспектом якості обробки аудіо буде виступати графік форми звукових хвиль (оскільки деякі сервіси надавали можливість завантажувати і обробляти лише фрагменти треків, графіки форми хвиль будуть наведені для відповідних фрагментів) а також розрахунок SNR індексу. Початково не оброблений трек в поганій якості має індекс 43, що вказує

на досить погану якість звуку [11]. Трек, оброблений вручну має індекс 76, що навпаки каже про дуже високу якість. Формула розрахунку SNR [11] (1.1):

$$SNR = 10\boldsymbol{log_{10}}(\frac{P_{signal}}{P_{noise}}). \quad (1.1)$$

Таблиця 1.1

Загальне порівняння інструментів автоматичного мастерингу аудіо

Сервіс	Жанри музики	Переваги	Недоліки	Зображення графіків та SNR
Sound Cloud	EDM, хіп-хоп, поп, інді	Інтеграція з SoundCloud, зручність використання	Обмежений контроль над параметрами	 До: 43. Після: 51
Landr	Рок, поп, хіп-хоп, EDM, джаз, фолк	Висока якість обробки, універсальність, доступність для різних жанрів	Обмеженість налаштувань	 До: 43. Після: 49
Master channel	Рок, поп, хіп-хоп, електронна музика	Інтеграція з музичними платформами, гнучкість	Висока вартість підписки	 До: 43. Після: 56
Izotope Ozone	Рок, класична музика, джаз, електронна музика	Висока якість обробки, широкий спектр налаштувань	Висока вартість, вимога певних знань для використання	 До: 43. Після: 61

Виходячи із порівняльної таблиці 1.1 можна сказати, що з автоматичним мастерингом аудіо з використанням штучного інтелекту найкраще впорався Izotope Ozone [6]. Також він має найбільшу кількість налаштувань серед наведених інструментів та має можливість аналізу та обробки аудіо в режимі реального часу.

2 АЛГОРИТМИ МАШИННОГО НАВЧАННЯ ДЛЯ ОБРОБКИ АУДІО

2.1 Види нейронних мереж для обробки аудіо рішень

Останні кілька років відбувається неабиякий прогрес в галузі машинного навчання. Це стосується і сфери роботи з аудіо. Сьогодні за допомогою технологій машинного навчання та, зокрема, глибокого навчання стало можливим автоматичне розпізнавання мовлення, музичної інформації, виявлення звукових подій та багато іншого, що до цього часу можливо було виконати лише людині. Це пов’язано, в першу чергу, з здатністю нейронних мереж ефективно знаходити закономірності і витягувати та моделювати складні патерни з аудіо даних. Різні типи нейронних мереж, такі як згорткові нейронні мережі (CNN), рекурентні нейронні мережі (RNN) та довготривала короткочасна пам'ять (LSTM), широко використовуються в обробці та аналізі аудіо сигналів завдяки саме цим дивовижним здібностям.

2.1.1 CNN: Застосування для аналізу спектрограм

Згорткові нейронні мережі (CNN) початково були розроблені для обробки зображень, але їх архітектура виявилася достатньо ефективною і для обробки аудіо сигналів [13]. Суть роботи такого виду нейронних мереж з аудіо полягає у тому, що будь-яку звукову хвилю можна перетворити на спектрограму – двовимірне представлення (зображення), що містить в собі інформацію про спектр частот сигналу в часі, де фактично вісь Y представляє собою значення частоти в Герцах, а вісь X представляє собою часову шкалу. CNN можуть виявляти локальні патерни та особливості в спектрограмі, такі як гармоніки, форманти, шуми та інші акустичні характеристики [14].

У дослідженні Понса та ін. [14] було показано, що використання згорткових фільтрів різних розмірів дозволяє моделі захоплювати як

короткочасні, так і довготривалі особливості аудіо сигналу. Цю особливість можна застосувати у вирішенні задач класифікації музичних жанрів, інструментів та розпізнавання мовлення. Крім того, прооцеси тренування моделі CNN, як і, власне, процеси обробки аудіо за допомогою заздалегідь натренованої моделі можуть бути ефективно реалізовані на графічних процесорах (GPU), що дозволяє обробляти великі обсяги даних у реальному часі [13]. Це робить згорткові нейронні мережі придатними для сценаріїв, що вимагають швидкості та ефективності.

Архітектура CNN для аудіо обробки включає:

- Вхідний шар, що приймає спектрограму або мел-спектрограму як вхідні дані.
- Згорткові шари, які застосовують фільтри для витягнення локальних ознак.
- Активізаційні функції, такі як ReLU, для введення нелінійності (2.1)
- Шари підвибірки (Pooling layers), які зменшують розмірність даних та покращують стійкість моделі до зсувів і деформацій.
- Повнозв'язані шари, що виконують кінцеву класифікацію або регресію.

$$f(x) = \max(0, x)\ . \qquad (2.1)$$

Формула операції згортки (2.2):

$$\mathrm{S(i,j)} = (\mathrm{X} * \mathrm{K})(\mathrm{i,j}) = \sum_{\mathrm{m}} \sum_{\mathrm{n}} \mathrm{X(i-m, j-n)} \cdot \mathrm{K(m,n)}, \qquad (2.2)$$

де:

- $S(i,j)$ – вихід згорткового шару в позиції (i,j)
- X – вхідне зображення або спектрограма.
- K – ядро (фільтр) згортки.
- m,n – індекси, що перебирають розмір фільтру.

В процесах обробки аудіо згорткові нейронні мережі насамперед застосовуються у задачах класифікації музики та жанрів: моделі CNN можуть виявляти специфічні патерни, характерні для різних жанрів музики [14], розпізнавання звукових подій: ідентифікація певних звуків у навколишньому середовищі, таких як звуки природи або шуми міста [15], розпізнавання мовлення: витягнення ознак для покращення точності моделей розпізнавання мовлення.

Основними перевагами згорткових нейронних мереж в задачах обробки аудіо сигналів є локальність – здатність виявляти локальні часово-частотні ознаки, спільні ваги – зменшення кількості параметрів моделі, інваріантність до зсувів: стійкість до невеликих змін у вхідних даних.

2.1.2 RNN: Обробка часових залежностей в аудіо

Рекурентні нейронні мережі (RNN) (2.3) спеціалізуються на обробці послідовних даних, таких як аудіо сигнали, де послідовність зразків має значення [16]. Через використання зворотних зв'язків вони призначені для роботи з такими даними, де поточний стан мережі залежить від попередніх входів. Це дозволяє рекурентним нейронним мережам зберігати інформацію про попередні стани, враховуючи контекст при обробці нових даних. Оскільки звук є послідовністю коливань тиску повітря з певними закономірностями, то при його обробці потрібно враховувати часові залежності [18]. Рекурентні нейронні мережі здатні моделювати ці часові залежності, що суттєво збільшує їх ефективність в задачах розпізнавання мовлення, синтезу аудіо та прогнозування часових рядів. Завдяки здатності враховувати попередні дані, рекурентні нейронні мережі можуть точно відтворювати складні структури та патерни, що присутні в послідовностях.

Стандартні рекурентні нейронні мережі (RNN) стикаються з низкою проблем, серед яких найбільш значущими є зникання та вибух градієнтів, а також обмежена пам'ять. При навчанні на довгих послідовностях градієнти можуть ставати надто малими або надто великими, що суттєво ускладнює процес навчання та може призводити до нестабільності моделі [18]. Крім того,

рекурентні нейронні мережі мають труднощі із запам'ятовуванням довготривалих залежностей, що обмежує їхню здатність ефективно враховувати контекст при обробці даних, що містять віддалені часові зв'язки. Незважаючи на ці обмеження, рекурентні нейронні мережі знаходять широке застосування в області обробки аудіо даних. Однією зі сфер є розпізнавання мовлення, де вони використовуються для моделювання послідовностей фонем або слів, дозволяючи точно розпізнавати вимовлені фрази [17]. Крім того, рекурентні нейронні мережі застосовуються для синтезу мовлення, генеруючи аудіо сигнали на основі текстових або інших вхідних даних, за допомогою чого відбувається створення природних та реалістичних голосових інтерфейсів тощо. Також ці мережі використовуються в аналізі музичних послідовностей, де вони допомагають прогнозувати наступні ноти або ритмічні структури. Завдяки цьому стає можливим процес автоматичного складання музики та підтримки творчих процесів.

Для подолання наявних у рекурентних нейронних мерезж обмежень були розроблені вдосконалені архітектури, такі як Long Short-Term Memory (LSTM) та Gated Recurrent Unit (GRU), які ефективніше зберігають інформацію протягом довших періодів часу, покращуючи таким чином продуктивність моделей у складних задачах обробки послідовних даних.

Формула оновлення прихованого стану RNN:

$$\mathrm{h_t} = \phi(\mathrm{W_{xh}x_t} + \mathrm{W_{hh}h_{t-1}} + \mathrm{b_h}) , \qquad (2.3)$$

де:

- h_t – поточний прихований стан.
- x_t – поточний вхідний вектор.
- $\mathrm{W_{xh}}, \mathrm{W_{hh}}$ – вагові матриці.
- b_h – вектор зміщення.
- ϕ – функція активації (зазвичай tanh або ReLU)
- $\mathrm{h_{t-1}}$ – попередній прихований стан.

2.1.3 LSTM: Переваги для обробки довгих послідовностей

Саме для вирішення проблем і недоліків рекурентних нейронних мереж було створено архітектуру довготривалої короткочасної пам'яті (LSTM). Long Term Short Memory (LSTM) є розширенням (покращенням) рекурентної нейронної мережі, яке вирішує проблему зникнення градієнта шляхом введення спеціальних «комірок пам'яті». Ці комірки дозволяють LSTM зберігати інформацію на довгих часових проміжках, ефективно моделюючи довготривалі залежності в даних [18]. У контексті аудіо обробки LSTM використовуються для задач, де важлива історія сигналу, наприклад, для генерації музики або в системах автоматичного перекладу мовлення. Довготривалі короткочасні пам’яті можуть комбінуватися із згротковими нейронними мережами у гібридних моделях, де остання витягує локальні особливості, а LSTM моделює їх часову динаміку [13].

LSTM модель складається з таких компонентів:

- Вхідні ворота (2.4): визначають, яку нову інформацію додати до комірки пам'яті.
- Забуваючі ворота (2.5): вирішують, яку інформацію з попереднього стану пам'яті слід зберегти.
- Комірка пам’яті (2.6, 2.7, 2.8): зберігає інформацію про стан системи.
- Вихідні ворота (2.9): контролюють, яку інформацію з комірки пам'яті передати на вихід.

$$i_t = \sigma(\mathrm{W_i x_t} + \mathrm{U_i h_{t-1}} + \mathrm{b_i}) , \quad (2.4)$$

$$\mathrm{f_t} = \sigma(\mathrm{W_f x_t} + \mathrm{U_f h_{t-1}} + \mathrm{b_f}) , \quad (2.5)$$

$$\tilde{\mathrm{c}}_t = \tanh(\mathrm{W_c x_t} + \mathrm{U_c h_{t-1}} + \mathrm{b_c}) , \quad (2.6)$$

$$\mathrm{c_t} = \mathrm{f_t} \odot \mathrm{c_{t-1}} + \mathrm{i_t} \odot \tilde{\mathrm{c}}_\mathrm{t} , \quad (2.7)$$

$$\mathrm{h_t} = \mathrm{o_t} \odot \tanh(\mathrm{c_t}) , \quad (2.8)$$

$$\mathrm{o_t} = \sigma(\mathrm{W_o x_t} + \mathrm{U_o h_{t-1}} + \mathrm{b_o}) , \quad (2.9)$$

де:

- x_t – вхідний вектор у момент часу t.

- h_{t-1} – прихований стан з попереднього часу.
- c_{t-1} – попередній стан комірки пам'яті.
- f_t – забуваючі ворота в момент часу *t*.
- i_t – вхідні ворота в момент часу *t*.
- o_t – вихідні ворота в момент часу *t*.
- $\tilde{c}_t$ – кандидат на оновлення стану комірки пам'яті.
- c_t – оновлений стан комірки пам'яті.
- h_t – оновлений прихований стан.
- W_f, W_i, W_c, W_o – вагові матриці для зв'язків від вхідного вектора x_t до відповідних воріт:
  - W_f – вагова матриця для забуваючих воріт.
  - W_i – вагова матриця для вхідних воріт.
  - W_c – вагова матриця для кандидата на стан комірки пам'яті.
  - W_o – вагова матриця для вихідних воріт.
- U_f, U_i, U_c, U_o – вагові матриці для зв'язків від попереднього прихованого стану h_{t-1} до відповідних воріт:
  - U_f – вагова матриця для забуваючих воріт.
  - U_i – вагова матриця для вхідних воріт.
  - U_c – вагова матриця для кандидата на стан комірки пам'яті.
  - U_o – вагова матриця для вихідних воріт.
- b_f, b_i, b_c, b_o – вектори зміщення для відповідних воріт:
  - b_f – зміщення для забуваючих воріт.
  - b_i – зміщення для вхідних воріт.
  - b_c – зміщення для кандидата на стан комірки пам'яті.
  - b_o – зміщення для вихідних воріт.
- σ – сигмоїдна функція активації (2.10):

$$\sigma(x) = \frac{1}{1+e^{-x}}. \qquad (2.10)$$

- $tanh$ – гіперболічний тангенс (2.11):

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \quad (2.11)$$

- $\odot$ – поелементне множення (Адамаровий добуток)

Серед переваг LSTM архітектури можна виділити зокрема:

- Здатність запам'ятовувати довготривалі залежності: завдяки спеціальній структурі комірок пам'яті та механізмам воріт.
- Гнучкість у обробці послідовних даних: ефективні для різних типів задач, пов'язаних з послідовностями.

2.1.4 Комбіновані архітектури: CNN + LSTM

При поєднанні згорткових нейронних мереж з довготривалими короткочасними пам’ятями отримана амальгама буде мати переваги обох архітектур: перша витягуватиме різноманітні просторові ознаки з вхідних даних, тоді як друга буде моделювати часові залежності цих ознак [22]. Нарешті, вихідний шар виконує кінцеву класифікацію або регресію, забезпечуючи точні результати на основі попередньо обробленої інформації. Комбінована архітектура має кілька суттєвих переваг. По-перше, завдяки здатності одночасно моделювати як просторові, так і часові патерни в даних, вона забезпечує покращену продуктивність, підвищуючи точність та надійність моделі. По-друге, така архітектура сприяє зменшенню перенавчання завдяки більш ефективному використанню даних і дозволяє моделі краще узагальнюватися на нових, невідомих вибірках. Комбіновані моделі знаходять широке застосування в різних сферах. Однією з основних областей є класифікація звуків навколишнього середовища, де вони підвищують точність розпізнавання різних звукових класів [15]. Крім того, такі моделі використовуються для розпізнавання емоцій в мовленні, аналізуючи інтонацію та просодію для визначення

емоційного стану мовця, що застосовується у створенні більш інтуїтивних та чутливих інтерактивних систем.

2.1.5 Архітектура U-Net

Модель U-Net (рис. 2.1) була вперше представлена для задач сегментації зображень у біомедичних дослідженнях [23] у 2014-2015 роках. Завдяки своїй унікальній архітектурі, яка поєднує шари згортки та розгортки, U-Net здатна ефективно витягувати та відновлювати інформацію на різних рівнях абстракції. У останні роки U-Net та її модифікації успішно застосовуються в обробці аудіо сигналів, зокрема в задачах розділення джерел звуку, покращення якості мовлення та інших аудіо застосуваннях [24, 25, 26] завдяки здібності моделі сегментувати спектрограми.

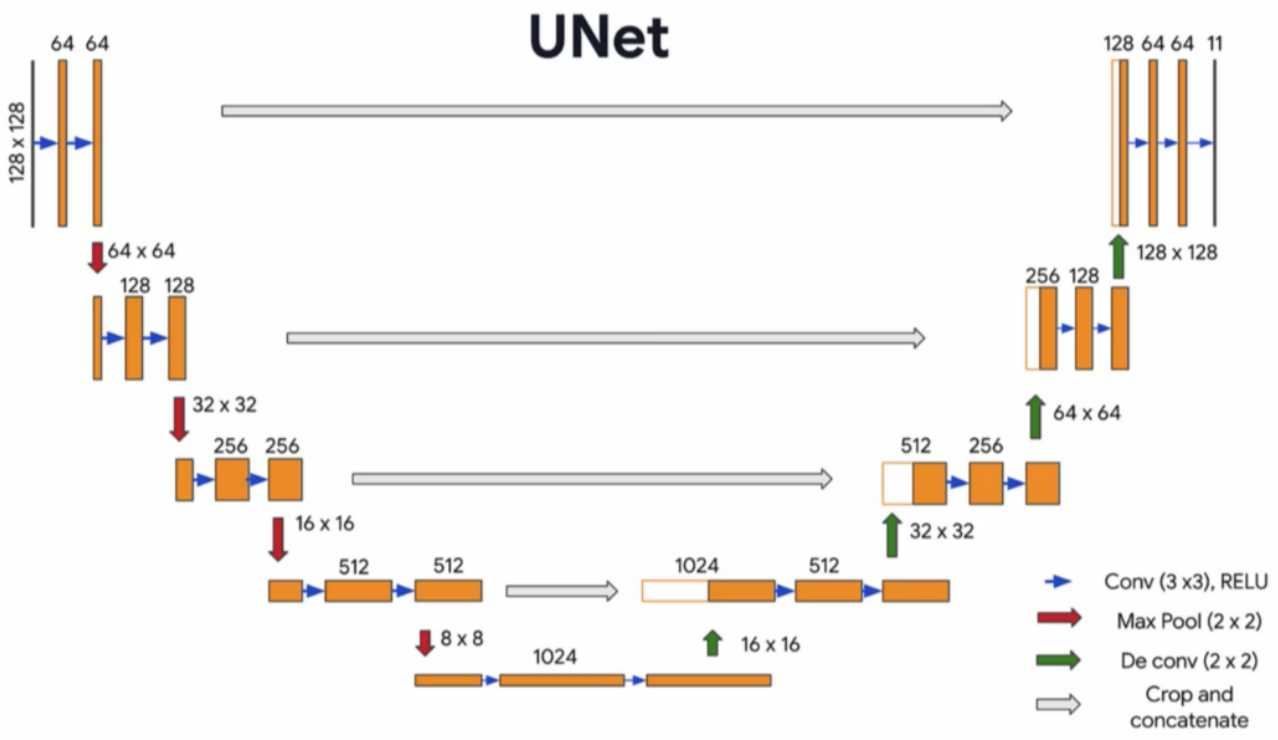


Рис. 2.1 Схема U-Net архітектури

U-Net складається з двох симетричних частин:

1. Контрактивна (стискаюча) частина: послідовність згорткових шарів, які витягують високорівневі ознаки з вхідного сигналу, поступово зменшуючи просторову роздільну здатність.

2. Експансивна (розширююча) частина: послідовність транспонованих згорткових шарів, які відновлюють роздільну здатність та поєднують витягнуті ознаки з відповідних рівнів контрактивної частини через skip connections.

Основні компоненти U-Net:

- Згорткові шари: виконують згортку вхідного сигналу з фільтрами для витягнення ознак.
- Max Pooling: зменшує розмірність даних, збільшуючи рецептивне поле.
- Up-Convolutions: збільшують розмірність даних, відновлюючи просторову роздільну здатність.
- Skip Connections: поєднують відповідні шари контрактивної та експансивної частин для збереження контекстної інформації.

Нехай x – вхідний сигнал. Контрактивна частина виконує послідовність операцій:

$$f_i = \sigma(W_i * f_{i-1} + b_i)\ , \qquad (2.12)$$

де:

- $f_0 = x$
- W_i – ваги згорткового шару на рівні i.
- $*$ – операція згортки.
- σ – функція активації (наприклад, ReLU)

Експансивна частина виконує транспоновані згортки та поєднання з відповідними шарами (2.13):

$$g_i = \sigma(W_i^{\top} * \mathrm{Concat}(g_{i+1}, f_i) + b_i), \quad (2.13)$$

де:

- $W_i^{\top}$ – транспоновані ваги згорткового шару.
- Concat – операція конкатенації по каналу.

Модель U-Net була адаптована для обробки аудіо сигналів, зокрема для задач відокремлення вокалу від інструментальної музики [24, 26], видалення шуму та артефактів з аудіо записів [25]. Виходячи з призначення U-Net моделі, її загалом можна використовувати в задачах обробки часово-частотних представлень – спектрограм аудіо сигналів [21].

Особливості обробки аудіо моделлю U-Net полягають у тому, що деякі варіації цієї моделі працюють безпосередньо з одномірними сигналами [26]. Сам же сигнал перетворюється у STFT або Мел-спектрограму для подальшої обробки нейронною мережею, що дозволяє застосовувати двовимірні згортки [24]. Skip Connections в свою чергу допомагають зберегти деталі високої частоти, важливі для якості звуку [23].

Одним прикладів застосування моделі U-Net в задачах, пов’язаних з роботою з аудіо, є розділення вокалу та інструментів. Jansson та ін. (2017) [24] представили модель на основі U-Net, яка використовує спектрограми як вхідні дані для розділення вокальних компонентів з музичних треків. Ця модель перевершує попередні підходи, точністю та чистотою розділення вокалу від супутніх інструментів. Іншим застосуванням є покращення якості мовлення. Seetharaman та Pardo (2019) [25] використали U-Net для задачі шумозаглушення в мовленні. Вони розробили глибокий автоенкодер на основі U-Net, який навчається відновлювати чистий аудіо сигнал з шумного вхідного аудіо. Це дозволяє суттєво підвищити якість мовлення навіть за наявності сильних фонових шумів. Також існує модифікація U-Net під назвою Wave-U-Net. Якщо оригінальна модель початково була розроблена для обробки зображень, а з аудіо вона могла взаємодіяти лише шляхом аналізу спектрограм, то Wave-U-Net початково створювалася як адаптований варіант U-Net для роботи з часовими

проміжками безпосередньою. Ця адаптація U-Net була запропонована Stoller та ін. (2018) [26]. Можливість роботи безпосередньо з аудіо дозволяє моделі вивчати часові залежності напряму з сигналу, за рахунок чого підвищується якість обробки звуку. Переваги Wave-U-Net в обробці аудіо значно підвищують її привабливість для різних сценаріїв використання і завдань. По-перше, завдяки Skip Connections зберігається деталізована інформація та відновлюються дрібні деталі сигналу [23]. По-друге, Wave-U-Net можна навчати без втрати роздільної здатності завдяки поєднанню згорток та розгорток, що дозволяє аналізувати дані на різних масштабах та дає можливість детально аналізувати як короткочасні, так і довготривалі залежності у сигналі [26]. Загалом, універсальність U-Net робить можливим процес адаптації цієї архітектури до різних задач обробки аудіо, як з використанням спектрограм, так і сирих сигналів [21].

У контексті процесів автоматизації мастерингу аудіо з використанням технологій машинного навчання, U-Net може бути застосована в задачах видалення шумів, корекції частотних характеристик, усунення артефактів [25] (загальне покращення якості звуку), ізоляції окремих інструментів або вокалу для подальшої індивідуальної обробки [24] та застосування стилістичних змін до аудіо сигналу на основі навчання на певних жанрах або стилях [14]. Завдяки своїй архітектурі, U-Net може навчатися складним перетворенням, необхідним для мастерингу [13]. Також модель нейронної мережі U-Net (та її модифікація Wave-U-Net) може бути інтегрована разом з LSTM для врахування довготривалих часових залежностей [21] та з архітектурами генеративних змагальних мереж (GAN) для покращення якості синтезованого аудіо [18]. Проте, ця модель не позбавлена і певних недоліків, а саме:

- Обчислювальна складність: глибокі моделі, такі як U-Net, можуть вимагати значних обчислювальних ресурсів [26].
- Якість даних: ефективність моделі залежить від якості та кількості тренувальних даних. Необхідно забезпечити різноманітність та репрезентативність датасету [15].

- Генералізація: забезпечення здатності моделі працювати з різними типами аудіо та жанрами. Можливе використання методів аугментації даних для покращення узагальнюючої здатності моделі [16].

Проте, ці недоліки не настільки значні у порівнянні з перевагами використання саме цієї моделі нейронної мережі у задачах обробки, а особливо, автоматичного мастерингу аудіо.

2.1.6 Інші архітектури нейронних мереж в аудіо обробці

Виділяють й інші архітектурні моделі нейронних мереж такі як:

- Автоенкодери, що використовуються для стиснення та відновлення аудіо даних або, наприклад, варіаційні енкодери (VAE) які в свою чергу дозволяють моделювати розподіл даних та генерувати нові зразки.
- Генеративні змагальні мережі (GAN), що використовуються для генерації реалістичних аудіо сигналів та їх різновид – SEGAN, що застосовується для покращення якості мовлення шляхом зниження рівня шуму [16].
- Трансформери – використовують механізм уваги для моделювання довготривалих залежностей. Застосовуються найчастіше в процесах синтезу мовлення, обробки природної мови.

Якщо порівнювати згорткові нейронні мережі з рекурентними нейронними мережами та довготривалими короткочасними пам’ятями, то варто зазначити що у висновку, перші краще підходять для витягнення локальних ознак у часово-частотному просторі, а другі ефективні для моделювання послідовних залежностей та контексту. Проте, у багатьох випадках комбіновані моделі у висновку показують найкращі результати, оскільки вони поєднують переваги обох підходів [22].

2.2 Методика створення датасету

Процес створення якісного та об’ємного датасету є невід’ємною складовою у підготовці до навчання будь-якої моделі нейронної мережі, в тому

числі це стосується навчання моделей машинного навчання для обробки аудіо. Від якості даних залежить продуктивність та точність моделі [13]. У цьому підрозділі буде розглянуто методику створення датасету, включаючи збір даних, аугментацію та підготовку даних до навчання моделі.

2.2.1 Ручний збір аудіо та налаштування

Джерела даних:

Для створення датасету необхідно зібрати велику кількість аудіо зразків, які відповідають поставленій задачі. Ці зразки мають бути якісними і, головне, консистентними – у випадку моделі для обробки аудіо датасет повинен складатися з треків, нормалізованих за гучністю, однакових за тривалістю та іншими характеристиками. Джерела даних можуть бути різноманітними. Одним із основних джерел є відкриті аудіо бібліотеки. Сьогодні існує багато відкритих аудіо баз даних, таких як UrbanSound8K, ESC-50, LibriSpeech, GTZAN Dataset тощо, які містять тисячі аудіо зразків різних категорій та жанрів [15]. Якщо специфіка задачі вимагає унікальних даних, можна здійснити власний запис аудіо з використанням мікрофонів, інструментів, синтезаторів та іншого обладнання. Крім того, інтернет пропонує безліч аудіо матеріалів, доступних для завантаження, однак при їх використанні важливо враховувати питання авторських прав та ліцензування. Загалом, комбінування різних джерел даних дозволяє створити обширний та різноманітний датасет, що сприяє підвищенню точності та надійності моделей обробки аудіо.

Формати аудіофайлів:

При виборі формату аудіо файлів потрібно враховувати що тип файлу має значення та впливає на якість і обсяг даних. Наприклад, WAV не має настільки сильного стиснення даних через що він є форматом, який зберігає аудіо майже без втрати якості. Він рекомендується для обробки та аналізу аудіо сигналів [13]. FLAC представляє собою формат з безвтратним стисненням, який дозволяє зберігати високу якість аудіо, одночасно зменшуючи розмір файлу майже вдвічі

у порівнянні з WAV, що стає в нагоді при зберіганні великих обсягів даних без компромісу щодо якості. З іншого боку, MP3 та AAC є стисненими форматами з втратами, які можуть не підходити для аналізу, де важлива висока якість звуку. Ці формати більше орієнтовані на зменшення розміру файлу за рахунок втрат в якості, що робить їх менш придатними для завдань, що потребують точного відтворення та обробки аудіо сигналів.

Параметри аудіо:

Крім вибору формату, важливо також враховувати параметри аудіо, які впливають на якість та ефективність обробки даних. Частота дискретизації зазвичай становить 44.1 кГц або 48 кГц. Чим вища частота дискретизації аудіо тим краща якість та більший розмір кінцевого файлу. Вибір конкретного значення залежить від специфіки задачі та вимог до якості [18]. Наприклад, вища (44.1 кГц і більше) частота дискретизація може більш точно відтворювати високочастотні компоненти звуку, тому для задач покращення якості аудіо важливо мати високий рівень частоти дискретизації. Водночас, для задач обробки голосу достатньо буде і середнього рівня частоти дискретизації (16 кГц – 20 кГц). Кількість біт також є важливим параметром, який визначає динамічний діапазон аудіо сигналу. Стандартні значення – 16-біт або 24-біт. Вища бітність створює більший динамічний діапазон, що дозволяє зберігати більше деталей у звукових сигналах. Також аудіо може бути моно або стерео. Цю характеристику визначає кількість каналів (1 для моно, 2 для стерео). Для спрощення обробки часто використовують моно сигнал, оскільки він потребує менше обчислювальних ресурсів і зберігає необхідну інформацію для багатьох завдань аналізу. Однак у задачах які вимагають високої точності стерео сигнал може бути необхідним, бо дозволяє зберегти просторову інформацію звуку.

Баланс та різноманітність даних:

Для того що б навчання моделі проходило максимально ефективно, дані повинні бути різноманітними: якщо задача передбачає класифікацію, необхідно

мати приблизно однакову кількість зразків для кожної категорії, а використання аудіо з різних джерел підвищує здатність моделі узагальнювати.

2.2.2 Аугментація з використанням Audiomentations

Аугментація даних – це техніка штучного збільшення обсягу датасету шляхом застосування різних трансформацій до існуючих зразків. Це дозволяє збільшити обсяг та різноманітність датасету без додаткових затрат на збір нових даних. Застосування аугментацій може збільшити різноманітність даних без необхідності збору додаткових зразків [15] та запобігти перенавчанню моделі. Audiomentations – це Python-бібліотека, спеціально розроблена для аугментації аудіо даних [16]. Вона надає набір інструментів для застосування різноманітних аудіо ефектів та перетворень, серед яких:

- Додавання шуму: імітація різних рівнів фонових шумів та додавання білого шуму до сигналу. (2.12)
- Зміна висоти тону: підвищення або пониження тону для створення варіацій.
- Зміна швидкості відтворення: розтягнення або стиснення сигналу в часі.
- Реверберація: імітація різних акустичних просторів.
- Випадкова зміна гучності.

Ці методи допомагають моделі стати стійкішою до змін у вхідних даних та покращують її здатність до узагальнення [15]. Проте у такого підходу є один основний недолік – для задач автоматизованого мастерингу аудіо недостатньо буде використовувати лише аугментований датасет, так як він складатиметься тільки з певного конкретного набору погіршень аудіо, в той час як задача вимагає обробки аудіо додатково з такими недоліками як неправильна компресія, нерівномірна АЧХ, частотні резонанси тощо. Тому для задач покращення якості аудіо необхідно щоб принаймні частина датасету складалася з аудіо погіршеного вручну або із застосуванням вище перелічених недоліків.

Формула накладання шуму на звуковий сигнал:

$$y_{aug}(t) = y(t) + \alpha \cdot n(t) , \quad (2.14)$$

де:

- $y(t)$ – оригінальний аудіо сигнал.
- $n(t)$ – шумовий сигнал.
- α – коефіцієнт масштабу шуму.

Приклад використання Audiomentations:

```
from audiomentations import Compose, AddGaussianNoise, TimeStretch, PitchShift
augment = Compose([
    AddGaussianNoise(min_amplitude=0.001, max_amplitude=0.015, p=0.5),
    TimeStretch(min_rate=0.8, max_rate=1.25, p=0.5),
    PitchShift(min_semitones=-4, max_semitones=4, p=0.5)
])
augmented_samples = augment(samples=samples, sample_rate=sample_rate)
```

Вплив аугментації на модель:

Аугментація може покращити узагальнюючу здатність моделі та підвищити її стійкість до різних шумів та спотворень [15]. Однак важливо не перевантажувати дані надмірною аугментацією, оскільки це може призвести до зниження точності моделі на реальних даних.

2.2.3 Підготовка даних

Після збору та аугментації даних необхідно провести їх підготовку:

- Нормалізація: вирівнювання рівнів гучності між різними аудіо файлами.

- Фрагментація: у випадку якщо датасет складається з треків різних за довжиною, необхідно розділити їх на однакові за довжиною фрагменти
- Конвертація формату: приведення всіх файлів до єдиного формату (WAV), частоти дискретизації (наприклад, 16 кГц) та розрядності (16 біт) [18].

Загалом виділяють кілька методів нормалізації аудіо, серед яких:

- Пікова нормалізація: Масштабує сигнал так, щоб його максимальна амплітуда дорівнювала заданому значенню. (2.13)
- RMS нормалізація: Масштабує сигнал на основі його середньоквадратичного значення. (2.14, 2.15)

$$y_{norm}(t) = \frac{y(t)}{\max|y(t)|}. \quad (2.15)$$

$$y_{norm}(t) = \frac{y(t)}{RMS(y(t))}, \quad (2.16)$$

де:

$$RMS(y(t)) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} y_i^2}. \quad (2.17)$$

Також рекомендується конвертувати всі аудіо файли до одного формату з однаковими параметрами [13]. В даному випадку буде проведено конвертування кожного аудіо файлу в формат WAV з частотою дискретизації 44.1 кГц, бітовою глибиною 16 біт. Для прикладу кожен трек буде переведено в моно задля забезпечення спрощення аналізу.

Інструменти для конвертації:

- FFmpeg: Потужний і зручний інструмент командного рядка для обробки аудіо та відео файлів.

Приклад команди:

ffmpeg -i input.mp3 -ar 44100 -ac 1 -sample_fmt s16 output.wav

- Librosa: Python-бібліотека для аналізу аудіо сигналів [13].

Також бажано видалити всі фрагменти тиші наявні в аудіо. Зазвичай тиша знаходиться на початку аудіо та в його кінці. Якщо аудіо файли містять довгі записи, їх можна сегментувати на менші фрагменти для спрощення обробки та збільшення кількості зразків.

Анотування даних:

Якщо задача вимагає наявності міток (наприклад, для класифікації), необхідно забезпечити коректне анотування аудіо зразків [14]:

- Ручне анотування: Прослуховування аудіо та призначення міток вручну.
- Автоматичне анотування: Використання алгоритмів для автоматичного призначення міток (може вимагати подальшої перевірки).

Розбиття на тренувальний, валідаційний та тестовий набори:

При підготовці даних для навчання моделей нейронних мереж потрібно правильно організувати структуру датасету. Для оцінки продуктивності моделі необхідно розділити датасет на три основні частини: тренувальний набір, валідаційний набір та тестовий набір. Тренувальний набір використовується для безпосереднього навчання моделі, дозволяючи їй вивчати закономірності та залежності у даних. Валідаційний набір застосовується для налаштування гіперпараметрів моделі та запобігання перенавчанню, підвищуючи її продуктивність без втрати здатності до узагальнення на нових даних. Тестовий набір служить для остаточної оцінки продуктивності моделі та вимірювання її точності і ефективності на незалежних від навчального процесу даних [21].

Типові пропорції розбиття датасету складають приблизно 70-80% для тренувального набору, 10-15% для валідаційного набору та 10-15% для тестового набору. За такого розподілу зберігається баланс між достатньою кількістю даних для навчання моделі та наявністю окремих наборів для її оцінки та налаштування.

Для належного рівня контролю якості моделей машинного навчання потрібно перевіряти аудіо файли на предмет пошкоджень або помилок. Для цього використовуються спеціалізовані скрипти, які автоматично перевіряють цілісність файлів та виявляють можливі пошкодження або некоректні дані. В свою чергу, аналіз розподілу даних допоможе виявити дисбаланси або аномалії у наборі даних. Це досягається за допомогою візуалізації та статистичного аналізу, що дозволяє ідентифікувати нерівномірне представлення певних класів або відхилення від очікуваного розподілу [19]. Виявлення таких проблем на ранніх етапах підготовки даних підвищує якість кінцевої моделі нейронної мережі та її здатності до точного прогнозування.

Також варто враховувати правові та етичні аспекти при використанні даних для створення датасету. Необхідно переконатися, що використання аудіо файлів не порушує авторських прав, а також якщо дані містять особисту інформацію (наприклад, записи голосів людей), необхідно дотримуватися норм конфіденційності.

Також процес підготовки даних можна автоматизовувати шляхом створення спеціальних скриптів або використання існуючих інструментів які автоматизовуватимуть наступні процеси:

- Завантаження та конвертація файлів.
- Аугментація даних.
- Нормалізація та фільтрація сигналів.
- Сегментація
- Розбиття датасету та створення метаданих.

Загалом, автоматизація зменшує ймовірність помилок та підвищує ефективність процесу [22].

2.3 Розробка нейромережевої моделі

Наступним етапом у створенні системи автоматизації процесу мастерингу аудіо з використанням технологій машинного навчання є безпосередньо розробка нейромережевої моделі для обробки аудіо сигналів. Цей розділ присвячений детальному опису процесу проектування архітектури моделі, вибору шарів, функцій активації, кількості нейронів, а також особливостей оптимізації для аудіо задач.

2.3.1 Архітектура моделі

Для розробки ефективної моделі обробки аудіо сигналів необхідно обрати відповідну архітектуру нейронної мережі. Для прикладу буде детально розглянуто модель нейронної мережі U-Net, адаптованої для аудіо задач [23]. U-Net була успішно застосована в обробці аудіо для задач розділення джерел звуку, шумозаглушення та покращення якості звуку [24, 26].

Структура моделі U-Net:

1. Контрактивна (стискаюча) частина (Енкодер):
    a. Послідовність згорткових шарів з функцією активації ReLU.
    b. Зменшення розмірності даних за допомогою Max Pooling.
    c. Витягнення високорівневих ознак з аудіо сигналу.
2. Експансивна (розширююча) частина (Декодер):
    a. Послідовність транспонованих згорткових шарів.
    b. Відновлення розмірності даних до початкового розміру.
    c. Використання Skip Connections для поєднання відповідних шарів енкодера та декодера, що дозволяє зберігати деталі сигналу [23].

Вхідний шар моделі приймає аудіосигнал у вигляді спектрограми. Використання саме спектрограм дозволяє застосовувати двовимірні згортки, ефективні для аналізу часово-частотних ознак [15]. Згорткові шари в свою чергу виконують згортку з ядрами фіксованого розміру (наприклад, 3×3). Також

згорткові шари містять функцію активації, в даному випадку – ReLU (2.18) для того щоб визначити як саме перетворити дані і передати їх наступному шару.

$$\mathrm{ReLU}(x) = \max(0, x)\ . \qquad (2.18)$$

Також на кожному рівні енкодера збільшується кількість фільтрів (наприклад, 64, 128, 256, 512) [24].

За зменшення розмірності даних вдвічі по кожній осі та за допомогу моделі вивчати більш абстрактні ознаки відповідає такий компонент нейронної мережі U-Net як Max Pooling. Транспоновані згортки ж використовуються в декодері для збільшення розмірності даних та поєднуються з відповідними шарами енкодера через Skip Connections, за допомогою яких здійснюється:

- Збереження інформації про дрібні деталі сигналу.
- Конкатенація ознак з енкодера та декодера (2.19):

$$g_i = \mathrm{Concat}(g_{i+1}, f_i)\ , \qquad (2.19)$$

де:

- g_{i+1} – ознаки з попереднього рівня декодера, f_i – ознаки з відповідного рівня енкодера.

Вихідний шар, залежно від задачі, використовує відповідну функцію активації:

- Сигмоїдна функція для бінарних задач: (2.20)
- Тангенс гіперболічний для регресійних задач: (2.21)

$$\sigma(x) = \frac{1}{1+e^{-x}}\ . \qquad (2.20)$$

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \quad (2.21)$$

Зазвичай, модель нейронної мережі U-Net має 4-5 рівнів згорткових шарів в енкодері та декодері [23]. Кількість фільтрів залежить від задачі і зазвичай починається з 64, подвоюючись на кожному рівні енкодера, потім зменшується на кожному рівні декодера. Також архітектура моделі нейронної мережі U-Net включає в себе наявність активаційних функцій. Вони приймають активну участь у забезпеченні нелінійності моделі та покращенні її здатності до навчання. Серед найпоширеніших активаційних функцій в даному випадку варто виділити дві:

1. ReLU (Rectified Linear Unit): Швидка та ефективна, допомагає уникнути проблеми зникання градієнтів [18]. Проте, стандартна ReLU може призводити до проблеми «вмирання нейронів», коли деякі нейрони перестають активуватися під час навчання.

2. Leaky ReLU: Вирішує проблему «вмирання нейронів» шляхом введення невеликого градієнта для від'ємних значень [18] (2.22):

$$Leaky\ ReLU(x) = \begin{cases} x, & x > 0 \\ x, & \alpha x \leq 0 \end{cases}, \quad (2.22)$$

де α – невелике додатне число (наприклад, 0.01).

Для стабілізації та прискорення навчання моделі використовується так звана пакетна нормалізація (Batch Normalization) [18] (2.23).

$$\hat{x} = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}, \quad (2.22)$$

де μ та σ – середнє значення та дисперсія в межах міні-пакету.

2.3.2 Особливості оптимізації для аудіо задач

Особливості оптимізації для аудіо задач полягають у виборі відповідних функцій втрат, оптимізаторів та методів регуляризації, для ефективного навчання моделі. Від обраної функції втрат буде залежати те, як модель коригуватиме свої прогнозування. Для регресійних задач, таких як відновлення аудіо сигналу, зазвичай використовується середньоквадратична похибка (MSE) [20]. Ця функція обчислюється за формулою: (2.23)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 , \qquad (2.23)$$

де:

- y_i – істинні значення.
- $\hat{y}_i$ – прогнозовані моделлю значення.

Для класифікаційних задач з бінарними або багатокласовими мітками часто застосовується крос-ентропія [20], яка визначається як: (2.24)

$$\text{CE} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) . \qquad (2.24)$$

Вибір оптимізатора є ще одним аспектом оптимізації моделі. Для прикладу, оптимізатор Adam – широко використовується завдяки своїй здатності адаптивно змінювати швидкість навчання для кожного параметра і поєднувати переваги методів AdaGrad та RMSProp [17]. Процес оновлення параметрів у Adam здійснюється за такими формулами: (2.25 – 2.28)

- Оновлення моментів першого порядку:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t , \qquad (2.25)$$

- Оновлення моментів другого порядку:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 , \qquad (2.26)$$

- Корекція зміщення:

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t}, \quad \widehat{v_t} = \frac{v_t}{1-\beta_2^t}, \tag{2.27}$$

- Оновлення параметрів:

$$\theta_{t+1} = \theta_t - \alpha \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}}+\epsilon}, \tag{2.28}$$

де:

- α – швидкість навчання.
- β_1 і β_2 – коефіцієнти експоненційного згладжування.
- ϵ – мале число для запобігання діленню на нуль.
- g_t– градієнт на кроці t

Для того щоб запобігти перенавчанню моделі нейронної мережі застосовуються методи регуляризації. Один із таких методів – Dropout. Його дія полягає у тому, що він випадково вимикає нейрони під час навчання, що допомагає уникнути перенавчання та покращує здатність моделі до узагальнення [18]. Також широко використовується L2-регуляризація, яка додає до функції втрат суму квадратів вагів. Це зменшує складність моделі нейронної мережі та запобігає її перенавчанню.

Налаштування гіперпараметрів, таких як швидкість навчання α, розмір міні-пакету (batch size) та кількість епох навчання, суттєво впливає на ефективність навчання моделі. Оптимальні значення цих параметрів можуть значно змінювати швидкість збіжності та кінцеву продуктивність моделі. Для визначення найкращих значень гіперпараметрів застосовують різноманітні методи. Байєсівська оптимізація – один з ефективних підходів, який використовує попередні результати для вибору наступних комбінацій гіперпараметрів [22]. Крім того, простіші методи, такі як Grid Search та Random Search, також можуть допомогти у пошуку оптимальних значень у заданих діапазонах. Під час розробки моделі бажано використовувати останні версії

бібліотек та інструментів. Фреймворки TensorFlow та PyTorch представляють собою потужні засоби для розробки та навчання нейронних мережевих моделей [19]. Високорівнева API Keras спрощує процес побудови моделей, та дозволяє швидко створювати різні за складністю архітектури. Для аналізу аудіо сигналів та їх перетворення у спектрограми корисною є бібліотека Librosa [15], а TensorBoard надає інструменти для візуалізації процесу навчання, моніторингу метрик та аналізу моделі [19].

Практичні аспекти розробки моделі включають підготовку даних. Аудіо сигнали часто перетворюються у спектрограми для подальшої обробки. Короткочасне перетворення Фур'є (STFT) використовується для отримання часово-частотного представлення сигналу: (2.29)

$$X(\omega,\tau) = \int_{-\infty}^{\infty} x(t)w(t-\tau)e^{-j\omega t}dt\,, \qquad (2.29)$$

де $w(t)$ – віконна функція.

Крім того, використовуються мел-спектрограми, які перетворюють частотну шкалу на мел-шкалу, що краще відповідає людському суб’єктивному сприйняттю звуку. Як було зазначено в попередньому пункті, важливо правильно розподілити датасет на тренувальний, валідаційний та тестовий набори [21], щоб оцінка продуктивності моделі була об’єктивною та щоб запобігти перенавчанню моделі нейронної мережі. Зручний моніторинг метрик, таких як функція втрат та точність, дозволяє відстежувати прогрес навчання та своєчасно виявляти можливі проблеми. Для оцінки моделі використовуються різні метрики якості. Наприклад, Signal-to-Noise Ratio (SNR) вимірює співвідношення сигналу до шуму, Perceptual Evaluation of Speech Quality (PESQ) оцінює якість мовлення з точки зору людського сприйняття, а Mean Opinion Score (MOS) є середньою оцінкою, яку дають слухачі [25]. Порівняння з базовими моделями (Baseline) допомагає визначити, наскільки нова модель покращує результати порівняно з простішими або попередніми підходами.

На завершення, розробка моделі нейронної мережі для автоматизації процесу мастерингу аудіо потребує відповідального і ретельного підходу до вибору архітектури, функцій активації та оптимізаторів, оскільки кожен з цих аспектів вносить свій відчутний вклад в те, наскільки швидко буде проходити навчання моделі, наскільки багато ресурсів буде споживати процес навчання, і, головне, наскільки точною буде натренована модель нейронної мережі безпосередньо в задачах мастерингу аудіо. Адаптована для аудіо задач модель U-Net, ефективно вирішуватиме проблеми обробки аудіо сигналів і забезпечить високу якість результатів [24, 26]. Особливу увагу слід приділити оптимізації та налаштуванню гіперпараметрів як і використанню сучасних інструментів для моніторингу та оцінки моделі. Сукупність всіх цих факторів сприятиме успішному впровадженню технологій машинного навчання в процес мастерингу аудіо.

2.4 Навчання моделі

Навчання нейромережевої моделі є одним з останніх етапів у створенні системи автоматизації процесу мастерингу аудіо з використанням технологій машинного навчання. У цьому процесі вибір функції втрат, налаштування параметрів навчання та використання інструментів для моніторингу приноситимуть найбільшу користь.

Одним із основних аспектів є вибір відповідної функції втрат, яка визначає, як модель оцінює свої помилки та коригує ваги під час навчання [20]. Для регресійних задач, пов'язаних з обробкою аудіо сигналів, часто застосовується середньоквадратична похибка (MSE) (2.23). Абсолютна середня похибка (MAE) використовується для зменшення впливу викидів (outliers) (2.30):

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \,. \qquad (2.30)$$

Іноді доцільно поєднувати MSE та MAE для врахування різних аспектів помилки (2.31):

$$\text{Loss} = \alpha \cdot \text{MSE} + (1 - \alpha) \cdot \text{MAE}\,, \qquad (2.31)$$

де α – ваговий коефіцієнт, що визначає внесок кожної складової.

Перцептивні функції втрат враховують особливості людського суб'єктивнгого сприйняття звуку. Наприклад, Мел-кепстральна відстань (Mel-Cepstral Distance) дозволяє оцінити різницю між спектральними характеристиками сигналів [21]. Налаштування параметрів навчання, таких як швидкість навчання (α) та розмір міні-пакету (batch size), впливає як на ефективність та стабільність процесу самого навчання [18], так і на використання процесом навчання ресурсів. Швидкість навчання визначає розмір кроку при оновленні вагів моделі. Надто велике значення може призвести до нестабільного навчання або до зникнення градієнта, тоді як надто мале – до повільної збіжності.

Швидкість навчання можна налаштувати різними способами:

1. Використання фіксованого значення протягом усього процесу.
2. Адаптивні оптимізатори, такі як Adam, автоматично змінюють швидкість навчання для кожного параметра [17].
3. Поступове зменшення швидкості навчання за певним графіком.

Розмір міні-пакету суттєво впливає на стабільність градієнтних оцінок. Менші розміри можуть покращити узагальнюючу здатність моделі, проте збільшують час навчання через більшу кількість оновлень [18]. Оптимальні значення цих параметрів часто визначають експериментально, хоч і існують такі методи як Grid Search або Random Search, що дозволяють перебрати можливі комбінації параметрів автоматично [22]. Байєсівська оптимізація є більш ефективним підходом, оскільки використовує попередні результати для інформованого вибору наступних значень [22].

Моніторинг процесу навчання допомагає виявити проблеми та оптимізувати модель [19]. Автоматизацію моніторингу може впровадити TensorBoard – інструмент, який дозволяє візуалізувати криві навчання, структуру моделі, розподіли вагів та інші аспекти. Це значно спрощує аналіз функції втрат та точності на тренувальному та валідаційному наборах. Для використання TensorBoard необхідно виконувати процес логування метрик під час навчання. Наприклад, у Python [27] з використанням PyTorch та TensorBoard це можна зробити так:

```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter(log_dir='logs')
for epoch in range(num_epochs):
    # Навчання моделі
    # ...
    writer.add_scalar('Loss/train', loss_value, epoch)
    writer.add_scalar('Accuracy/val', accuracy_value, epoch)
writer.close()
```

Після цього TensorBoard запускається командою у терміналі (за умови виконання активаційного скрипта віртуального оточення Python проекту):

```
tensorboard --logdir=logs
```

Після виконання команди, якщо встановлені всі необхідні залежності, TensorBoard стає доступним у браузері за адресою http://localhost:6006 (за необхідності порт можна змінити на інший). Подібний моніторинг дозволяє своєчасно виявити перенавчання або зникання градієнту та застосувати методи, такі як рання зупинка, якщо функція втрат на валідаційному наборі перестає зменшуватися [18].

Також навчання моделі включає ініціалізацію вагів, підготовку даних, цикл навчання з прямим та зворотним проходом, валідацію та тестування. Під час навчання модель постійно порівнює свої прогнозування з істинними значеннями, коригуючи ваги на основі обчислених помилок. Запобігання перенавчанню досягається завдяки процесу регуляризації та застосування методів, таких як Dropout або L2-регуляризація [18]. Метод ранньої зупинки також допомагає зупинити навчання до того, як модель почне перенавчатися на тренувальних даних.

Продуктивність моделі оцінюється за допомогою метрик, таких як MSE, Signal-to-Noise Ratio (SNR) та перцептивних метрик, як-от PESQ та STOI [25]. Візуалізація результатів через спектрограми та суб'єктивне прослуховування допомагає оцінити якість обробленого звуку. Порівняння з іншими моделями, включаючи базові та сучасні архітектури, дозволяє визначити ефективність розробленої моделі [14].

3. РОЗРОБКА ТА ВПРОВАДЖЕННЯ СИСТЕМИ

У процесі розробки системи автоматизованого аудіо мастерингу було створено два прототипи, кожен з яких базується на різних архітектурах нейронних мереж: LSTM та Wave-U-Net. Обидва прототипи реалізовані мовою Python з використанням фреймворку PyTorch для глибокого навчання та бібліотеки torchaudio для обробки аудіо сигналів. Графічний інтерфейс користувача було розроблено з використанням бібліотеки Tkinter у першому прототипі, другий же прототип не має графічного інтерфейсу, і всі операції виконуються шляхом введення консольних команд з різним набором аргументів.

3.1 Архітектура програмного рішення

3.1.1 Прототип на основі LSTM

Модель CRNNAudioModel поєднує згорткові шари та рекурентний шар LSTM для обробки аудіо сигналів. Такий підхід відповідає архітектурі CRNN (Convolutional Recurrent Neural Network), яка широко використовується для аналізу послідовних даних, включаючи аудіо та відео [47].

Структура моделі LSTM:

1. Згорткові шари: три послідовні згорткові шари з активацією ReLU витягують локальні особливості з аудіо сигналу. Перший шар приймає стерео сигнал з 2 каналами та генерує 16 вихідних каналів. Наступні шари збільшують кількість каналів до 32 та 128 відповідно.
2. Рекурентний шар LSTM: двонаправлений LSTM шар з 64 нейронами моделює часові залежності в сигналі. Використання двонаправленого LSTM дозволяє моделі враховувати як попередні, так і наступні стани сигналу [34].
3. Повнозв'язний шар: лінійний шар зменшує кількість каналів до 2, відновлюючи стерео формат сигналу.

Переваги використання LSTM:

- Моделювання довгострокових залежностей: LSTM може зберігати інформацію про попередні стани сигналу на довгі часові інтервали завдяки механізмам забування та запам'ятовування [33].
- Двонаправлений LSTM: враховує інформацію з майбутніх станів, що може бути корисно при обробці статичних аудіо файлів.

3.1.2 Обробка аудіо у прототипі LSTM

Модуль обробки аудіо включає конвертацію аудіо файлів з формату MP3 у WAV з використанням програми ffmpeg, нормалізацію сигналу та сегментацію аудіо для ефективної обробки. Функція process_audio_with_model обробляє аудіо сегментами з використанням моделі, що дозволяє оптимізувати використання пам'яті та ресурсів процесора. В прототипі реалізовано два варіанти навчання – парами заздалегідь підготовлених аудіо (трек в хорошій якості та трек в поганій), та за допомогою бібліотеки Audiomentations, в такому випадку достатньо мати лише файли в хорошій якості.

3.1.3 Графічний інтерфейс користувача (GUI) у прототипі LSTM

GUI розроблено з використанням бібліотеки Tkinter [28], яка входить до стандартної бібліотеки Python та не потребує додаткової установки. Інтерфейс складається з двох вкладок та надає можливості вибору аудіофайлу для обробки, завантаження навченої моделі (перша вкладка), налаштування параметрів навчання (кількість епох, швидкість навчання), запуску процесу обробки та збереження результатів (друга вкладка).

3.1.4 Збереження результатів у прототипі LSTM

Оброблені аудіо файли зберігаються у вибраному користувачем місці через GUI. Моделі також можуть бути збережені після навчання, що дозволяє їх повторне використання без необхідності перенавчання.

3.1.5 Прототип на основі Wave-U-Net

Модель використовує архітектуру Wave-U-Net, яка є модифікацією оригінальної U-Net, що спочатку була розроблена для сегментації зображень у медичній інформатиці [23].

Структура моделі Wave-U-Net:

1. Енкодер: утворений послідовністю згорткових блоків з двома згортковими шарами в кожному, активацією ReLU та нормалізацією із застосуванням BatchNorm. Після кожного блоку застосовується Max Pooling для зменшення розмірності сигналу та витягування патернів та особливостей.
2. Ботлнек: центральний шар мережі з найбільшою кількістю фільтрів, який концентрує інформацію.
3. Декодер: послідовність розгорткових (транспонованих згорткових) шарів, які збільшують розмірність сигналу. Після кожного розгорткового шару сигнал конкатенується з відповідним шаром енкодера завдяки механізму Skip Connection, що дозволяє передавати деталі високої роздільної здатності від енкодера до декодера.
4. Фінальний згортковий шар: зменшує кількість каналів до необхідного вихідного (2 для стерео у випадку прототипу).

Серед переваг використання моделі нейронної мережі архітектури U-Net можна виділити збереження просторової інформації через те що завдяки Skip Connection модель зберігає деталі сигналу на всіх рівнях абстракції [23] та гнучкість самої архітектури – U-Net може бути адаптована для різних розмірностей даних, включаючи одномірні сигнали [24]. Wave-U-Net є такою адаптацією, і саме вона була використана в прототипі через свою здатність працювати напряму зі звуком.

3.1.6 Обробка аудіо у прототипі Wave-U-Net

Підготовка даних здійснюється за допомогою класу AudioDataset, який створює датасет з високоякісних та низькоякісних аудіо. Також доступне

навчання парами аудіо в хорошій і поганій якості, так само як і можливість тренувати модель за допомогою GTZAN Dataset. Для збільшення різноманітності даних застосовуються різні аудіо аугментації, такі як додавання шуму, зміна висоти тону, розтягування у часі тощо [36]. Функція enhance_audio обробляє аудіо з використанням навченої моделі.

3.1.7 Взаємодія з програмою у прототипі Wave-U-Net

У цьому прототипі взаємодія здійснюється через командний рядок з використанням аргументів командного рядка. Це дозволяє більш гнучко налаштовувати параметри обробки та автоматизувати процес, хоч і не так зручно як керування програмою через графічний користувацький інтерфейс. Наприклад, користувач може вказати шлях до моделі, вхідного та вихідного аудіо файлів, а також налаштувати довжину сегмента на які будуть розділятися треки для обробки.

3.1.8 Збереження результатів у прототипі Wave-U-Net

Оброблені аудіо файли зберігаються у визначеному користувачем місці, при цьому параметри збереження можуть бути налаштовані через аргументи командного рядка.

3.2 Інтеграція моделі в програмне забезпечення

3.2.1 Використання PyTorch для роботи з моделлю

Фреймворк PyTorch було обрано для реалізації та навчання моделей нейронних мереж в обох прототипах [29]. PyTorch підтримує динамічні обчислювальні графи, що спрощує процес розробки та дебагінгу. Крім того, він має велику спільноту розробників та широкий набір інструментів для роботи з глибоким навчанням.

3.2.2 Інтеграція моделі LSTM

Модель LSTM інтегровано в GUI, що дозволяє користувачу взаємодіяти з нею безпосередньо через графічний інтерфейс. Функція load_model_weights_with_fallback забезпечує можливість завантаження вагів моделі з файлу, ігноруючи невідповідності в розмірностях шарів, що підвищує стійкість програми до змін у структурі моделі. Обробка аудіо здійснюється за допомогою функції process_audio_with_model, яка розбиває аудіо на сегменти та обробляє кожен з них окремо. Це дозволяє ефективно використовувати ресурси системи та забезпечує стабільну роботу з довгими аудіо файлами. Проте, через те що програма працює через графічний інтерфейс інтегрувати її в інші програми, наприклад, в VST плагін стає проблематично.

3.2.3 Інтеграція моделі Wave-U-Net

У прототипі Wave-U-Net керування процесами навчання та обробки аудіо здійснюються через командний рядок. Це надає можливість автоматизувати процес та інтегрувати його в інші системи чи скрипти. Модель можна навчити на власному датасеті, налаштувавши параметри навчання через аргументи командного рядка.

Серед особливостей інтеграції можна виділити:

- Сегментація аудіо: В обох прототипах аудіо розбивається на сегменти для забезпечення консистентності датасету, ефективної обробки та зменшення використання пам'яті. Це позитивно впливає на використання ресурсів при роботі з довгими аудіо файлами, де обробка всього файлу за один раз може призвести до перевантаження пам'яті [37].
- Використання GPU: Код передбачає можливість використання GPU для прискорення обчислень. Це забезпечує швидше навчання моделей та обробку аудіо, особливо при роботі з великими обсягами даних [29]. Важливо зауважити, що тренування моделі нейронної мережі доступне лише на GPU з підтримкою технології CUDA оскільки бібліотека PyTorch працює лише з цим типом ядер.

- Модульність коду: Функціональність програми розділена на окремі модулі (наприклад, модель, датасет, утиліти, файл конфігурації), що полегшує підтримку та розширення системи. Такий підхід відповідає принципам модульного програмування та спрощує процес розробки [1].

Додаткові джерела для поглиблення знань:

- LSTM: Для детальнішого розуміння роботи LSTM варто звернутися до оригінальної роботи Хохрейтера та Шмідхубера [33], де вони представили цю архітектуру та описали її переваги у моделюванні довгих послідовностей.
- Двоаправлений LSTM: Робота Грейвса та Шмідхубера [34] демонструє ефективність двонаправлених LSTM у задачах класифікації послідовностей та розпізнавання мови.
- U-Net: Оригінальна стаття Роннебергера та ін. [23] детально описує архітектуру U-Net та її застосування у сегментації медичних зображень. Вона підкреслює важливість механізму Skip Connection для збереження просторової інформації.
- Адаптація U-Net для аудіо: У роботі Янссона та ін. [24] показано, як здатність архітектури U-Net до сегментації зображень можна адаптувати для задач обробки аудіо, таких як відокремлення голосу від музичного супроводу.
- Аугментація аудіо даних: Стаття Парізі та ін. [36] описує різні методи аугментації аудіо даних та їх вплив на ефективність навчання моделей.

3.3 Опис програмного коду прототипів

3.3.1 Застосунок на основі LSTM

Застосунок на основі LSTM складається з кількох файлів та класів, які забезпечують повний цикл обробки аудіо від завантаження та підготовки даних до навчання моделі та обробки нових треків.

Файл main.py є основною точкою входу в програму. Він налаштовує тимчасові директорії та кэш PyTorch для оптимізації роботи на наявному обладнанні. Крім того, він імпортує необхідні модулі та визначає клас

AudioMasteringApp, який відповідає за графічний інтерфейс користувача (рис. 3.1, 3.2). Інтерфейс складається з двох вкладок: «Audio Processing» (рис. 3.1) та «Model Training» (3.2). У вкладці «Audio Processing» користувач може вибирати аудіо треки для обробки, завантажувати навчені моделі та зберігати оброблені треки. У вкладці «Model Training» надаються інструменти для вибору режиму навчання, налаштування гіперпараметрів та запуску процесу навчання моделі.

Файл model.py містить визначення класу CRNNAudioModel, який реалізує архітектуру CRNN (Convolutional Recurrent Neural Network) для обробки аудіо сигналів. Модель складається з трьох основних частин: згорткових шарів для виділення локальних ознак з аудіо сигналу, рекурентного шару LSTM для обробки послідовних даних та повнозв’язного шару для генерації вихідного сигналу.

Файл train.py відповідає за процес навчання моделі. Він включає функцію train_model, яка організовує завантаження даних, налаштування оптимізатора та функції втрат, а також сам процес навчання. Внутрішній клас AudioDataset відповідає за завантаження та попередню обробку аудіо даних. В залежності від режиму навчання, він може працювати з парами треків «поганий-хороший» або використовувати аугментації для генерації «поганих» треків з «хороших». Також реалізовано логування втрат у TensorBoard для моніторингу процесу навчання.

Файл utils.py містить допоміжні функції, необхідні для роботи застосунку. Функції включають конвертацію аудіо файлів з формату MP3 у WAV, завантаження вагів моделі з можливістю пропуску несумісних параметрів, обробку аудіо треків за допомогою моделі, а також нарізання аудіо файлів на сегменти фіксованої довжини. Функції convert_mp3_to_wav та convert_mp3_to_wav_simple використовують ffmpeg для конвертації форматів аудіо, а функція slice_audio_to_segments дозволяє розбити треки на однакові за довжиною сегменти (код програми знаходиться в додатку Б).

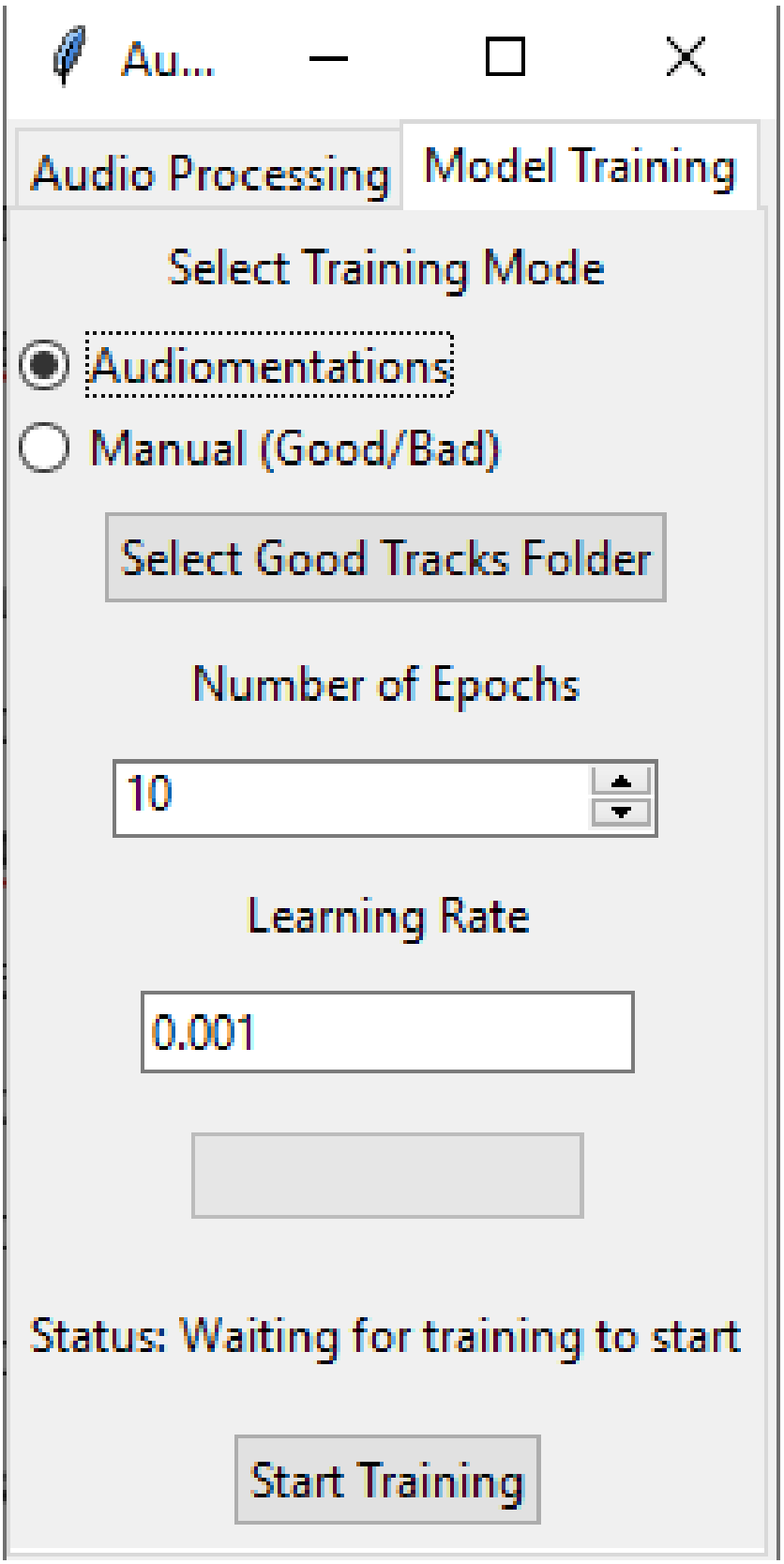


Рис. 3.1 Перша сторінка застосунку LSTM

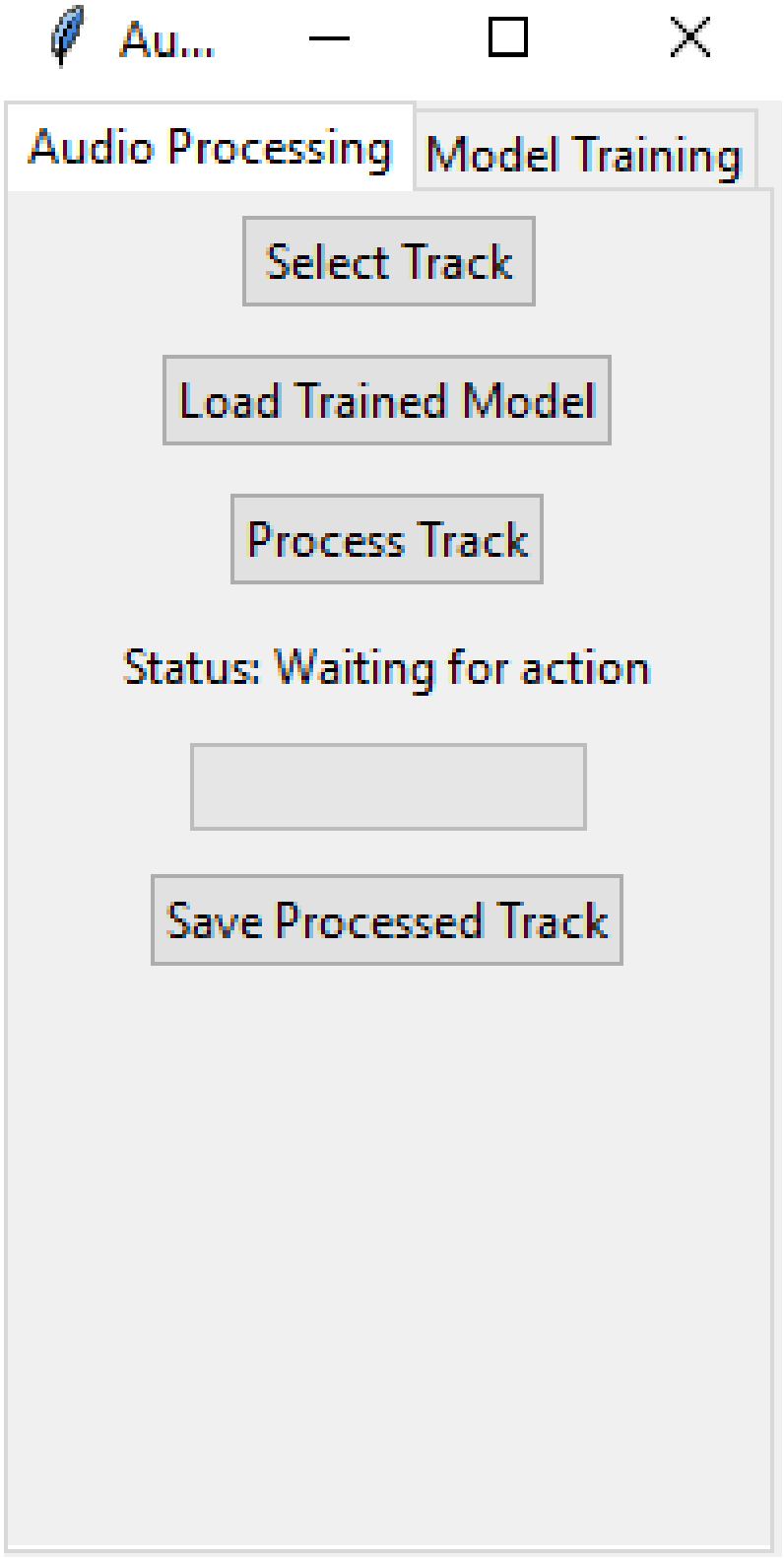


Рис. 3.2 Друга сторінка застосунку LSTM

На рисунку 3.3 зображено блок схему для взаємодії з графічним інтерфейсом застосунку.

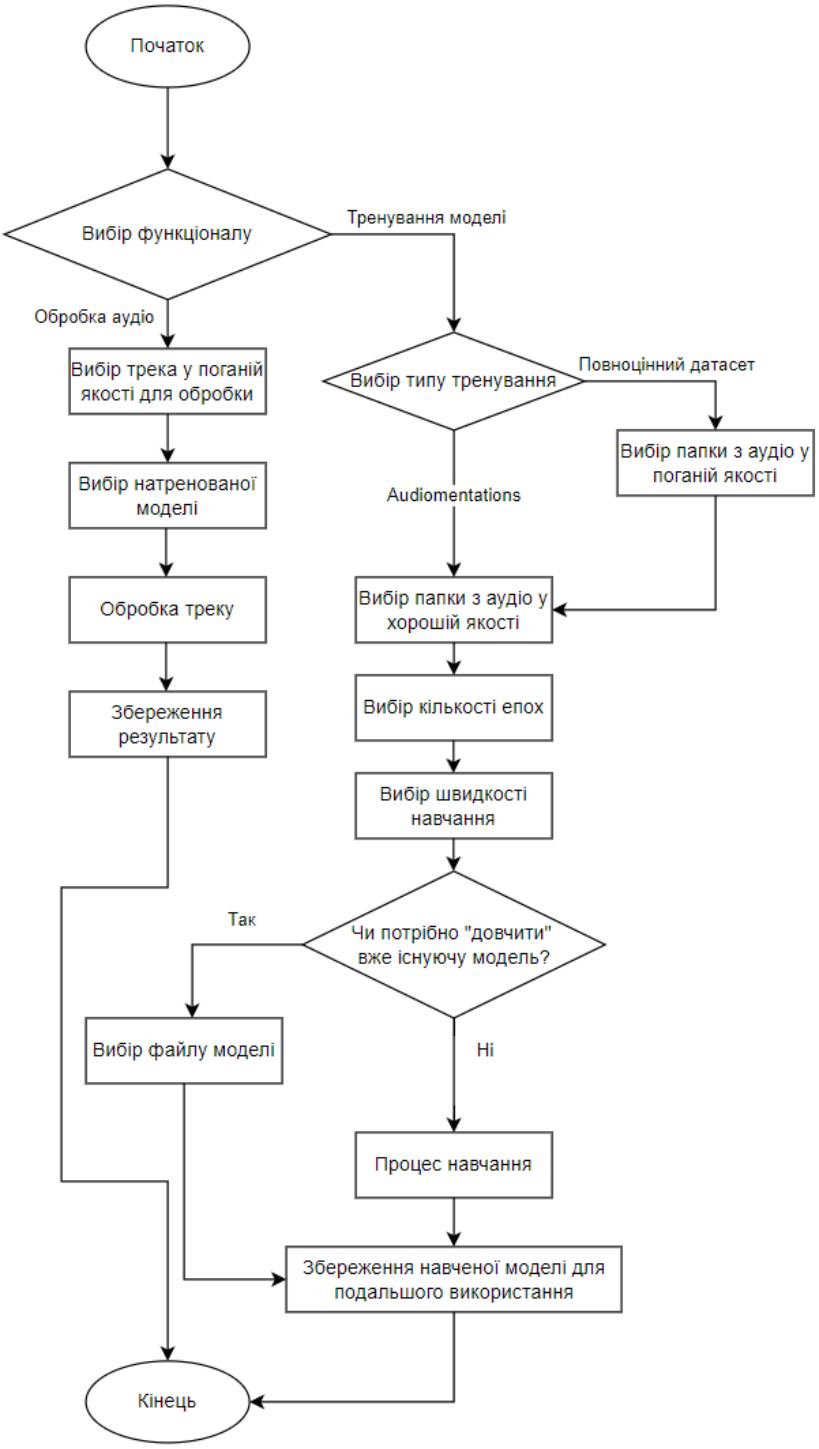


Рис. 3.3 Процес тренування моделі та обробки треків шляхом взаємодії з графічним інтерфейсом застосунку з використанням LSTM

3.3.2 Застосунок на основі Wave-U-Net

Структура проекту складається з декількох файлів і папок, які взаємодіють між собою для забезпечення повного робочого процесу від підготовки даних до обробки аудіо. Основні папки включають data/, де зберігаються вхідні дані, models/ для збереження навченої моделі, та runs/ для логів TensorBoard. Основні файли проекту включають main.py, model.py, train.py, dataset.py, enhance_audio.py, utils.py, augmentations.py, та config.py (код програмного застосунку знаходиться у додатку В).

Файл main.py є основною точкою входу в програму. Він відповідає за обробку аргументів командного рядка та визначення режиму роботи програми: навчання моделі або обробка аудіо. У режимі навчання він імпортує налаштування з файлу config.py та викликає функцію train_model з файлу train.py. У режимі обробки аудіо він використовує клас AudioProcessorApp з файлу enhance_audio.py для застосування навченої моделі до нового аудіо.

Файл model.py містить визначення класу моделі WaveUNet, яка, відповідно, реалізує архітектуру Wave-U-Net. Ця модель складається з енкодера, боттлнеку та декодера. Енкодер і декодер побудовані з використанням шарів згорток (Conv1d) і транспонованих згорток (ConvTranspose1d), відповідно, з активаційними функціями ReLU та опціональним Dropout для регуляризації. Боттлнек є центральною частиною моделі, що зв'язує енкодер і декодер. Модель приймає аудіо сигнал у часових проміжках та виводить оброблений сигнал у тому ж вигляді.

Файл train.py відповідає за процес навчання моделі. Він використовує класи з model.py та dataset.py для побудови моделі та підготовки даних відповідно. Функція train_model завантажує налаштування з файлу config.py, ініціалізує модель WaveUNet, визначає функцію втрат (nn.L1Loss) та оптимізатор (optim.AdamW). Вона також налаштовує TensorBoard для моніторингу процесу навчання. Під час навчання модель проходить через задану кількість епох, в кожній з яких обробляє дані, обчислює втрати та оновлює ваги моделі. Після завершення навчання модель зберігається в папці models/.

Файл dataset.py містить класи датасетів для підготовки та подачі даних у модель. Клас AudioDataset відповідає за завантаження пар «поганий-хороший» треків, їх нормалізацію, нарізання на сегменти та аугментацію. Він забезпечує модель необхідними даними для навчання перетворення «сирих» аудіо в оброблені. Якщо використовується датасет GTZAN, клас GTZANDataset обробляє треки з відповідної папки.

Файл enhance_audio.py реалізує логіку обробки аудіо з використанням навченої моделі. Клас AudioProcessorApp завантажує модель з файлу, вказаного в config.py, нарізає вхідний аудіо файл на сегменти та застосовує модель до кожного сегмента окремо. Після обробки всі сегменти об'єднуються та зберігаються як єдиний аудіо файл з покращеною якістю звуку.

Файл utils.py містить допоміжні функції, такі як конвертація аудіо файлів у формат WAV та нарізання аудіо на сегменти. Ці функції спрощують процес підготовки даних та забезпечують узгодженість формату аудіо файлів, що використовуються в проекті.

Файл augmentations.py містить функції для аугментації аудіо даних. Аугментація допомагає збільшити різноманітність даних для навчання, застосовуючи такі методи, як додавання випадкового шуму або зміна гучності сигналу. Це покращує здатність моделі до узагальнення та підвищує її стійкість до різних видів шумів та викривлень.

Файл config.py служить центральним місцем для зберігання всіх налаштувань та гіперпараметрів проекту. Тут вказуються параметри навчання, такі як кількість епох, швидкість навчання, розмір батчу та використання GPU. Також тут задаються параметри моделі, шляхи до даних, вибір джерел даних (пари треків або GTZAN), довжина сегментів та частота дискретизації. Це дозволяє легко змінювати налаштування проекту без необхідності редагувати код файлів.

Файл requirements.txt містить список всіх залежностей, необхідних для роботи проекту. Це включає бібліотеки torch, torchaudio, tqdm, matplotlib, tensorboard та інші.

Запуск програми та вибір режиму роботи, як було зазначено вище, реалізовується за допомогою команди консолі. Наприклад, для запуску програми в режимі навчання потрібно ввести таку команду (перед вводом кожної з команд необхідно активувати віртуальне середовище python та встановити необхідні залежності):

```
python main.py --mode train
```

Якщо в файлі config.py параметр use_pairs приймає значення True, а параметр use_gtzan значення False, то модель буде тренуватися, використовуючи тільки пари треків, і навпаки. Якщо ж обидва параметри прийматимуть значення True, то модель навчатиметься на датасеті, що складатиметься як з пар треків так і з аугментованих треків GTZAN Dataset.

Для того щоб запустити програму в режимі обробки аудіо потрібно ввести наступну команду (цей режим працюватиме коректно лише за наявності файлу навченої моделі у відповідній папці):

```
python main.py --mode enhance --input path/to/input_audio.wav --output path/to/output_audio.wav
```

3.4 Тестування та оцінка роботи системи

В ході проведення дослідження було проведено процес навчання моделей із використанням розроблених програмних застосунків. Процес навчання моделі LSTM відбувався з використанням потужностей CPU, а процес навчання моделі Wave-U-Net проводився вже з використанням GPU.

Конфігурація машини, на якій відбувалося тренування наступна:

- CPU – AMD Ryzen 7 5800X (8 ядер, 16 потоків, 4.5 ГГц)
- GPU – Nvidia GeForce RTX 4080
- RAM – 32 Гб DDR4 (3600 МГц)

Процес навчання відбувався з використанням комбінованого датасету – 100 пар треків, погіршених вручну, та GTZAN Dataset з аугментаціями. Було обрано лише два режими аугментацій – додавання шуму та зміна гучності оскільки такі режими як сповільнення аудіо та зміна тональності не є показниками якості мастерингу.

Для наочного порівняння швидкості навчання в залежності від налаштувань гіперпараметрів було проведено ряд тестів з різними значеннями гіперпараметрів, а саме:

- Кількість епох – 40, 80 та 100
- Швидкість навчання – 0.001 та 0.0001
- Датасет:
    - Тільки ручний датасет, створений з 100 пар аудіо в хорошій і поганій якостях
    - Ручний датасет та GTZAN Dataset
    - Лише GTZAN Dataset з використанням аугментацій

3.4.1 Дослідження впливу гіперпараметрів на швидкість навчання

Оскільки один сегмент триває 10 секунд, а частота дискретизації дорівнює 41000 Гц можна розрахувати кількість семплів в кожному сегменті перемноживши ці два значення. Кількість семплів становить 441000 для кожного сегменту довжиною в 10 секунд. Через відносно велику тривалість кожного сегменту допустимий розмір батча дорівнює 2. Таким чином модель повинна вміститися в буфер GPU.

Розрахунок кількості батчів на епоху:

- Тільки ручний датасет:
  - Кількість пар сегментів – 1800
  - Кількість батчів на епоху: 900 батчів (при розмірі батчу = 2)
- Ручний датасет + GTZAN
  - Кількість сегментів – 1800 (з ручного датасету) + 3000 (з GTZAN) = 4800 сегментів
  - Кількість батчів на епоху: 2400 батчів
- Тільки GTZAN:
  - Кількість сегментів – 3000
  - Кількість батчів на епоху – 1500 батчів

Для наявного обладнання час обробки одного батчу в середньому дорівнюватиме 0.5 секунди. В такому випадку час, витрачений на одну епоху складатиме:

- Тільки ручний датасет – приблизно 8 хвилин
- Комбіновний датасет – приблизно 22 хвилини
- Тільки GTZAN – приблизно 13 хвилин

Такий параметр як швидкість навчання напряму не має прямого впливу на час, витрачений на одну епоху. Проте, чим нижчий цей параметр, тим більше епох модель може потребувати для досягнення достатнього рівня точності. Тому цей параметр не буде включено в загальне порівняння. Результати порівняння швидкості навчання моделі залежно від значень гіперпараметрів наведено в таблиці 3.1

Таблиця 3.1

Порівняння швидкості навчання в залежності від налаштувань гіперпараметрів

Датасет	Кількість епох	Час на одну епоху	Загальний час
Ручний датасет	40	~8 хвилин	~5.3
	80	~8 хвилин	~10.7
	100	~8 хвилин	~13.3
Комбінований датасет	40	~22 хвилини	~14.7
	80	~22 хвилини	~29.2
	100	~22 хвилини	~36.7
GTZAN Dataset	40	~13 хвилин	~8.7
	80	~13 хвилин	~17.2
	100	~13 хвилин	~21.6

Результати експериментів були представлені тільки для моделі Wave-U-Net оскільки вона більш оптимізована для роботи безпосередньо з аудіо, ніж модель LSTM.

3.4.2 Результати обробки аудіо з використанням моделей LSTM та Wave-U-Net

На рисунках 3.4 та 3.5 наведено скріншоти утиліти для порівняння спектрограм двох аудіо.

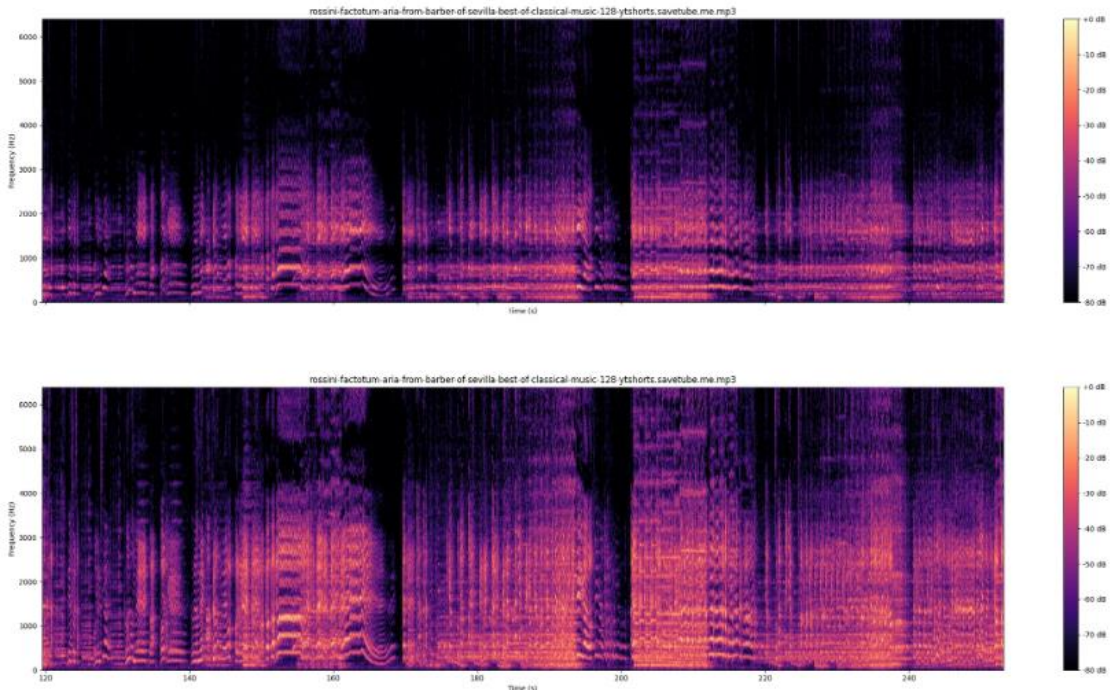


Рис. 3.4 Порівняння спектрограм до (зверху) та після (знизу) обробки моделлю LSTM

На скріншоті (рис. 3.4) видно, що після обробки класичного треку поганої якості моделлю LSTM на спектрограмі стало більше високих частот, проте наявні проблеми з балансом низьких частот.

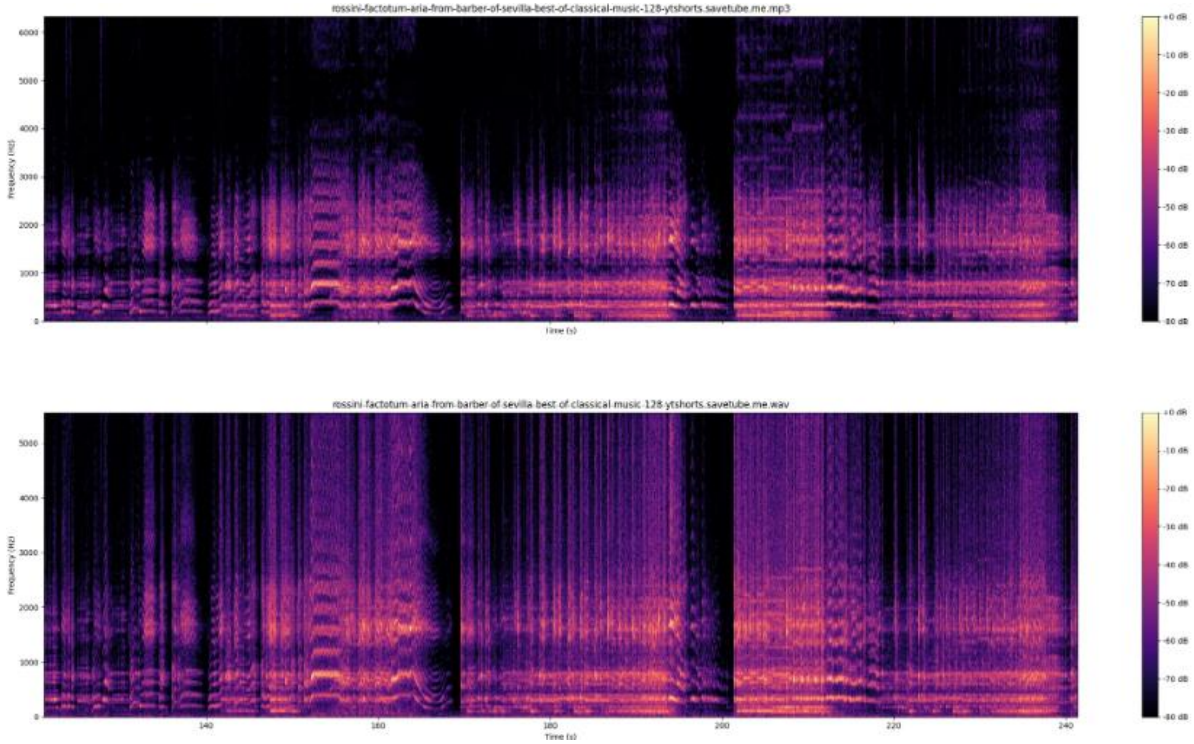


Рис. 3.5 Порівняння спектрограм до (зверху) та після (знизу) обробки моделлю Wave-U-Net

На скріншоті (рис. 3.5) видно, що після обробки класичного треку поганої якості моделлю Wave-U-Net на спектрограмі низький спектр частот перебуває у балансі з спектром високих частот. Відсутні артефакти та резонанси, проте наявна невелика кількість шумів.

3.4.3 Результати обробки аудіо з використанням моделей LSTM та Wave-U-Net

Хоча індекс SNR в даному випадку не може бути об’єктивним показником якості мастерингу, оскільки він лише демонструє співвідношення рівня сигналу ро рівня шуму, проте, він все одно буде включений у порівняння результатів.

Для моделі LSTM:

- До обробки – SNR = 45. Після обробки – SNR = 65

Для моделі Wave-U-Net:

- До обробки – SNR = 45. Після обробки – SNR = 60

Можна помітити що у другому випадку індекс нижчий, що означає гіршу якість звуку. Проте, це пов’язано лише з наявністю шумів в обробленому треку моделлю Wave-U-Net. Для виправлення цього дефекту потрібно розширювати ручний датасет, збільшуючи кількість пар треків для навчання.

4 ВИСНОВКИ

1. У ході роботи було здійснено глибокий аналіз предметної області аудіо мастерингу. Розглянуто основні етапи цього процесу: корекцію динамічного діапазону, налаштування тонального балансу та оптимізацію частотного спектра. Встановлено, що якість мастерингу значною мірою залежить від людського фактора, що зумовлює непослідовні результати й значні затрати часу та ресурсів. Досліджено технології спектрального аналізу аудіо, зокрема використання спектрограм для візуалізації та інтерпретації сигналів, а також метрики оцінки якості, як-от SNR та THD. Розглянуто різні види нейронних мереж для аудіообробки – CNN, RNN, LSTM – з урахуванням їх переваг для аналізу спектрограм та часових залежностей.

2. Проведено аналіз існуючих рішень, таких як Landr, Izotope Ozone та SoundCloud AI Mastering. Виявлено, що попри автоматизовані можливості, ці системи мають обмеження щодо якості обробки аудіо та простоти використання, що підкреслює потребу в розробці нової системи, позбавленої вищезазначених недоліків.

3. Особливу увагу приділено формуванню якісного датасету та адаптації моделей до специфіки аудіо сигналів. Для збільшення обсягу та різноманітності даних застосовано аугментацію (audiomentations). Дані були підготовлені шляхом конвертації у формат WAV, нормалізації та розділення на сегменти для формування навчальних батчів.

4. Створено дві нейромережеві моделі. Перша модель базується на LSTM і ефективно обробляє часові залежності в аудіо сигналах. Друга модель використовує адаптовану архітектуру Wave-U-Net, що зберігає деталі сигналу на різних рівнях завдяки механізму Skip Connection. Моделі були навчені з використанням відповідних функцій втрат (зокрема, середньоквадратичної похибки для спектральних характеристик). Налаштовано параметри навчання, включно зі швидкістю навчання та розміром батчу. Застосовано інструменти

моніторингу навчання (TensorBoard). Розроблено програмне рішення з модулями для обробки аудіо, графічним інтерфейсом користувача (Tkinter) та засобами збереження результатів. Інтеграція моделей здійснена з використанням PyTorch.

5. Система протестована та оцінена за допомогою об’єктивних метрик і суб’єктивного зворотного зв’язку користувачів. Аналіз спектрограм до й після мастерингу підтвердив покращення якості аудіо сигналу. Проведено порівняння розроблених моделей за точністю та швидкістю обробки. Встановлено, що обидві моделі добре справляються з задачами мастерингу аудіо, проте модель Wave-U-Net потребує об’ємнішого датасету для кращих результатів обробки.

ПЕРЕЛІК ПОСИЛАНЬ

1. Thomas Birtchnell: Listening without ears: Artificial intelligence in audio mastering. July 2018, журнал Big Data & Society 5(2):205395171880855. DOI: 10.1177/2053951718808553. Ліцензія CC BY-NC-ND 4.0.

2. Maciej Blaszke, Bozena Kostek: Musical Instrument Identification Using Deep Learning Approach. April 2022, журнал Sensors 22(8):3033. DOI: 10.3390/s22083033. Ліцензія CC BY 4.0.

3. Hajiali Yartire, Amir Hossein Hashemian, Saman Mohammadi: An Introduction to Sound Spectrum Level: Spectrum Analysis. January 2014. Журнал Middle East Journal of Scientific Research 21(12):2243-2246. DOI: 10.5829/idosi.mejsr.2014.21.12.21879

4. Gabriel Thuku Kimani: Fundamentals of sound design. August 2023. 3 книги SOUND DESIGN IN FILM: A KENYAN CINEMA BOOK SERIES Edition 1, chapter 1. Видавництво Utafiti Foundation

5. Image Line official FL Studio manual – [Електронний ресурс] – Режим доступу: https://www.image-line.com/fl-studio-learning/fl-studio-online-manual/ (дата звернення: 18.06.2024)

6. Native Instruments Izotope Ozone official guide – [Електронний ресурс] – Режим доступу: https://support.izotope.com/hc/en-us/articles/7350031568925-Ozone-10-Help-Documentation-Manuals (дата звернення: 18.06.2024)

7. Aisha Lichtner-Bajjaoui: A Mathematical Introduction to Neural Networks. January 2020 – [Електронний ресурс] – Режим доступу: https://diposit.ub.edu/dspace/bitstream/2445/180441/2/tfm_lichtner_bajjaoui_aisha.pdf (дата звернення: 18.06.2024)

8. Alex Smola: Introduction to Machine Learning. 2008. Видавництво: The press syndicate of the university of Cambridge. ISBN 0 521 82583 0 hardback.

9. Chen Manni: Intelligent audio mastering. September 2020. Thesis for: Master of Arts. Advisor: PerMagnus LINDBORG

10. VST documentation – [Електронний ресурс] – Режим доступу: https://store.vstapps.com/pages/vst-documentation (дата звернення 18.06.2024)

11. Mandar Sumant: Signal to Noise Ratio (SNR). 2014.

12. Unknown Eugene: Equillibrium – [Електронний ресурс (музика)] – Режим доступу: https://soundcloud.com/eugene_unknown/equilibrium?si=76680b64c49346c9adb55682f06bf01a&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing (дата звернення: 18.06.2024)

13. Stowell, D., & Plumbley, M. D. (2017). Deep Learning for Audio Signal Processing. IEEE Signal Processing Magazine, 34(6), 98-107.

14. Pons, J., Lidy, T., & Herrera, P. (2016). Experimenting with musically motivated convolutional neural networks. Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR).

15. Salamon, J., & Bello, J. P. (2017). Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification. IEEE Signal Processing Letters, 24(3), 279-283.

16. Eriksson, O. (2020). Audiomentations: A Python Library for Audio Data Augmentation.

17. Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations (ICLR).

18. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

19. TensorFlow Team. TensorBoard: Visualizing Learning.

20. Marques, G., & Nunes, P. (2020). A Survey on Loss Functions for Supervised Deep Learning Classification. arXiv preprint arXiv:2012.07421.

21. Kostić, D., & Yang, C.-H. H. (2021). Neural Network Architectures for Audio Processing. arXiv preprint arXiv:2106.01524.

22. Bergstra, J., Yamins, D., & Cox, D. D. (2013). Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision

Architectures. Proceedings of the 30th International Conference on Machine Learning (ICML).

23. Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv preprint arXiv:1505.04597.

24. Jansson, A., Humphrey, E., Montecchio, N., Bittner, R., Kumar, A., & Weyde, T. (2017). Singing Voice Separation with Deep U-Net Convolutional Networks. Proceedings of the 18th International Society for Music Information Retrieval Conference (ISMIR).

25. Seetharaman, P., & Pardo, B. (2019). Deep Autoencoder Based Speech Denoising. IEEE International Conference on Acoustics, Speech and Signal Processing.

26. Stoller, D., Ewert, S., & Dixon, S. (2018). Wave-U-Net: A Multi-Scale Neural Network for End-to-End Audio Source Separation. arXiv preprint arXiv:1806.03185.

27. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

28. Grayson, J. E. (2000). Python and Tkinter Programming. Manning Publications.

29. Paszke, A., Gross, S., Massa, F., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)*.

30. Lerch, A. (2012). An Introduction to Audio Content Analysis: Applications in Signal Processing and Music Informatics. Wiley-IEEE Press.

31. Pestana, P. D., & Reiss, J. D. (2014). Intelligent Audio Production Strategies Informed by Best Practices. *AES Convention 137*.

32. Hargis, G., Hernandez, A., Rouiller, S., et al. (2004). Developing Quality Technical Information: A Handbook for Writers and Editors (2nd ed.). Prentice Hall.

33. Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735-1780.

34. Graves, A., & Schmidhuber, J. (2005). Framewise Phoneme Classification with Bidirectional LSTM Networks. *Proceedings of the IEEE International Joint Conference on Neural Networks*, 2047-2052.

35. Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional Neural Networks: An Overview and Application in Radiology. *Insights into Imaging*, 9(4), 611-629.

36. Parisi, F., Cornelis, B., Maguire, L. P., & Cunningham, P. (2017). Unsupervised Feature Learning with Wavelets for Audio Classification. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 25(11), 2251-2260.

37. Gold, B., & Morgan, N. (2000). Speech and Audio Signal Processing: Processing and Perception of Speech and Music. Wiley.

38. Karlsson, J. (2019). Real-Time Streaming in Python with AsyncIO and AI. *International Journal of Computer Applications*, 178(17), 1-7.

39. Young, T., Hazarika, D., Poria, S., & Cambria, E. (2018). Recent Trends in Deep Learning Based Natural Language Processing. *IEEE Computational Intelligence Magazine*, 13(3), 55-75.

40. Zölzer, U. (Ed.). (2011). DAFX: Digital Audio Effects. John Wiley & Sons.

41. Kingma, D. P., & Welling, M. (2014). Auto-Encoding Variational Bayes. *International Conference on Learning Representations (ICLR)*.

42. McFee, B., Raffel, C., Liang, D., et al. (2015). librosa: Audio and Music Signal Analysis in Python. *Proceedings of the 14th Python in Science Conference*, 18-25.

43. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature*, 521(7553), 436–444.

44. O'Reilly, J. (2016). Digital Audio Signal Processing: An Introduction. Wiley.

45. Pirkle, W. (2019). Designing Audio Effect Plug-Ins in C++: With Digital Audio Signal Processing Theory. Focal Press.

46. Wang, Y., & Wang, J. (2018). An Overview of End-to-End Automatic Speech Recognition. *IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 599-602.

47. Sigtia, S., Bertin-Mahieux, T., & Dixon, S. (2016). Automatic Music Transcription Using a Fully Convolutional Deep Neural Network. *Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR)*, 175-181.

48. Graves, A., Mohamed, A., & Hinton, G. (2013). Speech Recognition with Deep Recurrent Neural Networks. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 6645-6649.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

АВТОМАТИЗАЦІЯ ПРОЦЕСУ МАСТЕРИНГУ АУДІО З ВИКОРИСТАННЯМ АЛГОРИТМІВ МАШИННОГО НАВЧАННЯ

Виконав: студент групи ПДМ-61 Лабун Євген Миколайович

Керівник: к.т.н., доц., доцент кафедри ІПЗ Золотухіна Оксана Анатоліївна

Київ - 2024

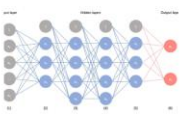
МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: Підвищення якості, швидкості та точності процесу мастерингу аудіо за рахунок автоматизації з використання алгоритмів машинного навчання.

Об'єкт дослідження: Процес мастерингу аудіо

Предмет дослідження: Моделі та методи машинного навчання для автоматизації мастерінгу аудіо.

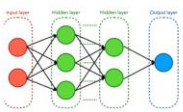
Актуальність роботи



Згорткові нейронні мережі (CNN)

CNN використовують для обробки зображень та сигналів.

Вони ефективні для виявлення та класифікації патернів у даних.



Рекурентні нейронні мережі (RNN)

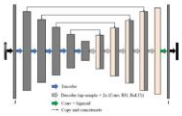
RNN обробляють послідовності даних, такі як текст, мова та аудіо.

RNN запам'ятовують контекст минулих входів, що дозволяє їм прогнозувати майбутні результати та моделювати часові залежності

Довгострокова короткочасна пам'ять (LSTM)

LSTM є різновидом RNN, який здатний обробляти довгі послідовності даних.

Вони запобігають градієнтному зникненню, що є проблемою для стандартних RNN.

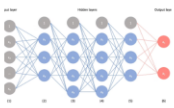


U-Net

U-Net використовується для сегментації зображень, тобто розбиття зображення на окремі області.

U-Net має архітектуру, схожу на пісочний годинник, що дозволяє ефективно обробляти зображення.

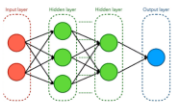
Основні недоліки існуючих підходів



Згорткові нейронні мережі (CNN)

Обмежені у роботі з послідовними даними

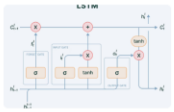
Не враховують часові залежності



Рекурентні нейронні мережі (RNN)

Схильні до проблеми зникання та вибуху градієнтів

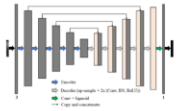
Важко навчати на довгих послідовностях



Довгострокова короткочасна пам'ять (LSTM)

Мають більше параметрів порівняно зі стандартними RNN

Вища обчислювальна складність

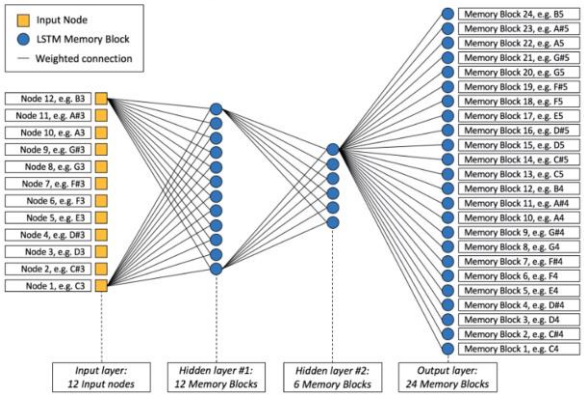


U-Net

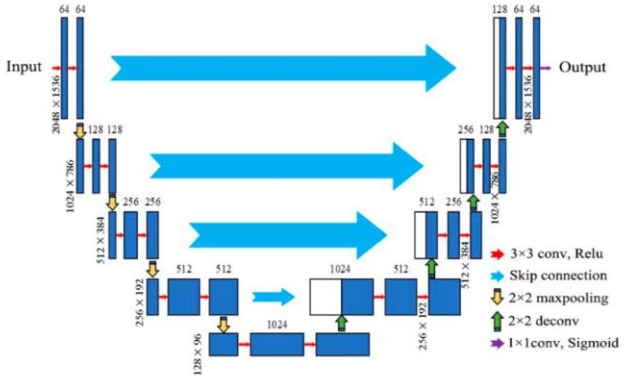
Вимагає ще більше обчислювальних ресурсів

Складніша архітектура для налаштування

Принцип роботи LSTM

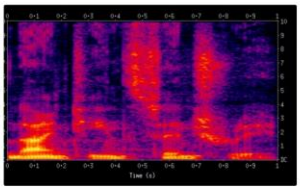


Принцип роботи моделі U-Net

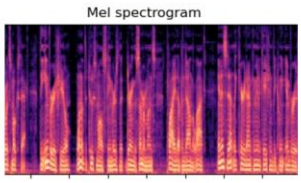


Способи представлення звуку

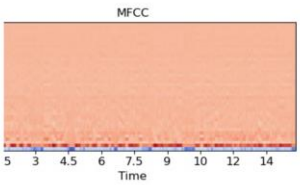
Вейвформа



Спектрограма

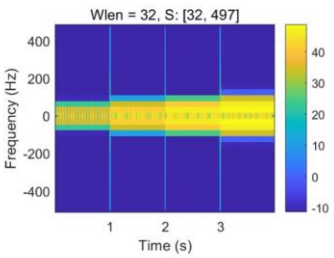


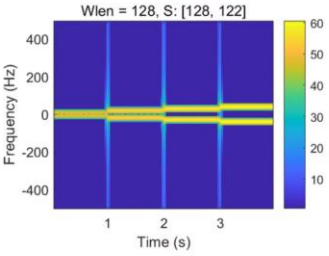
Мел-спектрограма



MFCCs (Mel-Frequency Cepstral Coefficients)

Модифікований метод





Вирішення проблем при навчанні на основі неконсистентного датасету

Аугментація даних

Створення додаткових даних шляхом застосування різних трансформацій.

Нормалізація

Приведення даних до єдиного масштабу для покращення збіжності.

Балансування датасету

Вирівнювання кількості прикладів кожного класу.

Обробка пропущених значень

Використання методів імпутації або видалення неповних записів.

Вплив налаштування гіперпараметрів на фінальний результат

Швидкість навчання

Визначає крок оновлення вагів моделі; занадто велика може призвести до нестабільності.

Розмір пакету (Batch Size)

Впливає на точність оцінки градієнта та швидкість навчання.

Архітектура моделі

Кількість шарів та нейронів впливає на здатність моделі до узагальнення.

Функції активації

Визначають нелінійність моделі та впливають на навчання.

Регуляризація та Dropout

Запобігають перенавчанню, покращуючи узагальнювальну здатність.

Залежність споживання ресурсів від налаштувань гіперпараметрів

1. **Кількість епох**

   Більша кількість епох може як покращити результат навчання (при цьому буде витрачено більше часу), так і погіршити його (у випадку коли модель почне перенавчатися).

2. **Розмір пакету**

   Збільшення Batch Size підвищує вимоги до пам'яті, але може прискорити навчання.

3. **Складність обчислень**

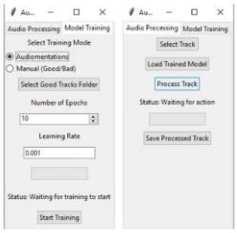
   Використання складних операцій збільшує час навчання.

4. **Обчислення на GPU**

   Можливість розподілу обчислень на кілька процесорів або GPU.

Практичний результат

Застосунок з використанням LSTM моделі



Складається з 4 файлів

- main.py - інтерфейс і керування навчанням
- model.py - опис і архітектура моделі
- train.py - логіка тренування моделі
- utils.py - допоміжні функції (конвертування форматів, сегментація аудіо тощо)

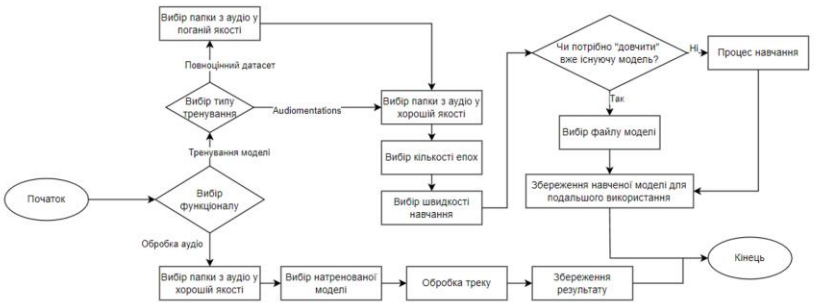
Застосунок з використанням Wave-U-Net моделі

Не має графічного інтерфейсу - керування відбувається шляхом вводу команд

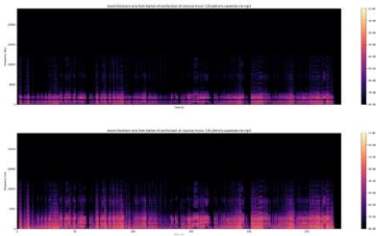
Складається з 8 файлів

- main.py - обробка команд і керування навчанням
- model.py - опис і архітектура моделі
- utils.py - допоміжні функції
- augmentations.py - отримання налаштувань для Audiomentations
- config.py - конфігураційний файл
- dataset.py/dataset_gtzan.py - процес підготовки датасету
- enhance_audio.py - завантаження навченої моделі та аудіо для обробки

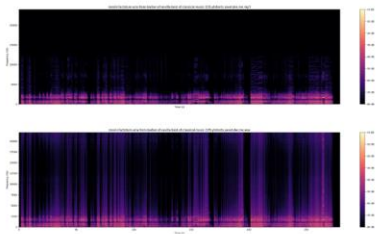
Процес тренування моделі та обробки треків шляхом взаємодії з графічним інтерфейсом застосунку з використанням LSTM



Практичний результат

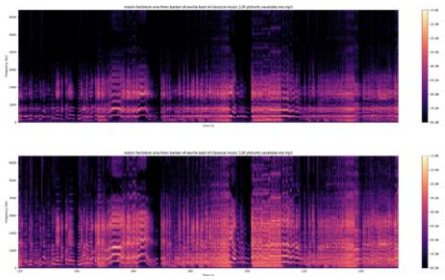


Обробка з використанням моделі LSTM

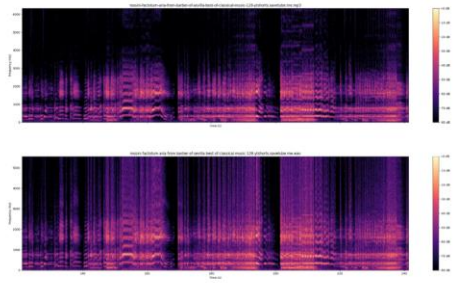


Обробка з використанням моделі U-Net

Практичний результат



Обробка з використанням моделі LSTM



Обробка з використанням моделі Wave-U-Net

Висновки

1. Розроблено та впроваджено методику збору та підготовки даних, що включає ручну підготовку датасету з треками високої та низької якості, а також використання аугментацій для його розширення.
2. Створено дві нейронні моделі:
    1. **LSTM архітектура** для роботи з часовими послідовностями, що забезпечує збереження контексту сигналу.
    2. **Wave-U-Net архітектура**, адаптована для аудіосигналів, яка дозволяє відновлювати спектральну інформацію з високою деталізацією.
3. Навчання моделей здійснено з використанням функцій втрат, налаштуванням гіперпараметрів та моніторингом за допомогою TensorBoard.
4. Розроблено утиліту для порівняння спектрограм до і після обробки, яка демонструє відновлення сигналів моделями LSTM і U-Net.
5. Проведено аналіз споживання ресурсів моделями під час тренування, а також їх продуктивності на основі якості відновлених аудіосигналів.

Публікації та апробація роботи

Тези доповідей:

1. Лабун Є. М. Використання нейронних мереж для автоматизації мастерингу аудіо. // IV Міжнародна науково-практична інтернет-конференція "Mechanisms of scientific and technical potential development" - Міжнародний електроний науково-практичний журнал "WayScience", с. 137
2. Лабун Є. М. Розробка VST (Virtual Studio Technology) плагіна на базі нейронних мереж для інтеграції у музичні редактори. // IV Міжнародна науково-практична інтернет-конференція "Mechanisms of scientific and technical potential development" - Міжнародний електроний науково-практичний журнал "WayScience", с. 139
3. Лабун Є.М. Створення датасету для навчання моделі нейронної мережі для мастерингу аудіо. //XXIV Міжнародна науково-практична конференція «Інформаційні технології та безпека» (подано до публікації)

ДОДАТОК Б. КОД ПРОГРАМНОГО ЗАСТОСУНКУ НА ОСНОВІ LSTM

main.py:

```
import tempfile
import tkinter as tk
from tkinter import ttk, filedialog
import os

import torch
import torchaudio

from model import CRNNAudioModel
from utils import prepare_dataset, convert_mp3_to_wav, slice_audio_to_segments, convert_mp3_to_wav_simple, \
    process_audio_with_model
from train import train_model

temp_dir = "D:/Temp"
os.makedirs(temp_dir, exist_ok=True)
tempfile.tempdir = temp_dir
os.environ["TMP"] = temp_dir
os.environ["TEMP"] = temp_dir

print(f"Тимчасова директорія встановлена на: {tempfile.gettempdir()}")

torch_cache_dir = "E:/TorchCache"
os.makedirs(torch_cache_dir, exist_ok=True)
os.environ["TORCH_HOME"] = torch_cache_dir
os.environ["TORCH_EXTENSIONS_DIR"] = torch_cache_dir
os.environ["CUDA_CACHE_PATH"] = torch_cache_dir

import psutil

def load_model_weights_with_fallback(model, model_path):
    """
    Завантажує ваги в модель, ігноруючи несумісні параметри.
    :param model: Екземпляр моделі (наприклад, CRNNAudioModel).
    :param model_path: Шлях до файлу з збереженою моделлю.
    """
    try:
        model.load_state_dict(torch.load(model_path))
        print("Модель успішно завантажена з strict=True.")
    except RuntimeError as e:
        print(f"Помилка при завантаженні моделі з strict=True: {e}")
        print("Спроба завантажити ваги з резервною логікою...")

        pretrained_dict = torch.load(model_path)
        model_dict = model.state_dict()

        filtered_dict = {k: v for k, v in pretrained_dict.items()
                         if k in model_dict and v.size() == model_dict[k].size()}

        model_dict.update(filtered_dict)
        model.load_state_dict(model_dict)
        print(f"Модель завантажена з частковими вагами. Завантажено {len(filtered_dict)} параметрів.")

    return model

def monitor_disk_usage(drive="C:/"):
    usage = psutil.disk_usage(drive)
    print(f"Використання диска на {drive}:")
    print(f"  Всього: {usage.total / (1024 ** 3):.2f} GB")
    print(f"  Використано: {usage.used / (1024 ** 3):.2f} GB")
    print(f"  Вільно: {usage.free / (1024 ** 3):.2f} GB")

monitor_disk_usage("C:/")
monitor_disk_usage("E:/")

class AudioMasteringApp:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Audio Mastering")

        self.model = None
        self.good_tracks_path = None
        self.bad_tracks_path = None
        self.selected_audio = None
        self.processed_audio = None
        self.training_mode = tk.StringVar(value="audiomentations")
        self.epochs = tk.IntVar(value=10)
        self.learning_rate = tk.DoubleVar(value=0.001)

        self.tab_control = ttk.Notebook(self.root)
        self._create_audio_processing_tab()
        self._create_training_tab()
        self.tab_control.pack(expand=1, fill="both")

    def _create_audio_processing_tab(self):
        tab1 = ttk.Frame(self.tab_control)
```

```
        self.tab_control.add(tab1, text="Обробка Аудіо")

        ttk.Button(tab1, text="Вибрати Трек", command=self._select_audio).pack(pady=5)
        ttk.Button(tab1, text="Завантажити Навчену Модель", command=self._load_trained_model).pack(pady=5)
        ttk.Button(tab1, text="Обробити Трек", command=self._process_audio_simple).pack(pady=5)

        self.label_status = ttk.Label(tab1, text="Статус: Очікування дії")
        self.label_status.pack(pady=5)

        self.progress_bar = ttk.Progressbar(tab1, orient="horizontal", mode="determinate", maximum=100)
        self.progress_bar.pack(pady=5)

        ttk.Button(tab1, text="Зберегти Оброблений Трек", command=self._save_audio).pack(pady=5)

    def _create_training_tab(self):
        tab2 = ttk.Frame(self.tab_control)
        self.tab_control.add(tab2, text="Навчання Моделі")

        ttk.Label(tab2, text="Виберіть Режим Навчання").pack(pady=5)
        ttk.Radiobutton(tab2, text="Audiomentations", variable=self.training_mode, value="audiomentations",
                        command=self._toggle_bad_tracks).pack(anchor=tk.W)
        ttk.Radiobutton(tab2, text="Ручний (Good/Bad)", variable=self.training_mode, value="manual",
                        command=self._toggle_bad_tracks).pack(anchor=tk.W)

        ttk.Button(tab2, text="Вибрати Папку Good Треків", command=self._select_good_tracks).pack(pady=5)

        self.bad_tracks_button = ttk.Button(tab2, text="Вибрати Папку Bad Треків", command=self._select_bad_tracks)
        self.bad_tracks_button.pack(pady=5)
        self.bad_tracks_button.pack_forget()

        ttk.Label(tab2, text="Кількість Епох").pack(pady=5)
        ttk.Spinbox(tab2, from_=1, to=100, textvariable=self.epochs).pack(pady=5)

        ttk.Label(tab2, text="Швидкість Навчання").pack(pady=5)
        ttk.Entry(tab2, textvariable=self.learning_rate).pack(pady=5)

        self.training_progress_bar = ttk.Progressbar(tab2, orient="horizontal", mode="determinate", maximum=100)
        self.training_progress_bar.pack(pady=10)

        self.training_status = ttk.Label(tab2, text="Статус: Очікування початку навчання")
        self.training_status.pack(pady=10)

        ttk.Button(tab2, text="Почати Навчання", command=self._start_training).pack(pady=5)

    def _toggle_bad_tracks(self):
        if self.training_mode.get() == "manual":
            self.bad_tracks_button.pack(pady=5)
        else:
            self.bad_tracks_button.pack_forget()

    def _select_audio(self):
        self.selected_audio = filedialog.askopenfilename(filetypes=[("Аудіо Файли", "*.mp3 *.wav")])
        if self.selected_audio:
            print(f"Вибрано Аудіо Трек: {self.selected_audio}")
            self.label_status.config(text="Аудіо трек вибрано.")

    def _load_trained_model(self):
        """
        Завантажує навчальну модель з переносом ваг при невідповідності.
        """
        model_path = filedialog.askopenfilename(filetypes=[("PyTorch Модель", "*.pth")])
        if model_path:
            self.model = CRNNAudioModel()
            self.model = load_model_weights_with_fallback(self.model, model_path)
            print(f"Модель завантажена з {model_path}")
            self.label_status.config(text="Модель успішно завантажена!")

    def _process_audio_simple(self):
        """
        Обробляє вибраний трек з використанням навченої моделі.
        """
        if not hasattr(self, "selected_audio") or not self.selected_audio:
            self.label_status.config(text="Файл аудіо не вибрано!")
            print("Файл аудіо не вибрано!")
            return

        if not self.model:
```

```
                self.label_status.config(text="Модель не завантажена!")
                print("Модель не завантажена!")
                return

            wav_path = convert_mp3_to_wav_simple(self.selected_audio, "temp_files/audio_to_process.wav")
            if not wav_path:
                self.label_status.config(text="Не вдалося конвертувати файл аудіо!")
                print("Не вдалося конвертувати файл аудіо!")
                return

            output_path = process_audio_with_model(wav_path, self.model, "cpu")

self.label_status.config(text=f"Оброблене аудіо збережено за адресою: {output_path}")
            print(f"Оброблене аудіо збережено за адресою: {output_path}")

        def _process_audio(self):
            if not self.selected_audio:
                print("Трек не вибрано. Будь ласка, виберіть аудіо трек.")

self.label_status.config(text="Спочатку виберіть трек!")
                return

            if not self.model:
                print("Навчена модель не завантажена. Неможливо обробити трек.")

self.label_status.config(text="Спочатку завантажте навчальну модель!")
                return

            from utils import slice_audio_to_segments

            wav_path = os.path.join("temp_files", "processed.wav")

convert_mp3_to_wav(self.selected_audio, wav_path)

            segments = slice_audio_to_segments(wav_path)
            print(f"Аудіо розділено на {len(segments)} сегментів.")

            processed_segments = []
            for i, segment in enumerate(segments):
                print(f"Обробка сегмента {i + 1}/{len(segments)}...")
                segment_input = segment.unsqueeze(0)
                with torch.no_grad():
                    processed_segment = self.model(segment_input).squeeze(0)
                processed_segments.append(processed_segment)

            processed_audio = torch.cat(processed_segments, dim=1)
            output_path = os.path.join("temp_files", "processed_complete.wav")
            torchaudio.save(output_path, processed_audio, 44100)

            print(f"Трек оброблено та збережено за адресою {output_path}")
            self.processed_audio = output_path
            self.label_status.config(text="Обробка завершена!")

        def _save_audio(self):
            if not self.processed_audio:
                print("Немає обробленого аудіо для збереження.")
                self.label_status.config(text="Немає доступного треку для збереження.")
                return

            save_path = filedialog.asksaveasfilename(defaultextension=".wav", filetypes=[("WAV Файли", "*.wav")])
            if save_path:
                import shutil
                shutil.copy(self.processed_audio, save_path)
                print(f"Оброблене аудіо збережено за адресою {save_path}")
                self.label_status.config(text="Трек успішно збережено!")

        def _select_good_tracks(self):
            self.good_tracks_path = filedialog.askdirectory()
            if self.good_tracks_path:
                print(f"Вибрано папку Good Треків: {self.good_tracks_path}")

        def _select_bad_tracks(self):
            self.bad_tracks_path = filedialog.askdirectory()
            if self.bad_tracks_path:
                print(f"Вибрано папку Bad Треків: {self.bad_tracks_path}")

        def _start_training(self):
            """
            Запускає процес навчання моделі.
            """
            GOOD_WAV_FOLDER = os.path.join("temp_files", "wav", "good")
            BAD_WAV_FOLDER = os.path.join("temp_files", "wav", "bad")
            GOOD_SEGMENTS_FOLDER = os.path.join("temp_files", "segments", "good")
            BAD_SEGMENTS_FOLDER = os.path.join("temp_files", "segments", "bad")
```

```
            os.makedirs(GOOD_WAV_FOLDER,
exist_ok=True)
            os.makedirs(BAD_WAV_FOLDER,
exist_ok=True)

os.makedirs(GOOD_SEGMENTS_FOLDER,
exist_ok=True)

os.makedirs(BAD_SEGMENTS_FOLDER,
exist_ok=True)

            print("Підготовка good треків...")
            good_files                              =
prepare_dataset(self.good_tracks_path,
GOOD_WAV_FOLDER)

            bad_files = []
            if self.training_mode.get() == "manual":
                print("Підготовка bad треків...")
                bad_files                           =
prepare_dataset(self.bad_tracks_path,
BAD_WAV_FOLDER)

                if len(good_files) != len(bad_files):
                    print("Синхронізація good та bad
треків...")
                    common_files = set(good_files) &
set(bad_files)
                    good_files = [file for file in
good_files if file in common_files]
                    bad_files = [file for file in bad_files
if file in common_files]

            print(f"Кінцева кількість good файлів:
{len(good_files)}")
            print(f"Кінцева кількість bad файлів:
{len(bad_files)}")

            print("Розрізання good треків на
сегменти...")

slice_audio_to_segments(GOOD_WAV_FOLDER,
GOOD_SEGMENTS_FOLDER)

            if self.training_mode.get() == "manual":
                print("Розрізання bad треків на
сегменти...")

slice_audio_to_segments(BAD_WAV_FOLDER,
BAD_SEGMENTS_FOLDER)

            if not self.model:
                self.model = CRNNAudioModel()

            log_dir                                 =
filedialog.askdirectory(title="Вибрати директорію для
логів TensorBoard")
            if not log_dir:
                print("Директорія для логів не
вибрана. Навчання скасовано.")
                return

            print("Початок навчання...")
self.training_status.config(text="Навчання в
процесі...")

            train_model(
                GOOD_SEGMENTS_FOLDER,
                BAD_SEGMENTS_FOLDER if
self.training_mode.get() == "manual" else None,
                self.model,
                self.epochs.get(),
                self.learning_rate.get(),

use_augmentations=(self.training_mode.get()
"audiomentations"),
                log_dir=log_dir
            )

            print("Навчання завершено!")

self.training_status.config(text="Навчання
завершено!")

            save_path                               =
filedialog.asksaveasfilename(
                defaultextension=".pth",
                filetypes=[("PyTorch Модель",
"*.pth")],
                title="Зберегти Навчену Модель"
            )
            if save_path:
                import torch
                torch.save(self.model.state_dict(),
save_path)
                print(f"Модель збережено за адресою
{save_path}")

self.training_status.config(text="Навчання
завершено! Модель збережено.")
            else:
                print("Модель не була збережена.")

self.training_status.config(text="Навчання
завершено! Модель не збережено.")

        def run(self):
            self.root.mainloop()


    if __name__ == "__main__":
        app = AudioMasteringApp()
        app.run()
```

model.py:

```
    import torch
    import torch.nn as nn


    class CRNNAudioModel(nn.Module):
        """
        CRNN для обробки аудіо.
```

```
        """
        def __init__(self):
            super(CRNNAudioModel, self).__init__()
            self.conv = nn.Sequential(
                nn.Conv1d(2, 16, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.Conv1d(16, 32, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.Conv1d(32, 128, kernel_size=3, stride=1, padding=1),
                nn.ReLU()
            )
            self.lstm = nn.LSTM(128, 64, batch_first=True, bidirectional=True)
            self.fc = nn.Linear(128, 2)

        def forward(self, x):
            """
            Прямий прохід через модель CRNN.
            """
            x = self.conv(x)
            x = x.permute(0, 2, 1)
            x, _ = self.lstm(x)
            x = self.fc(x)
            return x.permute(0, 2, 1)
```

train.py:

```
        import gc

        import torch
        import torchaudio
        import os
        from torch.utils.tensorboard import SummaryWriter
        from audiomentations import Compose, AddGaussianNoise, PitchShift, TimeStretch
        from tqdm import tqdm
        import tensorboard

        def train_model(good_tracks_path, bad_tracks_path, model, epochs, learning_rate, use_augmentations=False, log_dir="logs", batch_size=4):
            """
            Функція для навчання моделі CRNNAudioModel.
            """
            from torch.utils.data import DataLoader, Dataset
            from torch.nn.utils.rnn import pad_sequence

            device = "cpu"
            print(f"Використання пристрою: {device}")
            model = model.to(device)

            class AudioDataset(Dataset):
                def __init__(self, good_folder, bad_folder=None, use_augmentations=False):
                    self.good_files = sorted(os.listdir(good_folder))
                    self.bad_files = sorted(os.listdir(bad_folder)) if bad_folder else []
                    self.good_folder = good_folder
                    self.bad_folder = bad_folder
                    self.use_augmentations = use_augmentations
                    self.augment = Compose([
                        AddGaussianNoise(min_amplitude=0.001, max_amplitude=0.015, p=0.5),
                        PitchShift(min_semitones=-2, max_semitones=2, p=0.5),
                        TimeStretch(min_rate=0.8, max_rate=1.2, p=0.5)
                    ]) if use_augmentations else None

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

                def __getitem__(self, idx):
                    good_path = os.path.join(self.good_folder, self.good_files[idx])
                    good_waveform, _ = torchaudio.load(good_path)

                    if self.bad_files:
                        bad_path = os.path.join(self.bad_folder, self.bad_files[idx])
                        bad_waveform, _ = torchaudio.load(bad_path)
                    else:
                        bad_waveform = torch.tensor(self.augment(samples=good_waveform.numpy(), sample_rate=44100), dtype=torch.float32)

                    min_length = min(good_waveform.size(1), bad_waveform.size(1))
                    good_waveform = good_waveform[:, :min_length]
                    bad_waveform = bad_waveform[:, :min_length]

                    return bad_waveform, good_waveform

            dataset = AudioDataset(good_tracks_path, bad_tracks_path, use_augmentations)
            data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True, collate_fn=lambda x: tuple(zip(*x)))

            optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
            criterion = torch.nn.MSELoss()
            model.train()

            writer = SummaryWriter(log_dir=log_dir)

            for epoch in range(epochs):
```

```
            print(f"Епоха {epoch + 1}/{epochs}")
            epoch_loss = 0

            for batch_idx, (bad_batch, good_batch) in enumerate(tqdm(data_loader, desc=f"Прогрес Епохи {epoch + 1}")):
                bad_batch = pad_sequence([fix_length(x, 220500) for x in bad_batch], batch_first=True).to(device)
                good_batch = pad_sequence([fix_length(x, 220500) for x in good_batch], batch_first=True).to(device)

                bad_batch = pad_sequence(bad_batch, batch_first=True).to(device)
                good_batch = pad_sequence(good_batch, batch_first=True).to(device)

                optimizer.zero_grad()
                output = model(bad_batch)
                loss = criterion(output, good_batch)
                loss.backward()
                optimizer.step()

                epoch_loss += loss.item()

            writer.add_scalar("Loss/Epoka", epoch_loss, epoch)
            print(f"Втрати після епохи {epoch + 1}: {epoch_loss:.4f}")

        writer.close()
        print("Навчання завершено. Перевірте TensorBoard для логів.")
```

utils.py:

```
        import os
        import torchaudio
        import torch

        def convert_mp3_to_wav(input_path, output_path, target_sample_rate=44100):
            """
            Конвертує MP3 у WAV з заданою частотою дискретизації та стерео.
            """
            try:
                command = f"ffmpeg -i \"{input_path}\" -ar {target_sample_rate} -ac 2 \"{output_path}\" -y -loglevel error"
                result = os.system(command)
                if result != 0:
                    print(f"Помилка конвертації {input_path}. Файл може бути пошкодженим.")
                    return False
            except Exception as e:
                print(f"Не вдалося обробити {input_path}: {e}")
                return False
            return True

        def load_model_weights_with_fallback(model, model_path):
            """
            Завантажує ваги в модель, ігноруючи несумісні параметри.
            :param model: Екземпляр моделі (наприклад, CRNNAudioModel).
            :param model_path: Шлях до файлу з збереженою моделлю.
            """
            try:
                model.load_state_dict(torch.load(model_path))
                print("Модель успішно завантажена з strict=True.")
            except RuntimeError as e:
                print(f"Помилка при завантаженні моделі з strict=True: {e}")
                print("Спроба завантажити ваги з резервною логікою...")

                pretrained_dict = torch.load(model_path)
                model_dict = model.state_dict()

                filtered_dict = {k: v for k, v in pretrained_dict.items()
                                 if k in model_dict and v.size() == model_dict[k].size()}

                model_dict.update(filtered_dict)
                model.load_state_dict(model_dict)
                print(f"Модель завантажена з частковими вагами. Завантажено {len(filtered_dict)} параметрів.")

            return model

        def process_audio_with_model(input_wav, model, device, segment_length=5.0):
            """
            Обробляє трек з навченою моделлю сегментами.
            :param input_wav: Шлях до вхідного WAV-файлу.
            :param model: Навчена модель.
            :param device: Пристрій (CPU/GPU).
            :param segment_length: Довжина сегмента в секундах.
            :return: Шлях до обробленого WAV-файлу.
            """
            waveform, sample_rate = torchaudio.load(input_wav)
            waveform = waveform.to(device)

            segment_samples = int(segment_length * sample_rate)
            total_samples = waveform.size(1)

            processed_segments = []
            print(f"Обробка аудіо у сегментах по {segment_length} секунд...")
```

```
        for start in range(0, total_samples, segment_samples):
            end = min(start + segment_samples, total_samples)
            segment = waveform[:, start:end]

            if segment.size(1) < segment_samples:
                pad_size = segment_samples - segment.size(1)
                segment = torch.nn.functional.pad(segment, (0, pad_size))

            with torch.no_grad():
                processed_segment = model(segment.unsqueeze(0)).squeeze(0)

            processed_segments.append(processed_segment)

        processed_waveform = torch.cat(processed_segments, dim=1)

        output_path = "temp_files/processed_audio.wav"
        torchaudio.save(output_path, processed_waveform.cpu(), sample_rate)
        print(f"Оброблене аудіо збережено за адресою: {output_path}")
        return output_path

    def convert_mp3_to_wav_simple(input_path, output_path, target_sample_rate=44100):
        """
        Конвертує MP3 у WAV з заданою частотою дискретизації та стерео.
        :param input_path: Шлях до MP3-файлу.
        :param output_path: Шлях для збереження WAV-файлу.
        :param target_sample_rate: Бажана частота дискретизації.
        :return: Шлях до створеного WAV-файлу або None, якщо сталася помилка.
        """
        try:
            command = (
                f"ffmpeg -i \"{input_path}\" -ar {target_sample_rate} -ac 2 \"{output_path}\" -y -loglevel error"
            )
            result = os.system(command)
            if result != 0:
                print(f"Помилка конвертації {input_path} у WAV. Файл може бути пошкодженим.")
                return None
            print(f"Конвертовано {input_path} у формат WAV за адресою {output_path}.")
            return output_path
        except Exception as e:
            print(f"Виникла помилка під час конвертації MP3 у WAV: {e}")
            return None

    def slice_audio_to_segments(input_folder, output_folder, segment_length=10.0):
        """
        Розбиває WAV файли з папки на сегменти та зберігає їх.
        :param input_folder: Папка з WAV файлами.
        :param output_folder: Папка для збереження сегментів.
        :param segment_length: Довжина сегмента в секундах.
        """
        os.makedirs(output_folder, exist_ok=True)
        total_segments = 0

        for file in os.listdir(input_folder):
            if file.endswith(".wav"):
                input_path = os.path.join(input_folder, file)
                waveform, sample_rate = torchaudio.load(input_path)

                segment_samples = int(segment_length * sample_rate)
                num_segments = waveform.size(1) // segment_samples

                for i in range(num_segments):
                    start = i * segment_samples
                    end = start + segment_samples
                    segment = waveform[:, start:end]

                    segment_path = os.path.join(output_folder, f"{os.path.splitext(file)[0]}_seg{i}.wav")
                    torchaudio.save(segment_path, segment, sample_rate)

                total_segments += num_segments

        print(f"Створено {total_segments} сегментів з файлів у {input_folder}.")

    def fix_length(waveform, fixed_length):
        """
        Фіксує довжину waveform шляхом додавання нулів або обрізання.
        """
        if waveform.size(1) < fixed_length:
            pad_size = fixed_length - waveform.size(1)
            waveform = torch.nn.functional.pad(waveform, (0, pad_size))
        else:
            waveform = waveform[:, :fixed_length]
        return waveform

    def prepare_dataset(source_folder, output_folder, segment_length=10.0):
        """
        Підготовлює датасет: конвертує MP3 у WAV.
```

```
        :param source_folder: Папка з вихідними аудіо файлами (MP3).
        :param output_folder: Папка для збереження WAV.
        :return: Список успішно оброблених файлів.
        """
        os.makedirs(output_folder, exist_ok=True)
        processed_files = []

        for file in os.listdir(source_folder):
            if file.endswith(".mp3"):
                input_path = os.path.join(source_folder, file)
                wav_path = os.path.join(output_folder, f"{os.path.splitext(file)[0]}.wav")
                if not convert_mp3_to_wav(input_path, wav_path):
                    print(f"Пропуск {file} через помилку конвертації.")
                    continue
                processed_files.append(file)

        print(f"Оброблено {len(processed_files)} файлів.")
        return processed_files
```

ДОДАТОК В. КОД ПРОГРАМНОГО ЗАСТОСУНКУ НА ОСНОВІ WAVE-U-NET

main.py:

```
import argparse
from train import train_model
from enhance_audio import AudioProcessorApp
import config

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Wave-U-Net Покращення Аудіо")
    parser.add_argument('--mode', type=str, choices=['train', 'enhance'], required=True, help='Режим: train або enhance')
    parser.add_argument('--input', type=str, help='Шлях до вхідного аудіо файлу для покращення')
    parser.add_argument('--output', type=str, help='Шлях для збереження покращеного аудіо файлу')
    args = parser.parse_args()

    if args.mode == 'train':
        train_model(config)
    elif args.mode == 'enhance':
        if not args.input or not args.output:
            print("Будь ласка, вкажіть шляхи до вхідного та вихідного файлів для покращення.")
        else:
            app = AudioProcessorApp(config.model_save_path)

app.process_and_save_track(args.input, args.output)
```

model.py:

```
import torch
import torch.nn as nn

class WaveUNet(nn.Module):
    def __init__(self, in_channels=2, out_channels=2, num_channels=24, num_layers=12, kernel_size=15, dropout=0.0):
        super(WaveUNet, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.num_channels = num_channels
        self.num_layers = num_layers
        self.kernel_size = kernel_size
        self.dropout = dropout

        self.encoder = nn.ModuleList()
        self.decoder = nn.ModuleList()
        self.bottleneck = nn.Sequential(
            nn.Conv1d(num_channels * (2 ** (num_layers - 1)), num_channels * (2 ** num_layers), kernel_size, padding=kernel_size // 2),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout)
        )

        """
        Енкодер
        """
        for i in range(num_layers):
            in_ch = in_channels if i == 0 else num_channels * (2 ** (i - 1))
            out_ch = num_channels * (2 ** i)
            self.encoder.append(
                nn.Sequential(
                    nn.Conv1d(in_ch, out_ch, kernel_size, padding=kernel_size // 2),
                    nn.ReLU(inplace=True),
                    nn.Dropout(p=dropout)
                )
            )

        """
        Декодер
        """
        for i in reversed(range(num_layers)):
            in_ch = num_channels * (2 ** (i + 1))
            out_ch = num_channels * (2 ** i)
            self.decoder.append(
                nn.Sequential(
                    nn.ConvTranspose1d(in_ch, out_ch, kernel_size=kernel_size, stride=1, padding=kernel_size // 2),
                    nn.ReLU(inplace=True),
                    nn.Dropout(p=dropout)
                )
            )
```

```
        """
        Вихідний шар
        """
        self.final = nn.Conv1d(in_channels + num_channels, out_channels, kernel_size=1)

    def forward(self, x):
        """
        Прямий прохід через мережу.
        """
        # Зберігаємо вхід для пропущеного зв'язку
        x_in = x
        enc_outputs = []

        # Енкодер
        for enc in self.encoder:
            x = enc(x)
            enc_outputs.append(x)
            x = x[:, :, ::2]  # Децимація (зниження частоти дискретизації в 2 рази)

        # Ботлнек
        x = self.bottleneck(x)

        # Декодер
        for i, dec in enumerate(self.decoder):
            x = torch.nn.functional.interpolate(x, scale_factor=2, mode='linear', align_corners=False)
            x = dec(x)
            x = x + enc_outputs[-(i + 1)]  # Пропущений зв'язок

        # Вихідний шар
        x = torch.cat([x, x_in], dim=1)
        x = self.final(x)

        return x
```

utils.py:

```
import torch
import torchaudio
import os

def convert_to_wav(input_path, output_path):
    waveform, sample_rate = torchaudio.load(input_path)
    torchaudio.save(output_path, waveform, sample_rate)

def slice_audio(file_path, segment_length, output_dir, track_number):
    waveform, sample_rate = torchaudio.load(file_path)
    total_samples = waveform.shape[1]
    segment_samples = int(segment_length * sample_rate)
    num_segments = total_samples // segment_samples

    os.makedirs(output_dir, exist_ok=True)

    for i in range(num_segments):
        start = i * segment_samples
        end = start + segment_samples
        segment = waveform[:, start:end]
        segment_path = os.path.join(output_dir, f"track_{track_number}_segment_{i+1}.wav")
torchaudio.save(segment_path, segment, sample_rate)

    return num_segments
```

augmentations.py:

```
import torch

def augment_audio(waveform):
    """
    Аугментація аудіо: додавання шуму та випадкова зміна гучності.
    """
    # Додавання випадкового шуму
    noise = torch.randn_like(waveform) * 0.005
    waveform = waveform + noise

    # Випадкова зміна гучності
    gain = torch.FloatTensor(1).uniform_(0.9, 1.1).to(waveform.device)
    waveform = waveform * gain

    return waveform
```

config.py:

```
# Параметри навчання
num_epochs = 20
learning_rate = 0.0001
batch_size = 8
use_gpu = True

# Параметри моделі
in_channels = 2
out_channels = 2
num_channels = 24
num_layers = 8
kernel_size = 15
dropout = 0.05

# Шляхи до даних
bad_tracks_dir = 'data/bad_tracks'
good_tracks_dir = 'data/good_tracks'
gtzan_dir = 'data/gtzan_dataset'

# Вибір джерел даних
use_pairs = True  # Використовувати пари треків
use_gtzan = False  # Використовувати датасет GTZAN

# Шлях для збереження моделі
```

```
model_save_path = 'models/waveunet_model.pth'

# Довжина сегмента
segment_length = 131072  # Приблизно 3 секунди при 44.1 kHz

# Частота дискретизації
target_sample_rate = 44100

# Шлях для логів TensorBoard
tensorboard_log_dir = 'runs/waveunet_experiment'
```

dataset.py:

```
import torch
import torchaudio
from torch.utils.data import Dataset
from torchaudio.transforms import Resample
import os

class AudioDataset(Dataset):
    def __init__(self, bad_dir, good_dir, target_sample_rate=44100, segment_length=131072):
        self.bad_dir = bad_dir
        self.good_dir = good_dir
        self.target_sample_rate = target_sample_rate
        self.segment_length = segment_length

        self.bad_files = sorted([f for f in os.listdir(bad_dir) if f.endswith(".wav")])
        self.good_files = sorted([f for f in os.listdir(good_dir) if f.endswith(".wav")])

        """
        Перевіряємо, що кількість файлів збігається
        """
        assert len(self.bad_files) == len(self.good_files), "Кількість файлів у папках bad та good не збігається."

        """
        Створюємо список пар файлів
        """
        self.pairs = []
        for bad_file, good_file in zip(self.bad_files, self.good_files):
            self.pairs.append((os.path.join(bad_dir, bad_file), os.path.join(good_dir, good_file)))

        self.resampler = None

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

    def __getitem__(self, idx):
        bad_path, good_path = self.pairs[idx]

        """
        Завантаження та обробка "поганого" треку
        """
        bad_waveform, sample_rate = torchaudio.load(bad_path)
        if sample_rate != self.target_sample_rate:
            if self.resampler is None:
                self.resampler = Resample(orig_freq=sample_rate, new_freq=self.target_sample_rate)
            bad_waveform = self.resampler(bad_waveform)
        bad_waveform = bad_waveform / (bad_waveform.abs().max() + 1e-8)

        """
        Завантаження та обробка "хорошого" треку
        """
        good_waveform, sample_rate = torchaudio.load(good_path)
        if sample_rate != self.target_sample_rate:
            if self.resampler is None:
                self.resampler = Resample(orig_freq=sample_rate, new_freq=self.target_sample_rate)
            good_waveform = self.resampler(good_waveform)
        good_waveform = good_waveform / (good_waveform.abs().max() + 1e-8)

        """
        Перевіряємо, що довжини треків збігаються
        """
        min_length = min(bad_waveform.shape[1], good_waveform.shape[1])
        bad_waveform = bad_waveform[:, :min_length]
        good_waveform = good_waveform[:, :min_length]

        """
        Випадковий виріз сегмента
        """
        if min_length > self.segment_length:
            max_start = min_length - self.segment_length
            start = torch.randint(0, max_start + 1, (1,)).item()
            bad_segment = bad_waveform[:, start:start+self.segment_length]
            good_segment = good_waveform[:, start:start+self.segment_length]
        else:
            """
            Паддінг, якщо трек коротший за сегмент
            """
            pad_length = self.segment_length - min_length
```

```
                bad_segment = torch.nn.functional.pad(bad_waveform, (0, pad_length))
                good_segment = torch.nn.functional.pad(good_waveform, (0, pad_length))

            """
            Аугментація
            """
            bad_segment = self.augment_audio(bad_segment)

            return bad_segment, good_segment

        def augment_audio(self, waveform):
            """
            Аугментація аудіо: додавання шуму та випадкова зміна гучності.
            """
            # Додавання випадкового шуму
            noise = torch.randn_like(waveform) * 0.005
            waveform = waveform + noise

            # Випадкова зміна гучності
            gain = torch.FloatTensor(1).uniform_(0.9, 1.1).to(waveform.device)
            waveform = waveform * gain

            return waveform
```

datset_gtzan.py:

```
import os
import torch
import torchaudio
from torch.utils.data import Dataset
from torchaudio.transforms import Resample

class GTZANDataset(Dataset):
    def __init__(self, root_dir, target_sample_rate=22050, segment_length=65536):
        self.root_dir = root_dir
        self.target_sample_rate = target_sample_rate
        self.segment_length = segment_length

        """
        Створення списку файлів
        """
        self.file_list = []
        genres = os.listdir(root_dir)
        for genre in genres:
            genre_dir = os.path.join(root_dir, genre)
            if os.path.isdir(genre_dir):
                files = [os.path.join(genre_dir, f) for f in os.listdir(genre_dir) if f.endswith('.wav')]
                self.file_list.extend(files)
        self.resampler = None

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

    def __getitem__(self, idx):
        waveform, sample_rate = torchaudio.load(self.file_list[idx])
        if sample_rate != self.target_sample_rate:
            if self.resampler is None:
                self.resampler = Resample(orig_freq=sample_rate, new_freq=self.target_sample_rate)
            waveform = self.resampler(waveform)
        waveform = waveform / (waveform.abs().max() + 1e-8)

        """
        Випадковий виріз сегмента
        """
        if waveform.shape[1] > self.segment_length:
            max_start = waveform.shape[1] - self.segment_length
            start = torch.randint(0, max_start + 1, (1,)).item()
            segment = waveform[:, start:start+self.segment_length]
        else:
            """
            Паддінг, якщо трек коротший за сегмент
            """
            pad_length = self.segment_length - waveform.shape[1]
            segment = torch.nn.functional.pad(waveform, (0, pad_length))

        """
        Аугментація
        """
        segment = self.augment_audio(segment)

        return segment

    def augment_audio(self, waveform):
        """
        Аугментація аудіо: додавання шуму та випадкова зміна гучності.
        """
        # Додавання випадкового шуму
        noise = torch.randn_like(waveform) * 0.005
        waveform = waveform + noise

        # Випадкова зміна гучності
        gain = torch.FloatTensor(1).uniform_(0.9, 1.1).to(waveform.device)
        waveform = waveform * gain

        return waveform
```

enhance_audio.py:

```
import torch
import torchaudio
from model import WaveUNet

class AudioProcessorApp:
    def __init__(self, model_path, device=None):
        self.device = device if device else ("cuda" if torch.cuda.is_available() else "cpu")
        self.model = WaveUNet(in_channels=2, out_channels=2, num_channels=24, num_layers=12, kernel_size=15, dropout=0.0)

self.model.load_state_dict(torch.load(model_path, map_location=self.device))
        self.model.to(self.device)
        self.model.eval()

    def process_and_save_track(self, input_path, save_path):
        """
        Обробляє аудіо трек та зберігає результат.
        """
        try:
            print(f"Обробка треку: {input_path}")

            # Завантажуємо трек
            waveform, sample_rate = torchaudio.load(input_path)
            waveform = waveform / (waveform.abs().max() + 1e-8)

            # Обробка сегментами
            segment_length = 131072  # Повинен відповідати довжині сегмента, використовуваній при навчанні
            processed_waveform = self.process_waveform(waveform, segment_length)

            # Нормалізація вихідного waveform
            processed_waveform = processed_waveform / (processed_waveform.abs().max() + 1e-8)

            # Зберігаємо трек
            torchaudio.save(save_path, processed_waveform, sample_rate)
            print(f"Трек оброблено та збережено за адресою: {save_path}")

        except Exception as e:
            print(f"Помилка під час обробки: {e}")
            raise

    def process_waveform(self, waveform, segment_length):
        """
        Обробляє waveform сегментами.
        """
        total_samples = waveform.shape[1]
        processed_waveform = torch.zeros_like(waveform)

        with torch.no_grad():
            for start in range(0, total_samples, segment_length):
                end = min(start + segment_length, total_samples)
                segment = waveform[:, start:end]

                # Паддінг сегмента до потрібної довжини
                if segment.shape[1] < segment_length:
                    pad_length = segment_length - segment.shape[1]
                    segment = torch.nn.functional.pad(segment, (0, pad_length))

                segment = segment.to(self.device)

                output = self.model(segment.unsqueeze(0))
                output = output.squeeze(0).cpu()

                # Обрізання паддінгу, якщо був доданий
                output = output[:, :end - start]

                processed_waveform[:, start:end] = output

        return processed_waveform
```

train.py:

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, ConcatDataset
from model import WaveUNet
from dataset import AudioDataset
from dataset_gtzan import GTZANDataset
from tqdm import tqdm
import matplotlib.pyplot as plt
from torch.utils.tensorboard import SummaryWriter

def train_model(config):
    """
    Функція для навчання моделі Wave-U-Net.
    """
    device = "cuda" if config.use_gpu and torch.cuda.is_available() else "cpu"

    datasets = []

    if config.use_pairs:
        """
```

```
            Використовуємо пари треків
            """
            dataset_pairs = AudioDataset(
                bad_dir=config.bad_tracks_dir,
                good_dir=config.good_tracks_dir,
target_sample_rate=config.target_sample_rate,
segment_length=config.segment_length
            )
            datasets.append(dataset_pairs)

        if config.use_gtzan:
            """
            Використовуємо датасет GTZAN
            """
            dataset_gtzan = GTZANDataset(
                root_dir=config.gtzan_dir,
target_sample_rate=config.target_sample_rate,
segment_length=config.segment_length
            )
            datasets.append(dataset_gtzan)

        """
        Об'єднуємо датасети
        """
        if len(datasets) == 0:
            raise ValueError("Не вказано жодного джерела даних для навчання.")
        elif len(datasets) == 1:
            dataset = datasets[0]
        else:
            dataset = ConcatDataset(datasets)

        dataloader = DataLoader(dataset, batch_size=config.batch_size, shuffle=True, num_workers=4)

        """
        Ініціалізація моделі
        """
        model = WaveUNet(
            in_channels=config.in_channels,
            out_channels=config.out_channels,
            num_channels=config.num_channels,
            num_layers=config.num_layers,
            kernel_size=config.kernel_size,
            dropout=config.dropout
        )
        model = model.to(device)

        criterion = nn.L1Loss()
        optimizer = optim.AdamW(model.parameters(), lr=config.learning_rate, weight_decay=1e-5)

        """
        Ініціалізація TensorBoard
        """
        writer = SummaryWriter(log_dir=config.tensorboard_log_dir)

        """
        Переініціалізація ваг моделі (якщо необхідно)
        """
        for m in model.modules():
            if isinstance(m, (nn.Conv1d, nn.ConvTranspose1d)):
                torch.nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    torch.nn.init.zeros_(m.bias)

        loss_history = []

        for epoch in range(config.num_epochs):
            epoch_loss = 0
            model.train()
            with tqdm(dataloader, desc=f"Епоха [{epoch + 1}/{config.num_epochs}]", unit="batch") as pbar:
                for batch_idx, data in enumerate(pbar):
                    optimizer.zero_grad()

                    if config.use_pairs:
                        inputs, targets = data
                        inputs = inputs.to(device)
                        targets = targets.to(device)
                    else:
                        """
                        Якщо використовується лише GTZAN, навчаємо автоенкодер
                        """
                        inputs = data.to(device)
                        targets = inputs

                    outputs = model(inputs)

                    loss = criterion(outputs, targets)

                    loss.backward()

                    """
                    Кліпінг градієнтів
                    """
                    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

                    optimizer.step()
                    epoch_loss += loss.item()

                    pbar.set_postfix(loss=f"{loss.item():.4f}")

                    """
                    Логування в TensorBoard
                    """
                    global_step = epoch * len(dataloader) + batch_idx
                    writer.add_scalar('Loss/Train', loss.item(), global_step)
```

```
            avg_epoch_loss = epoch_loss / len(dataloader)
            loss_history.append(avg_epoch_loss)
            print(f"Епоха [{epoch + 1}/{config.num_epochs}], Середня Втрата: {avg_epoch_loss:.4f}")

            """
            Логування середнього значення функції втрат за епоху
            """
            writer.add_scalar('Loss/Epoch', avg_epoch_loss, epoch + 1)

            """
            Збереження моделі після кожної епохи (опціонально)
            """
            # torch.save(model.state_dict(), f"models/waveunet_epoch_{epoch+1}.pth")

        """
        Закриття TensorBoard writer
        """
        writer.close()

        """
        Збереження фінальної моделі
        """
        torch.save(model.state_dict(), config.model_save_path)
        print(f"Модель збережена за адресою {config.model_save_path}")

        """
        Побудова графіка функції втрат
        """
        plt.figure()
        plt.plot(range(1, config.num_epochs + 1), loss_history, label='Втрата Навчання')
        plt.xlabel('Епоха')
        plt.ylabel('Втрата')
        plt.title('Історія Втрат Навчання')
        plt.legend()
        plt.savefig('loss_history.png')
        plt.show()
```