КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра радіотехніки та радіоелектронних систем

«На правах рукопису»

Робота допущена до захисту в ЕК
рішенням кафедри радіотехніки та радіоелектронних систем
від 20 травня 2024 року, протокол № 111.
Завідувач кафедри доктор фіз.-мат. наук, професор
_____ Ігор АНІСІМОВ

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
на тему:

«ІНТЕЛЕКТУАЛЬНА СИСТЕМА ОБРОБКИ ЗВУКУ НА ОСНОВІ МАШИННОГО НАВЧАННЯ ДЛЯ РАДІОТЕХНІЧНИХ ПРИСТРОЇВ»

Виконав:
студент 4-го курсу
денної форми навчання
спеціальності 172 - Телекомунікації та радіотехніка
ОПП «Інформаційна безпека телекомунікаційних систем і мереж»
Мартинюк Стефан Володимирович _____

Науковий керівник:
канд. фіз.-мат. наук, доцент
Кононов Михайло Володимирович _____

Рецензент:
канд. фіз.-мат. наук, доцент
Єфіменко Світлана Володимирівна _____

Засвідчую, що у цій бакалаврській роботі
немає запозичень з праць інших авторів без
відповідних посилань
Студент Мартинюк Стефан Володимирович _____

Київ - 2024

РЕФЕРАТ

Дипломна робота: 58 с., 15 рис., 2 дод. (21с.), 13 джерел.

НЕЙРОННІ МЕРЕЖІ, СТИСНЕННЯ АУДІО, ПЕРІОДИЧНІ СИГНАЛИ, ГЕНЕРАТИВНО-ЗМАГАЛЬНІ МЕРЕЖІ, АКТИВАЦІЙНА ФУНКЦІЯ SNAKE, ВЕКТОРНЕ КВАНТУВАННЯ, СПЕКТРОГРАМИ.

Об’єкт розроблення – система стиснення аудіосигналів з використанням нейронних мереж.

Мета роботи – розробка модифікованих методів машинного навчання для стиснення аудіосигналів з високою точністю та нижчою вимогливістю до обчислювальних ресурсів.

У роботі розглянуто сучасні підходи до стиснення аудіосигналів за допомогою глибоких нейронних мереж. Зокрема, представлено аналіз проблем, з якими зустрічаються існуючі методи, такі як артефакти, що виникають під час синтезу, та велика вимогливість до обчислювальних ресурсів. Розроблені покращення включають використання нової функції активації Snake для кращого узагальнення періодичних сигналів, оптимізацію архітектури дискримінатора для точнішої роботи зі спектрограмами, та вдосконалення методів векторного квантування для зниження ресурсоємності.

Було розроблено модифіковану архітектуру нейронного аудіокодека, яка інтегрує удосконалені методи, та проведено тренування моделі на базі датасетів MUSDB18 та Common Voice. Результати показали покращення коефіцієнта стиснення та зниження необхідної обчислювальної потужності порівняно з існуючими рішеннями, такими як SoundStream.

Для подальших досліджень рекомендується розглянути можливість інтеграції розроблених методів у мобільні та вбудовані системи, що вимагають ефективного стиснення аудіо при низькій вимогливості до обчислювальних ресурсів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

VAE — варіаційний автокодер;

GAN — генеративна змагальна мережа;

STFT — віконне перетворення Фур’є;

MSD — багатомасштабний дискримінатор;

MPD — багатоперіодичний дискримінатор;

MRSD — дискримінатор спектрограм з множинною роздільною здатністю;

RVQ — залишковий векторний квантувач;

LUFS — гучність відносно повної шкали;

VQ-VAE — векторно-квантований варіаційний автокодер;

sg — оператор зупинки градієнта;

D — дискримінатор;

G — генератор;

Opus — стандарт стиснення аудіо;

SoundStream — нейронний аудіокодек для стиснення аудіо.

ВСТУП

Генеративне моделювання аудіо високої роздільної здатності є складним завданням через високу розмірність і різноманітну структуру на різних часових масштабах. Для вирішення цієї проблеми генерацію звуку часто поділяють на два етапи: прогнозування звуку на основі проміжного представлення, такого як мелоспектрограми, і прогнозування цього представлення на основі умовної інформації, наприклад, тексту. Це можна інтерпретувати як ієрархічну генеративну модель із проміжними змінними. Альтернативний підхід включає використання варіаційного автокодера (VAE) для прогнозування латентних змінних за певних умов, що виявилося успішним для синтезу мови.

Стиснення аудіо з втратами можна розглядати як задачу векторного квантування представлень автокодера з використанням кодової книги фіксованої довжини. Модель стиснення повинна, реконструювати аудіо з високою точністю без артефактів, досягати високого рівня стиснення та компактного представлення, зберігаючи високорівневу структуру та обробляти різні типи аудіо, такі як мова, музика, звуки навколишнього середовища, різні аудіокодування та частоти дискретизації, використовуючи єдину універсальну модель.

Незважаючи на досягнення у нейронних алгоритмах стиснення аудіо, таких як SoundStream і EnCodec, вони все ще можуть демонструвати звукові артефакти, як-то тональні артефакти та недосконале моделювання високих частот, що призводить до помітного відхилення від оригіналу. Крім того, ці моделі часто налаштовані на конкретні типи аудіосигналів і можуть мати труднощі з моделюванням загальних звуків.

Ця робота спрямована на розробку інтелектуальної системи обробки звуку на основі сучасних методів машинного навчання, яка забезпечуватиме високоякісне стиснення аудіосигналів.

1. ІНТЕЛЕКТУАЛЬНІ МЕТОДИ ОБРОБКИ АУДІОСИГНАЛІВ

Розвиток технологій машинного навчання та глибоких нейронних мереж відкриває нові можливості для обробки аудіосигналів. Зокрема, сучасні методи дозволяють значно покращити якість стиснення аудіо, генерацію нових звуків, та забезпечити точне моделювання аудіо у реальному часі. У цьому розділі розглядаються ключові методи та технології, що використовуються для обробки аудіосигналів з використанням генеративних моделей.

Першим підходом, який буде детально розглянуто, є генеративні змагальні мережі (GANs). Ці мережі дозволяють генерувати нові зразки аудіо, що максимально наближені до реальних даних, використовуючи два взаємодіючих компоненти: генератор та дискримінатор. GANs вже продемонстрували значний успіх у таких завданнях, як генерація зображень, тексту та аудіо, і продовжують залишатися однією з найактивніше досліджуваних технологій у цій сфері.

Далі буде розглянуто нейронний аудіокодек SoundStream, який здатний ефективно стискати аудіосигнали, зберігаючи високу якість відтворюваного звуку. SoundStream використовує інноваційні підходи, такі як залишковий векторний квантувач та змагальні втрати, що дозволяє досягти високого рівня стиснення без втрати якості.

Окрім цього, особлива увага приділяється новим активаційним функціям, таким як Snake. Ця функція забезпечує ефективне навчання нейронних мереж для моделювання періодичних функцій, що є важливим для задач прогнозування та обробки періодичних сигналів. Активаційна функція Snake демонструє високу точність та універсальність у різних задачах машинного навчання, що робить її цінним інструментом для обробки аудіо.

У цьому розділі детально аналізуються принципи роботи, переваги та обмеження кожного з підходів, а також їх застосування для обробки аудіосигналів.

1. 1. Генеративні змагальні мережі

Генеративні змагальні мережі [1], відомі як Generative Adversarial Networks (GANs), є класом машинних алгоритмів у сфері штучного інтелекту, що використовуються для генерації нових даних, подібних до наявних. Ця методика була запропонована Іаном Гудфеллоу і його колегами в 2014 році.

GANs складаються з двох нейронних мереж — генератора (G) і дискримінатора (D), які тренуються одночасно у змагальному режимі:

Генератор (G):

Мета: Генерувати зразки, що максимально наближені до реальних даних.

Вхід: Випадковий вектор z з латентного простору.

Вихід: Згенерований зразок даних G(z).

Дискримінатор (D):

Мета: Розрізняти справжні зразки даних і згенеровані зразки.

Вхід: Реальний зразок даних x або згенерований зразок G(z).

Вихід: Ймовірність того, що зразок є справжнім.

Процес навчання GANs полягає в одночасному оновленні ваг генератора і дискримінатора з використанням змагального підходу:

Навчання дискримінатора:

Дискримінатор тренується на реальних даних x з тренувальної вибірки, щоб максимально збільшити ймовірність класифікації їх як справжніх (D(x) → 1).

Дискримінатор тренується на згенерованих даних G(z), щоб мінімізувати ймовірність класифікації їх як справжніх (D(G(z)) → 0).

Навчання генератора:

Генератор тренується з метою обдурити дискримінатор, тобто згенеровані дані G(z) повинні бути класифіковані дискримінатором як справжні (D(G(z)) → 1).

Функція втрат для дискримінатора L_D і генератора L_G формулюється наступним чином:

$$L_D = -E_{x\sim p_{data}}[log\ D(x)] - E_{z\sim p_z}[log\ (1 - D(G(z))], \quad (1.1)$$

$$L_G = \quad - E_{z \sim p_z}[log\ (D(G(z))]\,, \tag{1.2}$$

Де p_{data} – розподіл справжніх даних, p_z – розподіл випадкових векторів в латентному просторі, $D(x)$ – ймовірність, що дискримінатор класифікує реальний зразок x як справжні, $D(G(z))$ – ймовірність, що дискримінатор класифікує згенерований зразок $G(z)$ як справжній.

Генеративні змагальні мережі є потужним підходом до генерації нових даних, які є подібними до наявних, використовуючи два змагальні компоненти — генератор і дискримінатор.

1. 2. Нейронний аудіокодек SoundStream

SoundStream [2] — нейронний аудіокодек, здатний ефективно стискати мову, музику та загальні аудіо при бітрейтах, що зазвичай використовуються для кодеків, спеціально призначених для мови. SoundStream використовує архітектуру, яка складається з повністю згорткової мережі енкодера/декодера і залишкового векторного квантувача, які тренуються спільно. Навчання поєднує змагальні та реконструкційні втрати для створення високоякісного аудіоконтенту з квантованих векторів.

Модель SoundStream, яка зображена на рисунку 1.1, складається з трьох основних компонентів: енкодер, залишкового векторного квантувача (Residual Vector Quantizer, RVQ) [3] і декодера.

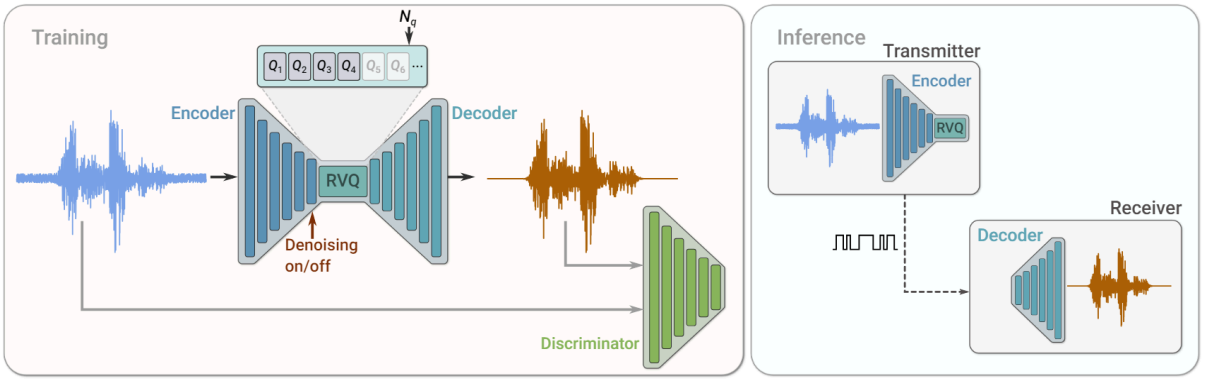


Рисунок 1.1 – Архітектура нейронного аудіокодека SoundStream [2]

Енкодер: перетворює вхідні аудіозаписи у послідовність векторів меншої розмірності.

Залишковий векторний квантувач (RVQ) [3]: кожен вектор квантувача відображається в простір дискретних кодових векторів, зменшуючи розмірність та стискаючи дані. Використовується кілька шарів квантувачів для покращення точності.

Декодер: реконструює аудіосигнал з квантованих векторів, застосовуючи зворотні згортки та блочні залишкові одиниці (Residual Units).

Залишковий векторний квантувач працює, поступово квантуючи залишки після кожного процесу квантування. Це дозволяє зменшити розмірність кодової книги та підвищити ефективність стиснення. Кількість квантувачів та розмір кодової книги налаштовуються залежно від цільового бітрейту. Використовується метод дроп-аут блоків квантування, тобто метод регуляризації нейронних мереж, який відкидає одиницю нейронної мережі (разом зі зв'язками) під час навчання із заданою ймовірністю. Це забезпечує масштабованість бітрейту та а також можливисть моделі працювати при різних бітрейтах без необхідності перенавчання.

Математично залишковий векторний квантувач для нейронного аудіокодека SoundStream можна описати наступним чином:

$$\widehat{y} = \sum_{i=1}^{N_q} Q_i(residual_i), \tag{1.3}$$

Де $\widehat{y}$ – квантований вектор, Q_i – функція квантування для і-го шару, $residual_i$ – залишок після квантування на і-му кроці, що обчислюється як:

$$residual_i = y - \sum_{j=1}^{i-1} Q_j(residual_j), \tag{1.4}$$

Для покращення якості відтворюваного аудіо, модель тренується за допомогою змагальних втрат, що дозволяють генератору створювати аудіо, яке важко відрізнити від оригінального, і реконструкційних втрат, що забезпечують

точність відновлення сигналу. Змагальні втрати обчислюються за допомогою дискримінаторів, що оцінюють відмінності між відтвореним та оригінальним аудіо на різних часових і частотних масштабах.

Формула змагальних втрат для дискримінатора:

$$L_D = E_x\left[\frac{1}{K}\sum_{k=0}^{K}\frac{1}{T_k}\sum_{t=1}^{T_k} max(0,\ 1 - D_{k,t}(x))\right] + \\ + E_x\left[\frac{1}{K}\sum_{k=0}^{K}\frac{1}{T_k}\sum_{t=1}^{T_k} max(0,\ 1 - D_{k,t}(G(x)))\right], \quad (1.5)$$

Де K – кількість дискримінаторів, T_k – кількість логітів (виходів нейронної мережі зі значеннями перед застосуванням функції активації) у виході k-го дискпримінатора, $D_{k,t}(x)$ – логіт дискримінатора для вхідного аудіо, $D_{k,t}(G(x)))$ – логіт дискримінатора для згенерованого аудіо $G(x)$. Ця формула визначає втрати дискримінатора, які використовуються для навчання дискримінатора розрізняти оригінальні та згенеровані аудіозаписи.

Змагальні втрати генератора визначаються наступною формулою:

$$L_{adv}^{G} = E_x\left[\frac{1}{K}\sum_{k=0}^{K}\frac{1}{T_k}\sum_{t=1}^{T_k} max(0,\ 1 - D_{k,t}(G(x)))\right], \quad (1.6)$$

Де L_{adv}^{G} змагальні втрати для генератора та $G(x)$ вектор згенерованого аудіо.

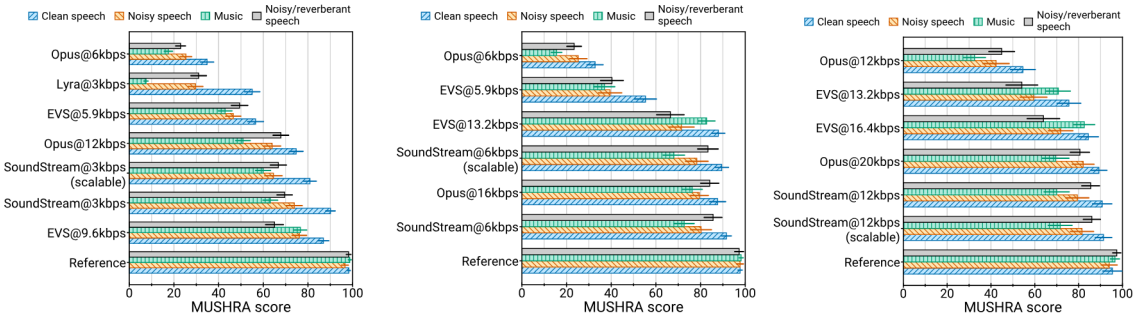
Загальні втрати генератора визначаються сумою змагальних втрати, втрат на основі ознак та мултишкальних спектральних втра помноженим на відповідні коефіцієнти для кожного типу втрат.

$$L_G = \lambda_{adv} L_{adv}^{G} + \lambda_{feat} L_{feat}^{G} + \lambda_{rec} L_{rec}^{G}, \quad (1.7)$$

Де L_{adv}^{G} – змагальні втрати для генератора, L_{feat}^{G} – втрати на основі ознак, L_{rec}^{G} – мультишкальні спектральні втрати та λ_{adv}, λ_{feat}, λ_{rec} – вагові коефіцієнти для кожного типу втрат.

Модель SoundStream оцінювалась за допомогою суб'єктивних (MUSHRA) та

об'єктивних методів (ViSQOL). В суб'єктивних оцінках, що на рисунку 1.2, SoundStream при бітрейті 3 кбіт/с перевершує Opus при 12 кбіт/с та підходить до EVS при 9,6 кбіт/с. Об'єктивні метрики, такі як ViSQOL, показують, що SoundStream забезпечує високу якість відтворення навіть при низьких бітрейтах. SoundStream перевершує Opus та EVS при різних бітрейтах у суб'єктивних тестах. Наприклад, при бітрейті 3 кбіт/с SoundStream демонструє кращу якість, ніж Opus при 6 кбіт/с і EVS при 5,9 кбіт/с. Навіть при збільшенні бітрейту до 6 кбіт/с і 12 кбіт/с SoundStream залишається кращим.



(a) Low bitrate. (b) Medium bitrate. (c) High bitrate.

Рисунок 1.2 – Результати суб'єктивного оцінювання за типами контенту. Смуги похибок позначають 95% довірчі інтервали.

SoundStream є нейронним аудіокодеком, який забезпечує високу якість стискання аудіо при низьких та середніх бітрейтах.

1. 3. Функція активації Snake

У роботі “Нейронні мережі не здатні вивчати періодичні функції і як це виправити” [4] розглянуто проблему навчання нейронних мереж для моделювання періодичних функцій. Автори доводять, що стандартні активаційні функції, такі як ReLU, tanh та sigmoid, не здатні ефективно екстраполювати періодичні функції. Для вирішення цієї проблеми запропоновано нову активаційну функцію $x + sin^2(x)$, яка поєднує в собі періодичну природу синусоїдальної функції та зберігає оптимізаційні властивості активаційних функцій на основі ReLU.

Періодичні функції мають властивість повторюваності з певним періодом. Для моделювання таких функцій необхідно, щоб нейронна мережа не тільки добре інтерполювала дані в межах тренувального діапазону, але й ефективно екстраполювала за його межами.

Для перевірки властивостей екстраполяції було проведено експерименти з використанням невеликої нейронної мережі з одним прихованим шаром, що складається з 512 нейронів. Дані тренування генерувалися шляхом вибірки з чотирьох різних аналітичних функцій у діапазоні [−5,5], з пропуском у діапазоні [−1,1]. Результати показали, що поведінка екстраполяції залежить від аналітичної форми активаційної функції: ReLU дивергує до ±∞, а tanh прагне до постійного значення.

Було доведено, що стандартні активаційні функції не здатні екстраполювати періодичні функції. Розглянуто дві теореми для функцій ReLU та tanh.

Теорема для ReLU:

$$\lim_{z \to \infty} \left|\left| f_{ReLU}(zu) - zW_u u - b_u \right|\right|^2 = 0, \tag{1.8}$$

де z — дійсний скаляр, u — одиничний вектор, W_u та b_u — матриця перетворення та зміщення відповідно. Це означає, що будь-яка нейронна мережа з активацією ReLU конвертується до лінійного перетворення у асимптотичному ліміті.

Теорема для tanh:

$$\lim_{z \to \infty} \left\| f_{tanh}(zu) - v_u \right\|^2 = 0, \quad (1.9)$$

де v_u — константний вектор, що залежить лише від u. Це означає, що нейронна мережа з активацією tanh екстраполює як постійна функція.

Щоб подолати обмеження стандартних активаційних функцій, автори запропонували використати активаційну функцію $x + sin^2(x)$, яку назвали "Snake". Ця функція поєднує в собі монотонність та періодичність, що дозволяє ефективно навчатися на періодичних даних.

Формула активаційної функції Snake:

$$Snake_a := x + \frac{1}{a} sin^2(ax) = x - \frac{1}{2} cos(2ax) + \frac{1}{2}, \quad (1.10)$$

де a — частота періодичної частини. Більше значення a а забезпечує вищу частоту.

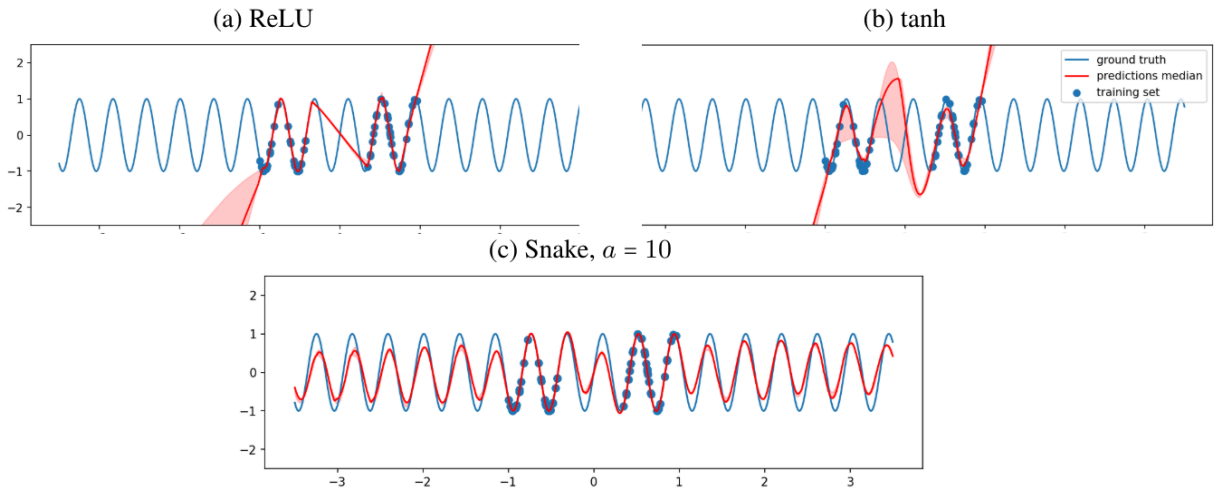


Рисунок 1.3 – Регресія простої sin функції з tanh, ReLU та Snake як функціями активації.

У роботі було проведено експерименти з регресією простої періодичної функції $sin(x)$ з використанням запропонованої активаційної функції у порівнянні з іншими стандартними активаційними функціями. Було показано, що Snake забезпечує значно кращу екстраполяцію періодичної природи функції.

Автори довели універсальну теорему екстраполяції, яка стверджує, що нейронна мережа з активацією Snake може апроксимувати будь-яку періодичну

функцію, що має сильні ознаки періодичності.

Теорема: Нехай $f(x)$ – шматочно-неперервна періодична функція з періодом L. Тоді нейронна мережа Snake з одним прихованим шаром і шириною N може сходитися до $f(x)$ рівномірно при $N \to \infty$.

$$\lim_{N \to \infty} f_{w_N}(x), \tag{1.12}$$

де $f_{w_N}(x)$ – нейронна мережа Snake з параметрами w_N.

Для забезпечення одиничної дисперсії вихідного сигналу кожного шару, необхідно використовувати правильну схему ініціалізації. Для активаційної функції Snake пропонується використовувати наступну дисперсію:

$$\sigma_a^2 = 1 + \frac{1 + e^{-8a^2} - 2e^{-4a^2}}{8a^2}, \tag{1.13}$$

що досягає максимуму при $a_{max} \approx 0.56045$.

Активаційна функція Snake була протестована на ряді завдань, включаючи класифікацію зображень, прогнозування атмосферної температури та прогнозування фінансових даних. У всіх завданнях Snake показала конкурентоспроможність або перевершила інші стандартні активаційні функції

Було проведено експерименти на датасеті CIFAR-10 з використанням ResNet-18. Я видно на рисунку 1.4, Активаційна функція Snake показала продуктивність, порівнянну з ReLU і Leaky-ReLU.

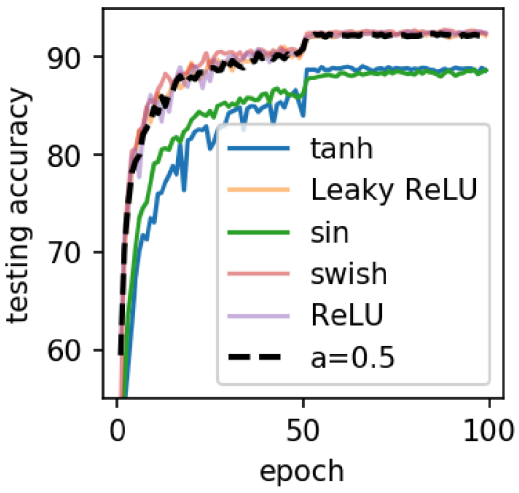


Рисунок 1.4 – Порівняння з іншими функціями активації на CIFAR10.

На рисунку 1.5 модель з активацією Snake змогла правильно екстраполювати періодичний характер даних температури на острові Мінамі-Торісіма. Для цього було використано нейронну мережу з двома прихованими шарами.

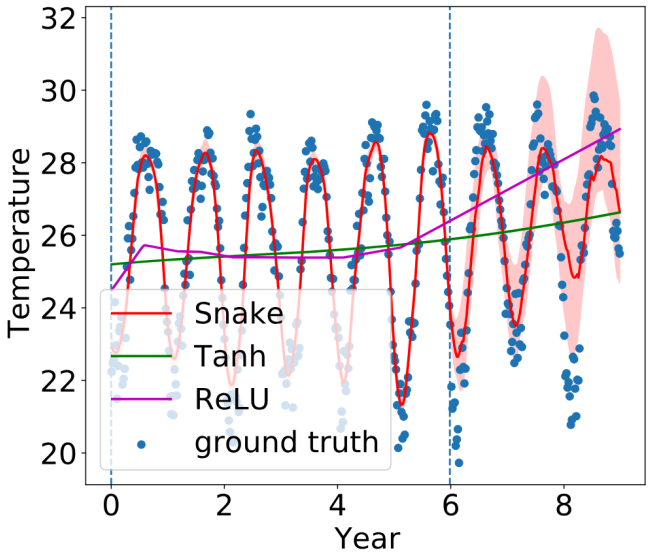


Рисунок 1.5 – Регресія середньої тижневої температури Мінамі-Торісіма з різними функціями активації. Для Snake, а розглядається як параметр, що навчається як параметр, що навчається, а червоний контур показує 90% довірчий інтервал.

Було проведено експерименти з прогнозування капіталізації ринку США за індексом Wilshire 5000. Я продемонстровано на рисунку 1.6, метод Snake показав значно кращі результати порівняно з іншими методами, включаючи традиційні ARIMA моделі.

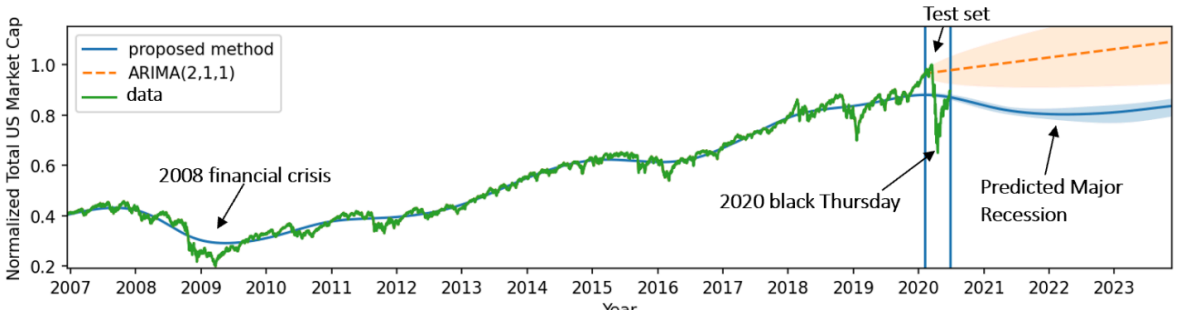


Рисунок 1.6 – Прогноз індексу Wilshire 5000, індикатора американської та світової економіки

Дослідження показало, що стандартні активаційні функції не здатні

ефективно моделювати періодичні функції. Запропонована активаційна функція $x + sin^2(x)$ забезпечує необхідні властивості для моделювання періодичних даних і демонструє конкурентоспроможність на стандартних задачах. Відомо, що аудіосигнали мають високу періодичність (особливо у голосових компонентах, музиці тощо). Тому є сенс у тому, що застосувавши функцію активації Snake до аудіообробки, можна отримати кращий результат при тренуванні генеративної змагальної моделі.

2. ПОКРАЩЕННЯ СИНТЕЗУ АУДІО СИГНАЛІВ НА БАЗІ ІСНУЮЧИХ МЕТОДІВ МАШИННОГО НАВЧАННЯ

Існуючі методи стиснення аудіо за допомогою нейронних мереж часто стикаються з проблемами, пов'язаними з артефактами, які виникають при синтезі аудіо. Також, попри високу степінь стиснення, нейронні мережі такого плану на практиці не застосовуються через їх вибагливість до ресурсів. Це дає простір для покращень наявної технології.

Одним із напрямків такого вдосконалення є розробка нових функцій активації, які б забезпечували краще узагальнення і реконструкцію періодичних сигналів. Функція активації Snake, зокрема, показала свою перевагу в моделюванні періодичних сигналів, забезпечуючи більшу точність і менші втрати при перекодуванні аудіо.

Удосконалення дискримінаторів у генеративно-змагальних мережах дозволяє більш точно аналізувати та відтворювати складність сигналів. Введення складних дискримінаторів, таких як множинний дискримінатор віконного перетворення Фур'є, сприяє кращому розпізнаванню і усуненню артефактів, що значно підвищує якість вихідних аудіофайлів.

Якщо підвищити ефективність використання кодових книг, можна зменшити вимоги до оперативної пам’яті, зменшивши вибагливість до ресурсів. Це можна зробити застосувавши векторне квантування [5] з впровадженням технік, як факторизовані коди та L2-нормалізовані коди відкинувши ресуроємні операції кластеризації k-середніх.

У цьому розділі детально розглядаються зазначені аспекти, їхнє вплив на процес синтезу аудіо та результати застосування на практиці.

2. 1. Нова функція активації для періодичних сигналів

Хоча сучасні неавторегресійні архітектури генерації звуку здатні створювати звук з високою точністю, вони часто демонструють артефакти висоти тону та періодичності [4]. Крім того, поширені функції активації нейронні мережі, такі як ReLU, відомі своєю нездатністю ефективно екстраполювати періодичні сигнали та

погано узагальнювати результати поза навчальним розподілом даних для синтезу звуку.

Для удосконалення моделі, було впроваджено функцію активації Snake [4]. Це дало змогу предствленій моделі краще узагальнювати періодичні сигнали звуку та збільшити точність відтворення звуку після декудування. Дана функція активації в даній реалізації має вигляд має вигляд визначається як

$$Snake_a := x - \frac{1}{2}cos(2ax) + \frac{1}{2}, \quad (2.1)$$

де a контролює частоту періодичної складової сигналу.

2. 2. Дискримінатор віконного перетворення Фур'є

У даній роботі використовуються багатомасштабні (MSD) та багатоперіодичні дискримінатори форми сигналу (MPD) [8], які покращують точність відтворення звуку. Проте, спектрограми згенерованого звуку все ще можуть виглядати розмитими через артефакти надмірного згладжування на високих частотах. Для усунення цих артефактів пропонується використання дискримінатора спектрограм з множинною роздільною здатністю (MRSD) [8], що допомагає зменшити артефакти висоти тону і періодичності [6]. Однак використання амплітудних спектрограм відкидає інформацію про фазу, яка могла б бути використана дискримінатором для корекції помилок фазового моделювання.

Більше того, високочастотне моделювання залишається складним завданням для цих моделей, особливо при високій частоті дискретизації. Для вирішення цих проблем у даній роботі застосовується складний дискримінатор віконного перетворення Фур'є на різних часових масштабах, який краще справляється з завданням на практиці та покращує фазове моделювання.

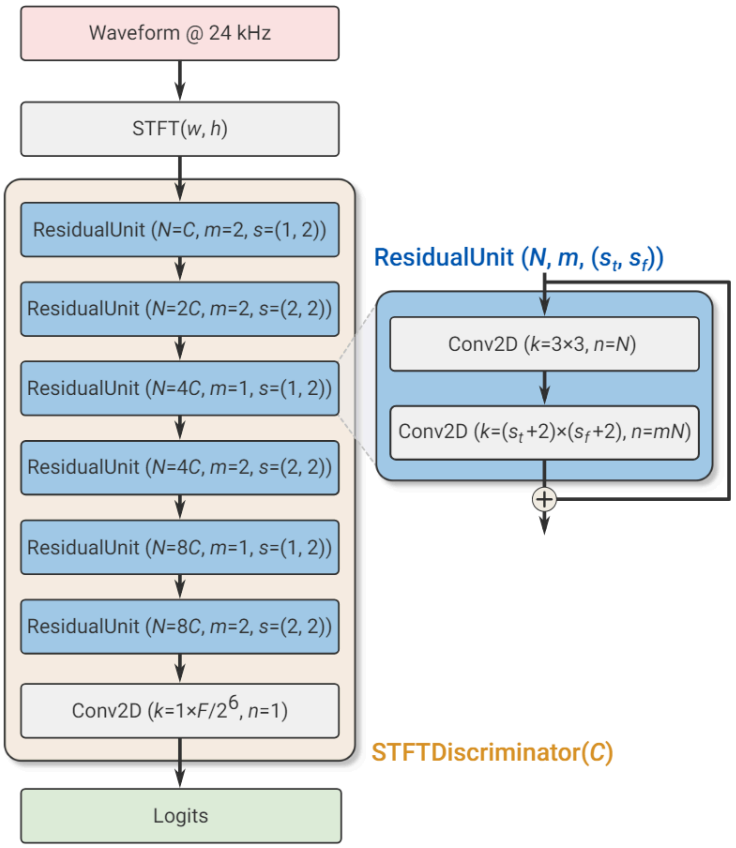


Рисунок 2.1 – Архітектура дискримінатора на основі віконного перетворення Фур'є (Short-time Fourier transform, STFT) [2]

Принцип роботи STFT дискримінатора можна описати наступним чином:

1. Вхідний аудіо-сигнал перетворюється за допомогою віконного перетворення Фур'є, що дає комплексні значення (дійсна та уявна частини), представлені для кожного часового вікна. Використовуючи параметри вікна, такі як довжина вікна (W) та крок зміщення (H), можна контролювати роздільну здатність аналізу.

2. STFT-сигнал подається на вхід повністю згорткового нейронного мережевого дискримінатора. Структура мережі дискримінатора буде складатись з початкового згорткового шару, послідовності блочних залишкових одиниць (Residual Units), кожна з яких складається з кількох згорткових шарів та останнього шару, який агрегує вихідні дані в одновимірний сигнал, що представляє логіти.

3. Аналогічно до хвильового дискримінатора, який аналізує вхідний

аудіо-сигнал на декількох роздільностях, STFT-дискримінатор опрацьовує сигнал на одній фіксованій роздільності з параметрами вікна і кроку зміщення.

Крім того, розбиття перетворення Фур'є на підсмуги покращує високочастотне прогнозування та зменшує артефакти аліасингу, оскільки дискримінатор вивчає дискримінаційні ознаки певного піддіапазону та подає сильніший градієнтний сигнал генератору.

2. 3. Покращене залишкове векторне квантування

Векторне квантування є популярним методом для тренування дискретних автоенкодерів, проте його застосування супроводжується рядом практичних труднощів. У базових VQ-VAE існує проблема низького використання кодової книги через невдалу ініціалізацію, що призводить до значної частини невикористаних кодів. Це зменшує ефективний розмір кодової книги, що, в свою чергу, призводить до зниження цільового бітрейту і поганої якості реконструкції аудіо.

Для подолання цієї проблеми, в останніх методах аудіокодеків використовують кластеризацію k-середніх для ініціалізації векторів кодової книги, а також вручну застосовують випадкові повторні запуски у випадках, коли певні коди не використовуються протягом кількох пакетів. Проте, виявилось, що модель модель з таким самим методом навчання кодової книги, все ще страждають від недостатнього використання частини кодів.

Для вирішення цієї проблеми використовується дві ключові техніки, для покращення використання кодової книги: факторизовані коди та L2-нормалізовані коди. Факторизація розділяє пошук кодів і їх вбудовування, здійснюючи пошук у низьковимірному просторі, тоді як вбудовування кодів знаходяться у високовимірному просторі [5]. Інтуїтивно це можна пояснити як пошук кодів, використовуючи тільки головні компоненти вхідного вектора, які максимально пояснюють варіацію даних. L2-нормалізація кодує та нормалізує вектори кодової книги, перетворюючи евклідову відстань на косинусну подібність, що сприяє стабільності та якості.

Ці дві техніки разом значно покращують використання кодової книги, а отже, ефективність бітрейту та якість реконструкції аудіо і при цьому є простими в реалізації. Модель може бути тренована з використанням оригінальних втрат кодової книги, без необхідності в ініціалізації кластеризації методом k-середніх або випадкових повторних запусків.

Модифікована операція квантування, визначається наступним чином:

$$z_q(x) = W_{out} e_k, \tag{2.2}$$

де

$$k = argmin_j \left|\left| l_2(W_{in} z_e(x)) - l_2(e_j) \right|\right|_2, \tag{2.3}$$

Тут, W_{in} та W_{out} є матрицями проекції, з W_{in} яка відображає вихід енкодера у проміжне представлення, а W_{out} перетворює це проміжне представлення в квантоване представлення $z_q(x)$. Зокрема, $W_{in} \in R^{D \times M}$ та $W_{out} \in R^{M \times D}$

де D є вихідним розміром енкодера, а M є розміром кодової книги, з $M \ll D$.

Функція втрат векторного квантувача тоді визначається для вимірювання помилки реконструкції і задається як:

$$z_{proj}(x) = W_{in} z_e(x), \tag{2.4}$$

$$k = \left|\left| sg\left[l_2(z_{proj}(x))\right] - l_2(e_k) \right|\right|_2^2 + \beta \left|\left| l_2(z_{proj}(x)) - sg\left[l_2(e_k)\right] \right|\right|_2^2, \tag{2.5}$$

де sg є оператором зупинки градієнта, що запобігає розповсюдженню градієнтів через e_k, і β є гіперпараметром, який контролює баланс між двома членами рівняння функції втрат.

2. 4. Архітектура нейронного аудіокодека

Архітектура нейронного більшою частиною була взята від SoundStream. В даному контексті було додано нову функцію активації Snake, що добре працює з періодичними данними для кращої роботи з адудіо. Ця архітектура використовує сучасні методології обробки сигналів, такі як використання розширених конволюцій та блоків залишкових одиниць для покращення вловлювання

контексту та глибшого навчання без втрати інформації при глибших шарах.

Модель складається з конволюційного енкодера, залишкового векторного квантувача та конволюційного декодера. Основний будівельний блок нейронної мережі - це конволюційний шар, який або збільшує, або зменшує з деяким кроком, за яким слідує шар залишкових блоків, що складається з конволюційних шарів, чередуючи з нелінійними активаціями Snake. Енкодер має 4 таких шари, кожен з яких знижує вхідну аудіохвилю на частоти [2, 4, 8, 8]. Декодер має 4 відповідних шари, які збільшують на частоти [8, 8, 4, 2]. Розмір на вхід декодера 1536. Всього модель має 76 млн параметрів, з яких 22 млн у енкодері і 54 млн у декодері.

Енкодер, архітектуру якого можна побачити на Рисунку 2.2, приймає вхідний аудіо сигнал та перетворює його в більші високорівневе представлення. Конволюція, що стоїть на початку енкодера, перетворює один аудіо канал у вектор, що предаставляє звуковий файл. Чотири послідовні блоки збільшують розмірність і одночасно зменшують просторовий розмір даних за допомогою кроку зміщення.

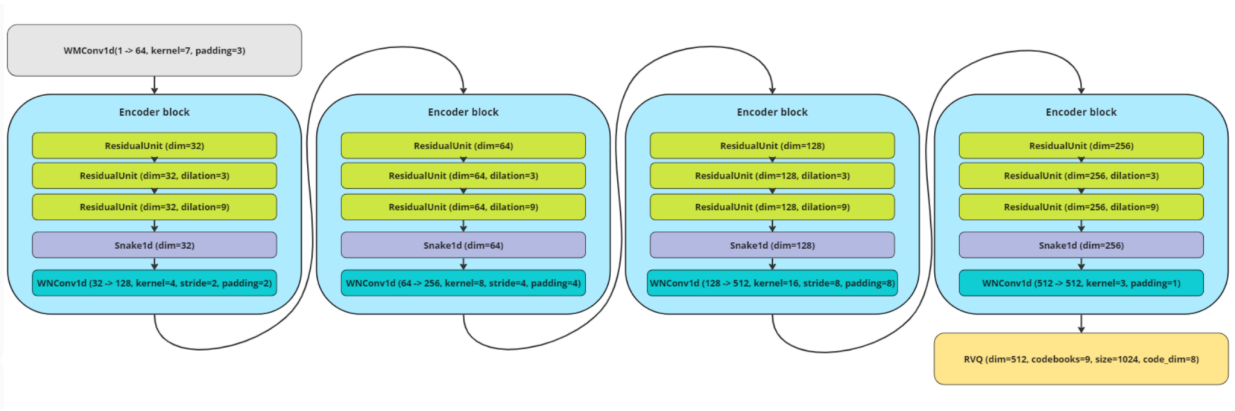


Рисунок 2.2 – Архітектура енкодера

Кожен блок містить залишкові одиниці з функцією активації Snake. Наприкінці стоїть векторний квантувач (RVQ), який перетворює неперервне представлення у дискретне за допомогою кількох кодових книг. Це дозволяє зберігати аудіо інформацію у більш компактному вигляді. Цей процес включає визначення і мінімізацію втрат кодової книги, що допомагає оптимізувати їх та наблизити вхідні даних до них.

Архітектуру декодера видно на Рисунку 2.3. Чотири блоки декодера, що послідовно відновлюють детальність аудіо сигналу, перетворюючи зменшену

просторову розмірність назад до вихідного розміру. Заключна конволюція та активація Tanh, нормалізує вихідний сигнал до діапазону [-1, 1]. Завдяки оберненим конволюціям і блокам залишкових одиницб, відновлений сигнал максимально наближається до оригіналу.

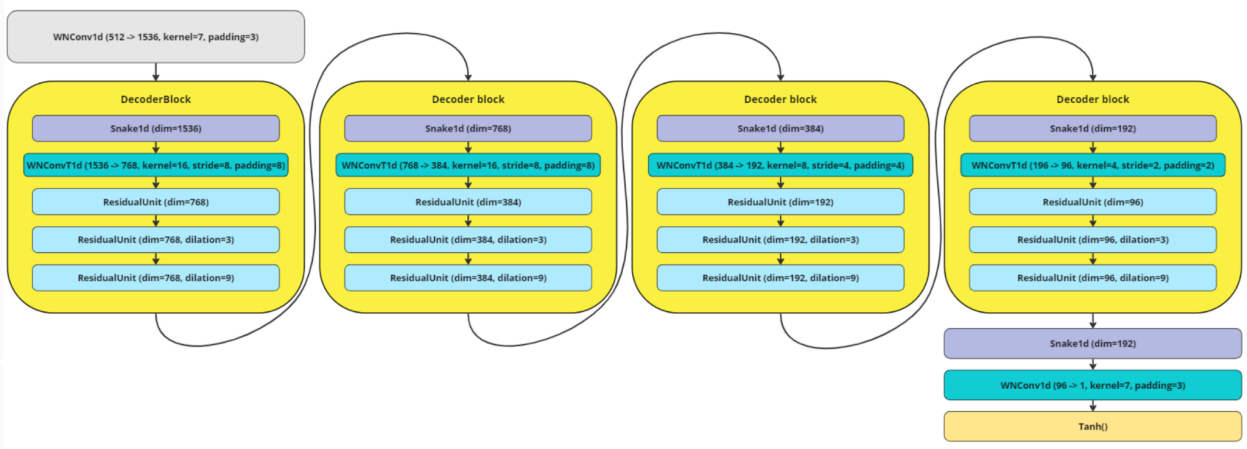


Рисунок 2.3 – Архітектура декодера

2. 5. Джерела даних

Модель була натренована на датасеті MUSDB18 [9] та датасеті Common Voice від Mozilla [10]. MUSDB18 це набір даних зі 150 музичних треків (тривалістю ~10 годин) різного жанру та ізольованими один від одного записами музичних інструментів, басу та вокалу, а також міксом аккомпонементу різних інструментів та вокалу одночасно. Всі сигнали в цьому датасеті є стереофонічними і закодовані на частоті 44.1 кГц. Що ж до Common Voice, то це відкрита багатомовна база данних голосів. Кожен запис у наборі даних складається з унікального MP3-файлу та відповідного текстового файлу. Наразі набір даних складається з 20 409 годин та 124 мовами.

Оглянемо кожен набір даних детально і почнемо з MUSDB18. Датасет MUSDB18 містить два основні підмножини: тренувальний набір, що складається зі 100 треків, та тестовий набір, що включає 50 треків. Для побудови та тестування моделей рекомендовано використовувати тренувальний набір, тоді як тестування може бути проведене на обох підмножинах. Джерела, з яких було отримано дані для MUSDB18, є різноманітними: 100 треків походять із набору даних DSD100, що в свою чергу базується на бібліотеці "Mixing Secrets' Free Multitrack Download

Library", 46 треків взято з MedleyDB, 2 треки були люб’язно надані компанією Native Instruments, 2 треки представлені канадською рок-групою The Easton Ellises в рамках конкурсу heise stems remix.

Файли MUSDB18 закодовані у форматі Native Instruments stems (.mp4), який складається з п'яти стереофонічних потоків, кожен з яких закодований у форматі AAC з бітрейтом 256 кбіт/с. Для кожного файлу мікс відповідає сумі всіх сигналів. Через окреме кодування міксу у форматі AAC можливі незначні відмінності між сумою всіх джерел і самим міксом, однак ці відмінності не впливають на оцінку ефективності. Альтернативно, MUSDB18-HQ пропонує необроблені WAV-файли, що дозволяють моделям передбачати сигнали з високою пропускною здатністю до 22 кГц. Однак для порівняння з існуючою літературою з розділення джерел рекомендується використовувати стандартний набір MUSDB18.

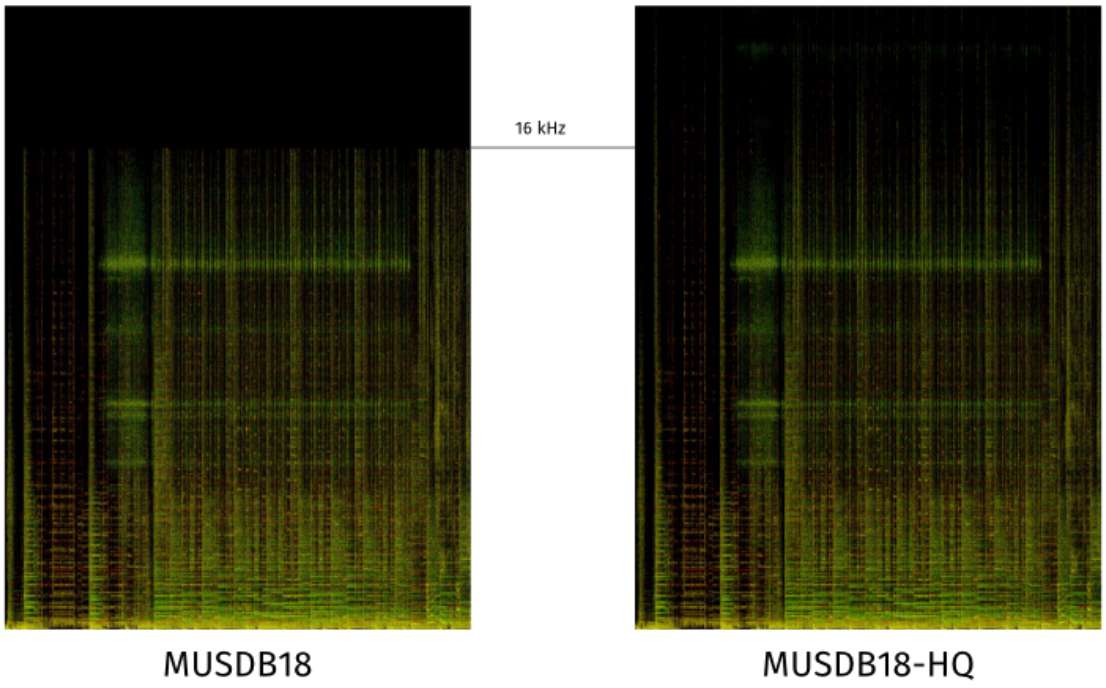


Рисунок 2.4 – Різниця між спектограмами аудіофайлу з MUSDB18 та MUSDB18-HQ

Файли MUSDB18 закодовані у форматі Native Instruments stems (.mp4), який складається з п'яти стереофонічних потоків, кожен з яких закодований у форматі

AAC з бітрейтом 256 кбіт/с. Для кожного файлу мікс відповідає сумі всіх сигналів. Через окреме кодування міксу у форматі AAC можливі незначні відмінності між сумою всіх джерел і самим міксом, однак ці відмінності не впливають на оцінку ефективності. Альтернативно, MUSDB18-HQ пропонує необроблені WAV-файли, що дозволяють моделям передбачати сигнали з високою пропускною здатністю до 22 кГц. Однак для порівняння з існуючою літературою з розділення джерел рекомендується використовувати стандартний набір MUSDB18.

Common Voice — це проєкт краудсорсингу, започаткований Mozilla, з метою створення вільної бази даних для програмного забезпечення розпізнавання мови. Проєкт підтримується волонтерами, які записують зразки речень за допомогою мікрофона та переглядають записи інших користувачів. Розшифровані речення зберігаються у базі даних голосів. Метою Common Voice є надання різноманітних зразків голосу.

Датасет маж 30 000 записів. Ілюстрацію розподілу за віком та статтю можна побачити на рисунку 2.5. З цих записів яких лише 11 194 мали вказаний вік, 11 233 — стать, а 9 827 — акцент. Ці дані показали, що набір даних є незбалансованим, і для подальшого використання необхідно виконати попередню обробку.

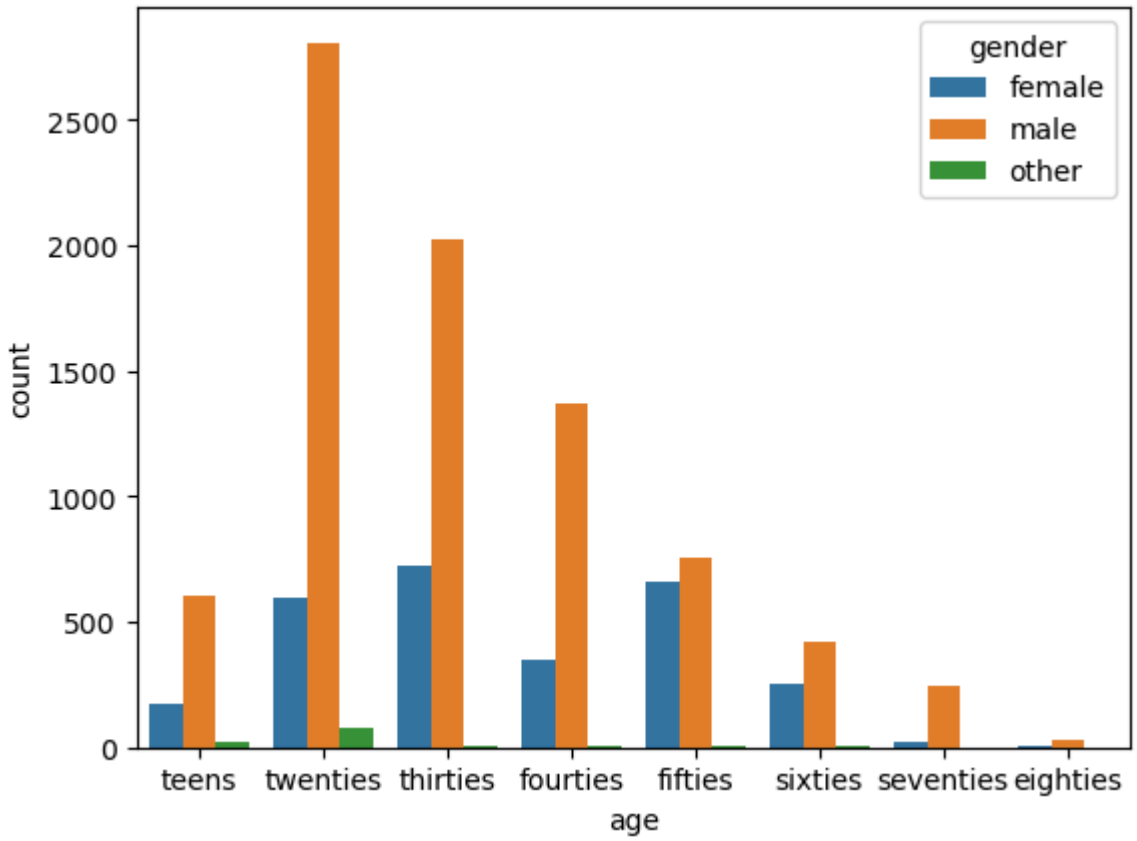


Рисунок 2.5 – Розподіл даних за віком та статтю в датасеті Common Voice

На етапі очищення даних було видалено записи з відсутніми значеннями. Було видалено стовпці, які не сприяють навчанню моделі, і перевірено типи даних атрибутів з їхнім відповідним коригуванням. Далі дані були збалансовані, забезпечивши наявність не більше 5000 точок даних для кожного вікового діапазону.

Після очищення даних було виконано екстракцію ознак з аудіофайлів з використанням бібліотеки librosa, рисунок 2.6. Були витягнуті спектральний центроїд, ширина смуги, частота зрізу та мелчастотні кепстральні коефіцієнти. Отримані ознаки використовувалися для створення нового датафрейму, який потім був масштабований і підготовлений для навчання моделі.

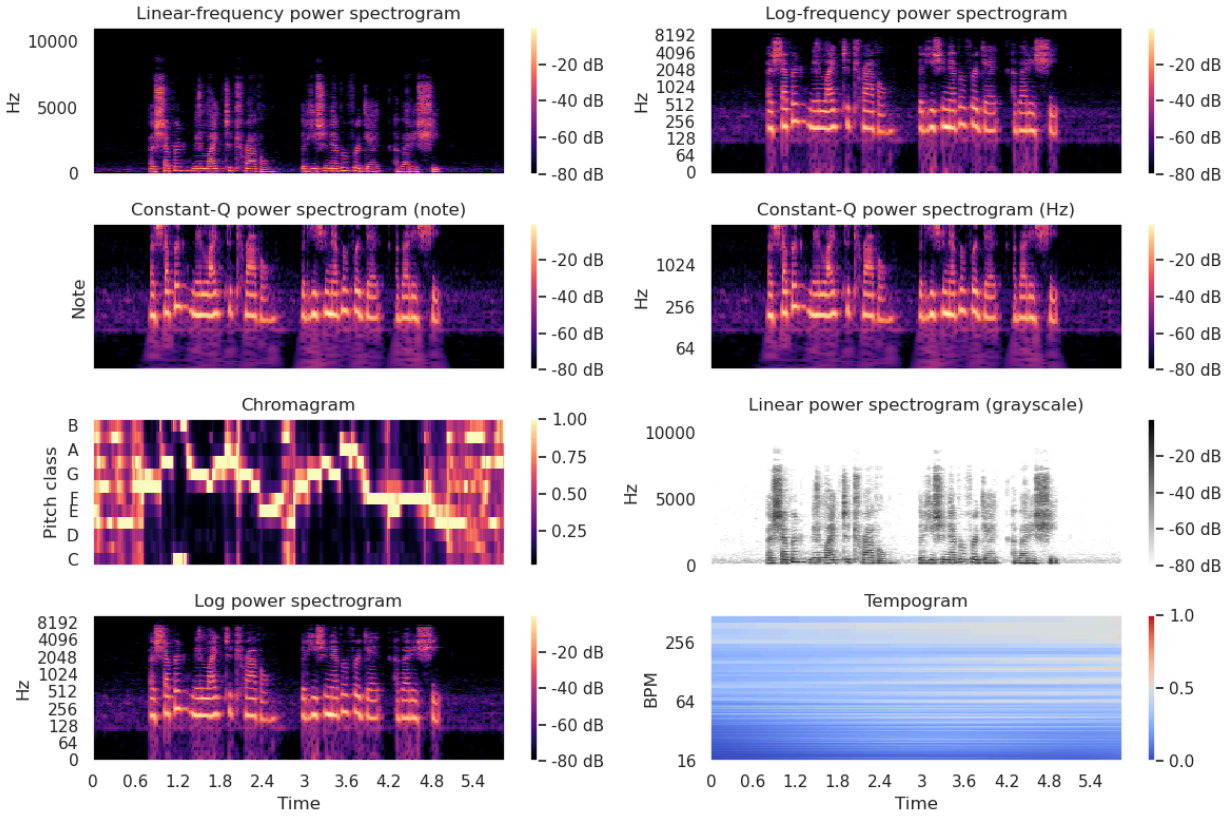


Рисунок 2.6 – Ілюстрація витягнутих ознак з аудіофайлу з датасету Common Voice

2. 6. Тренування моделі машинного навчання

Для тренування всі аудіо перекодовано до 44 кГц. Під час тренування витягуються короткі фрагменти з кожного аудіофайлу і відбувається їх нормалізуємо їх до -24 дБ віностно повної шкали (LUFS). Для збільшення даних для тренування застосовується випадковий рівномірний зсув фази.

В мовних даних Common Voice частота дискретизації є 8-16 кГц. При навчанні моделі на різних частотах дискретизації, виявилось, що отримана модель часто не відтворювала дані вище певної частоти. Ця порогова частота відповідала середній справжній частоті дискретизації набору даних. Для виправлення цього, набір даних розділяється на джерела даних, для яких є гарантія, що є повносмуговими - вони підтверджені, що містять енергію на частотах до бажаної частоти Найквіста (22,05 кГц) кодека - і джерела даних, де немає гарантій щодо того, яка буде максимальна частота. При вибірковому зразкуванні партій даних, іде перевірка, що вибрано елемент повної смуги. Нарешті, балансування даних,

щоб у кожній партії було рівну кількість елементів з кожного домену: мова, музика та звуки навколишнього середовища.

Для моделі використовуються мультиперіодичний дискримінатор і складний STFT дискримінатор. Для першого використовуються періоди [2, 3, 5, 7, 11], а для другого — довжини вікон віконного перетворення Фур'є [2048, 1024, 512], з кроком, що дорівнює 1/4 довжини вікна. Для розділення смуг STFT використовуються межі [0.0, 0.1, 0.25, 0.5, 0.75, 1.0]. Для втрат відновлення використовується відстань між лог-мел спектрограмами з довжинами вікон [32, 64, 128, 256, 512, 1024, 2048], з відповідною кількістю мелів для кожного з [5, 10, 20, 40, 80, 160, 320]. Довжина непересічної частини дорівнює 1/4 довжини вікна. Використовуються втрати узгодження ознак і кодової книги.

Для дослідження впливу різних факторів на модель, кожну модель навчали з розміром батч пакета 16 протягом 250 тисяч ітерацій. Це займає приблизно 30 годин на одному GPU. Для фінальної моделі навчання проводиться з розміром пакета 72 протягом 400 тисяч ітерацій. Тривалість уривків для навчання складає 0.38 секунд. Використовується оптимізатор AdamW з швидкістю навчання $1 \cdot 10^{-4}$, $\beta 1 = 0.79$, та $\beta 2 = 0.91$, як для генератора, так і для дискримінатора. Швидкість навчання зменшується на кожному кроці з $\gamma = 0.999994$.

Для підбору оптимальної конфігурації моделі та параметрів тренування використовувалися чотири об'єктивні метрики для порівняння одної моделі з іншою:

1. Відстань між логарифмічними мел-спектрограмами відновлених та вихідних хвильових форм

2. Відстань між логарифмічними спектрограмами амплітуд відновлених та вихідних хвильових форм. Ця метрика краще відображає точність відновлення високих частот порівняно з відстанню між логарифмічними мел-спектрограмами.

3. Відстань між хвильовими формами, подібна до співвідношення сигнал/шум, з модифікаціями, що роблять її інваріантною до масштабних відмінностей. У поєднанні зі спектральними метриками, вказує на якість фазового відновлення аудіо.

4. Ефективність бітрейту, що обчислюється як сума ентропії (у бітах) кожної кодової книги при застосуванні до великого тестового набору, поділена на кількість бітів для всіх кодових книг. Для ефективного використання бітрейту цей показник має наближатися до 100%, а нижчі відсотки вказують на недовикористання бітрейт.

Під час дослідження оптимальних параметрів було виявлено, що варіювання розмірності декодера впливає на продуктивність: менші моделі мали гірші показники. Однак модель з розмірністю декодера 1024 мала подібну продуктивність до базової моделі. Найбільший вплив мало заміщення функції активації Relu на активацію Snake, що призвело до значного покращення метрики відстані між хвильовими формами та інших показників. Було встановлено, що періодичний індуктивний ухил активації Snake корисний для генерації звукових хвиль. У фінальній моделі використовується розмірність декодера в 1536 та функція активації Snake.

При дослідженні впливу різних дискримінаторів на фінальний результат, виявлено, що мультичастотний STFT дискримінатор незначно покращує показники, за винятком показника відстані між хвильовими формами, де він трохи перевершує інші. Однак, при аналізі спектрограм згенерованих хвиль було виявлено, що мультичастотний дискримінатор зменшує артефакти аліасингу високих частот, які виникають через згорткові шари декодера. Аліасинг артефакти мають невеликий вплив на об'єктивні метрики, тому мультичастотний дискримінатор так і залишився для тренування.

Було встановлено, що змагальні втрати критичні для якості вихідного аудіо та ефективності бітрейту. Навчання лише з використанням втрат відновлення призводить до значного зниження ефективності бітрейту з 99% до 62% і зниження відстані між хвильовими формами з 9.12 до 1.07. Інші метрики, що вимірюють спектральну відстань, залишаються відносно незмінними, проте аудіо з цієї моделі містить багато артефактів, включаючи гул, через невміння відновлювати фазу. Заміна мультиперіодичного дискримінатора на одномасштабний дискримінатор, призвела до гірших показників, тому мультиперіодичний дискримінатор також

залишається.

Розмірність кодової книги значно впливає на ефективність бітрейту і, відповідно, якість відновлення. Встановлено, що оптимальною розмірністю є 8, тоді як занадто низька або висока розмірність, наприклад 2 або 512 призводить до значного погіршення метрик і зниження ефективності бітрейту.

Використання експоненційного ковзного середнього як методу навчання кодової книги, призводить до гірших метрик. Також це ускладнює реалізацію через необхідність ініціалізації k-середніх та випадкових перезапусків. Тому залишається простіший метод пошуку кодової книги. Встановлено, що вплив випадкового виключення для квантування значно впливає на кількісні метрики. Значення для випадкового виключення в 0.0 призводить до поганого відновлення з меншими кодовими книгами. Тому для остаточної моделі використовується занчення випадкового виключення блоків квантування в 0.5 (50% того, що блок квантування не прийматиме участь у навчанні), що забезпечує гарний баланс між якістю аудіо при повному бітрейті та нижчих бітрейтах.

Якщо не балансувати вибірку даних, то це призводить до погіршення метрик. Емпірично виявлено, що без збалансованої вибірки модель генерує хвилі з максимальною частотою близько 18 кГц. Це відповідає максимальній частоті, збереженій різними алгоритмами стиснення аудіо, такими як MPEG. Збалансована вибірка дозволяє моделі відновлювати як повнодіапазонне аудіо 44 кГц, так і аудіо з обмеженим діапазоном.

2. 7. Результати

Для оцінки ефективності запропонованої архітектури аудіокодеку з використанням новітніх технологій стиснення аудіо, було вибрано відомий датасет GTZAN [11]. Датасет складається з 1000 аудіофайлів, розподілених між 10 музичними жанрами, кожен тривалістю 30 секунд. Використання датасету GTZAN дозволяє оцінити ефективність кодеку в умовах різноманітності музичного матеріалу та специфіки звукозаписів, рисунок 2.7.

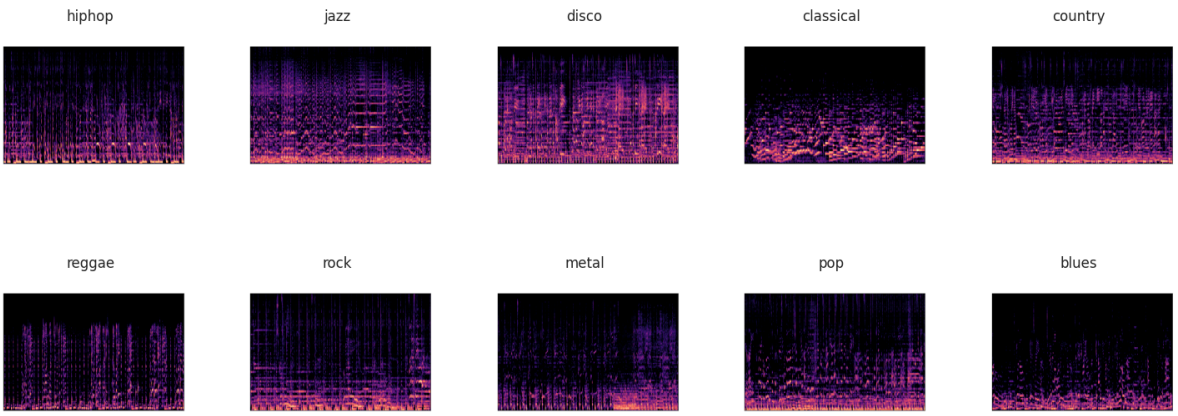


Рисунок 2.7 – Спектрограми для музики різних жанрів із датасету GTZAN

Цей датасет є хорошим інструменом для перевірки універсальності і адаптивності нових методів стиснення до різних жанрів музики.

Також було використано Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS) [12] датасет. який містить аудіофайли мовлення (16-бітні, 48 кГц .wav). Повний набір даних, що включає мову і спів, аудіо та відео (24.8 ГБ). RAVDESS містить 1440 файлів, які складаються з 60 різних сцен і 24 акторів, що сумарно дає 1440 файлів. У наборі даних представлено 24 професійні актори (12 жінок і 12 чоловіків). Мовні емоції включають вираження спокою, радості, смутку, гніву, страху, здивування та відрази. Кожна емоція виконується на двох рівнях інтенсивності (нормальному та сильному), з додатковим нейтральним вираженням.

Для оцінки методів стиснення використовувались відстані між логарифмічними мел-спектрограмами відновлених та вихідних хвильових форм та ViSQOL [13] метрика. ViSQOL (Virtual Speech Quality Objective Listener) є інструментом для об'єктивної оцінки якості мовлення. Він базується на порівнянні спектрограм еталонного і тестового сигналів за допомогою індексу подібності нейрограм (NSIM). Модель була розроблена з урахуванням специфічних проблем передачі голосу через Інтернет-протокол (VoIP), таких як джиттер, зсув частоти і затримки відтворення. ViSQOL працює в 5 етапів:

1. Попередня обробка: Потужність деградованого сигналу $y(t)$ масштабується до рівня еталонного сигналу $x(t)$. Після, еталонний і деградований сигнали перетворюються у спектрограми за допомогою короткочасного перетворення

Фур'є (STFT). Використовуються критичні смуги частот між 150 і 3400 Гц для вузькосмугових тестів та додаткові смуги до 8000 Гц для широкосмугових.

2. Тимчасове вирівнювання: Еталонний сигнал сегментується на патчі розміром 30 кадрів (480 мс) за частотними смугами. Використання NSIM для вирівнювання патчів: NSIM використовується для вирівнювання патчів еталонного сигналу з відповідними патчами тестового сигналу.

3. Прогнозування деформації: Для кожного еталонного патчу створюються альтернативні версії, які є на 1% і 5% довші і коротші від оригіналу. NSIM використовується для порівняння тестових патчів з оригінальними і альтернативними еталонними патчами. Якщо альтернатива має вищий індекс подібності, вона використовується для подальшого порівняння.

4. Порівняння подібності: Спектрограми розглядаються як зображення, а NSIM використовується для оцінки подібності між еталонною і деградованою спектрограмами. NSIM вимірює подібність за трьома факторами – яскравістю, контрастом і структурою.

5. Співвідношення подібності та оцінки якості: За допомогою сигмоїдальної функції NSIM перетворюється у оцінку якості MOS-LQOn (Mean Opinion Score – Listening Quality Objective for narrowband signals), яка має діапазон від 1 до 5.

Стиснення аудіофайлів виконувалось з використанням двох методів: Стиснення за домопогою моделі машинного навчання представленій у даній роботі та Opus — відомий стандарт стиснення аудіо.

За результатами тестів на датасеті GTZAN, рисунок 2.8, середній коефіцієнт стиснення для 1000 аудіофайлів, використовуючи нейронні мережі, склав 24.98, що свідчить про високу ефективність нової методології стиснення у порівнянні з традиційними підходами. В той же час, стандарт Opus показав коефіцієнт стиснення 14.56, що є менш ефективним на даному наборі даних. При цьому для метрики відстані відстані між логарифмічними мел-спектрограмами та оцінка ViSQOL у нейронного аудіокодека та Opus є майже на одному рівні.

Codec	Copression ratio	Mel distance ↓	ViSQOL ↑
ML Audio Encoder	24.98	0.93	4.18
Opus	14.56	0.88	4.15

Рисунок 2.8 – Результати тесту аудіокодеків на датасеті GTZAN

Під час тестування на датасеті RAVDESS, який містить 1440 аудіофайлів, було отримано наступні результати, рисунок 2.9. Використання нейронних мереж для стиснення аудіофайлів показало середній коефіцієнт стиснення 45.18. Це значення значно перевищує коефіцієнт стиснення стандартного кодека Opus, який становив 16.34, що свідчить про вищу ефективність нового методу стиснення. При цьому для обох метрик - відстані між логарифмічними мел-спектрограмами (Mel distance) та оцінки ViSQOL - результати були наступними: для нейронного аудіокодека відстань становила 0.77, а оцінка ViSQOL 4.61; для Opus ці показники були 1.15 та 4.12 відповідно.

Codec	Compression ratio	Mel distance ↓	ViSQOL ↑
ML Audio Encoder	45.18	0.77	4.61
Opus	16.34	1.15	4.12

Рисунок 2.9 – Результати тесту аудіокодеків на датасеті RVESS

Це означає, що новий метод забезпечує не лише більш ефективне стиснення, але й зберігає кращу якість аудіо за суб'єктивними та об'єктивними критеріями при роботі з голосовими данними. Це пов’язано з тим, що датасет для тренування з голосом був більшим за датасет з музикою. Це пояснює і більший коєфіцієнт стиснення та більшу точність відтворення голосових аудіофайлів.

Під кінець, в порівнянні з SoundSteram, удосконалена модель потребує значно менше оперативної пам’яті для обробки звуку. Для обробки одного аудіофайлу SoundStream необхідно більше 11 Гб оперативної пам’яті, тоді як для удосконаленої моделі потрібно всього приблизно 800 Мб.

ВИСНОВКИ

У цій роботі представлено алгоритм стиснення аудіо на основі нейронних мереж, який демонструє високу ефективність стиснення та відтворення аудіо без значних втрат якості на різних типах аудіоданих. Використання датасету GTZAN, що включає 1000 аудіофайлів з десяти музичних жанрів, дозволило об'єктивно оцінити переваги запропонованого методу. Середній коефіцієнт стиснення досягнуто на рівні 24.97, що є значно кращим за показник стандарту Opus (14.56).Тести на датасеті RAVDESS також показали значно вищий коефіцієнт стиснення для нейронного аудіокодека (45.18) порівняно з Opus (16.34).

Додатково, порівняння з попередніми системами, такими як SoundStream, показало значне зниження вимог до оперативної пам'яті з 11 ГБ до 800 МБ на аудіофайл, що робить технологію більш доступною для використання на пристроях з обмеженими ресурсами.

Основні інновації, внесені в дану розробку, охоплюють застосування функції активації Snake для покращення обробки періодичних аудіосигналів та розробку вдосконалених дискримінаторів віконного перетворення Фур'є для ефективнішого усунення артефактів і збільшення якості відтворення.

Для оцінки якості стиснення використовувались метрики відстані між логарифмічними мел-спектрограмами та ViSQOL. Новий метод забезпечує не лише більш ефективне стиснення, але й зберігає кращу якість аудіо за суб'єктивними та об'єктивними критеріями, що є важливим для різних аудіо доменів, включаючи музику та голосові дані.

Подальші дослідження мають на меті вдосконалення алгоритму для забезпечення ще вищої точності відтворення складних аудіосигналів та розширення його адаптивності до різних аудіо доменів. Також є можливість зменшити ресурсоємність, зменшивши кількість конволюційних блоків енкодера-декодера, що з одного боку зменшить якість та коєфіцієнт стисення але з іншого стане ще більш доступним для використання в портативних радіотехнічних пристроях.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Нейромережі GAN у створенні нових моделей. URL: http://jnas.nbuv.gov.ua/j-pdf/Kzms_2019_18_16.pdf (дата звернення: 23.06.2024).
2. SoundStream: an end-to-end neural audio codec. URL: https://arxiv.org/pdf/2107.03312 (дата звернення: 23.06.2024).
3. What is residual vector quantization? URL: https://www.assemblyai.com/blog/what-is-residual-vector-quantization/ (дата звернення: 23.06.2024).
4. Neural networks fail to learn periodic functions and how to fix It. URL: https://browse.arxiv.org/pdf/2006.08195v1 (дата звернення: 24.06.2024).
5. Generating diverse high-fidelity images. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/5f8e2fa1718d1bbcadf1cd9c7a54fb8c-Paper.pdf (дата звернення: 25.06.2024).
6. Chunked autoregressive gan for conditional waveform synthesis. URL: https://openreview.net/pdf?id=v3aeIsY_vVX (дата звернення: 24.06.2024).
7. Short-time fourier transforms. URL: https://course.ece.cmu.edu/~ece491/lectures/L25/STFT_Notes_ADSP.pdf (дата звернення: 23.06.2024).
8. Multi-band melgan: faster waveform generation for high-quality text-to-speech. URL: https://arxiv.org/pdf/2005.05106 (дата звернення: 26.06.2024).
9. Датасет MUSDB18 . URL: https://sigsep.github.io/datasets/musdb.html (дата звернення: 25.06.2024).
10. Датасет Common Voice. URL: https://commonvoice.mozilla.org/en/datasets (дата звернення: 25.06.2024).
11. Датасет GTZAN. URL: https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification (дата звернення: 26.06.2024).

12. Датасет RAVDESS. URL: https://www.kaggle.com/datasets/uwrfkaggler/ravdess-emotional-speech-audio/data (дата звернення: 31.06.2024).
13. ViSQOL: an objective speech quality model. URL: https://asmp-eurasipjournals.springeropen.com/articles/10.1186/s13636-015-0054-9 (дата звернення: 31.06.2024).

ДОДАТОК А
ПРОГРАМА ДЛЯ ПОБУДОВИ НЕЙРОННОЇ МЕРЕЖІ, ЩО СТИСКАЄ АУДІО

Програма для компіляції нейронної мережі, що стискає аудіо написана мовою Python використовуючи бібліотеку tensorflow.

```
import math
from typing import Union, List
from dataclasses import dataclass
from pathlib import Path

import numpy as np
from einops import rearrange

import tensorflow as tf
from tensorflow.keras.layers import Conv1D, Conv1DTranspose, Layer
from tensorflow_addons.layers import WeightNormalization

from audiotools import AudioSignal
from audiotools.ml import BaseModel

SUPPORTED_VERSIONS = ["1.0.0"]

@dataclass
class EncodedFile:
    codes: np.ndarray

    # Metadata
    chunk_length: int
    original_length: int
    input_db: float
    channels: int
    sample_rate: int
    padding: bool
    encoder_version: str

    def save(self, path):
        artifacts = {
            "codes": self.codes.astype(np.uint16),
            "metadata": {
                "input_db": np.float32(self.input_db),
                "original_length": self.original_length,
                "sample_rate": self.sample_rate,
                "chunk_length": self.chunk_length,
                "channels": self.channels,
                "padding": self.padding,
                "encoder_version": SUPPORTED_VERSIONS[-1],
            },
        }
        path = Path(path).with_suffix(".encoded")
        with open(path, "wb") as f:
            np.save(f, artifacts)
```

```
        return path

    @classmethod
    def load(cls, path):
        artifacts = np.load(path, allow_pickle=True)[()]
        codes = artifacts["codes"].astype(int)
        if artifacts["metadata"].get("encoder_version", None) not in SUPPORTED_VERSIONS:
            raise RuntimeError(
                f"Given file {path} can't be loaded with this version of descript-audio-codec."
            )
        return cls(codes=codes, **artifacts["metadata"])


class CodecMixin:
    @property
    def padding(self):
        if not hasattr(self, "_padding"):
            self._padding = True
        return self._padding

    @padding.setter
    def padding(self, value):
        assert isinstance(value, bool)

        layers = [
            l for l in self.layers if isinstance(l, (tf.keras.layers.Conv1D, tf.keras.layers.Conv2DTranspose))
        ]

        for layer in layers:
            if value:
                if hasattr(layer, "original_padding"):
                    layer.padding = layer.original_padding
            else:
                layer.original_padding = layer.padding
                layer.padding = 'valid'

        self._padding = value

    def get_delay(self):
        # Any number works here, delay is invariant to input length
        l_out = self.get_output_length(0)
        L = l_out

        layers = []
        for layer in self.layers:
            if isinstance(layer, (tf.keras.layers.Conv1D, tf.keras.layers.Conv2DTranspose)):
                layers.append(layer)

        for layer in reversed(layers):
            d = layer.dilation_rate[0]
            k = layer.kernel_size[0]
            s = layer.strides[0]

            if isinstance(layer, tf.keras.layers.Conv2DTranspose):
                L = ((L - d * (k - 1) - 1) / s) + 1
            elif isinstance(layer, tf.keras.layers.Conv1D):
                L = (L - 1) * s + d * (k - 1) + 1
```

```
        L = math.ceil(L)

    l_in = L

    return (l_in - l_out) // 2

def get_output_length(self, input_length):
    L = input_length
    # Calculate output length
    for layer in self.layers:
        if isinstance(layer, (tf.keras.layers.Conv1D, tf.keras.layers.Conv2DTranspose)):
            d = layer.dilation_rate[0]
            k = layer.kernel_size[0]
            s = layer.strides[0]

            if isinstance(layer, tf.keras.layers.Conv1D):
                L = ((L - d * (k - 1) - 1) / s) + 1
            elif isinstance(layer, tf.keras.layers.Conv2DTranspose):
                L = (L - 1) * s + d * (k - 1) + 1

            L = math.floor(L)
    return L

@tf.function
def compress(
    self,
    audio_path_or_signal: Union[str, Path, AudioSignal],
    win_duration: float = 1.0,
    verbose: bool = False,
    normalize_db: float = -16,
    n_quantizers: int = None,
) -> EncodedFile:
    audio_signal = audio_path_or_signal
    if isinstance(audio_signal, (str, Path)):
        audio_signal = AudioSignal.load_from_file_with_ffmpeg(str(audio_signal))

    self.eval()
    original_padding = self.padding
    original_device = audio_signal.device

    audio_signal = audio_signal.clone()
    original_sr = audio_signal.sample_rate

    resample_fn = audio_signal.resample
    loudness_fn = audio_signal.loudness

    # If audio is > 10 minutes long, use the ffmpeg versions
    if audio_signal.signal_duration >= 10 * 60 * 60:
        resample_fn = audio_signal.ffmpeg_resample
        loudness_fn = audio_signal.ffmpeg_loudness

    original_length = audio_signal.signal_length
    resample_fn(self.sample_rate)
    input_db = loudness_fn()

    if normalize_db is not None:
```

```
        audio_signal.normalize(normalize_db)
    audio_signal.ensure_max_of_audio()

    nb, nac, nt = audio_signal.audio_data.shape
    audio_signal.audio_data = audio_signal.audio_data.reshape(nb * nac, 1, nt)
    win_duration = (
        audio_signal.signal_duration if win_duration is None else win_duration
    )

    if audio_signal.signal_duration <= win_duration:
        # Unchunked compression (used if signal length < win duration)
        self.padding = True
        n_samples = nt
        hop = nt
    else:
        # Chunked inference
        self.padding = False
        # Zero-pad signal on either side by the delay
        audio_signal.zero_pad(self.delay, self.delay)
        n_samples = int(win_duration * self.sample_rate)
        # Round n_samples to nearest hop length multiple
        n_samples = int(math.ceil(n_samples / self.hop_length) * self.hop_length)
        hop = self.get_output_length(n_samples)

    codes = []
    range_fn = range if not verbose else tqdm.trange

    for i in range_fn(0, nt, hop):
        x = audio_signal[..., i : i + n_samples]
        x = x.zero_pad(0, max(0, n_samples - x.shape[-1]))

        audio_data = x.audio_data.numpy()
        audio_data = self.preprocess(audio_data, self.sample_rate)
        _, c, _, _, _ = self.encode(audio_data, n_quantizers)
        codes.append(c)
        chunk_length = c.shape[-1]

    codes = np.concatenate(codes, axis=-1)

    encoded_file = EncodedFile(
        codes=codes,
        chunk_length=chunk_length,
        original_length=original_length,
        input_db=input_db,
        channels=nac,
        sample_rate=original_sr,
        padding=self.padding,
        encoder_version=SUPPORTED_VERSIONS[-1],
    )

    if n_quantizers is not None:
        codes = codes[:, :n_quantizers, :]

    self.padding = original_padding
    return encoded_file

@tf.function
```

```
    def decompress(
        self,
        obj: Union[str, Path, EncodedFile],
        verbose: bool = False,
    ) -> AudioSignal:
        self.eval()
        if isinstance(obj, (str, Path)):
            obj = EncodedFile.load(obj)

        original_padding = self.padding
        self.padding = obj.padding

        range_fn = range if not verbose else tqdm.trange
        codes = obj.codes
        chunk_length = obj.chunk_length
        recons = []

        for i in range_fn(0, codes.shape[-1], chunk_length):
            c = codes[..., i : i + chunk_length]
            z = self.quantizer.from_codes(c)[0]
            r = self.decode(z)
            recons.append(r)

        recons = np.concatenate(recons, axis=-1)
        recons = AudioSignal(recons, self.sample_rate)

        resample_fn = recons.resample
        loudness_fn = recons.loudness

        # If audio is > 10 minutes long, use the ffmpeg versions
        if recons.signal_duration >= 10 * 60 * 60:
            resample_fn = recons.ffmpeg_resample
            loudness_fn = recons.ffmpeg_loudness

        recons.normalize(obj.input_db)
        resample_fn(obj.sample_rate)
        recons = recons[..., : obj.original_length]
        loudness_fn()
        recons.audio_data = recons.audio_data.reshape(
            -1, obj.channels, obj.original_length
        )

        self.padding = original_padding
        return recons

def WNConv1d(filters, kernel_size, **kwargs):
    return WeightNormalization(Conv1D(filters, kernel_size, **kwargs))

def WNConvTranspose1d(filters, kernel_size, **kwargs):
    return WeightNormalization(Conv1DTranspose(filters, kernel_size, **kwargs))

@tf.function
def snake(x, alpha):
    x = tf.reshape(x, [x.shape[0], x.shape[1], -1])
    x = x + tf.math.reciprocal(alpha + 1e-9) * tf.math.pow(tf.math.sin(alpha * x), 2)
    x = tf.reshape(x, [-1, x.shape[1], x.shape[2]])
    return x
```

```
class Snake1d(Layer):
    def __init__(self, channels):
        super().__init__()
        self.alpha = self.add_weight(name='alpha', shape=(1, channels, 1), initializer='ones', trainable=True)

    def call(self, x):
        return snake(x, self.alpha)


class WNConv1D(tf.keras.layers.Layer):
    def __init__(self, filters, kernel_size, **kwargs):
        super().__init__()
        self.conv = Conv1D(filters, kernel_size, **kwargs)
        self.weight_norm = tf.keras.layers.LayerNormalization(axis=-1)

    def call(self, inputs):
        return self.weight_norm(self.conv(inputs))

class VectorQuantize(tf.keras.layers.Layer):
    def __init__(self, input_dim: int, codebook_size: int, codebook_dim: int):
        super().__init__()
        self.codebook_size = codebook_size
        self.codebook_dim = codebook_dim

        self.in_proj = WNConv1D(codebook_dim, kernel_size=1)
        self.out_proj = WNConv1D(input_dim, kernel_size=1)
        self.codebook = self.add_weight(shape=(codebook_size, codebook_dim), initializer='random_normal',
trainable=True)

    def call(self, z):
        z_e = self.in_proj(z)
        z_q, indices = self.decode_latents(z_e)

        commitment_loss = tf.reduce_mean(tf.losses.mse(z_e, tf.stop_gradient(z_q)), axis=[1, 2])
        codebook_loss = tf.reduce_mean(tf.losses.mse(z_q, tf.stop_gradient(z_e)), axis=[1, 2])

        z_q = z_e + tf.stop_gradient(z_q - z_e)

        z_q = self.out_proj(z_q)

        return z_q, commitment_loss, codebook_loss, indices, z_e

    def embed_code(self, embed_id):
        return tf.nn.embedding_lookup(self.codebook, embed_id)

    def decode_code(self, embed_id):
        return tf.transpose(self.embed_code(embed_id), perm=[0, 2, 1])

    def decode_latents(self, latents):
        encodings = rearrange(latents, "b d t -> (b t) d")
        codebook = self.codebook

        encodings = tf.nn.l2_normalize(encodings, axis=-1)
        codebook = tf.nn.l2_normalize(codebook, axis=-1)

        dist = tf.reduce_sum(tf.square(encodings), axis=1, keepdims=True) - 2 * tf.matmul(encodings, codebook,
```

```
transpose_b=True) + tf.reduce_sum(tf.square(codebook), axis=1, keepdims=True)
        indices = rearrange(tf.argmax(-dist, axis=1), "(b t) -> b t", b=latents.shape[0])
        z_q = self.decode_code(indices)
        return z_q, indices

class ResidualVectorQuantize(tf.keras.layers.Layer):
    def __init__(self, input_dim: int = 512, n_codebooks: int = 9, codebook_size: int = 1024, codebook_dim:
Union[int, list] = 8, quantizer_dropout: float = 0.0):
        super().__init__()
        if isinstance(codebook_dim, int):
            codebook_dim = [codebook_dim for _ in range(n_codebooks)]

        self.n_codebooks = n_codebooks
        self.codebook_dim = codebook_dim
        self.codebook_size = codebook_size

        self.quantizers = [VectorQuantize(input_dim, codebook_size, codebook_dim[i]) for i in
range(n_codebooks)]
        self.quantizer_dropout = quantizer_dropout

    def call(self, z, n_quantizers: int = None):
        z_q = 0
        residual = z
        commitment_loss = 0
        codebook_loss = 0

        codebook_indices = []
        latents = []

        if n_quantizers is None:
            n_quantizers = self.n_codebooks
        if self.training:
            n_quantizers = tf.ones((tf.shape(z)[0],)) * (self.n_codebooks + 1)
            dropout = tf.random.uniform((tf.shape(z)[0],), minval=1, maxval=self.n_codebooks + 1,
dtype=tf.int32)
            n_dropout = int(tf.shape(z)[0] * self.quantizer_dropout)
            n_quantizers = tf.tensor_scatter_nd_update(n_quantizers, tf.reshape(tf.argsort(dropout)[:n_dropout],
(-1, 1)), dropout[:n_dropout])
            n_quantizers = tf.cast(n_quantizers, tf.int32)

        for i, quantizer in enumerate(self.quantizers):
            if not self.training and i >= n_quantizers:
                break

            z_q_i, commitment_loss_i, codebook_loss_i, indices_i, z_e_i = quantizer(residual)

            mask = tf.cast(tf.range(z.shape[0])[:, None] < n_quantizers[:, None], z_q_i.dtype)
            z_q += z_q_i * mask[:, None, None]
            residual -= z_q_i

            commitment_loss += tf.reduce_mean(commitment_loss_i * mask)
            codebook_loss += tf.reduce_mean(codebook_loss_i * mask)

            codebook_indices.append(indices_i)
            latents.append(z_e_i)

        codes = tf.stack(codebook_indices, axis=1)
```

```
        latents = tf.concat(latents, axis=1)

        return z_q, codes, latents, commitment_loss, codebook_loss

    def from_codes(self, codes):
        z_q = 0.0
        z_p = []
        n_codebooks = codes.shape[1]
        for i in range(n_codebooks):
            z_p_i = self.quantizers[i].decode_code(codes[:, i, :])
            z_p.append(z_p_i)

            z_q_i = self.quantizers[i].out_proj(z_p_i)
            z_q += z_q_i
        return z_q, tf.concat(z_p, axis=1), codes

    def from_latents(self, latents):
        z_q = 0
        z_p = []
        codes = []
        dims = np.cumsum([0] + [q.codebook_dim for q in self.quantizers])

        n_codebooks = np.where(dims <= latents.shape[1])[0].max()
        for i in range(n_codebooks):
            j, k = dims[i], dims[i + 1]
            z_p_i, codes_i = self.quantizers[i].decode_latents(latents[:, j:k, :])
            z_p.append(z_p_i)
            codes.append(codes_i)

            z_q_i = self.quantizers[i].out_proj(z_p_i)
            z_q += z_q_i

        return z_q, tf.concat(z_p, axis=1), tf.stack(codes, axis=1)

def init_weights(m):
    if isinstance(m, tf.keras.layers.Conv1D):
        m.kernel_initializer = tf.keras.initializers.TruncatedNormal(stddev=0.02)
        m.bias_initializer = tf.keras.initializers.Zeros()

class ResidualUnit(tf.keras.layers.Layer):
    def __init__(self, dim: int = 16, dilation: int = 1):
        super().__init__()
        pad = ((7 - 1) * dilation) // 2
        self.block = tf.keras.Sequential([
            Snake1d(dim),
            WNConv1d(dim, 7, dilation_rate=dilation, padding='same'),
            Snake1d(dim),
            WNConv1d(dim, 1)
        ])

    def call(self, x):
        y = self.block(x)
        pad = (x.shape[-1] - y.shape[-1]) // 2
        if pad > 0:
            x = x[..., pad:-pad]
        return x + y
```

```
class EncoderBlock(tf.keras.layers.Layer):
    def __init__(self, dim: int = 16, stride: int = 1):
        super().__init__()
        self.block = tf.keras.Sequential([
            ResidualUnit(dim // 2, dilation=1),
            ResidualUnit(dim // 2, dilation=3),
            ResidualUnit(dim // 2, dilation=9),
            Snake1d(dim // 2),
            WNConv1d(dim // 2, 2 * stride, strides=stride, padding='same')
        ])

    def call(self, x):
        return self.block(x)

class Encoder(tf.keras.layers.Layer):
    def __init__(self, d_model: int = 64, strides: list = [2, 4, 8, 8], d_latent: int = 64):
        super().__init__()
        self.block = [WNConv1d(1, d_model, 7, padding='same')]
        for stride in strides:
            d_model *= 2
            self.block += [EncoderBlock(d_model, stride=stride)]
        self.block += [Snake1d(d_model), WNConv1d(d_model, d_latent, 3, padding='same')]
        self.block = tf.keras.Sequential(self.block)
        self.enc_dim = d_model

    def call(self, x):
        return self.block(x)

class DecoderBlock(tf.keras.layers.Layer):
    def __init__(self, input_dim: int = 16, output_dim: int = 8, stride: int = 1):
        super().__init__()
        self.block = tf.keras.Sequential([
            Snake1d(input_dim),
            WNConvTranspose1d(input_dim, 2 * stride, strides=stride, padding='same'),
            ResidualUnit(output_dim, dilation=1),
            ResidualUnit(output_dim, dilation=3),
            ResidualUnit(output_dim, dilation=9)
        ])

    def call(self, x):
        return self.block(x)

class Decoder(tf.keras.layers.Layer):
    def __init__(self, input_channel, channels, rates, d_out: int = 1):
        super().__init__()
        layers = [WNConv1d(input_channel, channels, 7, padding='same')]
        for i, stride in enumerate(rates):
            input_dim = channels // 2**i
            output_dim = channels // 2 ** (i + 1)
            layers += [DecoderBlock(input_dim, output_dim, stride)]
        layers += [Snake1d(output_dim), WNConv1d(output_dim, d_out, 7, padding='same'),
tf.keras.layers.Activation('tanh')]
        self.model = tf.keras.Sequential(layers)

    def call(self, x):
        return self.model(x)
```

```
class MLAudioEncoder(BaseModel, CodecMixin):
    def __init__(self, encoder_dim: int = 64, encoder_rates: List[int] = [2, 4, 8, 8], latent_dim: int = None,
decoder_dim: int = 1536, decoder_rates: List[int] = [8, 8, 4, 2], n_codebooks: int = 9, codebook_size: int =
1024, codebook_dim: Union[int, list] = 8, quantizer_dropout: bool = False, sample_rate: int = 44100):
        super().__init__()
        self.encoder_dim = encoder_dim
        self.encoder_rates = encoder_rates
        self.decoder_dim = decoder_dim
        self.decoder_rates = decoder_rates
        self.sample_rate = sample_rate

        if latent_dim is None:
            latent_dim = encoder_dim * (2 ** len(encoder_rates))

        self.latent_dim = latent_dim
        self.hop_length = np.prod(encoder_rates)
        self.encoder = Encoder(encoder_dim, encoder_rates, latent_dim)

        self.n_codebooks = n_codebooks
        self.codebook_size = codebook_size
        self.codebook_dim = codebook_dim
        self.quantizer = ResidualVectorQuantize(input_dim=latent_dim, n_codebooks=n_codebooks,
codebook_size=codebook_size, codebook_dim=codebook_dim, quantizer_dropout=quantizer_dropout)

        self.decoder = Decoder(latent_dim, decoder_dim, decoder_rates)
        self.sample_rate = sample_rate
        self.apply(init_weights)
        self.delay = self.get_delay()

    def preprocess(self, audio_data, sample_rate):
        if sample_rate is None:
            sample_rate = self.sample_rate
        assert sample_rate == self.sample_rate
        length = audio_data.shape[-1]
        right_pad = math.ceil(length / self.hop_length) * self.hop_length - length
        audio_data = tf.pad(audio_data, [[0, 0], [0, 0], [0, right_pad]], mode='CONSTANT')
        return audio_data

    def encode(self, audio_data: tf.Tensor, n_quantizers: int = None):
        z = self.encoder(audio_data)
        z, codes, latents, commitment_loss, codebook_loss = self.quantizer(z, n_quantizers)
        return z, codes, latents, commitment_loss, codebook_loss

    def decode(self, z: tf.Tensor):
        return self.decoder(z)

    def call(self, audio_data: tf.Tensor, sample_rate: int = None, n_quantizers: int = None):
        length = audio_data.shape[-1]
        audio_data = self.preprocess(audio_data, sample_rate)
        z, codes, latents, commitment_loss, codebook_loss = self.encode(audio_data, n_quantizers)
        x = self.decode(z)
        return {
            "audio": x[..., :length],
            "z": z,
            "codes": codes,
            "latents": latents,
            "vq/commitment_loss": commitment_loss,
```

```
            "vq/codebook_loss": codebook_loss,
        }

if __name__ == "__main__":
    import numpy as np
    from functools import partial

    model = MLAudioEncoder()
    model.build((None, 1, 88200 * 2))
    model.summary()

    length = 88200 * 2
    x = tf.random.normal((1, 1, length))
    x = tf.Variable(x)

    with tf.GradientTape() as tape:
        out = model(x)["audio"]
    print("Input shape:", x.shape)
    print("Output shape:", out.shape)

    grad = tape.gradient(out, x)
    gradmap = tf.reduce_sum(tf.cast(grad != 0, tf.int32), axis=0)
    rf = tf.reduce_sum(tf.cast(gradmap != 0, tf.int32))

    print(f"Receptive field: {rf.numpy()}")

    x = AudioSignal(tf.random.normal((1, 1, 44100 * 60)), 44100)
    compressed = model.compress(x, verbose=True)
    decompressed = model.decompress(compressed, verbose=True)
```

ДОДАТОК Б
ПРОГРАМА ДЛЯ НАВЧАННЯ НЕЙРОННОЇ МЕРЕЖІ, ЩО СТИСКАЄ АУДІО

Програма для тренування нейронної мережі, що стискає аудіо написана мовою Python використовуючи бібліотеку tensorflow.

```
import typing
import os
import sys
import warnings
from dataclasses import dataclass
from pathlib import Path
from typing import List

import argbind
import numpy as np
import tensorflow as tf
import tensorflow_addons as tfa
from einops import rearrange

from audiotools import AudioSignal, ml, STFTParams
from audiotools.core import util
from audiotools.data import transforms
from audiotools.data.datasets import AudioDataset, AudioLoader, ConcatDataset
from audiotools.ml.decorators import timer, Tracker, when

def WNConv1d(filters, kernel_size, strides=1, padding='valid', act=True):
    conv = tf.keras.Sequential([tf.keras.layers.Conv1D(filters, kernel_size, strides=strides, padding=padding),
tf.keras.layers.LeakyReLU(0.1)]) if act else tf.keras.layers.Conv1D(filters, kernel_size, strides=strides,
padding=padding)
    return tf.keras.layers.LayerNormalization(axis=-1)(conv)

def WNConv2d(filters, kernel_size, strides=(1, 1), padding='valid', act=True):
    conv = tf.keras.Sequential([tf.keras.layers.Conv2D(filters, kernel_size, strides=strides, padding=padding),
tf.keras.layers.LeakyReLU(0.1)]) if act else tf.keras.layers.Conv2D(filters, kernel_size, strides=strides,
padding=padding)
    return tf.keras.layers.LayerNormalization(axis=-1)(conv)

class MPD(tf.keras.layers.Layer):
    def __init__(self, period):
        super().__init__()
        self.period = period
        self.convs = [WNConv2d(32, (5, 1), (3, 1), padding='same'), WNConv2d(128, (5, 1), (3, 1),
padding='same'), WNConv2d(512, (5, 1), (3, 1), padding='same'), WNConv2d(1024, (5, 1), (3, 1),
padding='same'), WNConv2d(1024, (5, 1), padding='same')]
        self.conv_post = WNConv2d(1, (3, 1), padding='same', act=False)

    def pad_to_period(self, x):
        t = x.shape[-1]
        x = tf.pad(x, [[0, 0], [0, 0], [0, self.period - t % self.period]], mode="REFLECT")
        return x
```

```
    def call(self, x):
        fmap = []
        x = self.pad_to_period(x)
        x = rearrange(x, "b c (l p) -> b c l p", p=self.period)
        for layer in self.convs:
            x = layer(x)
            fmap.append(x)
        x = self.conv_post(x)
        fmap.append(x)
        return fmap

class MSD(tf.keras.layers.Layer):
    def __init__(self, rate=1, sample_rate=44100):
        super().__init__()
        self.convs = [WNConv1d(16, 15, 1, padding='same'), WNConv1d(64, 41, 4, padding='same', act=True,
groups=4), WNConv1d(256, 41, 4, padding='same', act=True, groups=16), WNConv1d(1024, 41, 4,
padding='same', act=True, groups=64), WNConv1d(1024, 41, 4, padding='same', act=True, groups=256),
WNConv1d(1024, 5, 1, padding='same')]
        self.conv_post = WNConv1d(1, 3, 1, padding='same', act=False)
        self.sample_rate = sample_rate
        self.rate = rate

    def call(self, x):
        x = AudioSignal(x, self.sample_rate)
        x.resample(self.sample_rate // self.rate)
        x = x.audio_data
        fmap = []
        for l in self.convs:
            x = l(x)
            fmap.append(x)
        x = self.conv_post(x)
        fmap.append(x)
        return fmap

BANDS = [(0.0, 0.1), (0.1, 0.25), (0.25, 0.5), (0.5, 0.75), (0.75, 1.0)]

class MRD(tf.keras.layers.Layer):
    def __init__(self, window_length, hop_factor=0.25, sample_rate=44100, bands=BANDS):
        super().__init__()
        self.window_length = window_length
        self.hop_factor = hop_factor
        self.sample_rate = sample_rate
        self.stft_params = STFTParams(window_length=window_length, hop_length=int(window_length *
hop_factor), match_stride=True)
        n_fft = window_length // 2 + 1
        self.bands = [(int(b[0] * n_fft), int(b[1] * n_fft)) for b in bands]
        ch = 32
        convs = lambda: [WNConv2d(ch, (3, 9), (1, 1), padding='same'), WNConv2d(ch, (3, 9), (1, 2),
padding='same'), WNConv2d(ch, (3, 9), (1, 2), padding='same'), WNConv2d(ch, (3, 9), (1, 2),
padding='same'), WNConv2d(ch, (3, 3), (1, 1), padding='same')]
        self.band_convs = [convs() for _ in range(len(self.bands))]
        self.conv_post = WNConv2d(1, (3, 3), (1, 1), padding='same', act=False)

    def spectrogram(self, x):
        x = AudioSignal(x, self.sample_rate, stft_params=self.stft_params)
        x = tf.signal.stft(x, frame_length=self.window_length, frame_step=int(self.window_length *
```

```
self.hop_factor))
        x = tf.signal.stft(x)
        x = tf.cast(x, tf.complex64)
        x = rearrange(x, "b 1 f t c -> (b 1) c t f")
        x_bands = [x[..., b[0]:b[1]] for b in self.bands]
        return x_bands

    def call(self, x):
        x_bands = self.spectrogram(x)
        fmap = []
        x = []
        for band, stack in zip(x_bands, self.band_convs):
            for layer in stack:
                band = layer(band)
                fmap.append(band)
            x.append(band)
        x = tf.concat(x, axis=-1)
        x = self.conv_post(x)
        fmap.append(x)
        return fmap

class Discriminator(ml.BaseModel):
    def __init__(self, rates=[], periods=[2, 3, 5, 7, 11], fft_sizes=[2048, 1024, 512], sample_rate=44100,
bands=BANDS):
        super().__init__()
        discs = []
        discs += [MPD(p) for p in periods]
        discs += [MSD(r, sample_rate=sample_rate) for r in rates]
        discs += [MRD(f, sample_rate=sample_rate, bands=bands) for f in fft_sizes]
        self.discriminators = discs

    def preprocess(self, y):
        y = y - tf.reduce_mean(y, axis=-1, keepdims=True)
        y = 0.8 * y / (tf.reduce_max(tf.abs(y), axis=-1, keepdims=True) + 1e-9)
        return y

    def call(self, x):
        x = self.preprocess(x)
        fmaps = [d(x) for d in self.discriminators]
        return fmaps

class L1Loss(tf.keras.losses.Loss):
    def __init__(self, attribute: str = "audio_data", weight: float = 1.0, **kwargs):
        self.attribute = attribute
        self.weight = weight
        super().__init__(**kwargs)

    def call(self, x: AudioSignal, y: AudioSignal):
        if isinstance(x, AudioSignal):
            x = getattr(x, self.attribute)
            y = getattr(y, self.attribute)
        return tf.reduce_mean(tf.abs(x - y))


class SISDRLoss(tf.keras.losses.Loss):
    def __init__(self, scaling: int = True, reduction: str = "mean", zero_mean: int = True, clip_min: int = None,
weight: float = 1.0):
```

```
        self.scaling = scaling
        self.reduction = reduction
        self.zero_mean = zero_mean
        self.clip_min = clip_min
        self.weight = weight
        super().__init__()

    def call(self, x: AudioSignal, y: AudioSignal):
        eps = 1e-8
        references = x.audio_data if isinstance(x, AudioSignal) else x
        estimates = y.audio_data if isinstance(y, AudioSignal) else y

        nb = tf.shape(references)[0]
        references = tf.transpose(tf.reshape(references, (nb, -1, 1)), (0, 2, 1))
        estimates = tf.transpose(tf.reshape(estimates, (nb, -1, 1)), (0, 2, 1))

        mean_reference = tf.reduce_mean(references, axis=1, keepdims=True) if self.zero_mean else 0
        mean_estimate = tf.reduce_mean(estimates, axis=1, keepdims=True) if self.zero_mean else 0

        _references = references - mean_reference
        _estimates = estimates - mean_estimate

        references_projection = tf.reduce_sum(tf.square(_references), axis=-2) + eps
        references_on_estimates = tf.reduce_sum(_estimates * _references, axis=-2) + eps

        scale = (references_on_estimates / references_projection)[:, tf.newaxis] if self.scaling else 1

        e_true = scale * _references
        e_res = _estimates - e_true

        signal = tf.reduce_sum(tf.square(e_true), axis=1)
        noise = tf.reduce_sum(tf.square(e_res), axis=1)
        sdr = -10 * tf.math.log(signal / noise + eps) / tf.math.log(10.0)

        if self.clip_min is not None:
            sdr = tf.maximum(sdr, self.clip_min)

        if self.reduction == "mean":
            sdr = tf.reduce_mean(sdr)
        elif self.reduction == "sum":
            sdr = tf.reduce_sum(sdr)
        return sdr


class MultiScaleSTFTLoss(tf.keras.losses.Loss):
    def __init__(self, window_lengths: List[int] = [2048, 512], loss_fn: typing.Callable =
tf.keras.losses.MeanAbsoluteError(), clamp_eps: float = 1e-5, mag_weight: float = 1.0, log_weight: float = 1.0,
pow: float = 2.0, weight: float = 1.0, match_stride: bool = False, window_type: str = None):
        super().__init__()
        self.stft_params = [STFTParams(window_length=w, hop_length=w // 4, match_stride=match_stride,
window_type=window_type) for w in window_lengths]
        self.loss_fn = loss_fn
        self.log_weight = log_weight
        self.mag_weight = mag_weight
        self.clamp_eps = clamp_eps
        self.weight = weight
        self.pow = pow
```

```
    def call(self, x: AudioSignal, y: AudioSignal):
        loss = 0.0
        for s in self.stft_params:
            x.stft(s.window_length, s.hop_length, s.window_type)
            y.stft(s.window_length, s.hop_length, s.window_type)
            loss += self.log_weight * self.loss_fn(tf.math.log(tf.maximum(x.magnitude, self.clamp_eps) **
self.pow), tf.math.log(tf.maximum(y.magnitude, self.clamp_eps) ** self.pow))
            loss += self.mag_weight * self.loss_fn(x.magnitude, y.magnitude)
        return loss


class MelSpectrogramLoss(tf.keras.losses.Loss):
    def __init__(self, n_mels: List[int] = [150, 80], window_lengths: List[int] = [2048, 512], loss_fn:
typing.Callable = tf.keras.losses.MeanAbsoluteError(), clamp_eps: float = 1e-5, mag_weight: float = 1.0,
log_weight: float = 1.0, pow: float = 2.0, weight: float = 1.0, match_stride: bool = False, mel_fmin: List[float]
= [0.0, 0.0], mel_fmax: List[float] = [None, None], window_type: str = None):
        super().__init__()
        self.stft_params = [STFTParams(window_length=w, hop_length=w // 4, match_stride=match_stride,
window_type=window_type) for w in window_lengths]
        self.n_mels = n_mels
        self.loss_fn = loss_fn
        self.clamp_eps = clamp_eps
        self.log_weight = log_weight
        self.mag_weight = mag_weight
        self.weight = weight
        self.mel_fmin = mel_fmin
        self.mel_fmax = mel_fmax
        self.pow = pow

    def call(self, x: AudioSignal, y: AudioSignal):
        loss = 0.0
        for n_mels, fmin, fmax, s in zip(self.n_mels, self.mel_fmin, self.mel_fmax, self.stft_params):
            x_mels = x.mel_spectrogram(n_mels, mel_fmin=fmin, mel_fmax=fmax,
window_length=s.window_length, hop_length=s.hop_length, window_type=s.window_type)
            y_mels = y.mel_spectrogram(n_mels, mel_fmin=fmin, mel_fmax=fmax,
window_length=s.window_length, hop_length=s.hop_length, window_type=s.window_type)
            loss += self.log_weight * self.loss_fn(tf.math.log(tf.maximum(x_mels, self.clamp_eps) ** self.pow),
tf.math.log(tf.maximum(y_mels, self.clamp_eps) ** self.pow))
            loss += self.mag_weight * self.loss_fn(x_mels, y_mels)
        return loss


class GANLoss(tf.keras.losses.Loss):
    def __init__(self, discriminator):
        super().__init__()
        self.discriminator = discriminator

    def call(self, fake, real):
        d_fake = self.discriminator(fake.audio_data)
        d_real = self.discriminator(real.audio_data)
        return d_fake, d_real

    def discriminator_loss(self, fake, real):
        d_fake, d_real = self.call(fake, real)
        loss_d = 0
        for x_fake, x_real in zip(d_fake, d_real):
```

```
            loss_d += tf.reduce_mean(tf.square(x_fake[-1]))
            loss_d += tf.reduce_mean(tf.square(1 - x_real[-1]))
        return loss_d

    def generator_loss(self, fake, real):
        d_fake, d_real = self.call(fake, real)
        loss_g = 0
        for x_fake in d_fake:
            loss_g += tf.reduce_mean(tf.square(1 - x_fake[-1]))
        loss_feature = 0
        for i in range(len(d_fake)):
            for j in range(len(d_fake[i]) - 1):
                loss_feature += tf.reduce_mean(tf.abs(d_fake[i][j] - d_real[i][j]))
        return loss_g, loss_feature

warnings.filterwarnings("ignore", category=UserWarning)

AdamW = argbind.bind(tf.keras.optimizers.Adam, "generator", "discriminator")
Accelerator = argbind.bind(ml.Accelerator, without_prefix=True)

@argbind.bind("generator", "discriminator")
def ExponentialLR(optimizer, gamma: float = 1.0):
    return tf.keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=optimizer.learning_rate, decay_steps=100000, decay_rate=gamma, staircase=True
    )

MLAudioEncoder = argbind.bind(MLAudioEncoder)
Discriminator = argbind.bind(Discriminator)

AudioDataset = argbind.bind(AudioDataset, "train", "val")
AudioLoader = argbind.bind(AudioLoader, "train", "val")

filter_fn = lambda fn: hasattr(fn, "transform") and fn.__qualname__ not in [
    "BaseTransform",
    "Compose",
    "Choose",
]
tfm = argbind.bind_module(transforms, "train", "val", filter_fn=filter_fn)

filter_fn = lambda fn: hasattr(fn, "forward") and "Loss" in fn.__name__
losses = argbind.bind_module(loss, filter_fn=filter_fn)

def get_infinite_loader(dataloader):
    while True:
        for batch in dataloader:
            yield batch

@argbind.bind("train", "val")
def build_transform(augment_prob: float = 1.0, preprocess: list = ["Identity"], augment: list = ["Identity"],
postprocess: list = ["Identity"]):
    to_tfm = lambda l: [getattr(tfm, x)() for x in l]
    preprocess = transforms.Compose(*to_tfm(preprocess), name="preprocess")
    augment = transforms.Compose(*to_tfm(augment), name="augment", prob=augment_prob)
    postprocess = transforms.Compose(*to_tfm(postprocess), name="postprocess")
    transform = transforms.Compose(preprocess, augment, postprocess)
    return transform
```

```
@argbind.bind("train", "val", "test")
def build_dataset(sample_rate: int, folders: dict = None):
    datasets = []
    for _, v in folders.items():
        loader = AudioLoader(sources=v)
        transform = build_transform()
        dataset = AudioDataset(loader, sample_rate, transform=transform)
        datasets.append(dataset)

    dataset = ConcatDataset(datasets)
    dataset.transform = transform
    return dataset

@dataclass
class State:
    generator: MLAudioEncoder
    optimizer_g: tf.keras.optimizers.Adam
    scheduler_g: tf.keras.optimizers.schedules.ExponentialDecay

    discriminator: Discriminator
    optimizer_d: tf.keras.optimizers.Adam
    scheduler_d: tf.keras.optimizers.schedules.ExponentialDecay

    stft_loss: losses.MultiScaleSTFTLoss
    mel_loss: losses.MelSpectrogramLoss
    gan_loss: losses.GANLoss
    waveform_loss: losses.L1Loss

    train_data: AudioDataset
    val_data: AudioDataset

    tracker: Tracker

@argbind.bind(without_prefix=True)
def load(args, accel: ml.Accelerator, tracker: Tracker, save_path: str, resume: bool = False, tag: str = "latest",
load_weights: bool = False):
    generator, g_extra = None, {}
    discriminator, d_extra = None, {}

    if resume:
        kwargs = {"folder": f"{save_path}/{tag}", "map_location": "cpu", "package": not load_weights}
        tracker.print(f"Resuming from {str(Path('.').absolute())}/{kwargs['folder']}")
        if (Path(kwargs["folder"]) / "mlaudioencoder").exists():
            generator, g_extra = MLAudioEncoder.load_from_folder(**kwargs)
        if (Path(kwargs["folder"]) / "discriminator").exists():
            discriminator, d_extra = Discriminator.load_from_folder(**kwargs)

    generator = MLAudioEncoder() if generator is None else generator
    discriminator = Discriminator() if discriminator is None else discriminator

    tracker.print(generator)
    tracker.print(discriminator)

    generator = accel.prepare_model(generator)
    discriminator = accel.prepare_model(discriminator)

    with argbind.scope(args, "generator"):
```

```
        optimizer_g = AdamW(learning_rate=1e-4)
        scheduler_g = ExponentialLR(optimizer_g)
    with argbind.scope(args, "discriminator"):
        optimizer_d = AdamW(learning_rate=1e-4)
        scheduler_d = ExponentialLR(optimizer_d)

    if "optimizer.pth" in g_extra:
        optimizer_g.load_state_dict(g_extra["optimizer.pth"])
    if "scheduler.pth" in g_extra:
        scheduler_g.load_state_dict(g_extra["scheduler.pth"])
    if "tracker.pth" in g_extra:
        tracker.load_state_dict(g_extra["tracker.pth"])

    if "optimizer.pth" in d_extra:
        optimizer_d.load_state_dict(d_extra["optimizer.pth"])
    if "scheduler.pth" in d_extra:
        scheduler_d.load_state_dict(d_extra["scheduler.pth"])

    sample_rate = accel.unwrap(generator).sample_rate
    with argbind.scope(args, "train"):
        train_data = build_dataset(sample_rate)
    with argbind.scope(args, "val"):
        val_data = build_dataset(sample_rate)

    waveform_loss = losses.L1Loss()
    stft_loss = losses.MultiScaleSTFTLoss()
    mel_loss = losses.MelSpectrogramLoss()
    gan_loss = losses.GANLoss(discriminator)

    return State(
        generator=generator,
        optimizer_g=optimizer_g,
        scheduler_g=scheduler_g,
        discriminator=discriminator,
        optimizer_d=optimizer_d,
        scheduler_d=scheduler_d,
        waveform_loss=waveform_loss,
        stft_loss=stft_loss,
        mel_loss=mel_loss,
        gan_loss=gan_loss,
        tracker=tracker,
        train_data=train_data,
        val_data=val_data,
    )

@timer()
@tf.function
def val_loop(batch, state, accel):
    state.generator.eval()
    batch = util.prepare_batch(batch, accel.device)
    signal = state.val_data.transform(batch["signal"].clone(), **batch["transform_args"])

    out = state.generator(signal.audio_data, signal.sample_rate)
    recons = AudioSignal(out["audio"], signal.sample_rate)

    return {
        "loss": state.mel_loss(recons, signal),
```

```
        "mel/loss": state.mel_loss(recons, signal),
        "stft/loss": state.stft_loss(recons, signal),
        "waveform/loss": state.waveform_loss(recons, signal),
    }

@timer()
def train_loop(state, batch, accel, lambdas):
    state.generator.train()
    state.discriminator.train()
    output = {}

    batch = util.prepare_batch(batch, accel.device)
    with tf.no_grad():
        signal = state.train_data.transform(batch["signal"].clone(), **batch["transform_args"])

    with accel.autocast():
        out = state.generator(signal.audio_data, signal.sample_rate)
        recons = AudioSignal(out["audio"], signal.sample_rate)
        commitment_loss = out["vq/commitment_loss"]
        codebook_loss = out["vq/codebook_loss"]

    with accel.autocast():
        output["adv/disc_loss"] = state.gan_loss.discriminator_loss(recons, signal)

    state.optimizer_d.zero_grad()
    accel.backward(output["adv/disc_loss"])
    output["other/grad_norm_d"] = tf.clip_by_norm(state.discriminator.trainable_variables, 10.0)
    state.optimizer_d.apply_gradients(zip(state.optimizer_d.get_gradients(output["adv/disc_loss"],
state.discriminator.trainable_variables), state.discriminator.trainable_variables))
    state.scheduler_d.step()

    with accel.autocast():
        output["stft/loss"] = state.stft_loss(recons, signal)
        output["mel/loss"] = state.mel_loss(recons, signal)
        output["waveform/loss"] = state.waveform_loss(recons, signal)
        output["adv/gen_loss"], output["adv/feat_loss"] = state.gan_loss.generator_loss(recons, signal)
        output["vq/commitment_loss"] = commitment_loss
        output["vq/codebook_loss"] = codebook_loss
        output["loss"] = sum([v * output[k] for k, v in lambdas.items() if k in output])

    state.optimizer_g.zero_grad()
    accel.backward(output["loss"])
    output["other/grad_norm"] = tf.clip_by_norm(state.generator.trainable_variables, 1e3)
    state.optimizer_g.apply_gradients(zip(state.optimizer_g.get_gradients(output["loss"],
state.generator.trainable_variables), state.generator.trainable_variables))
    state.scheduler_g.step()
    accel.update()

    output["other/learning_rate"] = state.optimizer_g.learning_rate
    output["other/batch_size"] = signal.batch_size * accel.world_size

    return {k: v for k, v in sorted(output.items())}

def checkpoint(state, save_iters, save_path):
    metadata = {"logs": state.tracker.history}

    tags = ["latest"]
```

```
    state.tracker.print(f"Saving to {str(Path('.').absolute())}")
    if state.tracker.is_best("val", "mel/loss"):
        state.tracker.print(f"Best generator so far")
        tags.append("best")
    if state.tracker.step in save_iters:
        tags.append(f"{state.tracker.step // 1000}k")

    for tag in tags:
        generator_extra = {
            "optimizer.pth": state.optimizer_g.get_weights(),
            "scheduler.pth": state.scheduler_g.get_weights(),
            "tracker.pth": state.tracker.state_dict(),
            "metadata.pth": metadata,
        }
        accel.unwrap(state.generator).metadata = metadata
        accel.unwrap(state.generator).save_to_folder(f"{save_path}/{tag}", generator_extra)
        discriminator_extra = {
            "optimizer.pth": state.optimizer_d.get_weights(),
            "scheduler.pth": state.scheduler_d.get_weights(),
        }
        accel.unwrap(state.discriminator).save_to_folder(f"{save_path}/{tag}", discriminator_extra)

@tf.function
def save_samples(state, val_idx, logger):
    state.tracker.print("Saving audio samples")
    state.generator.eval()

    samples = [state.val_data[idx] for idx in val_idx]
    batch = state.val_data.collate(samples)
    batch = util.prepare_batch(batch, accel.device)
    signal = state.train_data.transform(batch["signal"].clone(), **batch["transform_args"])

    out = state.generator(signal.audio_data, signal.sample_rate)
    recons = AudioSignal(out["audio"], signal.sample_rate)

    audio_dict = {"recons": recons}
    if state.tracker.step == 0:
        audio_dict["signal"] = signal

    for k, v in audio_dict.items():
        for nb in range(v.batch_size):
            v[nb].cpu().write_audio_to_tb(f"{k}/sample_{nb}.wav", logger, state.tracker.step)

def validate(state, val_dataloader, accel):
    for batch in val_dataloader:
        output = val_loop(batch, state, accel)
    if hasattr(state.optimizer_g, "consolidate_state_dict"):
        state.optimizer_g.consolidate_state_dict()
        state.optimizer_d.consolidate_state_dict()
    return output

@argbind.bind(without_prefix=True)
def train(args, accel: ml.Accelerator, seed: int = 0, save_path: str = "ckpt", num_iters: int = 250000, save_iters:
list = [10000, 50000, 100000, 200000], sample_freq: int = 10000, valid_freq: int = 1000, batch_size: int = 12,
val_batch_size: int = 10, num_workers: int = 8, val_idx: list = [0, 1, 2, 3, 4, 5, 6, 7], lambdas: dict =
{"mel/loss": 100.0, "adv/feat_loss": 2.0, "adv/gen_loss": 1.0, "vq/commitment_loss": 0.25,
"vq/codebook_loss": 1.0}):
```

```
    util.seed(seed)
    Path(save_path).mkdir(exist_ok=True, parents=True)
    logger = ml.Logger(log_dir=f"{save_path}/logs") if accel.local_rank == 0 else None
    tracker = Tracker(writer=logger, log_file=f"{save_path}/log.txt", rank=accel.local_rank)

    state = load(args, accel, tracker, save_path)
    train_dataloader = accel.prepare_dataloader(
        state.train_data, start_idx=state.tracker.step * batch_size, num_workers=num_workers,
batch_size=batch_size, collate_fn=state.train_data.collate,
    )
    train_dataloader = get_infinite_loader(train_dataloader)
    val_dataloader = accel.prepare_dataloader(
        state.val_data, start_idx=0, num_workers=num_workers, batch_size=val_batch_size,
collate_fn=state.val_data.collate, persistent_workers=True if num_workers > 0 else False,
    )

    global train_loop, val_loop, validate, save_samples, checkpoint
    train_loop = tracker.log("train", "value", history=False)(
        tracker.track("train", num_iters, completed=state.tracker.step)(train_loop)
    )
    val_loop = tracker.track("val", len(val_dataloader))(val_loop)
    validate = tracker.log("val", "mean")(validate)

    save_samples = when(lambda: accel.local_rank == 0)(save_samples)
    checkpoint = when(lambda: accel.local_rank == 0)(checkpoint)

    with tracker.live:
        for tracker.step, batch in enumerate(train_dataloader, start=tracker.step):
            train_loop(state, batch, accel, lambdas)

            last_iter = tracker.step == num_iters - 1 if num_iters is not None else False
            if tracker.step % sample_freq == 0 or last_iter:
                save_samples(state, val_idx, logger)

            if tracker.step % valid_freq == 0 or last_iter:
                validate(state, val_dataloader, accel)
                checkpoint(state, save_iters, save_path)
                tracker.done("val", f"Iteration {tracker.step}")

            if last_iter:
                break

if __name__ == "__main__":
    args = argbind.parse_args()
    args["args.debug"] = int(os.getenv("LOCAL_RANK", 0)) == 0
    with argbind.scope(args):
        with Accelerator() as accel:
            if accel.local_rank != 0:
                sys.tracebacklimit = 0
            train(args, accel)
```