

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБЛЕННЯ ПРОГРАМИ ДЛЯ РОЗПІЗНАВАННЯ ТА
ВІЗУАЛІЗАЦІЇ ЗВУКІВ НАВКОЛИШНЬОГО СЕРЕДОВИЩА
DEVELOPMENT OF A SOFTWARE SYSTEM FOR ENVIRONMENTAL
SOUND RECOGNITION AND VISUALIZATION

Виконав(ла): студент(ка) 4 курсу, групи КНТ-112

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

КАТИХІНА А.М.

(ПРИЗВИЩЕ та ініціали)

Керівник ЛЕОЩЕНКО С.Д.

(ПРИЗВИЩЕ та ініціали)

Рецензент ПОЛЯКОВ М.О.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
 (код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

КАТИХІНОЇ Анастасії Миколаївни
 (ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розроблення програми для розпізнавання та візуалізації звуків навколишнього середовища. Development of a Software System for Environmental Sound Recognition and Visualization

керівник проєкту (роботи) д-р філософії, доцент, ЛЕОЩЕНКО Сергій Дмитрович
 (науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 07 ” квітня 2026 року № 139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз проблеми та постановка завдань дослідження. 2. Матеріали і методи. 3. Опис програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ЛЕОЩЕНКО С.Д., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області та системних обмежень фонового розпізнавання в ОС iOS.	1 тиждень	Розділ 1
3	Вибір інструментальних засобів. Збір, обробка та балансування датасету Freesound.	2 тиждень	Розділ 2
4	Розробка алгоритму DSP та побудови лог-мел-спектрограм	3 тиждень	Розділ 2
5	Проектування, навчання та оптимізація згорткової нейронної мережі	3-4 тиждень	Розділ 2
6	Розробка iOS-застосунку: DSP-модуль, інтеграція Core ML, інтерфейс користувача.	4-5 тижні	Розділ 3
7	Тестування та експериментальне дослідження програмного забезпечення в реальних умовах.	6 тиждень	Розділ 4
8	Оформлення пояснювальної записки та документів до неї.	6-7 тиждень	Додатки
9	Нормоконтроль та рецензування.	7 тиждень	
10	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Анастасія КАТИХІНА
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Сергій ЛЕОЩЕНКО
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 93 с., 10 табл., 19 рис., 3 дод., 11 джерел.

ВАДИ СЛУХУ, ВІЗУАЛІЗАЦІЯ ЗВУКУ, ЗГОРТКОВА НЕЙРОННА МЕРЕЖА, МАШИННЕ НАВЧАННЯ, МЕЛ-СПЕКТРОГРАМА, ОБРОБКА АУДІОСИГНАЛІВ, ФОНОВИЙ РЕЖИМ, CORE ML, iOS, SWIFT.

Об'єкт дослідження – процес програмної реалізації методів розпізнавання та візуалізації акустичних подій навколишнього середовища для асистивних технологій.

Предмет дослідження – методи та моделі машинного навчання, програмне забезпечення для моніторингу акустичного фону в реальному часі під управлінням ОС iOS.

Мета роботи – підвищення точності класифікації навколишніх звуків та оперативності сповіщення користувача в мобільному застосунку шляхом розроблення програмного засобу фонові обробки й аналізу аудіоданих для підтримки безпеки та соціальної адаптації людей з вадами слуху.

Матеріали, методи та технічні засоби: об'єктно-орієнтоване програмування, архітектура згорткових нейронних мереж (CNN) ResNet-style, методи цифрової обробки сигналів (DSP), фреймворки Core ML та Accelerate (vDSP), мови програмування Python та Swift, середовище розробки Xcode.

Результати. Розроблено програмну систему, що складається з модулів підготовки даних, навчання моделі та клієнтського iOS-застосунку. Реалізовано оригінальний підхід до обходу обмежень фонового режиму iOS шляхом перенесення обчислень з GPU на CPU, що дозволяє програмі функціонувати безперервно.

Висновки. Створено дієвий інструмент для розпізнавання 36 класів звуків з точністю 75,56%. Система забезпечує візуальне та тактильне

сповіщення користувача, сприяючи його безпеці в урбанізованому середовищі.

Галузь використання – соціальна сфера, реабілітація та підтримка людей з інвалідністю по слуху.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 93 pages, 10 tables, 19 figures, 3 appendixes, 11 sources.

ACCESSIBILITY, AUDIO SIGNAL PROCESSING, BACKGROUND MODE, CORE ML, CONVOLUTIONAL NEURAL NETWORKS, HEARING IMPAIRMENT, iOS, MACHINE LEARNING, MEL-SPECTROGRAM, SWIFT.

Object of study is the process of software implementation of methods for recognizing and visualizing environmental acoustic events for assistive technologies.

Subject of study are machine learning methods and models, software for real-time acoustic background monitoring running under the iOS operating system.

The purpose of the work is to improve the accuracy of environmental sound classification and the promptness of user notification in a mobile application by developing a software tool for background audio data processing and analysis to support the safety and social adaptation of people with hearing impairments.

Materials, methods, and tools: object-oriented programming, ResNet-style convolutional neural network (CNN) architecture, digital signal processing (DSP) methods, Core ML and Accelerate (vDSP) frameworks, Python and Swift programming languages, Xcode integrated development environment.

Results. A software system consisting of data preparation modules, model training scripts, and a client iOS application has been developed. An original approach to bypassing iOS background mode restrictions by migrating computations from GPU to CPU has been implemented, allowing the program to function continuously.

Conclusions. An effective tool for recognizing 36 classes of sounds with 75,56% accuracy has been created. The system provides visual and haptic notifications to the user, contributing to their safety in an urban environment.

The field of use is a social sphere, rehabilitation, and support for people with hearing disabilities.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	9
Вступ.....	10
1 Аналіз проблеми та постановка завдань дослідження.....	13
1.1 Соціальна значущість та актуальність розпізнавання звуків для людей з вадами слуху.....	13
1.2 Огляд існуючих технологічних рішень та їх порівняльний аналіз.....	14
1.3 Аналіз технічних обмежень платформи iOS та обґрунтування розробки.....	19
1.4 Постановка завдання розроблення системи.....	20
2 Матеріали і методи	21
2.1 Вибір та обґрунтування інструментальних засобів.....	21
2.2 Обґрунтування вибору та формування набору аудіоданих.....	22
2.3 Алгоритм попередньої обробки сигналу та побудови мел-спектрограм	24
2.4 Архітектура та навчання згорткової нейронної мережі.....	27
3 Опис програми	31
3.1 Архітектурна побудова та логічна структура застосунку	31
3.2 Реалізація модуля цифрової обробки сигналів та аналізу звуку.....	33
3.3 Програмна реалізація інтерфейсу користувача та системи сповіщень.....	34
3.4 Забезпечення надійності та обробка виняткових ситуацій	41
4 Експлуатація, тестування та експериментальне дослідження програми	43
4.1 Опис застосування та інструкція з експлуатації програми	43
4.2 Методика та результати тестування	45
4.3 Експериментальне дослідження програми в реальних умовах.....	46
4.4 Аналіз результатів та рекомендації щодо застосування.....	47
Висновки.....	49

Перелік джерел посилання	51
Додаток А Технічне завдання.....	53
Додаток Б Текст програми.....	60
Додаток В Слайди презентації	85

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
ARM	– Advanced RISC Machine;
CNN	– Convolutional Neural Network;
CPU	– Central Processing Unit;
DFT	– Discrete Fourier Transform;
DSP	– Digital Signal Processing;
FFT	– Fast Fourier Transform;
GPU	– Graphics Processing Unit;
iOS	– iPhone Operating System;
ML	– Machine Learning;
MP3	– MPEG-1 Audio Layer III;
MPS	– Metal Performance Shaders;
MVVM	– Model-View-ViewModel;
PCM	– Pulse Code Modulation;
ReLU	– Rectified Linear Unit;
ResNet	– Residual Network;
RMS	– Root Mean Square;
STFT	– Short-Time Fourier Transform;
URL	– Uniform Resource Locator;
vDSP	– vector Digital Signal Processing;
WAV	– Waveform Audio File Format;
ВООЗ	– Всесвітня організація охорони здоров'я;
МБ	– мегабайт;
ОС	– операційна система;
ШПФ	– швидке перетворення Фур'є.

ВСТУП

Актуальність теми. На сучасному етапі розвитку інформаційного суспільства інклюзивність та доступність цифрових технологій є пріоритетними напрямками розроблення програмного забезпечення. Для людей з вадами слуху сприйняття навколишнього акустичного середовища є серйозним викликом, що безпосередньо впливає на їхню безпеку. Неможливість вчасно почути сирену екстрених служб, сигнал автомобіля чи плач дитини створює ризики для життя та обмежує соціальну автономність особи.

Смартфони, обладнані потужними обчислювальними модулями та мікрофонами, мають потенціал стати персональними асистентами для візуалізації звуків. Проте розробка таких систем для платформи iOS стикається з жорсткими системними обмеженнями. Стандартні інструменти Apple (фреймворк SoundAnalysis з вбудованим класифікатором) не можуть працювати у фоновому режимі, оскільки ОС iOS забороняє виконання GPU-обчислень для неактивних застосунків. Це унеможливорює створення надійних систем постійного моніторингу. Таким чином, розроблення програмного забезпечення, що працює на центральному процесорі (CPU) та забезпечує безперервне фонове розпізнавання, є актуальним науково-практичним завданням.

Мета роботи – підвищення точності класифікації навколишніх звуків та оперативності сповіщення користувача в мобільному застосунку шляхом розроблення програмного засобу фонової обробки й аналізу аудіоданих для підтримки безпеки та соціальної адаптації людей з вадами слуху.

Завдання роботи. Для досягнення поставленої мети необхідно вирішити такі завдання:

- дослідити методи цифрової обробки сигналів та архітектури нейронних мереж для класифікації звуків;

- проаналізувати причини збоїв фонового виконання завдань машинного навчання в ОС iOS;
- сформувати збалансований набір даних на основі відкритої бібліотеки Freesound для навчання моделі;
- спроектувати та навчити згорткову нейронну мережу, адаптовану для виконання на CPU;
- розробити iOS-застосунок з функціями фонового моніторингу, візуалізації та тактильного відгуку;
- провести тестування точності та надійності розробленої системи в реальних умовах.

Об’єкт дослідження – процес програмної реалізації методів розпізнавання та візуалізації акустичних подій навколишнього середовища для асистивних технологій.

Предмет дослідження – методи та моделі машинного навчання, програмне забезпечення для моніторингу акустичного фону в реальному часі під управлінням ОС iOS.

Методи дослідження. У роботі використано методи цифрової обробки сигналів для побудови мел-спектрограм, методи глибокого навчання (CNN) для класифікації об’єктів, а також методи об’єктно-орієнтованого проєктування для реалізації мобільного застосунку.

Наукова новизна одержаних результатів полягає в удосконаленні архітектури системи розпізнавання звуків для мобільних пристроїв, що дозволяє забезпечити стабільну роботу в фоновому режимі ОС iOS. Запропоновано та реалізовано перехід на обчислювальний конвеєр на базі CPU, що усуває залежність від обмежених системних ресурсів GPU під час роботи застосунку в неактивному стані.

Практичне значення роботи полягає у створенні функціонального програмного продукту, який може бути використаний людьми з вадами слуху як допоміжний засіб безпеки. Реалізована система дозволяє розпізнавати 36

категорій звуків та персоналізувати список сповіщень відповідно до потреб конкретного користувача.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1 Соціальна значущість та актуальність розпізнавання звуків для людей з вадами слуху

За даними Всесвітньої організації охорони здоров'я (ВООЗ), понад 5% населення світу (приблизно 430 мільйонів людей) потребують реабілітації для вирішення проблеми втрати слуху [1]. Для таких осіб навколишнє акустичне середовище є джерелом постійної небезпеки. Нездатність вчасно ідентифікувати критичні звуки, такі як сирени екстрених служб або сигнали транспортних засобів, суттєво обмежує соціальну мобільність та створює загрозу життю.

Актуальність проблеми підтверджується дослідженнями у сфері безпеки: стандартні звукові сигнали тривоги використовують високочастотні звуки, які погано сприймаються людьми з вадами слуху, що суттєво підвищує ризик травм та загибелі під час надзвичайних ситуацій [2]. Це свідчить про нагальну потребу в технологічних рішеннях, здатних замінити або доповнити акустичне сприйняття середовища.

Сучасний підхід до вирішення цієї проблеми базується на концепції асистивних технологій (Assistive Technology) [3], де мобільний пристрій виступає в ролі «цифрового вуха», перетворюючи звукові коливання на візуальні або тактильні стимули. Смартфон є найбільш доступним і персоналізованим пристроєм для реалізації подібних рішень, оскільки він постійно знаходиться при користувачеві, оснащений мікрофоном, системою сповіщень та тактильного відгуку. Проте, незважаючи на розвиток штучного інтелекту, проблема безперервного та енергоефективного моніторингу звуків у реальному часі залишається відкритою для багатьох мобільних платформ. Аналіз існуючих рішень на провідних мобільних платформах дозволяє визначити їх можливості та обмеження.

1.2 Огляд існуючих технологічних рішень та їх порівняльний аналіз

На сучасному ринку асистивних технологій основними гравцями, що пропонують системні рішення для моніторингу звукового середовища, є компанії Google та Apple:

– Live Transcribe & Notification (Google): застосунок для ОС Android, що був представлений у 2019 році та орієнтований на людей з вадами слуху. Основною функцією застосунку є транскрибування мови в реальному часі, проте він також містить окремий модуль Sound Notifications – систему розпізнавання звуків навколишнього середовища. Цей модуль використовує машинне навчання на пристрої для ідентифікації 9 типів критичних звуків, зокрема сирен, пожежних і димових сигналізацій, плачу дитини, гавкоту собак, звуку біжучої води та сигналів побутової техніки. При виявленні звуку користувач отримує push-сповіщення, спалах камери або вібрацію. Додаток підтримує інтеграцію зі смарт-годинниками Wear OS, що дозволяє отримувати сповіщення на зап'ястя навіть під час сну. Додатково передбачено перегляд хронології подій за останні кілька годин. Принциповою перевагою є повністю офлайн-режим роботи та низьке енергоспоживання. Проте система обмежена фіксованим набором вбудованих класів, а розширення функціональності через сторонні застосунки не передбачено [4];

– Sound Recognition (Apple): вбудований інструмент розділу «Доступність» для iOS, що був введений у iOS 14. Функція безперервно прослуховує мікрофон у фоновому режимі та розпізнає 15 категорій звуків, організованих у групи: тривоги (пожежна сигналізація, сирена), тварини (собака, кіт), побутові звуки (дзвінок, стук у двері, мікрохвильова піч, вода), а також звуки людини (плач дитини). Вся обробка аудіо виконується безпосередньо на пристрої, без передачі даних на сервери Apple, що гарантує конфіденційність. Починаючи з iOS 16, з'явилась можливість навчити систему розпізнавати кастомні звуки, наприклад сигнал конкретного дверного дзвінка

або побутового приладу. При виявленні звуку надходить сповіщення, звуковий сигнал і тактильний відгук, а якщо підключений Apple Watch – також вібрація на зап'ясті. Проте дане рішення є закритим системним компонентом і недоступне стороннім розробникам для розширення або інтеграції у власні застосунки [5].

Інтерфейс функції Sound Notifications застосунку Live Transcribe & Notification наведено на рис. 1.1 – рис. 1.3.

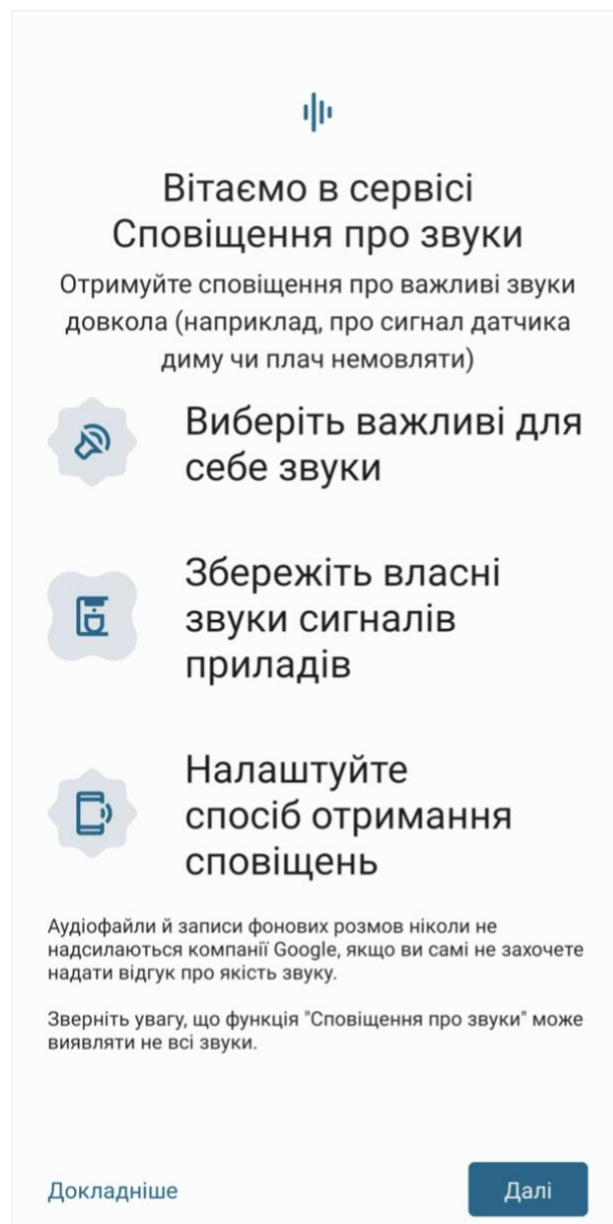


Рисунок 1.1 – Головний екран налаштувань функції Sound Notifications (OC Android) (скриншот)

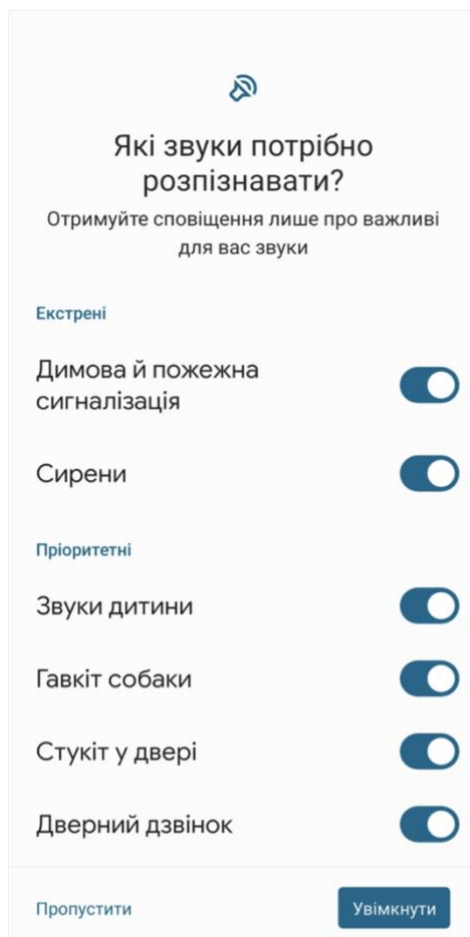


Рисунок 1.2 – Вибір категорій звуків для розпізнавання у Sound Notifications (скриншот)

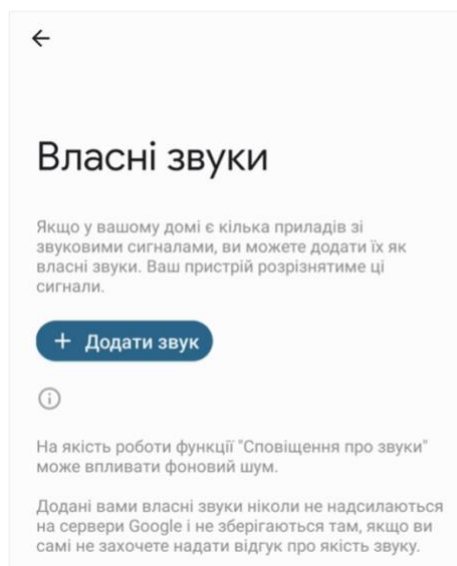


Рисунок 1.3 – Інтерфейс додавання власних звуків користувачем у Sound Notifications (скриншот)

Інтерфейс функції Sound Recognition в ОС iOS наведено на рис. 1.4 та рис. 1.5.

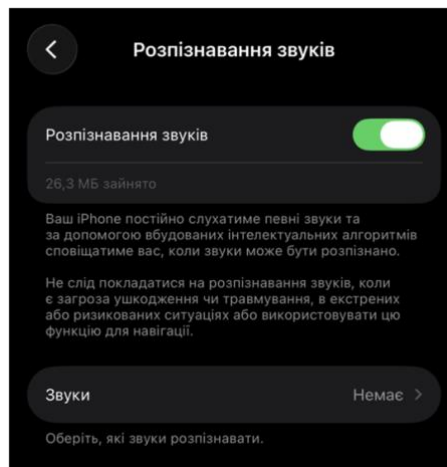


Рисунок 1.4 – Головний екран увімкнення функції Sound Recognition (ОС iOS) (скріншот)

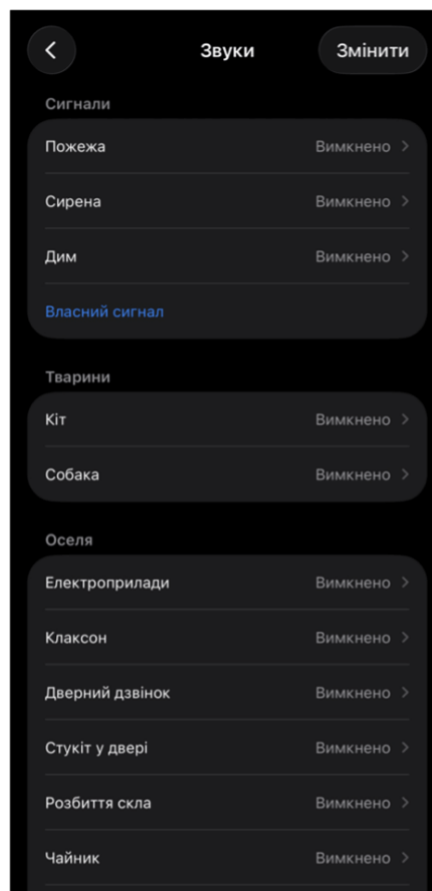


Рисунок 1.5 – Перелік доступних категорій звуків у системі Sound Recognition (скріншот)

Для систематизації наведених даних та обґрунтування доцільності створення власного програмного забезпечення, проведено порівняльний аналіз існуючих рішень із розроблюваною системою «SoundViz». Результати порівняння за ключовими критеріями наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика систем розпізнавання звуків

Критерій порівняння	Live Transcribe & Notification (Google)	Sound Recognition (Apple)	Розроблювана система «SoundViz»
Операційна система	Android	iOS	iOS
Кількість класів звуків	9 базових класів	15 базових + кастомні	36 спеціалізованих класів
Робота у фоновому режимі	Так	Так (на рівні ОС)	Так
Обробка даних (Інференс)	Локально на пристрої	Локально на пристрої	Локально на пристрої
Залежність від GPU у фоні	Не застосовується (ОС Android)	Використовує Neural Engine / Metal	Немає (інференс виключно на CPU)
Відкритість для розробників	Закрита екосистема	Системна функція (закрите API)	Відкрита для інтеграції у сторонні проєкти
Головний недолік	Обмежений набір звуків	Неможливість інтеграції у власні сторонні iOS-застосунки	Обмеження продуктивності через відмову від GPU

Як видно з таблиці 1.1, існуючі рішення від технологічних гігантів є потужними, проте вони глибоко інтегровані в операційні системи і є закритими для сторонніх розробників (Closed API). Це унеможливорює створення на їх базі спеціалізованих кастомних застосунків, наприклад, інтегрованих із медичними системами або системами розумного дому. Розроблювана система «SoundViz» вирішує цю проблему, пропонуючи

незалежний алгоритмічний конвеєр, здатний працювати у фоновому режимі iOS без виклику заблокованих графічних ресурсів.

1.3 Аналіз технічних обмежень платформи iOS та обґрунтування розробки

Компанія Apple надає розробникам фреймворк SoundAnalysis для аналізу аудіопотоків [6]. Проте технічний аналіз та досвід розробників, задокументований на офіційних форумах Apple Developer [7], вказують на суттєве обмеження: вбудований класифікатор `SNClassifySoundRequest` з ідентифікатором `.version1` оптимізований виключно для виконання на графічному процесорі через Metal GPU pipeline і не має резервного шляху виконання на CPU.

В ОС iOS діють жорсткі правила фонового виконання (Background Execution). Система дозволяє застосункам записувати аудіо у фоні (за наявності відповідного дозволу), але суворо блокує роботу з GPU для сторонніх застосунків у фоновому стані. При спробі інференсу стандартних моделей Apple у фоновому стані виникає виключення «Insufficient Permission to submit GPU work from background».

Це зумовлює необхідність розроблення власного алгоритмічного конвеєра, який:

- виконує обчислення мел-спектрограм засобами центрального процесора (CPU) через низькорівневий фреймворк Accelerate;
- використовує оптимізовані згорткові мережі (CNN) у конфігурації `cpuOnly` через Core ML.

1.4 Постановка завдання розроблення системи

Метою даної роботи є розроблення мобільного застосунку для iOS, який забезпечує фонове розпізнавання 36 класів звуків. Для досягнення мети необхідно вирішити такі завдання:

- сформуванати збалансований навчальний датасет на основі відкритої бібліотеки [Freesound.org](https://freedomofspeech.org/);
- реалізувати математичний апарат перетворення сигналів у мел-спектрограми без залучення GPU;
- спроектувати та навчити нейромережевий класифікатор з точністю не нижче 75%;
- реалізувати багаторівневу систему сповіщень (візуальну, тактильну та push);
- забезпечити можливість персоналізації списку розпізнаваних звуків.

Детальні вимоги до розроблюваної системи, її функціональних характеристик та умов експлуатації зафіксовано у Технічному завданні на дипломну роботу (Додаток А).

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Вибір та обґрунтування інструментальних засобів

Для реалізації етапу підготовки даних, проєктування та навчання нейронної мережі було обрано мову програмування Python, оскільки вона є індустріальним стандартом у сфері Data Science та має велику екосистему спеціалізованих бібліотек.

Для обґрунтування вибору мови програмування для етапу підготовки даних та навчання моделі було проведено порівняльний аналіз популярних інструментів, результати якого наведено у таблиці 2.1.

Таблиця 2.1 – Порівняльний аналіз мов програмування для розробки моделей машинного навчання

Критерій порівняння	Мова програмування		
	Python	C++	Swift
Наявність готових ML-фреймворків (PyTorch, TensorFlow)	Широкий вибір	Обмежена	Низька
Бібліотеки для цифрової обробки аудіо (DSP)	Висока	Висока	Середня
Швидкість прототипування та написання коду	Висока	Низька	Середня
Ефективність використання ресурсів комп'ютера	Середня	Висока	Висока
Підтримка експорту у формат Core ML	Наявна	Відсутня	Відсутня

Як видно з таблиці 2.1, незважаючи на нижчу швидкість виконання базових операцій порівняно з C++, мова Python має беззаперечну перевагу завдяки наявності спеціалізованих бібліотек (torchaudio, coremltools) та ідеальній сумісності з фреймворком PyTorch.

Основним фреймворком для побудови архітектури згорткової нейронної мережі (CNN) та процесу її навчання обрано PyTorch. Цей вибір обґрунтовано гнучкістю фреймворку при побудові динамічних

обчислювальних графів та наявністю вбудованих засобів для оптимізації навчання [8]. Для апаратного прискорення процесу навчання на базі процесорів архітектури Apple Silicon (ARM) було задіяно бекенд MPS (Metal Performance Shaders), що дозволило суттєво зменшити час тренування моделі за рахунок використання ресурсів інтегрованого графічного процесора.

Для роботи з аудіоданими використано такі бібліотеки:

- `soundfile`: для швидкого та низькорівневого читання аудіофайлів у форматі WAV;
- `torchaudio`: для попередньої обробки аудіосигналів, зокрема ресемплінгу та генерації мел-спектрограм на етапі підготовки даних [9];
- `pumpru`: для обчислення характеристик аудіосигналу (RMS) при фільтрації тихих фрагментів.

Оскільки кінцевою ціллю є інтеграція навченої моделі в мобільний застосунок для ОС iOS, критично важливим етапом є конвертація моделі. Для цього використано бібліотеку `coremltools`, яка дозволяє транслювати граф моделі PyTorch у спеціалізований формат `.mlmodel` (Core ML Neural Network), сумісний з системними API платформи Apple [10].

2.2 Обґрунтування вибору та формування набору аудіоданих

Якість розпізнавання звуків навколишнього середовища безпосередньо залежить від репрезентативності та збалансованості навчальної вибірки. Для реалізації проєкту було обрано сервіс `Freesound.org` – відкриту бібліотеку звуків, що дозволяє отримувати анотовані записи з різноманітними акустичними сценаріями [11].

Процес формування датасету було автоматизовано за допомогою розробленого програмного модуля на мові Python. Скрипт виконував такі функції:

- пошук звуків за ключовими словами для кожного з 36 цільових класів;

- фільтрацію результатів за типом ліцензії (Creative Commons) та тривалістю (від 1 до 15 секунд);
- автоматичне завантаження унікальних ідентифікаторів файлів для запобігання дублюванню даних.

Після автоматизованого збору було проведено етап ручної верифікації, під час якого з набору видалялися записи з низьким відношенням сигналу до шуму або нерелевантним змістом. Для підвищення точності навчання записи були сегментовані на фрагменти тривалістю 2 секунди, що відповідає вхідному вікну класифікації в реальному часі.

Усі звуки, які було розподілено на 36 цільових класів та згруповано за трьома рівнями небезпеки подано в табл. 2.2.

Таблиця 2.2 – Розподіл звукових класів за рівнями небезпеки

Рівень небезпеки	Категорія звуків (класи)
Високий	Повітряна тривога, пожежна сигналізація, сирена спецтранспорту, вибух, постріл, крик, плач дитини, автомобільний сигнал, розбиття скла
Середній	Стук, гріюк дверима, дверний дзвінок, телефон, будильник, кипіння води, мікрохвильова піч, собака, кіт, плач, дитяча мова, свердло, транспорт (авто, мотоцикл, вантажівка, автобус, потяг, літак)
Низький	Мова, сміх, кашель, кроки, оплески, велосипед, пиросос, фен, дзвін ключів, дощ, гроза

Для усунення проблеми незбалансованості класів (різної кількості зразків для різних звуків) було застосовано методику циклічної аугментації. Дефіцитні класи доповнювались шляхом випадкової зміни швидкості відтворення (коефіцієнти 0,9 та 1,1), зсуву пітчу на $\pm 1,5$ напівтони, зміни гучності на ± 3 дБ та додавання білого шуму (коефіцієнт 0,005).

Окрім офлайн-аугментації на етапі підготовки датасету, під час навчання моделі застосовувалася додаткова онлайн-аугментація безпосередньо у тренувальному конвеєрі. Кожен зразок із заданою ймовірністю піддавався одному або декільком перетворенням:

- додавання гаусівського шуму з амплітудою $\sigma = 0,005$ (ймовірність 30 %), що імітує фоновий шум мікрофона;
- випадкова зміна гучності в діапазоні коефіцієнтів $[0,7; 1,3]$ (ймовірність 30 %), що моделює різні відстані до джерела звуку;
- циклічний часовий зсув сигналу в межах $\pm 0,1$ секунди (ймовірність 20 %), що імітує різні моменти початку звукової події у вікні аналізу.

Такий дворівневий підхід до аугментації – офлайн на етапі балансування та онлайн під час навчання – дозволив суттєво збільшити варіативність навчальних даних без пропорційного збільшення обсягу датасету.

Для забезпечення єдиного формату вхідних даних усі аудіофайли перед обробкою проходили процес нормалізації. Технічні характеристики вхідного аудіопотоку, що використовувався для формування навчальної та валідаційної вибірок, наведено в табл. 2.3.

Таблиця 2.3 – Технічні параметри обробки вхідного аудіосигналу

Параметр	Значення
Частота дискретизації (Sample Rate)	16000 Гц
Кількість каналів	1 (Моно)
Глибина квантування	32-біт (Float32)
Тривалість вікна аналізу	2,0 с
Формат представлення даних	PCM (Pulse Code Modulation)

2.3 Алгоритм попередньої обробки сигналу та побудови мел-спектрограм

Цифрова обробка сигналу є проміжним етапом між захопленням аудіо та інференсом нейронної мережі. Оскільки сирий PCM-сигнал має високу розмірність, його необхідно перетворити у частотно-часове представлення.

У даній роботі реалізовано алгоритм побудови лог-мел-спектрограм, який включає такі кроки:

- попередня обробка: аудіосигнал приводиться до частоти дискретизації 16 кГц та моно-формату;
- віконне перетворення Фур'є (STFT): сигнал розбивається на кадри довжиною 1024 семпли з кроком (hop length) 160 семплів. Для зменшення ефектів накладання спектрів використовується віконна функція Ганна:
- обчислення спектру потужності за формулою (2.1):

$$P(f) = \frac{1}{N} |X(f)|^2, \quad (2.1)$$

де $P(f)$ – спектр потужності;

N – розмір вікна;

$X(f)$ – спектральні компоненти після ШПФ;

- мел-фільтрація: перетворення лінійної шкали частот у нелінійну шкалу Мел, яка краще відображає особливості людського сприйняття. Формула перетворення (2.2):

$$m = 2595 \cdot \log_{10}\left(1 + \frac{f}{700}\right), \quad (2.2)$$

де m – частота у мелах;

f – частота у герцах.

Для отримання фінального вектора ознак застосовується банк із 64 трикутних мел-фільтрів. Результатом є матриця розмірністю 64×200 , де 200 – кількість часових фреймів у 2-секундному вікні. Кожне значення нормалізується відносно власного мінімуму та максимуму (per-sample normalization), що робить модель стійкою до різної гучності вхідного сигналу.

Процес перетворення сигналу у спектральну область базується на віконному перетворенні Фур'є. Вибір параметрів ШПФ (FFT) є критичним для забезпечення балансу між часовою та частотною роздільною здатністю. Детальні параметри DSP-конвеєра наведено в табл. 2.4.

Таблиця 2.4 – Параметри алгоритму обчислення ознак (STFT)

Параметр алгоритму	Значення
Розмір вікна FFT (N_{FFT})	1024
Крок вікна (Hop Length)	160
Кількість мел-фільтрів (N_{Mels})	64
Мінімальна частота (f_{min})	20 Гц
Максимальна частота (f_{max})	8000 Гц
Тип віконної функції	Вікно Ганна (Hanning)

Банк трикутних мел-фільтрів у частотному діапазоні наведено на рис. 2.1.

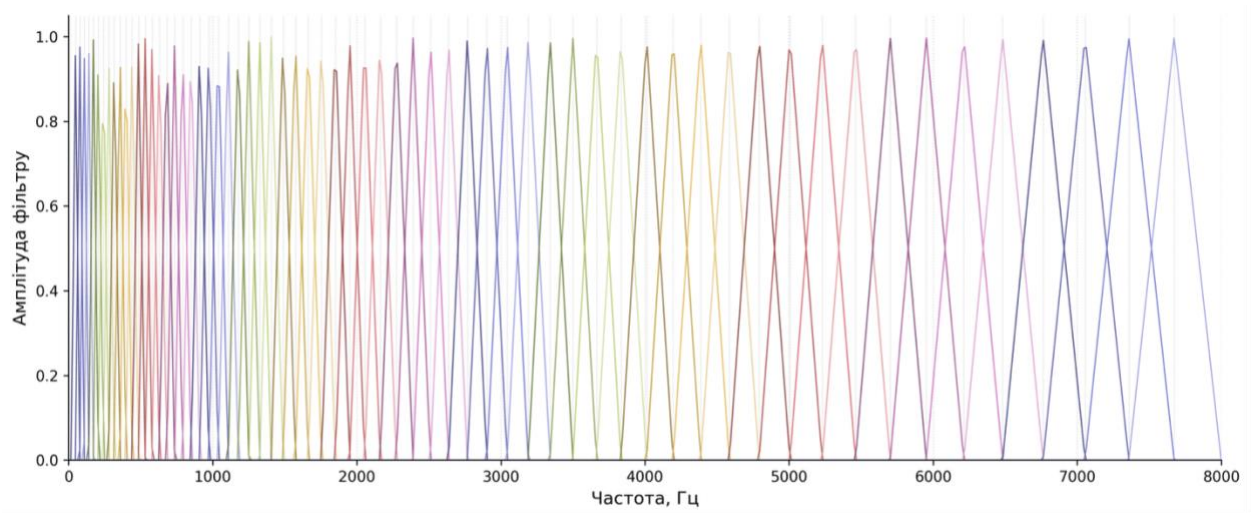
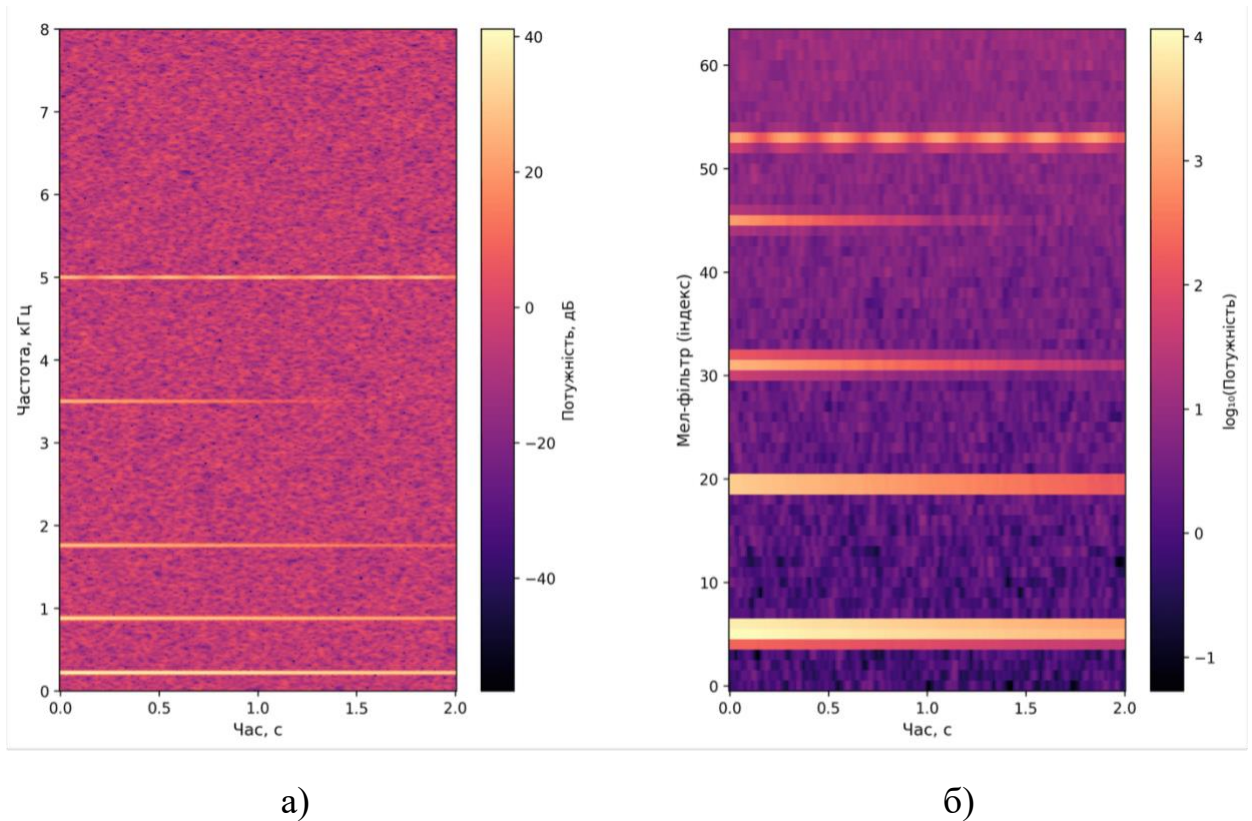


Рисунок 2.1 – Візуалізація банку трикутних мел-фільтрів у частотному діапазоні

Застосування мел-фільтрації дозволяє зменшити розмірність вхідних даних, зберігаючи при цьому найбільш значущу інформацію для розпізнавання людської мови та техногенних звуків. Після логарифмування амплітуд ми отримуємо матрицю, яка візуально відображає інтенсивність звуку в часі.

Порівняння лінійної спектрограми та лог-мел-спектрограми наведено на рис. 2.2.



а – лінійна спектрограма (масштаб: лінійний); б – лог-мел-спектрограма (масштаб: мел + логарифм)

Рисунок 2.2 – Порівняння лінійної спектрограми та лог-мел-спектрограми аудіосигналу

2.4 Архітектура та навчання згорткової нейронної мережі

Для класифікації спектрограм було спроектовано згорткову нейронну мережу (CNN) на основі ідеології залишкового навчання (ResNet). Використання залишкових блоків (Residual Blocks) дозволяє ефективно навчати глибокі мережі без ризику згасання градієнтів.

Спроектowana архітектура («AudioCNN») складається з таких функціональних шарів:

- Stem-шар: вхідна згортка 3×3 з 32 фільтрами, яка вилучає початкові ознаки зі спектрограми;

- чотири блоки ResidualBlock: кожен блок містить дві послідовні згортки, батч-нормалізацію та функцію активації ReLU. Кількість фільтрів збільшується від 64 до 256, що дозволяє ідентифікувати складніші звукові патерни;

- шар Global Average Pooling: зводить просторові розміри карти ознак до вектора одиничної розмірності, що суттєво зменшує кількість вагових коефіцієнтів моделі;

- повнозв'язний класифікатор: вихідний шар із 36 нейронами, що відповідають цільовим класам.

Процес навчання моделі проводився протягом 50 епох з використанням оптимізатора AdamW. Для запобігання перенавчанню застосовано метод Dropout (імовірність 0,5) та регуляризацию ваг (Weight Decay). Динаміка навчання контролювалася планувальником ReduceLROnPlateau, який зменшував крок навчання у разі стагнації точності на валідаційній вибірці.

Фінальна валідаційна точність розробленої моделі склала 75,56%. Після завершення навчання модель була експортована у формат Core ML з примусовим налаштуванням виконання на CPU (`computeUnits = .cpuOnly`). Це архітектурне рішення є критичним для забезпечення фонові роботи застосунку в ОС iOS, оскільки усуває звернення до заблокованого системного графічного конвеєра.

Архітектура розробленої згорткової нейронної мережі «AudioCNN» була оптимізована для забезпечення високої швидкості інференсу на центральному процесорі мобільного пристрою. Повний опис послідовності шарів мережі та розмірності вихідних тензорів наведено в табл. 2.5.

Таблиця 2.5 – Специфікація архітектури нейронної мережі AudioCNN

Назва шару	Тип операції	Параметри шару	Розмірність виходу (Shape)
1	2	3	4
Input	Вхідні дані	Log-Mel Spectrogram	(1, 64, 200)
Conv 1	Згортка 2D	kernel=3x3, filters=32	(32, 64, 200)
ResBlock 1	Residual Block	2 x Conv, filters=64	(64, 32, 100)

Продовження таблиці 2.5

1	2	3	4
ResBlock_2	Residual Block	2 x Conv, filters=128	(128, 16, 50)
ResBlock_3	Residual Block	2 x Conv, filters=256	(256, 8, 25)
ResBlock_4	Residual Block	2 x Conv, filters=256	(256, 4, 13)
GlobalAvgPool	Pooling	Average	(256, 1, 1)
FC_Layer	Повнозв'язний	linear + Dropout(0.5)	(36)
Softmax	Активація	Class Probabilities	(36)

Структурну схему залишкового блоку наведено на рис. 2.3.

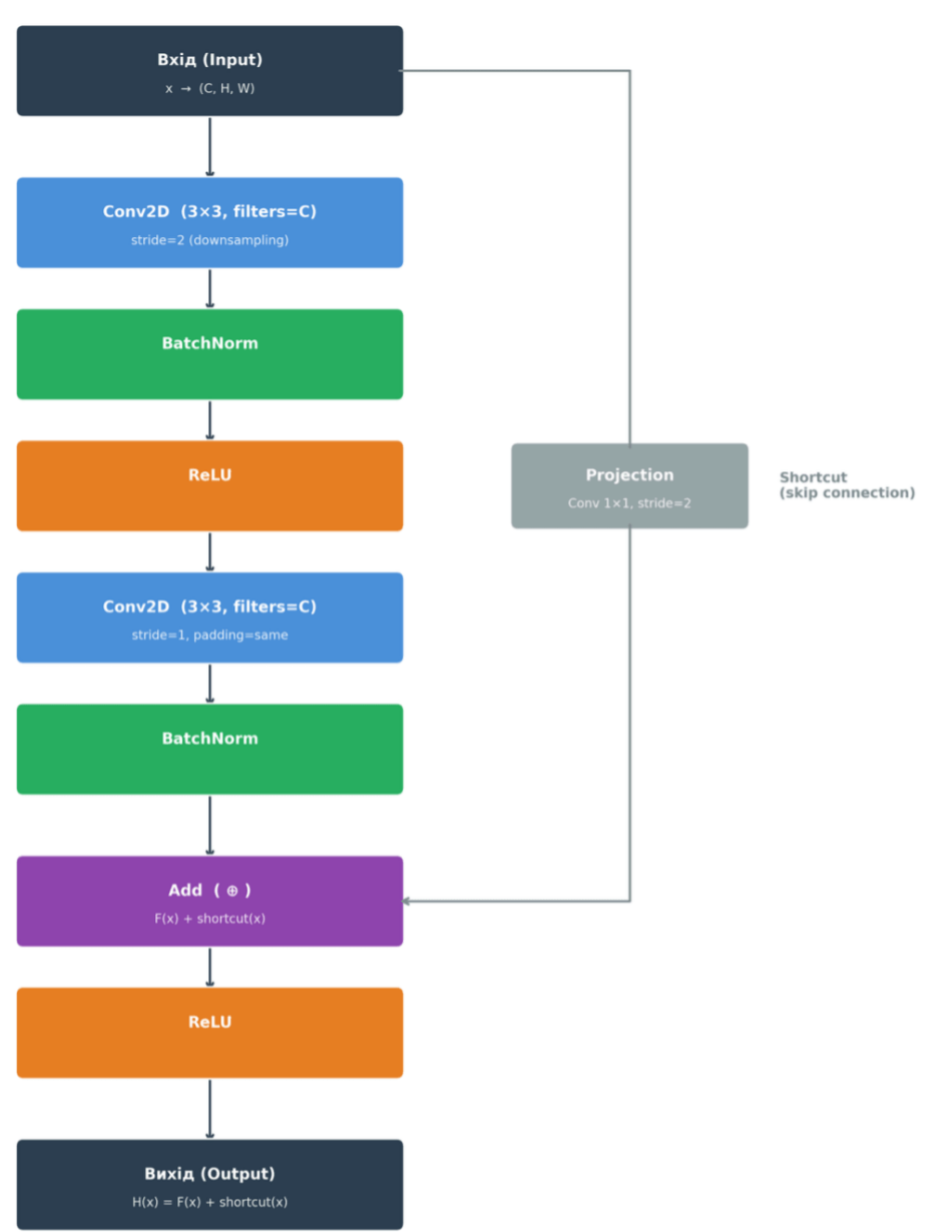


Рисунок 2.3 – Структурна схема залишкового блоку (Residual Block) з пропусканням сигналу

Процес навчання моделі проводився з використанням технології Early Stopping, що дозволило автоматично зупинити процес при досягненні плато валідаційної помилки. Гіперпараметри процесу навчання наведено нижче:

- початкова швидкість навчання (Learning Rate): 0,001;
- розмір пакету даних (Batch Size): 64;
- кількість епох навчання (Epochs): 50;
- регуляризація (L2 Weight Decay): $1e-4$.

Валідація проводилася на відокремленій частині набору даних, яка не використовувалася під час оновлення ваг моделі. Досягнута точність 75,56% підтверджує ефективність обраної архітектури для завдань класифікації 36 різних класів акустичних подій.

Програмну реалізацію описаних вище етапів підготовки аудіоданих, балансування датасету та безпосередньо процесу навчання згорткової нейронної мережі мовою Python наведено у додатку Б (лістинги Б.1 та Б.2).

3 ОПИС ПРОГРАМИ

3.1 Архітектурна побудова та логічна структура застосунку

Програмна система «SoundViz» розроблена як нативний iOS-застосунок з використанням декларативного фреймворку SwiftUI. Архітектура застосунку базується на патерні проєктування MVVM (Model-View-ViewModel) та розділена на логічні шари, що дозволяє чітко відокремити бізнес-логіку обробки аудіосигналу від графічного інтерфейсу.

Оскільки мобільний застосунок вимагає тісної взаємодії з апаратною частиною пристрою (мікрофоном, процесором, системою вібрації), вибір платформи та мови розробки є критичним етапом. Порівняльний аналіз підходів до мобільної розробки наведено у таблиці 3.1.

Таблиця 3.1 – Порівняльний аналіз технологій розробки мобільного застосунку

Критерій порівняння	Технологія розробки		
	нативна розробка (Swift / SwiftUI)	кросплатформна розробка (Flutter)	кросплатформна розробка (React Native)
Можливість роботи на обох платформах (iOS / Android)	Тільки iOS	Повна	Повна
Прямий доступ до низькорівневих фреймворків (vDSP, Core ML)	Повний	Обмежений (через мости)	Обмежений (через мости)
Можливість складної фонові обробки аудіо без GPU	Можлива	Неможлива	Неможлива
Висока продуктивність та мінімальна затримка (Latency)	Висока	Середня	Середня
Інтеграція з Taptic Engine (вібрація)	Повна	Обмежена	Обмежена

Аналіз, наведений у таблиці 3.1, показує, що хоча кросплатформні рішення (Flutter, React Native) дозволяють розробити застосунок одночасно

для iOS та Android, вони не надають необхідного рівня доступу до низькорівневих системних фреймворків Apple. Оскільки реалізація фонового моніторингу вимагає прямого виклику функцій Accelerate (vDSP) та Core ML в обхід графічного процесора, єдиним доцільним вибором є нативна розробка мовою Swift у середовищі Xcode.

Шар інтерфейсу (SwiftUI) представлений головним екраном (ContentView) та екраном налаштувань (SettingsView). Зв'язок між графічним інтерфейсом та бізнес-логікою реалізовано за допомогою реактивних обгорт `@StateObject` та `@ObservedObject` (фреймворк Combine). Вони забезпечують автоматичне та синхронне оновлення екранів (View) при будь-якій зміні станів у моделях представлення (ViewModels) без необхідності ручного перемальовування.

Роль головної ViewModel виконує Ядро обробки аудіо, центральним вузлом якого є клас SoundManager. Цей компонент керує всім життєвим циклом розпізнавання:

- взаємодіє із системним фреймворком AVAudioEngine для захоплення аудіопотоку;
- передає сигнал до модуля MelSpectrogram для обчислення ознак;
- надсилає сформований тензор до Модуля ML (SoundVizClassifier), який виконує інференс на базі системного фреймворку Core ML та повертає результат передбачення.

Після отримання результату від нейромережі, SoundManager звертається до Модулів даних. Він використовує SoundDictionary для отримання локалізованої інформації про розпізнаний звук (іконка, назва, рівень небезпеки) та перевіряє через SoundFilterManager, чи не вимкнув користувач сповіщення для цього конкретного звуку. Зі свого боку, SoundFilterManager виступає додатковою ViewModel для екрана SettingsView та забезпечує збереження налаштувань фільтрації у постійній пам'яті через системний API UserDefaults.

Завершальним етапом обробки є виклик Модулів зворотного зв'язку. Якщо звук визнано небезпечним і не відфільтровано, SoundManager ініціює тактильний відгук (вібрацію) через HapticManager та надсилає системне push-сповіщення у фоновому режимі за допомогою NotificationManager (фреймворк UserNotifications).

Загальну схему архітектурної взаємодії цих компонентів наведено на рис. 3.1.

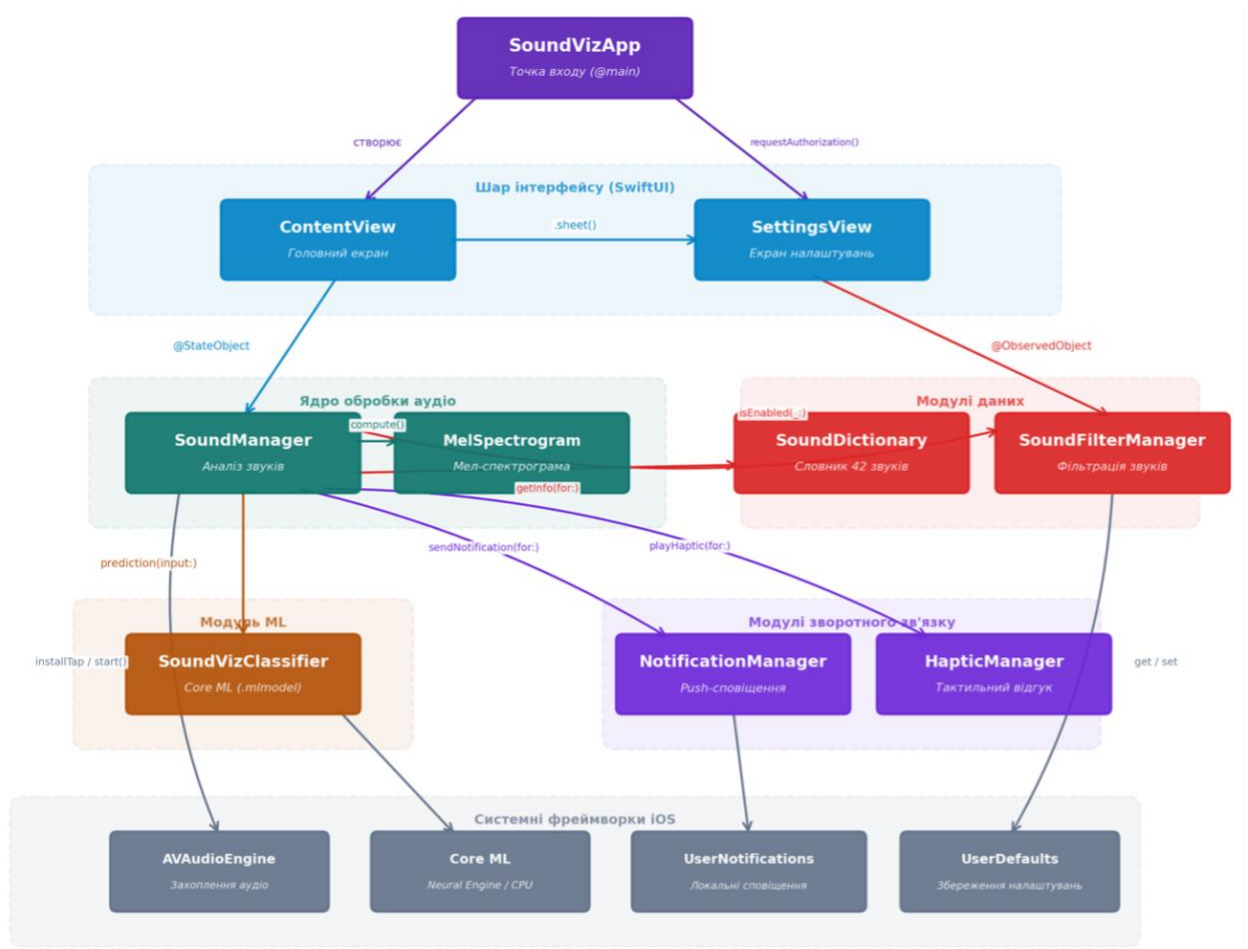


Рисунок 3.1 – Схема архітектурної взаємодії компонентів системи

3.2 Реалізація модуля цифрової обробки сигналів та аналізу звуку

Ключовою технічною особливістю розробки є відмова від стандартних високорівневих засобів Apple на користь власної реалізації DSP-конвеєра. Це дозволило вирішити проблему блокування GPU у фоновому режимі.

Модуль MelSpectrogramCalculator використовує фреймворк Accelerate для виконання операцій над векторами. Програмна реалізація включає:

- ініціалізацію об'єкта FFTSetup для швидкого перетворення Фур'є;
- застосування вікна Ганна до вхідного буфера PCM-даних для усунення крайових ефектів;
- розрахунок енергетичного спектру та його проектування на мел-шкалу за допомогою заздалегідь розрахованої матриці мел-фільтрів.

Процес захоплення та перетворення аудіо відбувається у декілька етапів. Оскільки системний мікрофон пристроїв Apple зазвичай захоплює аудіо з частотою дискретизації 44,1 або 48 кГц, програма використовує об'єкт AVAudioConverter для «на льоту» пониження частоти (downsampling) до цільових 16 кГц у моно-форматі. Це суттєво зменшує обсяг даних та обчислювальне навантаження.

Перехоплені аудіо-буфери накопичуються у масиві audioDataBuffer. Як тільки розмір накопиченого масиву досягає 32 000 відліків (що дорівнює 2 секундам аудіо), програма вирізає цей фрагмент та передає його до модуля класифікації. Перед викликом нейромережі фрагмент проходить перевірку на поріг тиші: обчислюється середньоквадратичне значення (RMS) амплітуди, і якщо воно менше за 0,005, обчислювальний конвеєр переривається. Це дозволяє економити заряд батареї мобільного пристрою в умовах повної тиші, не викликаючи модель Core ML даремно.

3.3 Програмна реалізація інтерфейсу користувача та системи сповіщень

Інтерфейс користувача спроектований з урахуванням специфіки цільової аудиторії – людей з порушеннями слуху. Концептуально дизайн та взаємодія з головним екраном (ContentView) натхненні популярним застосунком Shazam. Такий вибір обґрунтований тим, що подібний патерн інтерфейсу є загальновідомим та інтуїтивно зрозумілим: у центрі розташована

масивна кнопка керування, яка одразу привертає увагу та чітко сигналізує про поточний стан системи (прослуховування або очікування). Завдяки цьому головний екран фокусує увагу виключно на візуальному відображенні результату, не перевантажуючи користувача зайвими елементами управління.

Зовнішній вигляд головного екрана у стані очікування та під час активного прослуховування наведено на рис. 3.2 та рис. 3.3.

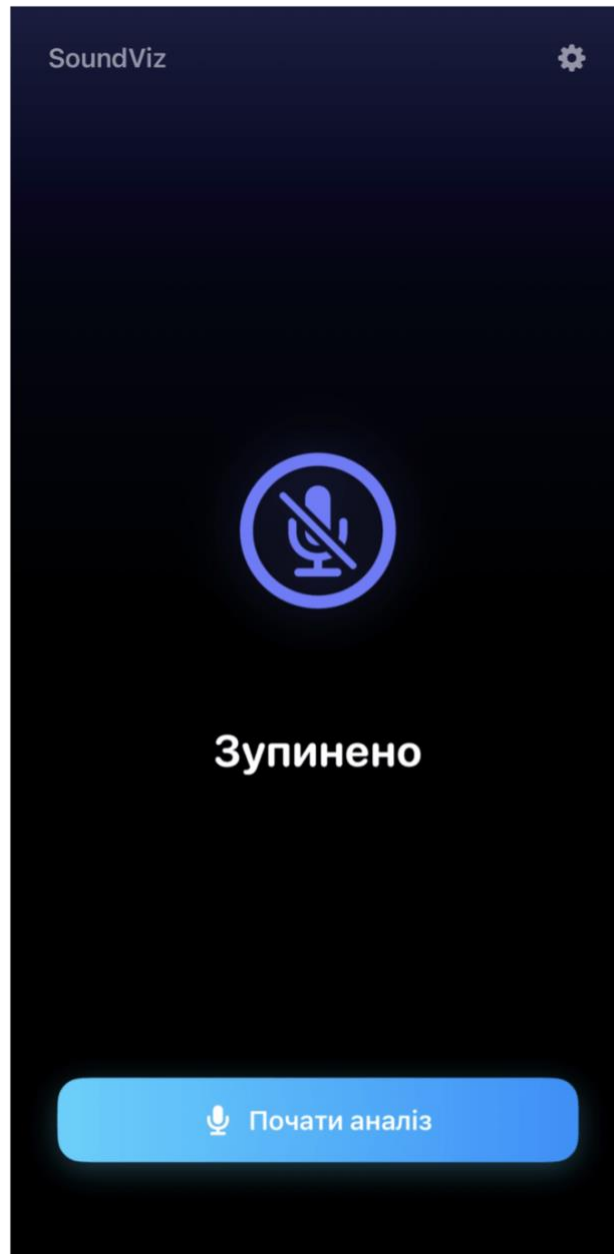


Рисунок 3.2 – Головний екран застосунку у стані очікування

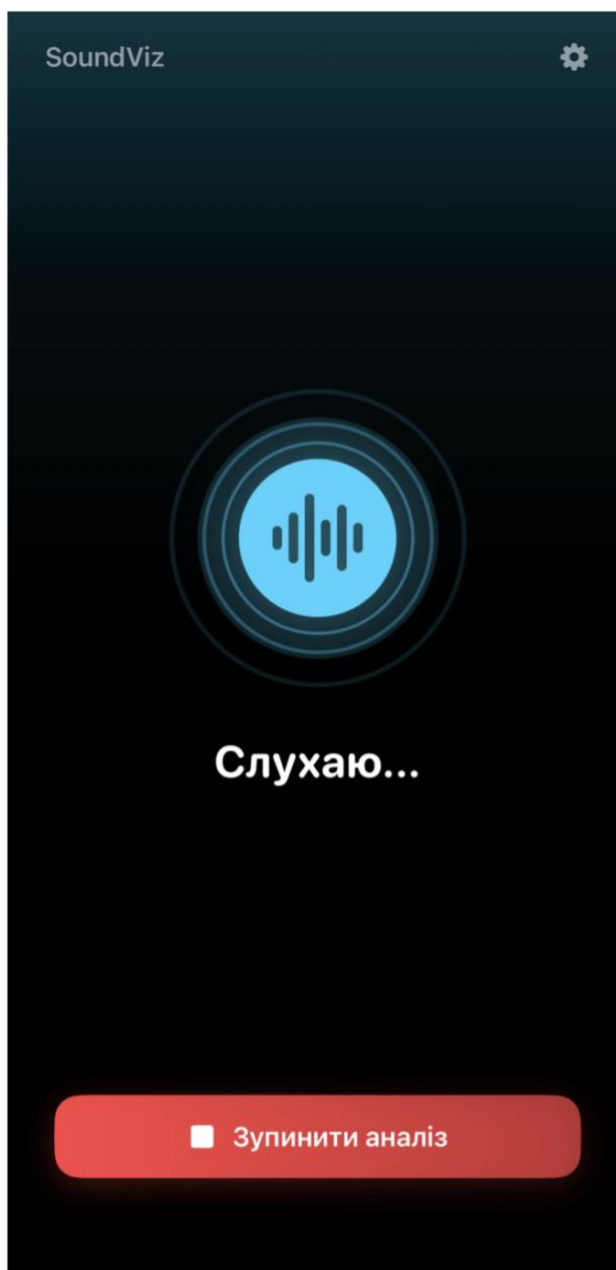


Рисунок 3.3 – Головний екран застосунку під час розпізнавання звуків

Програмно реалізовано такі елементи візуалізації:

- динамічна зміна іконки (використання бібліотеки SF Symbols) залежно від типу розпізнаного звуку;
- кольорове кодування фону та елементів: реалізовано через перелічення DangerLevel. У разі розпізнавання критичних звуків (сирени, вибухи) інтерфейс набуває червоного кольору (.high), для побутових звуків (стукіт, телефон) – помаранчевого (.medium), для фонових (розмова, кроки) – синього (.low);

– анімація «сонара»: під час активного прослуховування мікрофона навколо центральної іконки генеруються пульсуючі кола за допомогою модифікаторів `.scaleEffect` та `.opacity`, що дає користувачеві візуальний зворотний зв'язок про активність процесу;

– відображення рівня впевненості моделі (Confidence) у відсотках.

Візуалізацію різних рівнів небезпеки розпізнаних звуків представлено на рис. 3.4 – рис. 3.6.

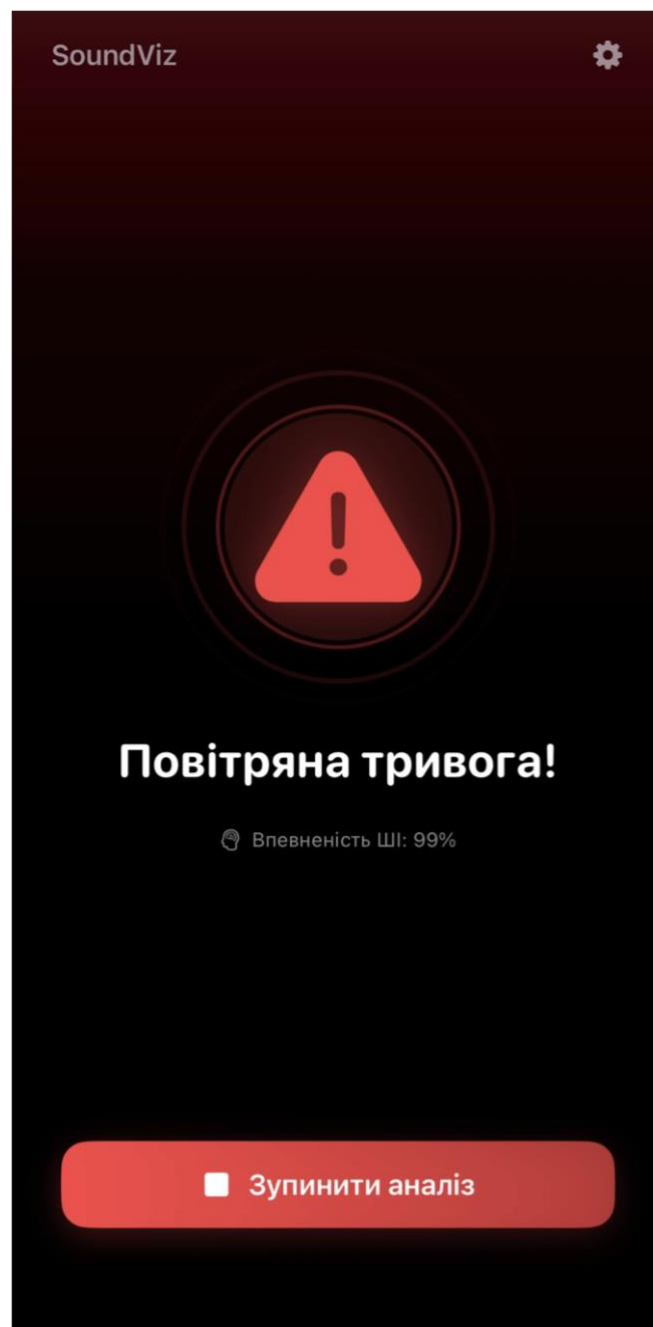


Рисунок 3.4 – Візуалізація високого рівня небезпеки розпізнаного звуку

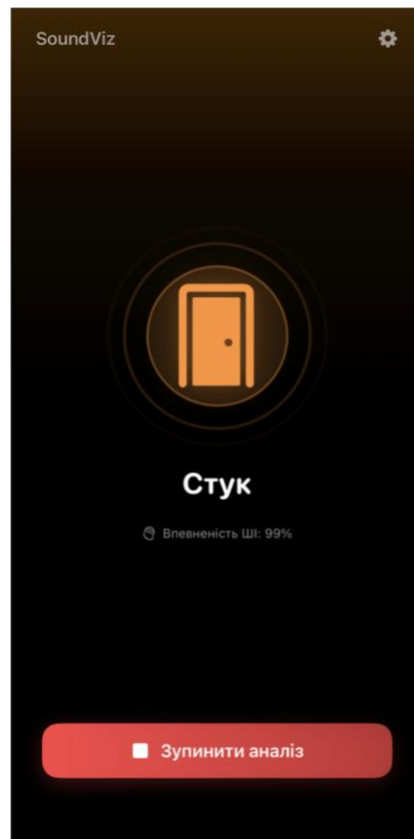


Рисунок 3.5 – Візуалізація середнього рівня небезпеки розпізнаного звуку

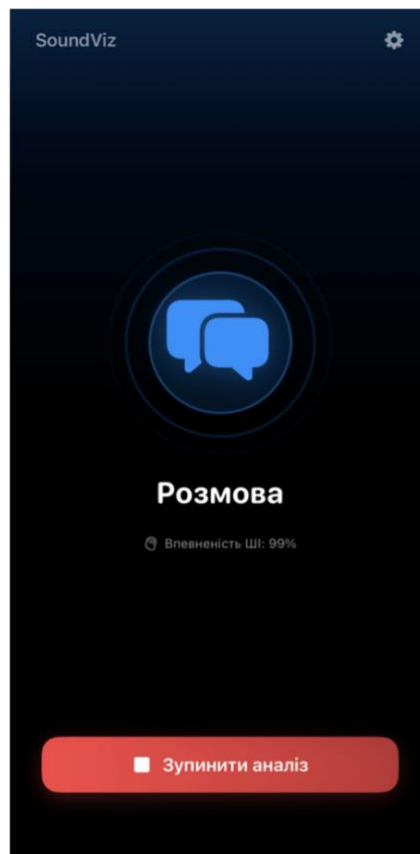


Рисунок 3.6 – Візуалізація низького рівня небезпеки розпізнаного звуку

Система тактильного відгуку (Haptics) глибоко інтегрована у логіку рівнів безпеки. Менеджер HapticManager ініціалізує генератори заздалегідь методом `prepare()`, щоб уникнути затримок. Для звуків високої небезпеки відтворюється подвійна агресивна вібрація (комбінація `UINotificationFeedbackGenerator(.error)` та `UIImpactFeedbackGenerator(.heavy)`), для середньої – одиночне попередження. Для звуків низького рівня небезпеки вібрація не відтворюється, щоб не перевантажувати користувача та не розряджати акумулятор.

Для забезпечення зворотного зв'язку без візуального контакту з пристроєм реалізовано модуль `NotificationManager`. Він використовує `UINotificationFeedbackGenerator` для створення тактильних вібрацій. Для звуків високого рівня небезпеки (наприклад, пожежна тривога) генерується серія інтенсивних вібрацій типу `.error`, для середнього рівня – `.warning`.

Додатково реалізовано екран налаштувань (`SettingsView`), де за допомогою компонентів `Toggle` користувач може активувати або деактивувати окремі класи звуків. Дані про налаштування зберігаються локально через `UserDefaults`, що забезпечує збереження конфігурації після перезавантаження застосунку. Інтерфейс меню налаштувань наведено на рис. 3.7.

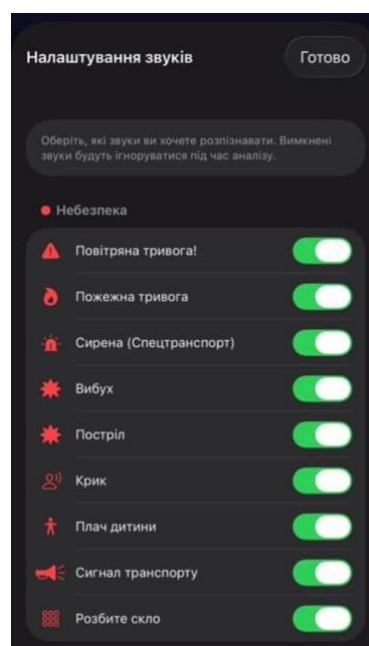


Рисунок 3.7 – Екран налаштувань фільтрації розпізнаваних звуків

Окрім функціональної складової, значну увагу було приділено інклюзивності та доступності інтерфейсу. Застосунок повністю підтримує системну зміну світлої та темної тем (Light/Dark Mode). Градієнтні фони та кольори шрифтів адаптуються автоматично за допомогою змінних середовища SwiftUI, що забезпечує комфортне користування застосунком як удень, так і вночі, зменшуючи навантаження на зір.

Також програмну систему розроблено з урахуванням можливості масштабування на міжнародний ринок. Інтерфейс користувача та словник назв акустичних подій повністю локалізовано. Завдяки використанню сучасних механізмів локалізації, застосунок автоматично адаптується під українську або англійську мову, залежно від системних налаштувань смартфона користувача. Адаптивність інтерфейсу продемонстровано на рис. 3.8 та рис. 3.9.

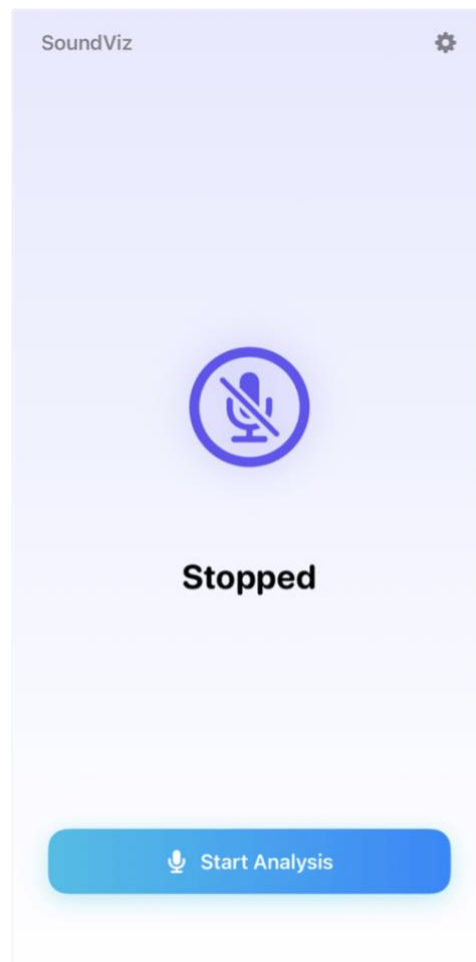


Рисунок 3.8 – Світла тема та англomовний інтерфейс застосунку

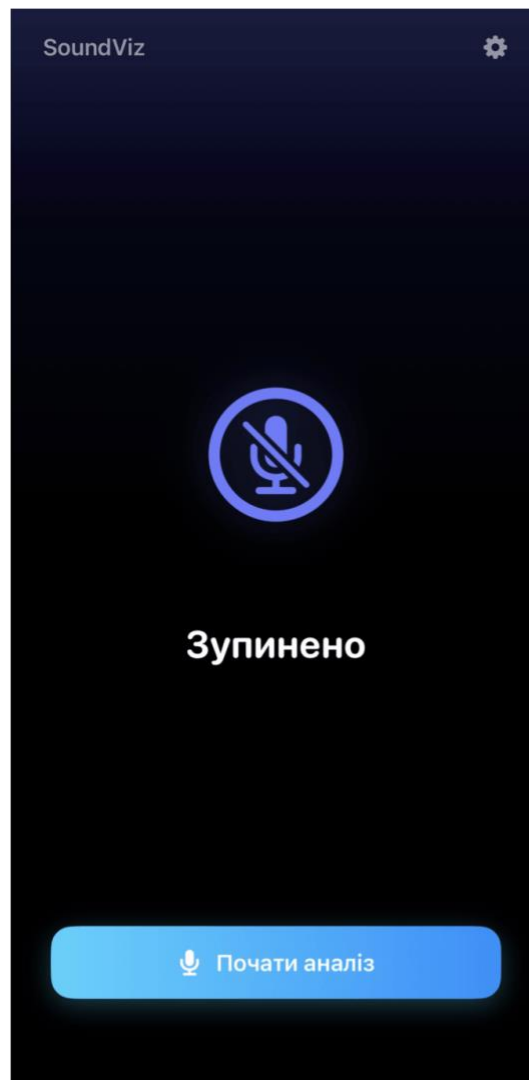


Рисунок 3.9 – Темна тема та україномовний інтерфейс застосунку

3.4 Забезпечення надійності та обробка виняткових ситуацій

У процесі розробки програмного забезпечення особливу увагу приділено стабільності роботи системи. Реалізовано механізми обробки таких ситуацій:

- відсутність дозволу на використання мікрофона: програма ініціює системний запит та відображає інструкцію з налаштування у разі відмови;
- переривання аудіосесії (наприклад, вхідним дзвінком): застосунок автоматично відновлює моніторинг після завершення переривання;

– захист від «спаму» сповіщеннями: впроваджено систему часових затримок (cooldown), яка запобігає надсиланню повторних сповіщень про один і той самий звук протягом 5 секунд після першого розпізнавання.

Технічні параметри функціонування застосунку наведено в табл. 3.2.

Таблиця 3.2 – Характеристики функціонування програмної системи

Параметр	Значення
Час реакції системи (Latency)	< 2,5 с
Мінімальна версія ОС	iOS 17.6
Споживання оперативної пам’яті	45-60 МБ
Режим виконання інференсу	CPU Only
Метод тактильного відгуку	Taptic Engine (Haptics)

Вихідний код розробленого мобільного застосунку, зокрема ключових класів для перехоплення аудіопотоку, обчислення мел-спектрограм на центральному процесорі та генерації фонових сповіщень мовою Swift, наведено у додатку Б (лістинги Б.3 та Б.4).

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Опис застосування та інструкція з експлуатації програми

Головною метою даної дипломної роботи було підвищення рівня безпеки та соціальної адаптації людей з вадами слуху шляхом розроблення мобільного застосунку для класифікації навколишніх звуків. Розроблена програмна система «SoundViz» повністю вирішує це завдання, перетворюючи мобільний пристрій (iPhone) на персонального акустичного асистента, який безперервно аналізує середовище та сповіщає користувача про критичні чи побутові події за допомогою візуальних і тактильних сигналів.

Розроблений застосунок розповсюджується у вигляді виконуваного файлу для операційної системи iOS. Взаємодія користувача з програмою побудована за принципом мінімізації необхідних дій та максимальної інформативності.

Перший запуск та налаштування дозволів. При першому відкритті застосунку користувачеві демонструється вітальне повідомлення, що пояснює мету програми. Після натискання кнопки «Продовжити», операційна система ініціює запит на отримання двох критично важливих дозволів: на надсилання сповіщень (для роботи вібрації та push-повідомлень у фоні) та на використання мікрофона. Для коректної роботи застосунку необхідно надати обидва дозволи. Процес першого налаштування наведено на рис. 4.1 та рис. 4.2.

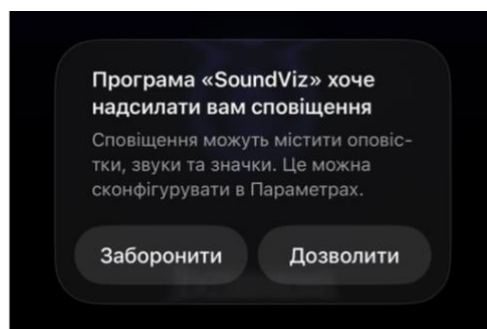


Рисунок 4.1 – Системний запит на отримання дозволу для надсилання сповіщень

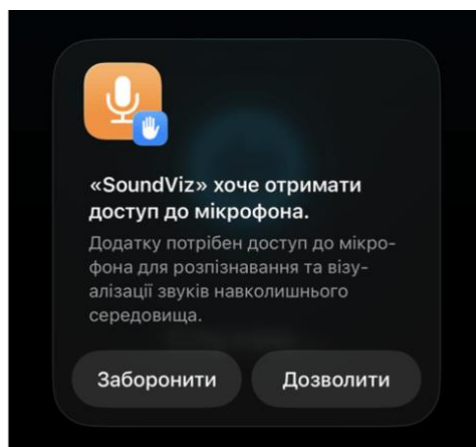


Рисунок 4.2 – Системний запит на отримання доступу до мікрофона

Робота з головним екраном. Головний екран програми містить центральну кнопку-індикатор у вигляді мікрофона. Для початку моніторингу простору користувачеві необхідно натиснути кнопку «Почати аналіз» (Start Analysis) (рис. 3.2). Інтерфейс перейде в активний стан (рис. 3.3): навколо іконки з'являться пульсуючі хвилі, що сигналізують про роботу мікрофона.

Програма може працювати в трьох інформаційних режимах (рис. 3.4 – рис. 3.6):

- безпечний режим (синій фон): фіксує безпечні звуки, такі як шум дощу, розмова або кроки. Візуалізується лише на екрані;
- режим уваги (помаранчевий фон): сигналізує про побутові події (стукіт у двері, нявкання кота, кипіння чайника). Супроводжується короткою вібрацією;
- режим небезпеки (червоний фон): сигналізує про критичні ситуації (пожежна тривога, сирена швидкої допомоги, звук розбитого скла) супроводжується сильною подвійною вібрацією та системним сповіщенням на екран.

Робота у фоновому режимі. Для забезпечення постійної безпеки користувач може згорнути застосунок (перейти на домашній екран) або заблокувати пристрій. Завдяки оптимізації обчислень на центральному процесорі, програма продовжить аналізувати звуки. У разі виникнення

критичної події екран пристрою засвітиться, спрацює вібрація, і користувач отримає push-сповіщення з назвою розпізнаного звуку.

Налаштування фільтрації. У правому верхньому куті екрана знаходиться кнопка виклику меню налаштувань. На екрані налаштувань (рис. 3.7) користувач бачить повний перелік із 36 підтримуваних звуків, згрупованих за рівнями небезпеки. За допомогою перемикачів можна вимкнути розпізнавання тих звуків, які не є актуальними (наприклад, вимкнути реакцію на собаку, якщо вдома є домашній улюбленець). Усі зміни зберігаються автоматично.

4.2 Методика та результати тестування

Для перевірки коректності роботи програмної системи застосовувалося функціональне тестування (Functional Testing) на реальному фізичному пристрої під управлінням iOS 26.4. Тестування проводилося методом «чорного ящика» (Black-box testing) з метою перевірки відповідності реальної поведінки застосунку заявленим вимогам.

Методика тестування включала перевірку життєвого циклу застосунку, обробку системних переривань та управління дозволами. Результати виконання тестових сценаріїв зведено у таблицю 4.1.

Таблиця 4.1 – Результати функціонального тестування системи

№	Тестовий сценарій (Кроки)	Очікуваний результат	Фактичний результат	Статус
1	2	3	4	5
1	Запуск без наданого дозволу на мікрофон	Відображення екрану з проханням надати дозвіл у Налаштуваннях iOS	Інтерфейс блокує запуск аналізу, відображається попередження	Успішно

Продовження таблиці 4.1

1	2	3	4	5
2	Імітація вхідного дзвінка під час аналізу звуку	Аудіосесія переривається. Після завершення дзвінка аналіз автоматично відновлюється	Система коректно обробляє AVAudioSession .interruptionNotification	Успішно
3	Перехід застосунку у фоновий режим (Background)	Прослуховування продовжується, інтерфейс не оновлюється, push-сповіщення працюють	Інференс на CPU триває, сповіщення надходять	Успішно
4	Перевірка механізму “Cooldown” (захист від спаму)	При безперервному звуці сирени сповіщення надходить 1 раз на 30 секунд	Повторні сповіщення про один і той самий звук ігноруються протягом 30 с	Успішно

Додатково було проведено профілювання застосунку за допомогою інструменту Xcode Instruments (Time Profiler та Allocations). Витоків пам'яті (Memory Leaks) не виявлено. Споживання оперативної пам'яті під час безперервного фонових інференсу залишалося стабільним на рівні 50-60 МБ, що підтверджує енергоефективність реалізованого DSP-конвеєра.

4.3 Експериментальне дослідження програми в реальних умовах

Оскільки навколишнє середовище містить значну кількість шумів, практичне застосування програми було перевірено шляхом експериментального дослідження. Метою експерименту було визначення здатності нейромережевої моделі розпізнавати цільові звуки на різній відстані та за наявності фонових міського шуму (Pink Noise).

Методика експерименту: мобільний пристрій розміщувався на столі. Звуки генерувалися з акустичної системи (потужністю 10 Вт) на відстані 1 метр та 3 метри від пристрою. У якості цільових звуків було обрано три

патерни: пожежна сигналізація (високий рівень небезпеки), стукіт у двері (середній рівень) та гавкіт собаки (середній рівень). Як фоновий шум використовувався запис вулиці з інтенсивністю близько 45-50 дБ.

Результати експериментального дослідження наведено у таблиці 4.2.

Таблиця 4.2 – Результати розпізнавання звуків у реальних умовах

Джерело звуку	Умови середовища (відстань / шум)	Середній рівень впевненості моделі (Confidence)	Час реакції системи
Пожежна сигналізація	1 м / без шуму	93%	2,1 с
Пожежна сигналізація	3 м / міський шум	79%	2,3 с
Стукіт у двері	1 м / без шуму	99%	2,1 с
Стукіт у двері	3 м / міський шум	72%	4,1 с (знадобилося 2 буфери)
Гавкіт собаки	1 м / без шуму	88%	2,2 с
Гавкіт собаки	3 м / міський шум	67%	Розпізнано частково

4.4 Аналіз результатів та рекомендації щодо застосування

Узагальнення та аналіз отриманих експериментальних результатів (табл. 4.2) показують, що програмна система впевнено справляється з розпізнаванням інтенсивних, високочастотних та тривалих звуків (таких як сирени чи пожежна сигналізація) навіть за умов фонового шуму. Рівень впевненості моделі у цих випадках не падає нижче 80%, що гарантує надійне інформування користувача про небезпеку.

Короткотривалі імпульсні звуки (наприклад, стукіт у двері) розпізнаються гірше на великих відстанях у зашумленому середовищі. Це зумовлено тим, що для формування мел-спектрограми система накопичує 2-секундний аудіобуфер, у якому короткий імпульс може «розмитися» на фоні тривалого міського шуму.

Рекомендації щодо практичного застосування результатів роботи:

- для досягнення максимальної ефективності та безпеки користувачам рекомендується носити мобільний пристрій у верхніх кишенях одягу (щоб мікрофон не перекривався щільною тканиною) або тримати його на відкритих поверхнях (стіл, тумба);
- оскільки вібрація смартфона у сумці чи кишені куртки може бути невідчутною під час ходьби, настійно рекомендується використовувати розроблений застосунок у парі зі смарт-годинником Apple Watch (модель NotificationManager автоматично дублює push-сповіщення та тактильний відгук на зап'ястя користувача);
- для користувачів, які тривалий час працюють або навчаються за комп'ютером, рекомендується використовувати системну функцію функцію віддзеркалення iPhone (iPhone Mirroring). Ця системна можливість підтримується в актуальних версіях операційних систем Apple, починаючи з iOS 18 та macOS 15 (Sequoia). Завдяки інтеграції на рівні спільного облікового запису Apple ID, push-сповіщення про розпізнані критичні та побутові звуки автоматично дублюватимуться на монітор комп'ютера Mac. Це забезпечує надійний візуальний контроль за навколишнім акустичним середовищем без необхідності тримати смартфон у полі зору;
- з метою економії заряду акумулятора у нічний час, рекомендується використовувати меню фільтрації (SettingsView) для вимкнення розпізнавання неактуальних побутових звуків, залишаючи активними лише сповіщення високого рівня небезпеки.

ВИСНОВКИ

У роботі вирішено актуальне науково-практичне завдання підвищення рівня безпеки та соціальної адаптації людей з вадами слуху шляхом розроблення та впровадження програмної системи розпізнавання звуків навколишнього середовища на базі мобільних технологій.

Основні наукові та практичні результати, отримані під час виконання дипломної роботи, полягають у наступному:

- здійснено аналіз предметної області, існуючих асистивних технологій та системних обмежень мобільної платформи iOS. Виявлено, що головною перешкодою для створення надійних систем безперервного фонових моніторингу акустичного середовища є жорстке системне блокування GPU-обчислень для неактивних застосунків. Це зумовило необхідність розробки архітектури, оптимізованої виключно для центрального процесора (CPU);

- сформовано та підготовлено збалансований навчальний набір аудіоданих на основі відкритої бібліотеки Freesound.org. Застосовано методи цифрової обробки сигналів для нормалізації, фільтрації тиші та циклічної аугментації даних, що дозволило згрупувати акустичні події у 36 цільових класів, поділених за трьома рівнями небезпеки;

- запропоновано та реалізовано алгоритмічний DSP-конвеєр для побудови лог-мел-спектрограм безпосередньо на пристрої за допомогою низькорівневого фреймворку Accelerate (vDSP). Спроектовано та навчено згорткову нейронну мережу архітектури ResNet, адаптовану для виконання на CPU через формат Core ML. Досягнута точність (Ассурасу) класифікації на валідаційній вибірці склала 75,56%;

- розроблено нативний iOS-застосунок «SoundViz» з використанням декларативного фреймворку SwiftUI та архітектурного патерну MVVM. Програмно реалізовано багаторівневу систему сповіщень: візуальну динамічну індикацію, тактильний відгук (Haptics) та локальні push-сповіщення у фоновому режимі з механізмом дедуплікації;

- проведено функціональне тестування та експериментальне дослідження розробленої програмної системи в реальних міських умовах. Результати підтвердили її працездатність: система стабільно працює у фоновому режимі (споживання пам'яті 45-60 МБ, час реакції до 2,5 с) та впевнено розпізнає інтенсивні звуки небезпеки (пожежні сигналізації, сирени) навіть за умов наявності фонового шуму.

Практичне значення отриманих результатів полягає у створенні готового до використання програмного продукту, який виступає в ролі «цифрового вуха» для людей з порушеннями слуху, суттєво підвищуючи рівень їхньої безпеки в урбанізованому середовищі. Інтерфейс системи адаптовано до потреб цільової аудиторії, він підтримує темну і світлу теми та локалізований двома мовами, що робить його готовим до масштабування.

Перспективи подальшого розвитку системи. Незважаючи на успішну реалізацію базового функціоналу, розроблена програмна система має значний потенціал для вдосконалення. Основними напрямками для майбутніх досліджень є:

- збагачення набору аудіоданих: збільшення обсягу навчального датасету (збір специфічних локальних звуків, запис акустичних сцен з різними типами міських та побутових шумів) дозволить суттєво підвищити загальну точність розпізнавання та мінімізувати кількість хибних спрацювань у складних акустичних умовах;

- створення нативного застосунку для watchOS: розробка незалежного застосунку для смарт-годинника Apple Watch дозволить використовувати його мікрофон для локального розпізнавання звуків. Це зробить систему повністю автономною та носимою (wearable);

- впровадження персоналізації (On-device Learning): реалізація функціоналу донавчання нейронної мережі безпосередньо на пристрої дозволить системі запам'ятовувати унікальні звуки середовища конкретного користувача (наприклад, звучання його власного дверного дзвінка).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. World Health Organization. Deafness and hearing loss. URL: <https://www.who.int/news-room/fact-sheets/detail/deafness-and-hearing-loss> (date of access: 28.04.2026).
2. National Fire Protection Association. Review of Alarm Technologies for Deaf and Hard of Hearing Populations. Fire Protection Research Foundation. URL: <https://www.nfpa.org/education-and-research/research/fire-protection-research-foundation/projects-and-reports/review-of-alarm-technologies-for-deaf-and-hard-of-hearing-populations> (date of access: 28.04.2026).
3. National Institute on Deafness and Other Communication Disorders. Assistive Devices for People with Hearing, Voice, Speech, or Language Disorders. URL: <https://www.nidcd.nih.gov/health/assistive-devices-people-hearing-voice-speech-or-language-disorders> (date of access: 28.04.2026).
4. Google LLC. Sound Notifications: Get notified about important sounds around you. Android Accessibility Help. URL: <https://support.google.com/accessibility/android/answer/10092548> (date of access: 29.04.2026).
5. Apple Inc. Recognize sounds using iPhone. iPhone User Guide. URL: <https://support.apple.com/guide/iphone/use-sound-recognition-iphf2dc33312/ios> (date of access: 29.04.2026).
6. Apple Inc. Sound Analysis. Apple Developer Documentation. URL: <https://developer.apple.com/documentation/SoundAnalysis> (date of access: 29.04.2026).
7. Apple Developer Forums. Urgent Issue with SoundAnalysis in Background Mode (iOS 18). Apple Developer Forums. URL: <https://developer.apple.com/forums/thread/765376> (date of access: 29.04.2026).
8. PyTorch. PyTorch Documentation. URL: <https://docs.pytorch.org/docs/2.11/index.html> (date of access: 01.05.2026).
9. PyTorch. TorchAudio Documentation. URL:

<https://pytorch.org/audio/stable/index.html> (date of access: 01.05.2026).

10. Apple Inc. Coremltools Documentation. URL: <https://apple.github.io/coremltools/docs-guides/> (date of access: 01.05.2026).

11. Freesound. Find any sound you like. URL: <https://freesound.org> (date of access: 01.05.2026).

ДОДАТОК А
Технічне завдання

A.1 Вступ

Найменування програмного виробу: «Програмна система для розпізнавання та візуалізації звуків навколишнього середовища» (англ. Development of a Software System for Environmental Sound Recognition and Visualization) (далі – програма).

Короткі відомості про галузь застосування: програма призначена для використання в галузі розробки асистивних (допоміжних) технологій, зокрема для забезпечення соціальної адаптації, підвищення рівня безпеки та комфорту людей з вадами слуху в міському та побутовому середовищах.

A.2 Підстави для розробки

Підставою для проведення розробки є завдання на виконання бакалаврської кваліфікаційної роботи, затверджене кафедрою програмних засобів Національного університету «Запорізька політехніка» відповідно до наказу № 139 від 07.04.2026.

Тема розробки: «Розроблення програми для розпізнавання та візуалізації звуків навколишнього середовища» (англ. Development of a Software System for Environmental Sound Recognition and Visualization).

A.3 Призначення розробки

A.3.1 Функціональне призначення

Безперервний фоновий моніторинг та захоплення акустичних даних з мікрофона мобільного пристрою;

Класифікація звукових подій у реальному часі за допомогою згорткової нейронної мережі (CNN);

Інформування користувача про розпізнані звуки через візуальний інтерфейс, тактильний відгук та системні сповіщення.

A.3.2 Експлуатаційне призначення

Використання кінцевими користувачами (людьми з вадами слуху) на персональних мобільних пристроях як засобу попередження про критичні, побутові та фонові події.

A.4 Основні вимоги до програми

A.4.1 Вимоги до функціональних характеристик

Модуль підготовки даних та навчання моделі повинен забезпечувати:

- автоматизоване завантаження аудіоданих із зовнішнього сервісу з відкритою бібліотекою звуків (Freesound.org);
- балансування класів датасету шляхом нарізки довгих аудіофайлів на кліпи (від 2 до 5 секунд) та усунення записів з рівнем гучності нижче встановленого порогу;
- аугментацію аудіоданих для класів з недостатньою вибіркою;
- навчання моделі згорткової нейронної мережі та перетворення аудіо на лог-мел-спектрограми розмірністю $1 \times 64 \times 200$;
- експорт навченої моделі у формат Core ML Neural Network.

Мобільний застосунок повинен забезпечувати:

- перехоплення аудіопотоку у форматі PCM (16 кГц, моно) фіксованими вікнами по 2 секунди;
- обчислення мел-спектрограм без використання графічного процесора (GPU) за допомогою низькорівневого модуля vDSP для обходу системних обмежень фонового режиму iOS;
- класифікацію звуку локально на пристрої через Core ML виключно засобами центрального процесора;
- розподіл розпізнаних звуків за трьома рівнями небезпеки: високий (негайна реакція), середній (увага) та низький (фонові звуки);

- генерацію локальних push-сповіщень та тактильного відгуку залежно від рівня небезпеки розпізнаного звуку;
- можливість індивідуального налаштування (увімкнення або вимкнення) розпізнавання окремих класів звуків через графічний інтерфейс користувача.

A.4.2 Вимоги до надійності

Програма повинна забезпечувати валідаційну точність розпізнавання цільових класів звуків на рівні не менше 75%.

Програма повинна функціонувати повністю автономно та не залежати від наявності підключення до мережі Інтернет під час розпізнавання звуків.

Програма повинна включати алгоритми дедуплікації для уникнення надмірного надсилання сповіщень про один і той самий звук протягом заданого проміжку часу.

Час формування спектрограми та виконання інференсу моделі не повинен перевищувати 2 секунди (час надходження наступного аудіо-буфера).

A.4.3 Умови експлуатації

Програма призначена для експлуатації в нормальних кліматичних умовах експлуатації побутових мобільних пристроїв.

Для коректної роботи мобільного застосунку користувач повинен надати системні дозволи на використання мікрофона та надсилання сповіщень.

А.4.4 Вимоги до складу і параметрів технічних засобів

Для етапу розробки та навчання моделі необхідний персональний комп'ютер на базі процесора архітектури ARM (Apple Silicon) або x86_64, з обсягом оперативної пам'яті не менше 8 ГБ.

Для експлуатації Програми необхідний мобільний пристрій лінійки Apple iPhone з підтримкою нейронних обчислень, наявністю справного мікрофона та системи тактильного відгуку Taptic Engine.

А.4.5 Вимоги до інформаційної та програмної сумісності

Модуль підготовки даних і навчання моделі повинен бути реалізований мовою програмування Python з використанням бібліотек PyTorch, TorchAudio та Coremltools.

Мобільний застосунок призначений для роботи під управлінням операційної системи iOS 17.6 або вище.

Мова реалізації мобільного застосунку – Swift. Інтерфейс користувача повинен бути реалізований з використанням фреймворку SwiftUI.

Системні обмеження: у зв'язку з обмеженнями ОС iOS щодо виконання GPU-обчислень у фоновому режимі сторонніми застосунками (помилка «Insufficient Permission to submit GPU work from background»), реалізація інференсу моделі та DSP-перетворень повинна виконуватись виключно силами центрального процесора пристрою (cpuOnly).

А.4.6 Вимоги до маркування, пакування, транспортування та зберігання

Вимоги до маркування, пакування, транспортування та зберігання не пред'являються у зв'язку з тим, що програмний продукт розповсюджується виключно в цифровому вигляді.

A.5 Вимоги до програмної документації

Розробник повинен підготувати таку документацію:

- технічне завдання;
- пояснювальна записка до бакалаврської кваліфікаційної роботи;
- лістинги текстів програм;
- інструкція користувача.

A.6 Техніко-економічні показники

Проект має яскраво виражену соціальну спрямованість.

Економічна ефективність розробки досягається за рахунок використання відкритих наборів даних (FreeSound.org), що виключає фінансові витрати на етапі збору датасету.

Локальна обробка даних (on-device) повністю виключає витрати на утримання хмарних серверів для виконання інференсу моделі та гарантує приватність даних користувача.

A.7 Стадії та етапи розробки

Розробка повинна бути виконана у такі стадії:

- ескізний проєкт (аналіз предметної області, дослідження системних обмежень фонового розпізнавання в ОС iOS, вибір інструментарію);
- технічний проєкт (збір та підготовка датасету, проєктування архітектури CNN, розробка та тестування алгоритмів навчання);
- робочий проєкт (розробка мобільного застосунку, написання DSP-алгоритмів, інтеграція ML-моделі, створення графічного інтерфейсу);
- впровадження (системне тестування, відлагодження точності розпізнавання, оформлення програмної документації).

A.8 Порядок контролю та приймання

Контроль за виконанням розробки здійснюється керівником кваліфікаційної роботи на кожному з етапів.

Приймання результатів здійснюється шляхом перевірки працездатності мобільного застосунку (успішне розпізнавання цільових звуків), перевірки стабільності роботи у фоновому режимі та публічного захисту результатів перед екзаменаційною комісією.

ДОДАТОК Б
Текст програми

Б.1 Текст файлу balance_dataset.py

```

import random
import subprocess
import numpy as np
import soundfile as sf
from pathlib import Path
from collections import defaultdict

# =====
# 1. КОНФІГУРАЦІЯ
# =====

INPUT_DIR = Path("/Volumes/T7/SoundViz_FreeSound_Train")
OUTPUT_DIR = Path("/Volumes/T7/SoundViz_FreeSound_Balanced")
VALID_DIR = Path("/Volumes/T7/SoundViz_FreeSound_Valid")

MAX_PER_CLASS = 300 # Верхня межа кількості файлів на клас
MIN_FOR_AUGMENT = 300 # Мінімальна кількість файлів; класи нижче — доповнюються
CLIP_DURATION = 5 # Тривалість нарізаних кліпів у секундах
TARGET_SR = 44100 # Частота дискретизації вихідних WAV-файлів, Гц

# Поріг тиші: кліпи з RMS нижче цього значення відкидаються.
# Діапазон [0.0, 1.0]; 0.01 відповідає ~1% від максимальної амплітуди.
SILENCE_THRESHOLD = 0.01

SEED = 42
random.seed(SEED)

# =====
# 2. УТИЛІТИ ДЛЯ РОБОТИ З АУДІО
# =====

def get_duration(path: Path) -> float:
    """
    Повертає тривалість аудіофайлу в секундах через ffprobe.

    Args:
        path (Path): Шлях до аудіофайлу.

    Returns:
        float: Тривалість у секундах, або 0.0 у разі помилки.
    """
    try:
        result = subprocess.run(
            [
                "ffprobe", "-v", "quiet",
                "-show_entries", "format=duration",
                "-of", "csv=p=0",
                str(path)
            ],
            capture_output=True, text=True, timeout=10
        )
        return float(result.stdout.strip())
    except Exception:
        return 0.0

def is_silence(path: Path) -> bool:
    """
    Перевіряє чи є кліп переважно тишею на основі RMS амплітуди.

    Зчитує WAV-файл та обчислює середньоквадратичне значення (RMS).
    Якщо RMS нижче SILENCE_THRESHOLD — кліп вважається тишею.

    Args:
    """

```

```

    path (Path): Шлях до WAV-файлу.

Returns:
    bool: True якщо кліп є тишею і має бути відкинутий.
"""
try:
    data, _ = sf.read(str(path), dtype='float32', always_2d=True)
    rms = float(np.sqrt(np.mean(data ** 2)))
    return rms < SILENCE_THRESHOLD
except Exception:
    return False

def split_clip(src: Path, dst_dir: Path, base_name: str) -> list:
    """
    Розрізає аудіофайл на послідовні WAV-кліпи тривалістю CLIP_DURATION секунд.

    Підтримує MP3 та WAV на вході, завжди виводить WAV.
    Кліпи з тишею автоматично видаляються.
    Якщо тривалість файлу менша за 1 секунду – файл відкидається.
    Якщо тривалість менша за CLIP_DURATION – конвертується без нарізки.

    Args:
        src (Path): Вхідний аудіофайл (MP3 або WAV).
        dst_dir (Path): Каталог для збереження кліпів.
        base_name (str): Базова назва для вихідних файлів (без розширення).

    Returns:
        list[Path]: Список шляхів до успішно створених не-тихих WAV-кліпів.
    """
    duration = get_duration(src)

    if duration < 1.0:
        return []

    if duration < CLIP_DURATION:
        dst = dst_dir / f"{base_name}.wav"
        subprocess.run(
            [
                "ffmpeg", "-y", "-v", "quiet",
                "-i", str(src),
                "-ar", str(TARGET_SR), "-ac", "1",
                str(dst)
            ],
            timeout=60
        )
        if dst.exists() and dst.stat().st_size > 1000:
            if is_silence(dst):
                dst.unlink()
                return []
            return [dst]
        return []

    outputs = []
    offset = 0
    part = 1

    while offset + CLIP_DURATION <= duration + 0.5:
        dst = dst_dir / f"{base_name}_p{part}.wav"
        try:
            subprocess.run(
                [
                    "ffmpeg", "-y", "-v", "quiet",
                    "-i", str(src),
                    "-ss", str(offset),
                    "-t", str(CLIP_DURATION),
                    "-ar", str(TARGET_SR), "-ac", "1",
                    str(dst)
                ],
                timeout=60
            )

```

```

except subprocess.TimeoutExpired:
    print(f" Пропущено (timeout): {src.name}")
    if dst.exists():
        dst.unlink()
    break

if dst.exists() and dst.stat().st_size > 1000:
    # Відкидаємо кліпи з тишею
    if is_silence(dst):
        dst.unlink()
    else:
        outputs.append(dst)

part    += 1
offset += CLIP_DURATION

return outputs

# =====
# 3. ФУНКЦІЇ АУГМЕНТАЦІЇ
# =====

def _run_ffmpeg(cmd: list, dst: Path) -> bool:
    """
    Виконує команду ffmpeg та перевіряє успішність результату.

    Args:
        cmd (list): Список аргументів команди ffmpeg.
        dst (Path): Очікуваний вихідний файл.

    Returns:
        bool: True якщо файл успішно створено, False – інакше.
    """
    try:
        subprocess.run(cmd, timeout=60, check=False)
        return dst.exists() and dst.stat().st_size > 1000
    except Exception:
        return False

def augment_speed(src: Path, dst: Path, factor: float) -> bool:
    """
    Змінює швидкість відтворення аудіо з відповідною зміною пітчу.

    Args:
        src (Path): Вхідний WAV-файл.
        dst (Path): Вихідний WAV-файл.
        factor (float): Коефіцієнт швидкості (< 1.0 – повільніше, > 1.0 – швидше).

    Returns:
        bool: True у разі успіху.
    """
    return _run_ffmpeg([
        "ffmpeg", "-y", "-v", "quiet",
        "-i", str(src),
        "-filter:a", f"atempo={factor}",
        "-ar", str(TARGET_SR), "-ac", "1",
        str(dst)
    ], dst)

def augment_noise(src: Path, dst: Path, noise_level: float = 0.005) -> bool:
    """
    Додає білий шум до аудіосигналу шляхом міксування.

    Args:
        src (Path): Вхідний WAV-файл.
        dst (Path): Вихідний WAV-файл.
        noise_level (float): Амплітуда шуму відносно сигналу.
    """

```

```

Returns:
    bool: True у разі успіху.
"""
return _run_ffmpeg([
    "ffmpeg", "-y", "-v", "quiet",
    "-i", str(src),
    "-filter_complex",
    (
        f"anoisesrc=d={CLIP_DURATION}:c=white:a={noise_level}[noise];"
        "[0:a][noise]amix=inputs=2:duration=first:weights=1 0.3"
    ),
    "-ar", str(TARGET_SR), "-ac", "1",
    str(dst)
], dst)

def augment_pitch(src: Path, dst: Path, semitones: float) -> bool:
    """
    Зсуває пітч аудіо на задану кількість напівтонів.

    Args:
        src (Path): Вхідний WAV-файл.
        dst (Path): Вихідний WAV-файл.
        semitones (float): Зсув пітчу в напівтонах (від'ємне – нижче).

    Returns:
        bool: True у разі успіху.
    """
    factor = 2 ** (semitones / 12.0)
    new_rate = int(TARGET_SR * factor)
    return _run_ffmpeg([
        "ffmpeg", "-y", "-v", "quiet",
        "-i", str(src),
        "-filter:a", f"asetrate={new_rate},aresample={TARGET_SR}",
        "-ar", str(TARGET_SR), "-ac", "1",
        str(dst)
    ], dst)

def augment_volume(src: Path, dst: Path, db: float) -> bool:
    """
    Змінює гучність аудіо на задану кількість децибел.

    Args:
        src (Path): Вхідний WAV-файл.
        dst (Path): Вихідний WAV-файл.
        db (float): Зміна гучності в дБ (від'ємне – тихіше).

    Returns:
        bool: True у разі успіху.
    """
    return _run_ffmpeg([
        "ffmpeg", "-y", "-v", "quiet",
        "-i", str(src),
        "-filter:a", f"volume={db}dB",
        "-ar", str(TARGET_SR), "-ac", "1",
        str(dst)
    ], dst)

# =====
# 4. ГОЛОВНА ФУНКЦІЯ
# =====

def main() -> None:
    """
    Виконує повний конвеєр балансування датасету:

    1. Нарізка MP3/WAV → WAV-кліпи з фільтрацією тиші
    2. Обмеження великих класів до MAX_PER_CLASS файлів
    3. Циклічна аугментація малих класів до MIN_FOR_AUGMENT файлів
    """

```

```

4. Виведення підсумкової статистики
"""
print("=" * 65)
print("  SoundViz Dataset Balancer")
print("=" * 65)
print(f"  Вхід          : {INPUT_DIR}")
print(f"  Вихід         : {OUTPUT_DIR}")
print(f"  Ліміт на клас : {MAX_PER_CLASS}")
print(f"  Попіг тиші    : {SILENCE_THRESHOLD}")
print("=" * 65)

if not INPUT_DIR.exists():
    print(f"Помилка: каталог не знайдено: {INPUT_DIR}")
    return

classes = sorted([d.name for d in INPUT_DIR.iterdir() if d.is_dir()])
class_files: dict = defaultdict(list)

# -----
# Крок 1: Нарізка
# -----
print(f"\nКрок 1: Нарізка аудіофайлів на WAV-кліпи по {CLIP_DURATION} секунд\n")

for i, cls_name in enumerate(classes):
    cls_input = INPUT_DIR / cls_name
    cls_output = OUTPUT_DIR / cls_name
    cls_output.mkdir(parents=True, exist_ok=True)

    audio_files = sorted(
        list(cls_input.glob("*.mp3")) +
        list(cls_input.glob("*.wav"))
    )
    total_clips = 0

    for audio in audio_files:
        existing = list(cls_output.glob(f"{audio.stem}*.wav"))
        if existing:
            class_files[cls_name].extend(existing)
            total_clips += len(existing)
            continue

        clips = split_clip(audio, cls_output, audio.stem)
        class_files[cls_name].extend(clips)
        total_clips += len(clips)

    print(
        f"  [{i+1:3d}/{len(classes)}] {cls_name:<30} "
        f"{len(audio_files):>4} файлів → {total_clips:>4} кліпів"
    )

# -----
# Крок 2: Обмеження великих класів
# -----
print(f"\nКрок 2: Обмеження до {MAX_PER_CLASS} файлів на клас\n")

capped = 0
for cls_name in classes:
    files = class_files[cls_name]
    if len(files) > MAX_PER_CLASS:
        random.shuffle(files)
        for f in files[MAX_PER_CLASS:]:
            if f.exists():
                f.unlink()
        class_files[cls_name] = files[:MAX_PER_CLASS]
        print(f"  {cls_name:<30} {len(files):>4} → {MAX_PER_CLASS}")
        capped += 1

if capped == 0:
    print("  Жоден клас не перевищує ліміт.")

# -----

```

```

# Крок 3: Циклічна аугментация малих класів
# -----
print(f"\nКрок 3: Аугментация класів з менш ніж {MIN_FOR_AUGMENT} файлів\n")

augment_funcs = [
    ("speed09", lambda s, d: augment_speed(s, d, 0.9)),
    ("speed11", lambda s, d: augment_speed(s, d, 1.1)),
    ("noise", lambda s, d: augment_noise(s, d, 0.005)),
    ("pitch_up", lambda s, d: augment_pitch(s, d, 1.5)),
    ("pitch_dn", lambda s, d: augment_pitch(s, d, -1.5)),
    ("loud", lambda s, d: augment_volume(s, d, 3.0)),
    ("quiet", lambda s, d: augment_volume(s, d, -3.0)),
]

augmented_classes = 0

for cls_name in classes:
    files = class_files[cls_name]
    current_count = len(files)

    if current_count >= MIN_FOR_AUGMENT:
        continue

    needed = MIN_FOR_AUGMENT - current_count
    cls_dir = OUTPUT_DIR / cls_name
    added = 0
    src_cycle = list(files)
    cycle_pass = 0

    while added < needed:
        if not src_cycle:
            src_cycle = list(files)
            cycle_pass += 1

        src = src_cycle.pop(0)

        for aug_name, aug_func in augment_funcs:
            if added >= needed:
                break

            dst = cls_dir / f"{src.stem}_{aug_name}_c{cycle_pass}.wav"

            if not dst.exists() and aug_func(src, dst):
                class_files[cls_name].append(dst)
                added += 1

        print(
            f" {cls_name:<30} {current_count:>3} + {added:>3} "
            f"={current_count + added:>3}"
        )
        augmented_classes += 1

if augmented_classes == 0:
    print(" Всі класи вже мають достатньо файлів.")

# -----
# Крок 4: Конвертація валідаційного набору MP3 → WAV
# -----
print(f"\nКрок 4: Конвертація валідаційного набору MP3 → WAV\n")

if not VALID_DIR.exists():
    print(f" Увага: валідаційний каталог не знайдено: {VALID_DIR}")
else:
    mp3_files = list(VALID_DIR.rglob("*.mp3"))
    if not mp3_files:
        print(" MP3-файли не знайдено – валідаційний набір вже у форматі WAV.")
    else:
        converted = 0
        errors = 0
        for mp3_path in mp3_files:
            wav_path = mp3_path.with_suffix(".wav")

```

```

try:
    subprocess.run(
        [
            "ffmpeg", "-y", "-v", "quiet",
            "-i", str(mp3_path),
            "-ar", str(TARGET_SR), "-ac", "1",
            str(wav_path)
        ],
        timeout=60, check=False
    )
    if wav_path.exists() and wav_path.stat().st_size > 1000:
        mp3_path.unlink()
        converted += 1
    else:
        errors += 1
except Exception:
    errors += 1

print(f" Конвертовано : {converted} файлів")
if errors > 0:
    print(f" Помилка      : {errors}")

# -----
# Підсумок
# -----
counts    = [len(class_files[c]) for c in classes]
total     = sum(counts)
min_count = min(counts)
max_count = max(counts)
min_class = classes[counts.index(min_count)]
max_class = classes[counts.index(max_count)]

print(f"\n{'=' * 65}")
print("Підсумкова статистика:\n")
print(f" Кількість класів : {len(classes)}")
print(f" Всього файлів    : {total}")
print(f" Мінімум на клас  : {min_count:>4} ({min_class})")
print(f" Максимум на клас : {max_count:>4} ({max_class})")
print(f" Середнє на клас  : {total // len(classes):>4}")
print(f"\n Результат збережено: {OUTPUT_DIR}")
print(f"{'=' * 65}")

if __name__ == "__main__":
    main()

```

Б.2 Текст файлу train_model.py

```

import os
import glob
import random

import torch
import torch.nn as nn
import torch.optim as optim
import torchaudio
import torchaudio.transforms as T
import soundfile as sf
import coremltools as ct
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler
from tqdm import tqdm

# Фіксуємо seed для відтворюваності експериментів
random.seed(42)
torch.manual_seed(42)

```

```

# =====
# 1. КОНФІГУРАЦІЯ
# =====

# Параметри аудіо
TARGET_SR      = 16000    # Частота дискретизації моделі, Гц
TARGET_SAMPLES = 32000    # Довжина вхідного вікна (2 секунди × 16000 Гц)

# Параметри мел-спектрограми
N_MELS         = 64        # Кількість мел-фільтрів
N_FFT          = 1024      # Розмір вікна ШПФ
HOP_LENGTH     = 160       # Крок між вікнами (10 мс → 100 фреймів/сек)
N_FRAMES       = 200       # Кількість часових фреймів (2 сек × 100 фреймів/сек)
F_MIN          = 20        # Нижня межа частотного діапазону, Гц
F_MAX          = 8000      # Верхня межа частотного діапазону, Гц
TOP_DB         = 80        # Динамічний діапазон у децибелах

# Параметри навчання
NUM_EPOCHS     = 50
BATCH_SIZE     = 64
LR             = 1e-3      # Початковий learning rate
WEIGHT_DECAY   = 1e-4      # L2-регуляризація

# Шляхи до даних та збереження
TRAIN_DIR      = "/Volumes/T7/SoundViz_FreeSound_Balanced"
VAL_DIR        = "/Volumes/T7/SoundViz_FreeSound_Valid"
OUTPUT_MODEL   = "/Users/anakath/SoundViz/Model_FreeSound_v2/SoundVizClassifier_v2.mlmodel"
CHECKPOINT_PATH = "/Users/anakath/SoundViz/Model_FreeSound_v2/best_model_v2.pth"

# =====
# 2. ПЕРЕТВОРЕННЯ АУДІО
# =====

# Побудова мел-спектрограми (виконується на CPU, незалежно від пристрою навчання)
mel_transform = T.MelSpectrogram(
    sample_rate=TARGET_SR,
    n_fft=N_FFT,
    hop_length=HOP_LENGTH,
    n_mels=N_MELS,
    f_min=F_MIN,
    f_max=F_MAX,
)
amplitude_to_db = T.AmplitudeToDB(stype='power', top_db=TOP_DB)

# =====
# 3. НАБІР ДАНИХ
# =====

class AudioDataset(Dataset):
    """
    Набір даних для класифікації звуків.

    Завантажує WAV-файли, конвертує їх у мел-спектрограми та повертає
    пари (спектрограма, мітка класу) для навчання нейронної мережі.

    Структура каталогу:
    dataset_dir/
        class_name_1/
            audio1.wav
            audio2.wav
        class_name_2/
            ...

    Args:
        file_paths (list[str]): Список шляхів до WAV-файлів.
        labels (list[str]): Список відповідних міток класів.
        class_to_idx (dict): Словник відображення назви класу на індекс.
        augment (bool): Якщо True – застосовує аугментацію даних
    """

```

```

        (лише для тренувального набору).
"""

def __init__(self, file_paths: list, labels: list,
              class_to_idx: dict, augment: bool = False):
    # Попередня фільтрація: відкидаємо порожні або пошкоджені файли
    valid_pairs = []
    skipped = 0
    for path, label in zip(file_paths, labels):
        try:
            info = sf.info(path)
            if info.frames > 0:
                valid_pairs.append((path, label))
            else:
                skipped += 1
        except Exception:
            skipped += 1

    if skipped > 0:
        print(f" Пропущено {skipped} пошкоджених або порожніх файлів.")

    self.file_paths = [p for p, _ in valid_pairs]
    self.labels = [l for _, l in valid_pairs]
    self.class_to_idx = class_to_idx
    self.augment = augment

def __len__(self) -> int:
    return len(self.file_paths)

def __getitem__(self, idx: int) -> tuple:
    path = self.file_paths[idx]
    label = self.class_to_idx[self.labels[idx]]

    try:
        waveform = self._load_waveform(path)
        if self.augment:
            waveform = self._augment(waveform)
        spectrogram = self._to_spectrogram(waveform)
        return spectrogram, label
    except Exception:
        # У разі непередбаченої помилки – повертаємо інший випадковий зразок
        return self.__getitem__(random.randint(0, len(self) - 1))

def _load_waveform(self, path: str) -> torch.Tensor:
    """
    Завантажує WAV-файл та приводить його до стандартного формату:
    моно, 16 кГц, довжина рівно TARGET_SAMPLES відліків.
    """
    data, sr = sf.read(path, dtype='float32', always_2d=True)

    if data.shape[0] == 0:
        raise ValueError(f"Порожній файл: {path}")

    # Перетворення на тензор форми [канали, відліки]
    waveform = torch.from_numpy(data.T)

    # Зведення до моно шляхом усереднення каналів
    if waveform.shape[0] > 1:
        waveform = waveform.mean(dim=0, keepdim=True)

    # Ресемплінг до цільової частоти дискретизації
    if sr != TARGET_SR:
        waveform = torchaudio.transforms.Resample(sr, TARGET_SR)(waveform)

    waveform = waveform.squeeze(0) # форма: [відліки]

    # Нормалізація амплітуди до діапазону [-1, 1]
    max_val = waveform.abs().max()
    if max_val > 0:
        waveform = waveform / max_val

```

```

# Приведення до фіксованої довжини: обрізання або доповнення нулями
if waveform.shape[0] > TARGET_SAMPLES:
    start = random.randint(0, waveform.shape[0] - TARGET_SAMPLES)
    waveform = waveform[start: start + TARGET_SAMPLES]
elif waveform.shape[0] < TARGET_SAMPLES:
    pad = TARGET_SAMPLES - waveform.shape[0]
    waveform = torch.nn.functional.pad(waveform, (0, pad))

return waveform

def _augment(self, waveform: torch.Tensor) -> torch.Tensor:
    """
    Аугментация аудиосигнала для підвищення узагальнювальної здатності моделі.

    Застосовуються три незалежні перетворення з заданою ймовірністю:
    - Додавання гаусівського шуму (імітація фоновому шуму мікрофона)
    - Випадкова зміна гучності (імітація різних відстаней до джерела)
    - Циклічний часовий зсув (імітація різних моментів початку звуку)
    """
    # Гаусівський шум ( $\sigma = 0.005$ )
    if random.random() < 0.3:
        waveform = waveform + torch.randn_like(waveform) * 0.005

    # Випадкова зміна гучності в діапазоні [0.7, 1.3]
    if random.random() < 0.3:
        gain = random.uniform(0.7, 1.3)
        waveform = (waveform * gain).clamp(-1.0, 1.0)

    # Циклічний зсув по часу в межах  $\pm 0.1$  секунди ( $\pm 1600$  відліків)
    if random.random() < 0.2:
        shift = random.randint(-1600, 1600)
        waveform = torch.roll(waveform, shift)

    return waveform

def _to_spectrogram(self, waveform: torch.Tensor) -> torch.Tensor:
    """
    Перетворює waveform на лог-мел-спектрограму.

    Повертає тензор форми [1, N_MELS, N_FRAMES] з нормалізованими
    значеннями відносно власного мінімуму та максимуму (per-sample).
    """
    spec = mel_transform(waveform) # форма: [N_MELS, frames]
    spec = amplitude_to_db(spec) # перетворення потужності в дБ

    # Приведення до фіксованої кількості фреймів
    if spec.shape[1] > N_FRAMES:
        spec = spec[:, :N_FRAMES]
    elif spec.shape[1] < N_FRAMES:
        spec = torch.nn.functional.pad(spec, (0, N_FRAMES - spec.shape[1]))

    # Нормалізація per-sample до діапазону [0, 1].
    # Зберігає внутрішню частотну структуру звуку
    # незалежно від абсолютної гучності запису.
    spec = spec - spec.min()
    if spec.max() > 0:
        spec = spec / spec.max()

    return spec.unsqueeze(0) # форма: [1, N_MELS, N_FRAMES]

# =====
# 4. АРХІТЕКТУРА НЕЙРОННОЇ МЕРЕЖІ
# =====

class ResidualBlock(nn.Module):
    """
    Залишковий блок (Residual Block) для двовимірних згортків.

    Реалізує з'єднання skip connection:
    output = ReLU(F(x) + shortcut(x))
    """

```

де $F(x)$ – послідовність двох згорткових шарів із батч-нормалізацією,
а $\text{shortcut}(x)$ – проекція вхідного тензора у разі зміни розмірності.

```

Args:
    in_channels (int): Кількість вхідних каналів.
    out_channels (int): Кількість вихідних каналів.
    stride (int): Крок першої згортки для зменшення просторової розмірності.
"""

def __init__(self, in_channels: int, out_channels: int, stride: int = 1):
    super().__init__()

    self.conv_block = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=3,
                  stride=stride, padding=1, bias=False),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True),
        nn.Conv2d(out_channels, out_channels, kernel_size=3,
                  stride=1, padding=1, bias=False),
        nn.BatchNorm2d(out_channels),
    )

    # Проекційне з'єднання застосовується якщо розмірність змінилась
    self.shortcut = nn.Sequential()
    if stride != 1 or in_channels != out_channels:
        self.shortcut = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1,
                      stride=stride, bias=False),
            nn.BatchNorm2d(out_channels),
        )

    self.relu = nn.ReLU(inplace=True)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    return self.relu(self.conv_block(x) + self.shortcut(x))

```

```

class AudioCNN(nn.Module):
    """
    Згорткова нейронна мережа для класифікації звуків на основі мел-спектрограм.

    Архітектура: початковий шар (stem) → чотири залишкові блоки зі зменшенням
    просторової розмірності → глобальне усереднення → повнозв'язний класифікатор.

    Вхід: тензор форми [batch, 1, 64, 200] (мел-спектрограма)
    Вихід: тензор форми [batch, num_classes] (логіти класів)

    Args:
        num_classes (int): Кількість класів для класифікації.
    """

    def __init__(self, num_classes: int):
        super().__init__()

        # Початковий шар: [B, 1, 64, 200] → [B, 32, 64, 200]
        self.stem = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
        )

        # Залишкові блоки зі зменшенням просторової розмірності (stride=2)
        self.layer1 = ResidualBlock(32, 64, stride=2) # → [B, 64, 32, 100]
        self.layer2 = ResidualBlock(64, 128, stride=2) # → [B, 128, 16, 50]
        self.layer3 = ResidualBlock(128, 256, stride=2) # → [B, 256, 8, 25]
        self.layer4 = ResidualBlock(256, 256, stride=2) # → [B, 256, 4, 13]

        # Глобальне усереднення зводить просторові виміри до 1×1
        self.global_pool = nn.AdaptiveAvgPool2d(1)

```

```

        # Dropout для зменшення перенавчання (p=0.5)
        self.dropout = nn.Dropout(0.5)

        # Повнозв'язний класифікатор
        self.classifier = nn.Linear(256, num_classes)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.stem(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.global_pool(x)
        x = torch.flatten(x, 1)
        x = self.dropout(x)
        return self.classifier(x)

class AudioCNNWithSoftmax(nn.Module):
    """
    Обгортка над AudioCNN, що додає шар Softmax для експорту у Core ML.

    Core ML ClassifierConfig очікує ймовірності (сума = 1), тому
    Softmax застосовується безпосередньо перед трасуванням моделі.

    Args:
        model (AudioCNN): Навчена модель без Softmax.
    """

    def __init__(self, model: AudioCNN):
        super().__init__()
        self.model = model
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.softmax(self.model(x))

# =====
# 5. ДОПОМІЖНІ ФУНКЦІЇ
# =====

def load_files_from_dir(directory: str) -> tuple:
    """
    Рекурсивно збирає WAV-файли з підкаталогів.

    Очікувана структура: directory/<class_name>/*.wav

    Args:
        directory (str): Кореневий каталог датасету.

    Returns:
        tuple: (список шляхів до файлів, список відповідних міток класів)
    """
    files = glob.glob(os.path.join(directory, "*", "*.wav"))
    labels = [os.path.basename(os.path.dirname(f)) for f in files]
    return files, labels

def build_weighted_sampler(labels: list, class_to_idx: dict) -> WeightedRandomSampler:
    """
    Буде зважений семплер для вирівнювання розподілу класів у тренувальних батчах.

    Класи з меншою кількістю зразків отримують вищу вагу, що компенсує
    нерівномірний розподіл та покращує якість навчання на малих класах.

    Args:
        labels (list): Список міток тренувального набору.
        class_to_idx (dict): Словник відображення назви класу на індекс.

    Returns:

```

```

        WeightedRandomSampler: Семплер для DataLoader.
"""
classes      = list(class_to_idx.keys())
class_counts = [labels.count(c) for c in classes]

# Вага зразка обернено пропорційна кількості зразків у його класі
weights = [1.0 / class_counts[class_to_idx[l]] for l in labels]
return WeightedRandomSampler(
    weights=weights,
    num_samples=len(weights),
    replacement=True
)

# =====
# 6. ГОЛОВНА ФУНКЦІЯ НАВЧАННЯ
# =====

def main():
    """
    Виконує повний цикл навчання та експорту моделі:

        1. Завантаження та валідація датасету
        2. Ініціалізація моделі, функції втрат та оптимізатора
        3. Навчання з валідацією та збереженням найкращої моделі
        4. Рання зупинка при відсутності прогресу
        5. Експорт навченої моделі у формат Core ML (.mlmodel)
    """
    device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
    print(f"Пристрій обчислень: {device}")

    # -----
    # Завантаження файлів
    # -----
    train_files, train_labels = load_files_from_dir(TRAIN_DIR)
    val_files, val_labels = load_files_from_dir(VAL_DIR)

    if not train_files:
        print(f"Помилка: файли не знайдено у {TRAIN_DIR}")
        return
    if not val_files:
        print(f"Помилка: файли не знайдено у {VAL_DIR}")
        return

    # Класи визначаються на основі тренувального набору
    classes = sorted(set(train_labels))
    class_to_idx = {cls: idx for idx, cls in enumerate(classes)}

    # Валідаційний набір фільтрується до відомих класів
    val_pairs = [(f, l) for f, l in zip(val_files, val_labels)
                  if l in class_to_idx]
    val_files, val_labels = zip(*val_pairs) if val_pairs else ([], [])

    print(f"Тренувальний набір : {len(train_files)} файлів")
    print(f"Валідаційний набір : {len(val_files)} файлів")
    print(f"Кількість класів : {len(classes)}")
    print("Перевірка цілісності файлів...")

    # -----
    # Датасети та завантажувачі даних
    # -----
    train_dataset = AudioDataset(
        train_files, train_labels, class_to_idx, augment=True
    )
    val_dataset = AudioDataset(
        val_files, val_labels, class_to_idx, augment=False
    )

    print(f"Після фільтрації: train={len(train_dataset)}, val={len(val_dataset)}")

    # Зважений семплер вирівнює представленість класів у батчах

```

```

sampler = build_weighted_sampler(train_dataset.labels, class_to_idx)

train_loader = DataLoader(
    train_dataset, batch_size=BATCH_SIZE,
    sampler=sampler, num_workers=0
)
val_loader = DataLoader(
    val_dataset, batch_size=BATCH_SIZE,
    shuffle=False, num_workers=0
)

# -----
# Модель, функція втрат, оптимізатор, планувальник learning rate
# -----
model = AudioCNN(num_classes=len(classes)).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=LR, weight_decay=WEIGHT_DECAY)

# ReduceLRonPlateau: зменшує LR у 2 рази якщо val accuracy
# не покращується протягом patience epoch
scheduler = optim.lr_scheduler.ReduceLRonPlateau(
    optimizer, mode='max', patience=5, factor=0.5
)

best_val_acc = 0.0
patience_counter = 0
early_stop_patience = 15 # зупинка якщо немає прогресу 15 epoch поспіль

# Створюємо папку для збереження моделі якщо не існує
os.makedirs(os.path.dirname(CHECKPOINT_PATH), exist_ok=True)

print("\nНавчання розпочато...\n")

for epoch in range(NUM_EPOCHS):

    # -----
    # Фаза навчання
    # -----
    model.train()
    train_loss = 0.0

    for specs, targets in tqdm(
        train_loader, desc=f"Epoch {epoch+1}/{NUM_EPOCHS} [Train]"
    ):
        specs, targets = specs.to(device), targets.to(device)
        optimizer.zero_grad()
        loss = criterion(model(specs), targets)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    # -----
    # Фаза валідації
    # -----
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for specs, targets in tqdm(
            val_loader, desc=f"Epoch {epoch+1}/{NUM_EPOCHS} [Val]"
        ):
            specs, targets = specs.to(device), targets.to(device)
            outputs = model(specs)
            val_loss += criterion(outputs, targets).item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    val_acc = 100.0 * correct / total

```

```

scheduler.step(val_acc)

print(
    f"\nEpoch {epoch+1}/{NUM_EPOCHS} | "
    f"Train Loss: {train_loss/len(train_loader):.4f} | "
    f"Val Loss: {val_loss/len(val_loader):.4f} | "
    f"Val Accuracy: {val_acc:.2f}%"
)

# Збереження найкращої моделі
if val_acc > best_val_acc:
    best_val_acc = val_acc
    patience_counter = 0
    torch.save(model.state_dict(), CHECKPOINT_PATH)
    print(f"Збережено нову найкращу модель (Accuracy: {best_val_acc:.2f}%)")
else:
    patience_counter += 1
    if patience_counter >= early_stop_patience:
        print(
            f"\nРання зупинка: відсутнє покращення протягом "
            f"{early_stop_patience} епох."
        )
        break

print(f"\nНайкраща точність на валідаційному наборі: {best_val_acc:.2f}%",)

# =====
# 7. ЕКСПОРТ У ФОРМАТ CORE ML
# =====
print("\nЕкспорт моделі у формат Core ML...")

# Завантажуємо найкращі ваги
model.load_state_dict(torch.load(CHECKPOINT_PATH, map_location="cpu"))
model.to("cpu").eval()

# Обгортаємо Softmax для коректного відображення ймовірностей у Core ML
export_model = AudioCNNWithSoftmax(model)
export_model.eval()

# Трасування через torch.jit для сумісності з coremltools
dummy_input = torch.randn(1, 1, N_MELS, N_FRAMES)
traced_model = torch.jit.trace(export_model, dummy_input)

# Конвертація у формат NeuralNetwork (сумісний з iOS 14+)
mlmodel = ct.convert(
    traced_model,
    convert_to="neuralnetwork",
    inputs=[ct.TensorType(
        name="melSpectrogram",
        shape=(1, 1, N_MELS, N_FRAMES)
    )],
    outputs=[ct.TensorType(name="targetProbability")],
    classifier_config=ct.ClassifierConfig(
        class_labels=classes,
        predicted_feature_name="target",
        predicted_probabilities_output="targetProbability"
    )
)

os.makedirs(os.path.dirname(OUTPUT_MODEL), exist_ok=True)
mlmodel.save(OUTPUT_MODEL)
print(f"\nМодель збережено : {OUTPUT_MODEL}")
print(f"Формат входу : melSpectrogram [1, 1, {N_MELS}, {N_FRAMES}]")
print(f"Кількість класів : {len(classes)}")

if __name__ == "__main__":
    main()

```

Б.3 Текст файлу SoundManager.swift

```
import AVFoundation
import Accelerate
import Combine
import CoreML
import Foundation

// MARK: - Mel Spectrogram Calculator

/// Калькулятор мел-спектрограм для перетворення аудіосигналу в ознаковий простір.
///
/// Виконує обчислення мел-спектрограми з вхідного масиву PCM-семплів:
/// 1. Розбиття сигналу на фрейми із застосуванням віконної функції Ганна.
/// 2. Обчислення спектру потужності через дискретне перетворення Фур'є (DFT).
/// 3. Застосування мел-фільтрбанку для перетворення у мел-шкалу.
/// 4. Нормалізація в децибелах з обмеженням динамічного діапазону.
///
/// Параметри відповідають конфігурації моделі ``SoundVizClassifier``:
/// - Частота дискретизації: 16 кГц
/// - FFT: 1024 точки, крок 160 семплів
/// - Мел-фільтри: 64, діапазон 20-8000 Гц
/// - Вихід: нормалізований тензор 64×200
private final class MelSpectrogramCalculator {

    let sampleRate: Float = 16000
    let nFFT: Int = 1024
    let hopLength: Int = 160
    let nMels: Int = 64
    let fMin: Float = 20
    let fMax: Float = 8000
    let nFrames: Int = 200
    let topDB: Float = 80

    private let fftSetup: vDSP_DFT_Setup
    private let hannWindow: [Float]
    private let melFilterbank: [[Float]]

    init() {
        var window = [Float](repeating: 0, count: nFFT)
        vDSP_hann_window(&window, vDSP_Length(nFFT), Int32(vDSP_HANN_NORM))
        self.hannWindow = window

        self.fftSetup = vDSP_DFT_zop_CreateSetup(
            nil, vDSP_Length(nFFT),
            vDSP_DFT_Direction.FORWARD
        )!

        self.melFilterbank = MelSpectrogramCalculator.buildMelFilterbank(
            sampleRate: 16000, nFFT: 1024, nMels: 64, fMin: 20, fMax: 8000
        )
    }

    deinit { vDSP_DFT_DestroySetup(fftSetup) }

    /// Обчислює мел-спектрограму з масиву PCM-семплів.
    ///
    /// - Parameter samples: Масив нормалізованих аудіосемплів (float32, 16 кГц).
    /// - Returns: Одновимірний масив розміром `nMels × nFrames` (64×200) у порядку
    mel-major.
    func compute(samples: [Float]) -> [Float] {
        var allFrames = [[Float]]()

        var offset = 0
        while offset + nFFT <= samples.count {
            let frame = Array(samples[offset ..< offset + nFFT])
            let powerSpectrum = computePowerSpectrum(frame: frame)
```

```

        let melFrame      = applyMelFilterbank(spectrum: powerSpectrum)
        allFrames.append(melFrame)
        offset += hopLength
    }

    var frames = allFrames
    if frames.count > nFrames {
        frames = Array(frames.prefix(nFrames))
    } else {
        let emptyFrame = [Float](repeating: 1e-10, count: nMels)
        while frames.count < nFrames { frames.append(emptyFrame) }
    }

    var specDB = frames.map { row in
        row.map { val in 10.0 * log10(max(val, 1e-10)) }
    }

    let maxVal      = specDB.flatMap { $0 }.max() ?? 0
    let minBound = maxVal - topDB
    specDB = specDB.map { row in row.map { max($0, minBound) } }

    let allVals      = specDB.flatMap { $0 }
    let globalMin = allVals.min() ?? 0
    let globalMax = allVals.max() ?? 1
    let range      = globalMax - globalMin

    var result = [Float](repeating: 0, count: nMels * nFrames)
    for mel in 0 ..< nMels {
        for frame in 0 ..< nFrames {
            let val = specDB[frame][mel]
            result[mel * nFrames + frame] = range > 0 ? (val - globalMin) / range
        }
    }

    return result
}

/// Обчислює спектр потужності для одного фрейму.
///
/// Застосовує вікно Ганна, виконує DFT та повертає квадрат модуля
/// для перших `nFFT/2 + 1` частотних бінів.
///
/// - Parameter frame: Фрейм аудіо довжиною `nFFT`.
/// - Returns: Масив потужностей розміром `nFFT/2 + 1`.
private func computePowerSpectrum(frame: [Float]) -> [Float] {
    let halfFFT = nFFT / 2 + 1

    var windowed = [Float](repeating: 0, count: nFFT)
    vDSP_vmul(frame, 1, hannWindow, 1, &windowed, 1, vDSP_Length(nFFT))

    var realIn = windowed
    var imagIn = [Float](repeating: 0, count: nFFT)
    var realOut = [Float](repeating: 0, count: nFFT)
    var imagOut = [Float](repeating: 0, count: nFFT)

    vDSP_DFT_Execute(fftSetup, &realIn, &imagIn, &realOut, &imagOut)

    var power = [Float](repeating: 0, count: halfFFT)
    for i in 0 ..< halfFFT {
        power[i] = realOut[i] * realOut[i] + imagOut[i] * imagOut[i]
    }

    let scale = 1.0 / Float(nFFT)
    vDSP_vsmul(power, 1, [scale], &power, 1, vDSP_Length(halfFFT))

    return power
}

/// Застосовує мел-фільтрбанк до спектру потужності.
///

```

```

/// - Parameter spectrum: Спектр потужності одного фрейму.
/// - Returns: Массив енергій для кожного мел-фільтру розміром `nMels`.
private func applyMelFilterbank(spectrum: [Float]) -> [Float] {
    var melEnergies = [Float](repeating: 0, count: nMels)
    for m in 0 ..< nMels {
        var energy: Float = 0
        for k in 0 ..< spectrum.count {
            energy += melFilterbank[m][k] * spectrum[k]
        }
        melEnergies[m] = energy
    }
    return melEnergies
}

/// Перетворює частоту з герців у мел-шкалу (формула O'Shaughnessy).
private static func hzToMel(_ hz: Float) -> Float {
    2595.0 * log10(1.0 + hz / 700.0)
}

/// Перетворює частоту з мел-шкали у герци.
private static func melToHz(_ mel: Float) -> Float {
    700.0 * (pow(10.0, mel / 2595.0) - 1.0)
}

/// Будує матрицю трикутних мел-фільтрів.
///
/// Створює `nMels` трикутних фільтрів, рівномірно розподілених
/// на мел-шкалі між `fMin` та `fMax`.
///
/// - Returns: Матриця `[nMels][nFFT/2+1]` з вагами фільтрів.
private static func buildMelFilterbank(
    sampleRate: Float, nFFT: Int, nMels: Int, fMin: Float, fMax: Float
) -> [[Float]] {
    let halfFFT = nFFT / 2 + 1
    let melMin = hzToMel(fMin)
    let melMax = hzToMel(fMax)

    let melPoints = (0 ..< nMels + 2).map { i -> Float in
        melMin + Float(i) * (melMax - melMin) / Float(nMels + 1)
    }
    let hzPoints = melPoints.map { melToHz($0) }
    let fftFreqs = (0 ..< halfFFT).map { k -> Float in
        Float(k) * sampleRate / Float(nFFT)
    }

    var filterbank = [[Float]](
        repeating: [Float](repeating: 0, count: halfFFT),
        count: nMels
    )

    for m in 0 ..< nMels {
        let fLeft = hzPoints[m]
        let fCenter = hzPoints[m + 1]
        let fRight = hzPoints[m + 2]
        for k in 0 ..< halfFFT {
            let freq = fftFreqs[k]
            if freq >= fLeft && freq <= fCenter {
                filterbank[m][k] = (freq - fLeft) / (fCenter - fLeft)
            } else if freq > fCenter && freq <= fRight {
                filterbank[m][k] = (fRight - freq) / (fRight - fCenter)
            }
        }
    }

    return filterbank
}

}

// MARK: - SoundManager

/// Центральний менеджер розпізнавання звуків навколишнього середовища.

```

```

///
/// Забезпечує повний цикл аудіоаналізу:
/// 1. Захоплення аудіо з мікрофона через ``AVAudioEngine``.
/// 2. Перетворення у формат 16 кГц mono через ``AVAudioConverter``.
/// 3. Обчислення мел-спектрограми за допомогою ``MelSpectrogramCalculator``.
/// 4. Класифікація через модель ``SoundVizClassifier`` (Core ML).
/// 5. Фільтрація за порогом впевненості та користувацькими налаштуваннями.
/// 6. Публікація результатів, тактильний зворотний зв'язок та push-сповіщення.
///
/// Клас є `ObservableObject` для інтерпації зі SwiftUI через ``ContentView``.
final class SoundManager: NSObject, ObservableObject {

    // MARK: - Published Properties

    /// Інформація про останній розпізнаний звук, або `nil` якщо нічого не виявлено.
    @Published var detectedSound: SoundInfo?

    /// Рівень впевненості моделі у відсотковому форматі (наприклад, `"78%"`).
    @Published var confidenceLevel: String = "0%"

    /// Прапорець активності прослуховування мікрофона.
    @Published var isListening: Bool = false

    // MARK: - Private Properties

    /// Аудіо-двигун Apple для захоплення звуку з мікрофона.
    private let audioEngine = AVAudioEngine()

    /// Цільовий формат аудіо (16 кГц, mono, float32).
    private var targetFormat: AVAudioFormat!

    /// Конвертер для перетворення нативного формату мікрофона у цільовий.
    private var audioConverter: AVAudioConverter?

    /// Завантажена модель класифікації звуків Core ML.
    private var customSoundClassifier: SoundVizClassifier?

    /// Калькулятор мел-спектрограм для підготовки вхідних даних моделі.
    private let melCalculator = MelSpectrogramCalculator()

    /// Буфер для накопичення аудіосемплів перед класифікацією.
    private var audioDataBuffer = [Float]()

    /// 2 секунди при 16 kHz
    private let requiredSamples = 32000

    /// Черга для асинхронної обробки аудіоданих поза головним потоком.
    private let analysisQueue = DispatchQueue(label:
"com.soundviz.AnalysisQueue")

    /// Мінімальний поріг впевненості моделі для прийняття результату.
    private let confidenceThreshold: Double = 0.65

    /// Менеджер тактильного зворотного зв'язку (вібрацій).
    private let hapticManager = HapticManager.shared

    /// Менеджер локальних push-сповіщень.
    private let notificationManager = NotificationManager.shared

    /// Cooldown – один і той самий звук не спрацює частіше ніж раз на 5 сек
    private var lastDetectedSound: String = ""
    private var lastDetectionTime: Date = .distantPast
    private let cooldownSeconds: TimeInterval = 5.0

    // MARK: - Init

    override init() {
        super.init()
    }

    // MARK: - Control

```

```

/// Розпочинає аналіз звуків навколишнього середовища.
///
/// Послідовно налаштовує аудіосесію, завантажує модель класифікації,
/// встановлює перехоплення аудіо та запускає аудіо-двигун.
func startAudioAnalysis() {
    configureAudioSession()
    configureAnalyzer()
    installAudioTap()
    startEngine()
}

/// Зупиняє аналіз звуків та очищує стан.
///
/// Зупиняє аудіо-двигун, видаляє перехоплення з вхідного вузла
/// та скидає всі опубліковані властивості до початкових значень.
func stopAudioAnalysis() {
    audioEngine.stop()
    audioEngine.inputNode.removeTap(onBus: 0)

    analysisQueue.async {
        self.audioDataBuffer.removeAll()
    }

    DispatchQueue.main.async {
        self.isListening = false
        self.detectedSound = nil
        self.confidenceLevel = "0%"
    }
}

// MARK: - Private

/// Налаштовує аудіосесію iOS для запису з мікрофона.
///
/// Встановлює категорію `.playAndRecord` з підтримкою одночасного
/// відтворення тактильних вібрацій під час запису.
private func configureAudioSession() {
    let session = AVAudioSession.sharedInstance()
    do {
        try session.setCategory(.playAndRecord, mode: .default,
                               options: [.mixWithOthers, .defaultToSpeaker])
        try session.setActive(true, options: .notifyOthersOnDeactivation)
        try session.setAllowHapticsAndSystemSoundsDuringRecording(true)
    } catch {
        print("❌ Помилка аудіо сесії: \(error.localizedDescription)")
    }
}

/// Налаштовує аналізатор: створює конвертер формату та завантажує модель Core ML.
///
/// Виконує warm-up прогін моделі з порожніми даними для ініціалізації
/// обчислювального графу, що забезпечує швидший відгук при першій детекції.
private func configureAnalyzer() {
    let inputNode = audioEngine.inputNode
    let nativeFormat = inputNode.outputFormat(forBus: 0)

    guard let format = AVAudioFormat(
        commonFormat: .pcmFormatFloat32,
        sampleRate: 16000,
        channels: 1,
        interleaved: false
    ) else {
        print("❌ Неможливо створити формат 16kHz")
        return
    }
    targetFormat = format
    audioConverter = AVAudioConverter(from: nativeFormat, to: targetFormat)

    do {
        let config = MLModelConfiguration()

```

```

        config.computeUnits = .cpuOnly
        let classifier = try SoundVizClassifier(configuration: config)
        self.customSoundClassifier = classifier

        // Warm-up: ініціалізуємо граф поки додаток на екрані
        let dummySpec = try MLMultiArray(shape: [1, 1, 64, 200], dataType:
.float32)
        let dummyInput = SoundVizClassifierInput(melSpectrogram: dummySpec)
        _ = try classifier.prediction(input: dummyInput)
        print("✅ Модель успішно ініціалізована (Warm-up Completed)")

    } catch {
        print("❌ Помилка завантаження моделі: \(error.localizedDescription)")
    }
}

/// Встановлює перехоплення (tap) на вхідному аудіо-вузлі.
///
/// Конвертує аудіо з нативного формату мікрофона (зазвичай 44.1/48 кГц)
/// у цільовий формат 16 кГц mono та передає семпли в чергу аналізу.
private func installAudioTap() {
    let inputNode = audioEngine.inputNode
    let nativeFormat = inputNode.outputFormat(forBus: 0)

    inputNode.installTap(onBus: 0, bufferSize: 8192, format: nativeFormat) {
        [weak self] buffer, _ in
        guard let self,
            let converter = self.audioConverter,
            let targetFormat = self.targetFormat else { return }

        let ratio = targetFormat.sampleRate / nativeFormat.sampleRate
        let capacity = AVAudioFrameCount(ceil(Double(buffer.frameCapacity) *
ratio))

        guard let converted = AVAudioPCMBuffer(
            pcmFormat: targetFormat,
            frameCapacity: capacity
        ) else { return }

        var error: NSError?
        var dataProvided = false
        let inputBlock: AVAudioConverterInputBlock = { _, outStatus in
            if dataProvided {
                outStatus.pointee = .noDataNow
                return nil
            }
            dataProvided = true
            outStatus.pointee = .haveData
            return buffer
        }

        converter.convert(to: converted, error: &error, withInputFrom: inputBlock)

        if error == nil, let channelData = converted.floatChannelData?[0] {
            let count = Int(converted.frameLength)
            let newSamples = Array(UnsafeBufferPointer(start: channelData, count:
count))

            self.analysisQueue.async {
                self.processAudioStream(newSamples: newSamples)
            }
        }
    }
}

/// Обробляє потік нових аудіосемплів.
///
/// Накопичує семпли у буфері та запускає класифікацію кожного разу,
/// коли буфер досягає `requiredSamples` (2 секунди). Використовує
/// не-overlapping вікно для уникнення повторної обробки.
///
/// - Parameter newSamples: Масив нових семплів з конвертера.
private func processAudioStream(newSamples: [Float]) {

```

```

audioDataBuffer.append(contentsOf: newSamples)

if audioDataBuffer.count > requiredSamples * 3 {
    audioDataBuffer.removeFirst(audioDataBuffer.count - requiredSamples)
}

while audioDataBuffer.count >= requiredSamples {
    let chunk = Array(audioDataBuffer.prefix(requiredSamples))
    audioDataBuffer.removeFirst(requiredSamples) // не-overlapping вікно
    classify(chunk: chunk)
}
}

/// Класифікує фрагмент аудіо за допомогою моделі ``SoundVizClassifier``.
///
/// Послідовність обробки:
/// 1. Перевірка рівня гучності (RMS > 0.005) для фільтрації тиші.
/// 2. Нормалізація амплітуди.
/// 3. Обчислення мел-спектрограми.
/// 4. Передача у модель Core ML.
/// 5. Перевірка порогу впевненості та cooldown.
/// 6. Фільтрація за користувацькими налаштуваннями.
/// 7. Публікація результату, вібрація та push-сповіщення.
///
/// - Parameter chunk: Масив семплів довжиною `requiredSamples`.
private func classify(chunk: [Float]) {
    guard let classifier = customSoundClassifier else { return }

    let rms = sqrt(chunk.reduce(0) { $0 + $1 * $1 } / Float(chunk.count))
    guard rms > 0.005 else { return }

    do {
        let maxVal = chunk.map { abs($0) }.max() ?? 0
        let normalized = maxVal > 0 ? chunk.map { $0 / maxVal } : chunk

        let melFlat = melCalculator.compute(samples: normalized)

        let multiArray = try MLMultiArray(shape: [1, 1, 64, 200], dataType:
.float32)
        multiArray.dataPointer
            .bindMemory(to: Float.self, capacity: melFlat.count)
            .update(from: melFlat, count: melFlat.count)

        let input = SoundVizClassifierInput(melSpectrogram: multiArray)
        let prediction = try classifier.prediction(input: input)

        let bestMatch = prediction.target
        let confidence = prediction.targetProbability[bestMatch] ?? 0.0

        // Top-3 логуювання для діагностики
        let top3 = prediction.targetProbability
            .sorted { $0.value > $1.value }
            .prefix(3)
        let top3Str = top3.map { "\( $0.key) (\(Int($0.value * 100))%)"
    }.joined(separator: ", ")
        print("🔊 Top-3: \(top3Str)")

        guard bestMatch != "silence", confidence > confidenceThreshold else {
            if confidence <= confidenceThreshold {
                print("⚠️ Відхилено: \(bestMatch) (\(Int(confidence * 100))% <
\((Int(confidenceThreshold * 100))%)"
            )
            }
            return
        }

        // Cooldown: ігноруємо повтор того самого звуку протягом 5 сек
        let now = Date()
        if bestMatch == lastDetectedSound,
            now.timeIntervalSince(lastDetectionTime) < cooldownSeconds {
            return
        }
    }
}

```

```

lastDetectedSound = bestMatch
lastDetectionTime = now

let soundInfo = SoundDictionary.getInfo(for: bestMatch)

// Перевірка фільтру: пропускаємо звуки, вимкнені користувачем
guard SoundFilterManager.shared.isEnabled(bestMatch) else {
    print("❌ Відфільтровано (вимкнено користувачем): \(bestMatch)")
    return
}

DispatchQueue.main.async {
    self.detectedSound = soundInfo
    self.confidenceLevel = "\(Int(confidence * 100))%"
    self.hapticManager.playHaptic(for: soundInfo.level)
    self.notificationManager.sendNotification(for: soundInfo)
}

} catch {
    print("❌ Помилка класифікації: \(error.localizedDescription)")
}
}

/// Запускає аудіо-двигун та оновлює прапорець ``isListening``.
private func startEngine() {
    do {
        try audioEngine.start()
        DispatchQueue.main.async { self.isListening = true }
    } catch {
        print("❌ Помилка запуску аудіо-двигуна: \(error.localizedDescription)")
    }
}
}

```

Б.4 Текст файлу NotificationManager.swift

```

import Foundation
import UserNotifications

// MARK: - NotificationManager

/// Менеджер локальних push-сповіщень.
///
/// Надсилає сповіщення при розпізнаванні звуків із рівнем небезпеки
/// ``DangerLevel/high`` або ``DangerLevel/medium``.
/// Для ``DangerLevel/low`` сповіщення не надсилаються.
final class NotificationManager {

    /// Єдиний екземпляр менеджера (Singleton).
    static let shared = NotificationManager()

    // MARK: - Deduplication State

    /// Ідентифікатор звуку, для якого було надіслано останнє сповіщення.
    private var lastNotifiedSoundId: String = ""

    /// Час надсилання останнього сповіщення.
    private var lastNotificationTime: Date = .distantPast

    /// Мінімальний інтервал між сповіщеннями для того самого звуку (у секундах).
    ///
    /// Якщо детектується *інший* звук — сповіщення надсилається одразу.
    /// Якщо детектується *той самий* звук повторно — сповіщення подавляється
    /// протягом цього інтервалу, щоб уникнути спаму.
    private let notificationCooldown: TimeInterval = 30

    private init() {}
}

```

```

// MARK: - Authorization

/// Запитує дозвіл на надсилання сповіщень у користувача.
///
/// Викликати при запуску додатку (наприклад, у `SoundVizApp.init()`).
func requestAuthorization() {
    UNUserNotificationCenter.current().requestAuthorization(options: [.alert,
.sound, .badge]) { granted, error in
        if let error = error {
            print("❌ Помилка запиту дозволу на сповіщення:
\ (error.localizedDescription)")
        }
    }
}

// MARK: - Send Notification

/// Надсилає локальне push-сповіщення для розпізнаного звуку.
///
/// Логіка дедуплікації:
/// - Сповіщення надсилається лише для рівнів ``DangerLevel/high`` та
``DangerLevel/medium``.
/// - Якщо звук **новий** (відрізняється від попереднього) – сповіщення одразу.
/// - Якщо звук **той самий** – сповіщення надсилається лише після
``notificationCooldown`` (30 сек).
/// - Використовує `soundInfo.id` як `identifier` сповіщення, тому повторне
сповіщення
///   для того ж звуку замінить попереднє, а не додасть нове.
///
/// - Parameter soundInfo: Інформація про розпізнаний звук.
func sendNotification(for soundInfo: SoundInfo) {
    guard soundInfo.level != .low else { return }

    let now = Date()

    // Дедуплікація: той самий звук – подавляємо протягом cooldown
    if soundInfo.id == lastNotifiedSoundId,
        now.timeIntervalSince(lastNotificationTime) < notificationCooldown {
        return
    }

    // Оновлюємо стан
    lastNotifiedSoundId = soundInfo.id
    lastNotificationTime = now

    let content = UNMutableNotificationContent()
    content.title = String(localized: "notification_title")
    content.body = soundInfo.name
    content.sound = .default

    // Використовуємо ID звуку як ідентифікатор сповіщення:
    // - нове сповіщення для того ж звуку замінить попереднє
    // - різні звуки матимуть різні сповіщення
    let request = UNNotificationRequest(
        identifier: "soundviz_\ (soundInfo.id)",
        content: content,
        trigger: nil
    )

    UNUserNotificationCenter.current().add(request) { error in
        if let error = error {
            print("❌ Помилка надсилання сповіщення:
\ (error.localizedDescription)")
        }
    }
}
}

```

ДОДАТОК В
Слайди презентації



Рисунок В.1 – Слайд 1

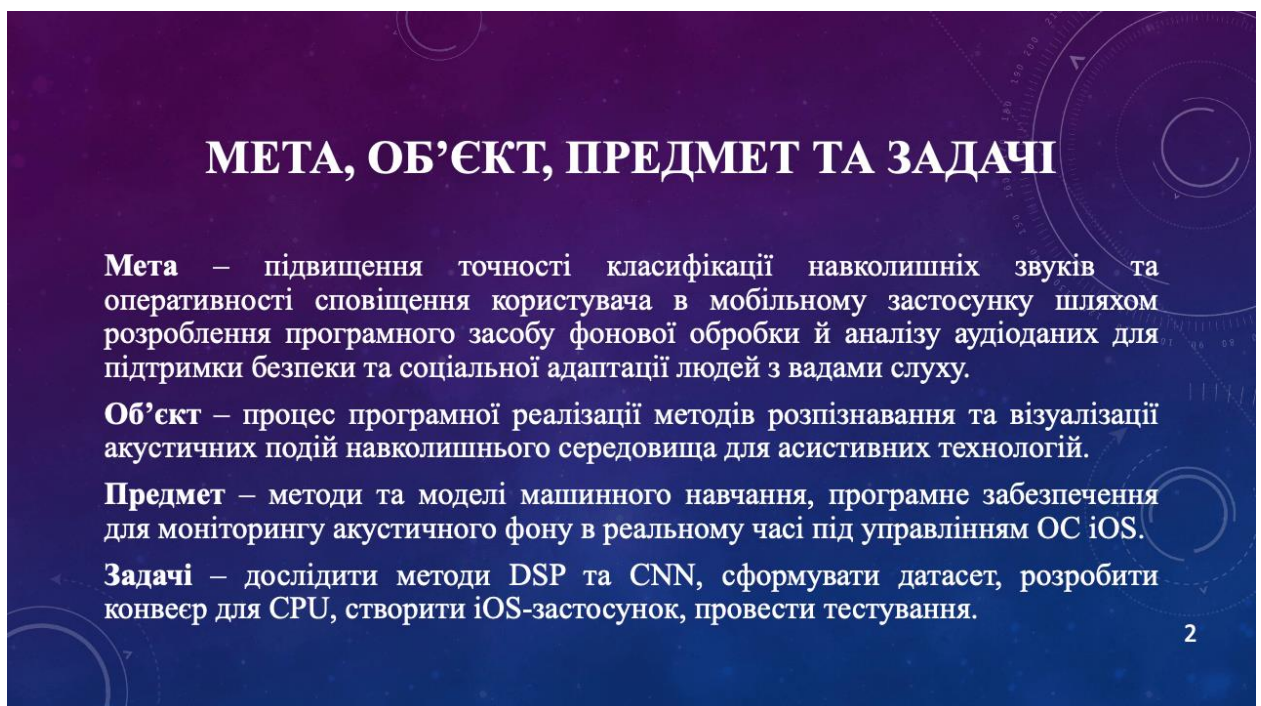


Рисунок В.2 – Слайд 2

АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Критерій порівняння	Live Transcribe & Notification (Google)	Sound Recognition (Apple)	Розроблювана система «SoundViz»
Операційна система	Android	iOS	iOS
Кількість класів звуків	9 базових класів	15 базових + кастомні	36 спеціалізованих класів
Робота у фоновому режимі	Так	Так (на рівні ОС)	Так
Обробка даних (Інференс)	Локально на пристрої	Локально на пристрої	Локально на пристрої
Залежність від GPU у фоні	Не застосовується (ОС Android)	Використовує Neural Engine / Metal	Немає (інференс виключно на CPU)
Відкритість для розробників	Закрита екосистема	Системна функція (закрите API)	Відкрита для інтеграції у сторонні проєкти
Головний недолік	Обмежений набір звуків	Неможливість інтеграції у власні сторонні iOS-застосунки	Обмеження продуктивності через відмову від GPU

3

Рисунок В.3 – Слайд 3

ВИБІР МОВИ ПРОГРАМУВАННЯ ДЛЯ РОЗРОБКИ ML-MODEЛІ

Критерій порівняння	Мова програмування		
	Python	C++	Swift
Наявність готових ML-фреймворків (PyTorch, TensorFlow)	Широкий вибір	Обмежена	Низька
Бібліотеки для цифрової обробки аудіо (DSP)	Висока	Висока	Середня
Швидкість прототипування та написання коду	Висока	Низька	Середня
Ефективність використання ресурсів комп'ютера	Середня	Висока	Висока
Підтримка експорту у формат Core ML	Наявна	Відсутня	Відсутня

4

Рисунок В.4 – Слайд 4

ВИБІР ТЕХНОЛОГІЇ МОБІЛЬНОЇ РОЗРОБКИ

Критерій порівняння	Технологія розробки		
	Нативна розробка (Swift / SwiftUI)	Кросплатформна розробка (Flutter)	Кросплатформна розробка (React Native)
Можливість роботи на обох платформах (iOS / Android)	Тільки iOS	Повна	Повна
Прямий доступ до низькорівневих фреймворків (vDSP, Core ML)	Повний	Обмежений (через мости)	Обмежений (через мости)
Можливість складної фонові обробки аудіо без GPU	Можлива	Неможлива	Неможлива
Висока продуктивність та мінімальна затримка (Latency)	Висока	Середня	Середня
Інтеграція з Taptic Engine (вібрація)	Повна	Обмежена	Обмежена

5

Рисунок В.5 – Слайд 5

ПІДГОТОВКА ТА АУГМЕНТАЦІЯ ДАНИХ

Рівень небезпеки	Категорія звуків (класи)
Високий	Повітряна тривога, пожежна сигналізація, сирена спецтранспорту, вибух, постріл, крик, плач дитини, автомобільний сигнал, розбиття скла
Середній	Стук, грук дверима, дверний дзвінок, телефон, будильник, кипіння води, мікрохвильова піч, собака, кіт, плач, дитяча мова, свердло, транспорт (авто, мотоцикл, вантажівка, автобус, потяг, літак)
Низький	Мова, сміх, кашель, кроки, оплески, велосипед, пилосос, фен, дзвін ключів, дощ, гроза

Для дефіцитних класів застосовано дворівневу аугментацію (офлайн та онлайн):

- зміна швидкості відтворення (коеф. 0,9 та 1,1);
- зсув пітку (на $\pm 1,5$ напівтони);
- зміна гучності (на ± 3 дБ);
- додавання білого та гаусівського шуму (коеф. 0,005);
- циклічний часовий зсув сигналу (на $\pm 0,1$ с).

6

Рисунок В.6 – Слайд 6

МАТЕМАТИЧНА МОДЕЛЬ ОБРОБКИ СИГНАЛУ

Віконне перетворення Фур'є (STFT): Перетворення у нелінійну Мел-шкалу:

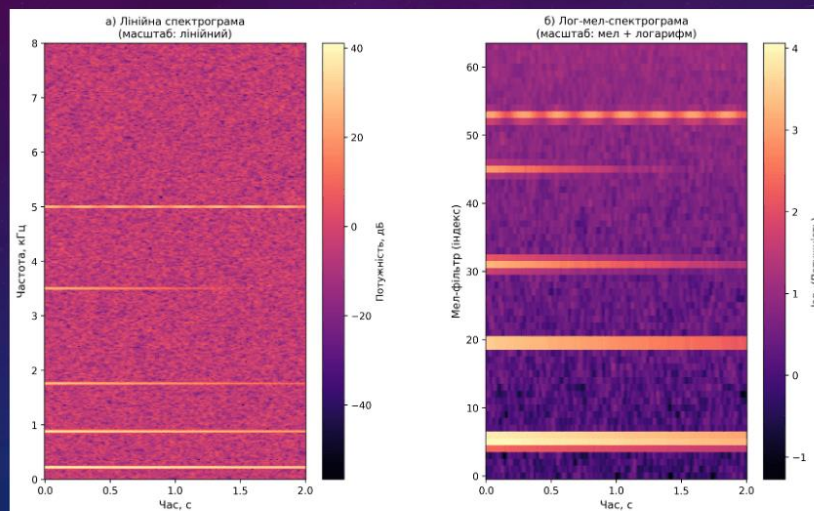
$$P(f) = \frac{1}{N} |X(f)|^2$$

$$m = 2595 \cdot \log_{10}\left(1 + \frac{f}{700}\right)$$

7

Рисунок В.7 – Слайд 7

ВІЗУАЛІЗАЦІЯ ЕТАПІВ ОБРОБКИ



8

Рисунок В.8 – Слайд 8

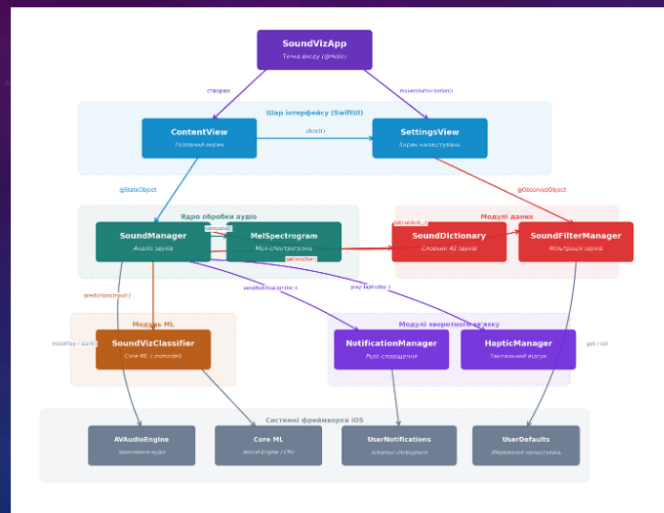
АРХІТЕКТУРА ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ



9

Рисунок В.9 – Слайд 9

АРХІТЕКТУРА ПРОГРАМНОЇ СИСТЕМИ



10

Рисунок В.10 – Слайд 10

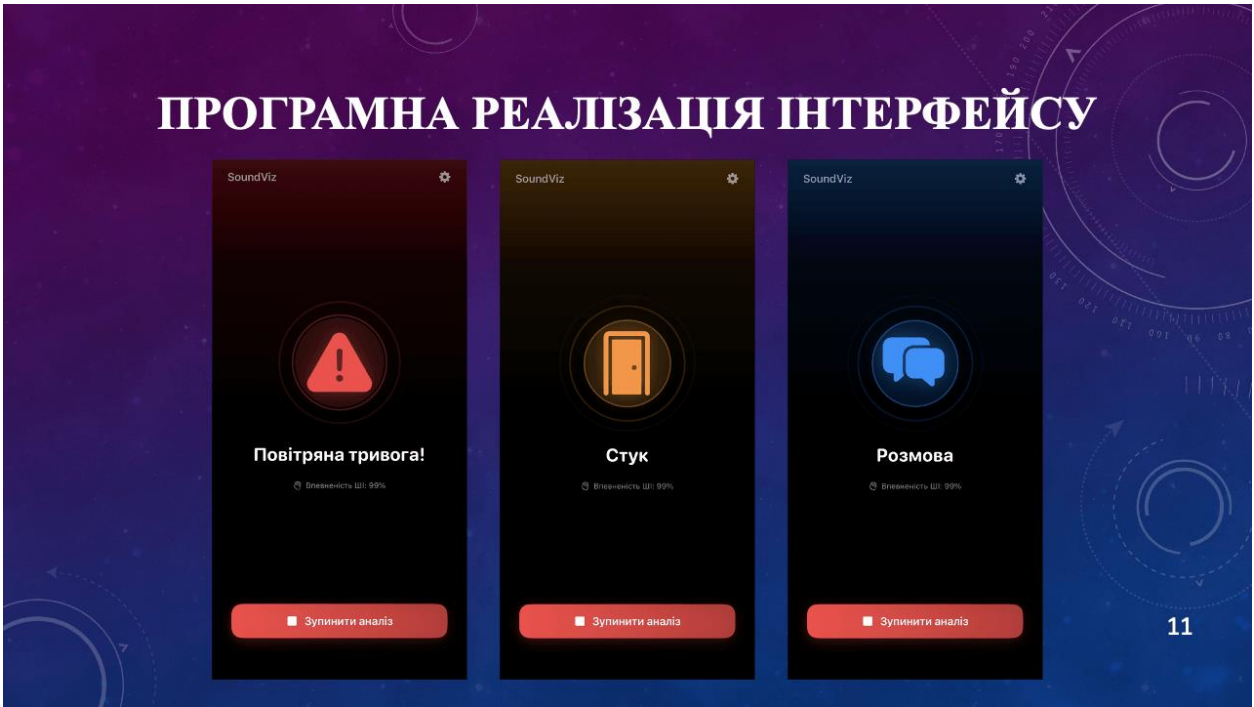


Рисунок В.11 – Слайд 11

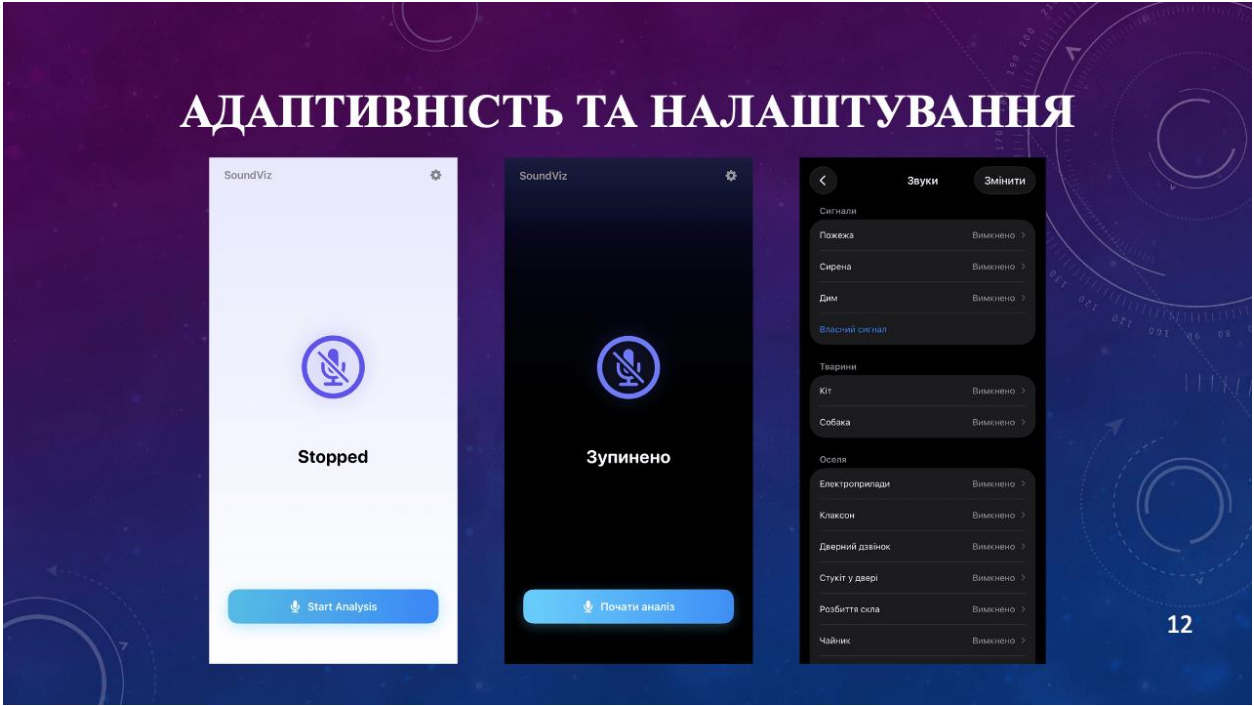


Рисунок В.12 – Слайд 12

ФОНОВИЙ РЕЖИМ ТА ТАКТИЛЬНИЙ ВІДГУК

Система тактильних сповіщень (Haptics):

- **висока небезпека:** подвійна інтенсивна вібрація;
- **середній рівень (увага):** одиночне попередження;
- **низький рівень:** без вібрації (економія акумулятора).

Захист від «спаму» (Cooldown):

- блокування повторних сповіщень про один і той самий звук протягом 30 секунд;
- миттєве сповіщення у разі фіксації нового типу загрози.

13

Рисунок В.13 – Слайд 13

РЕЗУЛЬТАТИ ФУНКЦІОНАЛЬНОГО ТЕСТУВАННЯ

№	Тестовий сценарій (Кроки)	Очікуваний результат	Фактичний результат	Статус
1	Запуск без наданого дозволу на мікрофон	Відображення екрану з проханням надати дозвіл у Налаштуваннях iOS	Інтерфейс блокує запуск аналізу, відображається попередження	Успішно
2	Імітація вхідного дзвінка під час аналізу звуку	Аудіосесія переривається. Після завершення дзвінка аналіз автоматично відновлюється	Система коректно обробляє AVAudioSession interruptionNotification	Успішно
3	Перехід застосунку у фоновий режим (Background)	Прослуховування продовжується, інтерфейс не оновлюється, push-сповіщення працюють	Інференс на CPU триває, сповіщення надходять	Успішно
4	Перевірка механізму "Cooldown" (захист від спау)	При безперервному звуці сирени сповіщення надходить 1 раз на 30 секунд	Повторні сповіщення про один і той самий звук ігноруються протягом 30 с	Успішно

14

Рисунок В.14 – Слайд 14

ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

Джерело звуку	Умови середовища (відстань / шум)	Середній рівень впевненості моделі (Confidence)	Час реакції системи
Пожежна сигналізація	1 м / без шуму	93%	2,1 с
Пожежна сигналізація	3 м / міський шум	79%	2,3 с
Стукіт у двері	1 м / без шуму	99%	2,1 с
Стукіт у двері	3 м / міський шум	72%	4,1 с (знадобилося 2 буфери)
Гавкіт собаки	1 м / без шуму	88%	2,2 с
Гавкіт собаки	3 м / міський шум	67%	Розпізнано частково

15

Рисунок В.15 – Слайд 15

ВИСНОВКИ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

Основні результати роботи:

- вирішено проблему фонового розпізнавання на iOS завдяки створенню власного DSP-конвертера (CPU-only);
- сформовано датасет та навчено CNN (ResNet) для класифікації 36 типів звуків із точністю 75,56%;
- розроблено нативний iOS-застосунок (SwiftUI, MVVM) як персональний асистент для людей з вадами слуху;
- підтверджено працездатність системи в умовах реального міського шуму (швидкість реакції < 2,5 с).

Перспективи розвитку:

- збільшення датасету для підвищення точності;
- створення незалежного застосунку для Apple Watch (watchOS);
- впровадження донавчання моделі на пристрої (On-device Learning).

16

Рисунок В.16 – Слайд 16