МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«СИСТЕМА АВТОМАТИЧНОГО РОЗПІЗНАВАННЯ УКРАЇНСЬКОГО МОВЛЕННЯ НА ОСНОВІ ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ»

Рівень «магістр»
Галузь знань 12 «Інформаційні технології»
Спеціальність 122 «Комп’ютерні науки»
Виконала здобувачка 2 курсу
Антіпіна Надія Сергіївна

Керівник к.т.н., доцент
Трофименко Олена Григорівна

Рецензент: к.пед.н., доцент
Лобода Юлія Геннадіївна

Одеса – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

Факультет кібербезпеки та інформаційних технологій
Кафедра інформаційні технології
Галузь знань: 12 Інформаційні технології
Спеціальність: 122 Комп'ютерні науки
Назва освітньої програми: Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Н.І. Логінова
_____ «___» _____ 2025 року

З А В Д А Н Н Я
на кваліфікаційну (магістерську) роботу
здобувачці Антіпіній Надії Сергіївні

1. Тема роботи: «Система автоматичного розпізнавання українського мовлення на основі згорткових нейронних мереж»
   Керівник роботи: к.т.н, доцент Трофименко Олена Григорівна
   затверджені наказом НУ «ОЮА» від «10» жовтня 2025 року № 3182-06.
2. Строк подання здобувачем роботи «25» листопада 2025 року.
3. Дата видачі завдання «02» червня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Вибір теми, вивчення літературних джерел та складання плану роботи	15.09.2025	
2	Підготовка першого розділу роботи та подання його керівнику	01.10.2025	
3	Підготовка другого розділу роботи та подання його керівнику	14.10.2025	
4	Підготовка третього розділу роботи та подання його керівнику	01.11.2025	
5	Підготовка вступу та висновків	07.11.2025	
6	Доопрацювання роботи з урахуванням зауважень керівника	17.11.2025	
7	Подача електронного варіанта роботи для перевірки на плагіат	20.11.2025	

Здобувач _____*Н.С. Антіпіна*_____
(підпис) (прізвище та ініціали)

Керівник роботи _____*О.Г. Трофименко*_____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Система автоматичного розпізнавання українського мовлення на основі згорткових нейронних мереж» на здобуття другого (магістерського) рівня вищої освіти за спеціальністю 122 «Комп'ютерні науки», освітньою програмою «Комп'ютерні науки». Робота містить 12 рисунків, 5 таблиць, 2 додатки, 38 літературних джерел. Робота виконана на 77 сторінках основного тексту і 91 сторінках загального тексту.

Метою роботи є дослідження та реалізація моделі системи автоматичного розпізнавання українського мовлення для транскрибування відео- та аудіофайлів на основі згорткових нейронних мереж та принципів BERT-архітектури.

Об'єктом дослідження є процеси автоматизованого розпізнавання мовлення на основі згорткових нейронних мереж.

Предметом дослідження є методи та засоби реалізації нейронної моделі автоматичного розпізнавання українського мовлення.

В першому розділі досліджено особливості систем автоматичного розпізнавання українського мовлення, наведено архітектурні та алгоритмічні виклики під час розробки моделей, проаналізовано сучасні методи глибинного навчання в ASR-задачах та порівняння наявних моделей. У другому розділі здійснено проєктування архітектури системи автоматичного розпізнавання мовлення вибір стека інструментів для її реалізації. У третьому розділі здійснено програмну реалізацію системи автоматичного розпізнавання мовлення, досліджено якість моделі за допомогою метрик точності та швидкодії, а також порівняння з наявними системами.

Ключові слова: автоматичне розпізнавання мовлення, згорткові нейронні мережі, машинне навчання.

ABSTRACT

Antipina N.S. Automatic Speech Recognition System for Ukrainian using CNN. – Master's qualifying work on specialty 122 "Computer science".

The work consists of 77 pages of main text and 91 pages of total text, comprising 3 chapters, 5 tables, 12 figures, 2 appendice, and 38 used sources.

The purpose of the work is to investigate and implement an automatic speech recognition (ASR) model for Ukrainian speech, designed to transcribe audio and video files using convolutional neural networks and principles of BERT-based architecture.

The object of the study is the processes of automated speech recognition based on convolutional neural networks.

The subject of the study is the methods and tools for implementing a neural model for Ukrainian automatic speech recognition.

In the first section, the specifics of automatic speech recognition systems for the Ukrainian language were examined, architectural and algorithmic challenges in developing ASR models were outlined, modern deep-learning approaches applied in ASR tasks were analyzed. In the second section, the architecture of the speech recognition system was designed, and an appropriate technology stack and instrumental tools for its implementation were selected. The third section presented the program implementation of the ASR system, evaluated the quality of the developed model using accuracy and performance metrics, and provided a comparison with existing recognition systems.

Keywords: automatic speech recognition, convolutional neutral networks, machine learning.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ANN – Artificial Neural Network, штучна нейронна мережа

BERT – Bidirectional Encoder Representations from Transformers, двонапрямлені подання з кодувальника трансформерів

CER – Character Error Rate, частота помилок у символах

CNN – Convolutional Neural Network, згорткова нейронна мережа

CTC – Connectionist Temporal Classification, зв’язкова тимчасова класифікація

DL – Deep Learning, глибинне навчання

FFN – Feed-Forward Network, мережа прямого поширення

FFT – Fast Fourier Transform, швидке перетворення Фур’є

GELU – Gaussian Error Linear Unit, лінійна одиниця з похибкою Гаусса

GPU – Graphics Processing Unit, графічний процесор

LSTM – Long Short-Term Memory, довга короткострокова пам’ять

MFCC – Mel-Frequency Cepstral Coefficients, мел-частотні кепстральні коефіцієнти

ReLU – Rectified Linear Unit, випрямлена лінійна одиниця

RNN – Recurrent Neural Network, рекурентна нейронна мережа

RTF – Real-Time Factor, коефіцієнт реального часу

SER – Sentence Error Rate, частота помилок у реченнях

WER – Word Error Rate, частота помилок у словах

ВСТУП

У сучасному світі за умов стрімкого розвитку цифрових технологій системи автоматичного розпізнавання мовлення набувають все більшої популярності. Вони широко використовуються в голосових асистентах, системах керування пристроями, автоматичному перекладі, службах підтримки, медичних застосунках та інших сферах, де необхідна взаємодія людини з машиною через природну мову. Одним із ключових напрямів удосконалення таких систем є підвищення їхньої точності та адаптації до конкретних мов, зокрема української.

Попри значний прогрес у галузі, переважна більшість високоточних технологій автоматичного розпізнавання мовлення створена для англійської або інших глобальних мов з великими відкритими корпусами даних. Для української мови, яка має складну фонетичну й морфологічну структуру, кількість доступних даних для тренування глибоких нейронних мереж суттєво менша, а доступ до потужних попередньо навчених моделей розпізнавання мовлення, адаптованих для української мови, є дуже обмеженим. Такі моделі зазвичай мають значну обчислювальну складність, що теж обмежує їх доступність.

У зв’язку з цим актуальним є створення відкритої моделі розпізнавання українського мовлення, яка може навчатися на доступних обсягах аудіоданих та не потребує потужних обчислювальних ресурсів, проте здатна забезпечувати високий рівень точності та інтегруватися у реальні вебсервіси, мобільні застосунки чи то інформаційні системи для здійснення різних задач розпізнавання мовлення: створення автоматичних субтитрів для відеоконтенту, протоколювання зустрічей, архівування інтерв’ю, створення мультимедійних освітніх матеріалів, впровадження голосового керування українською мовою тощо.

Метою роботи є дослідження та реалізація моделі системи автоматичного розпізнавання українського мовлення для транскрибування відео- та аудіофайлів на основі згорткових нейронних мереж та принципів BERT-архітектури.

Об’єктом дослідження є процеси автоматизованого розпізнавання мовлення на основі згорткових нейронних мереж.

Предметом дослідження є методи та засоби реалізації нейронної моделі автоматичного розпізнавання українського мовлення.

Відповідно до поставленої мети завданнями роботи є:

– аналіз сучасних методів та підходів до задач автоматичного розпізнавання мовлення, порівняння існуючих ASR-моделей;

– проєктування запропонованої архітектури моделі автоматичного розпізнавання мовлення та її обґрунтування;

– реалізація системи розпізнавання мовлення та навчання на українському наборі даних;

– оцінка точності роботи системи за допомогою метрик та порівняння з сучасними аналогами;

– аналіз результатів та визначення напрямів подальшого вдосконалення моделі.

Наукова новизна роботи полягає в урахуванні специфіки складної фонетичної та морфологічної структури української мови при використанні глибоких нейронних моделей в розробці системи автоматичного розпізнавання мовлення, оскільки переважна більшість високоточних технологій розпізнавання мовлення працюють для англійською чи то іншими глобальними мовами з великими відкритими корпусами даних.

Публікації з апробацією кваліфікаційної роботи:

1. Антіпіна Н.С. Архітектурні та алгоритмічні виклики навчання ASR-моделей для розпізнавання українського мовлення. *Кібербезпека в сучасному світі: актуальні виклики :* матер. VI міжнар. наук.-практ. конф. (28 листопада 2025 р.). Одеса, 2025 (депоновано).

2. Антіпіна Н.С., Разінкін Н.С. Інтеграція систем розпізнавання мовлення в автоматизовані співбесіди для оптимізації підбору персоналу на підприємстві. *Управління ресурсним забезпеченням господарської діяльності підприємств реального сектору економіки:* матер. X всеукр. наук.-практ. конф. (13 листопада 2025 р.) Полтава, 2025 (депоновано).

1 ОСОБЛИВОСТІ АВТОМАТИЧНОГО РОЗПІЗНАВАННЯ МОВЛЕННЯ

1.1 Поняття та алгоритм розпізнавання мовлення

Засоби машинного навчання набувають все більшу популярність та затребуваність через свою здатність до аналізу великих обсягів даних, виявлення складних закономірностей, самовдосконалення шляхом ітеративного процесу оновлення внутрішніх параметрів, генерування прогнозів і варіантів рішень на їхній основі [1]. Однією з поширених галузей застосування машинного навчання, що наразі активно розвивається та вдосконалюється, є розпізнавання мови.

Розпізнавання мови – це аналіз та перетворення аудіоданих з голосовими сигналами на текстовий набір даних. Технологія, яка дозволяє ідентифікувати та класифікувати музику, навколишні звуки, людську мову, а також перетворювати мовлення людини на текст або команди, знайшла використання в повсякденному житті через смартфони, голосові асистенти (наприклад, Siri та Alexa), смарт-прилади з функцією голосового управління, системи перекладу, автоматичні служби підтримки, засоби автоматичного транскрибування відео- й аудіофайлів, допоміжні інструменти для людей з вадами слуху тощо. Алгоритми розпізнавання мови та аудіоданих можуть відрізнятися за своїми задачами: транскрибування мовлення людини та перетворення його на доступні команди (speech recognition); визначення особи людини за голосом (speaker recognition); аналіз емоційного стану людини (emotion recognition); розпізнавання звуків навколишнього середовища (sound recognition); генерування аудіо з вхідного тексту (speech synthesis) [2].

Принцип роботи систем автоматичного розпізнавання мовлення полягає у використанні моделей нейронних мереж, здатних сприймати звуковий сигнал, виділяти мовні ознаки та автоматично інтерпретувати їх як певну послідовність слів. Для тренування таких моделей в машинному навчанні застосовують підхід навчання з вчителем (supervised learning), який полягає в поданні мічених наборів даних як вхідної незалежної змінної, на прикладі якої модель навчатиметься прогнозувати вихідні значення. Алгоритм виявляє закономірності, навчається на основі спостережень і робить прогнози, які виправляються, доки не буде досягнуто

високого рівня точності. Після навчання модель може прогнозувати результат для нових або невідомих наборів вхідних даних без потреби в правильних відповідях [3].

Вміст набору даних для реалізації навчання з учителем поділяють на такі частини:

– навчальний набір – 70–80% матеріалу від загального набору даних, який містить правильні відповіді в якості міток і використовується для навчання моделі розпізнавати закономірності та патерни даних;

– валідаційний набір – приблизно 10–15% даних для контролю якості набутих системою навичок для узагальнення даних та коригування налаштувань гіперпараметрів;

– тестовий набір – решта 10–15%, які використовуються після завершення навчання для оцінки продуктивності моделі рівня точності вихідного результату на незнайомих даних [4].

Підхід навчання з учителем в системах автоматичного розпізнавання мовлення забезпечує поступове вдосконалення здатності моделі до узагальнення патернів людської мови шляхом мінімізації різниці між транскрипцією правильної відповіді та вихідного передбачення тексту.

Процес розпізнавання мовлення є послідовним ланцюгом перетворень, націлених на отримання найбільш точного результату. Алгоритм розпізнавання мовлення зображений на рис. 1.1.

Відповідно до цього процес розпізнавання мовлення має такі етапи:

1) Попередня обробка аудіо даних

Перед поданням моделі нейронної мережі вхідних даних потрібно зчитати, оцифрувати та попередньо опрацювати вхідні аудіозаписи, щоб підвищити якість, стабільність та репрезентативність цих даних. Для цього аудіозаписи шляхом дискретизації перетворюють на цифровий сигнал, нормалізують за амплітудою, приводять до цільової частоти та однакової тривалості, застосовують алгоритми для зниження шуму.

Рисунок 1.1 – Алгоритм процесу розпізнавання мовлення

Аудіозаписи розбиваються на короткі інтервали (фрейми) тривалістю 10-30 мс, після чого до кожного з них застосовується віконна функція (Геммінга або Гаусса) для зменшення стрибків амплітуди на їхніх межах. Для того, щоб навчити модель сприймати більшу кількість варіацій вхідних даних, на цьому етапі може проводитися базове аугментування випадкової вибірки або копій аудіозаписів шляхом їхнього зміщення в часі, змінення висоти тону або швидкості сигналу [5].

2) Вилучення ознак

До підготовлених фреймів, які наразі показують лише зміни амплітуди сигналу в часі, застосовується швидке перетворення Фур’є (FFT), яке дозволяє розкласти сигнал на окремі частоти й визначити амплітуду кожної. Це перетворює сигнал із часової області у частотну, в результаті чого формується спектр сигналу. Щоб побачити часову залежність частот, спектр сигналу обчислюється в коротких часових вікнах короткочасним перетворенням Фур’є та формує спектрограму з поданням часу по осі *X*, частоти в логарифмічній шкалі по осі *Y* та амплітуди через

зміну кольору. В таких спектрограмах перетворення звичайних частот у мел-шкалу дозволяє подати зображення звуку більш наближеним до того, як його чує людина. Мел-спектрограми відображають ключові особливості аудіо й зазвичай є найзручнішою формою подання вхідних даних для глибинних нейронних мереж. Для ще більшої адаптації до людського мовлення здійснюється розрахунок мел-частотних кепстральних коефіцієнтів (MFCC), які містять стиснене подання мел-спектрограми лише в межах частотного діапазону, що сприймається людиною. Після формування мел-спектрограм можливе здійснення повторної аугментації даних із застосуванням маскування частоти та часу шляхом накладання горизонтальних та вертикальних смуг на зображення спектрограми. Така обробка сприяє підвищенню пристосованості моделі до шумів, швидкості, висоти голосу та здатності до узагальнення [6].

Приклад мел-спектрограми подано на рис. 1.2. Разом з аудіоданими здійснюється підготовка транскрипцій аудіо (міток) шляхом їхнього розбиття на символи та привласнення ідентифікаторів. Такі цільові мітки разом з мел-спектрограмами або MFCC є вхідним набором даних для навчання моделі розпізнавання мовлення.

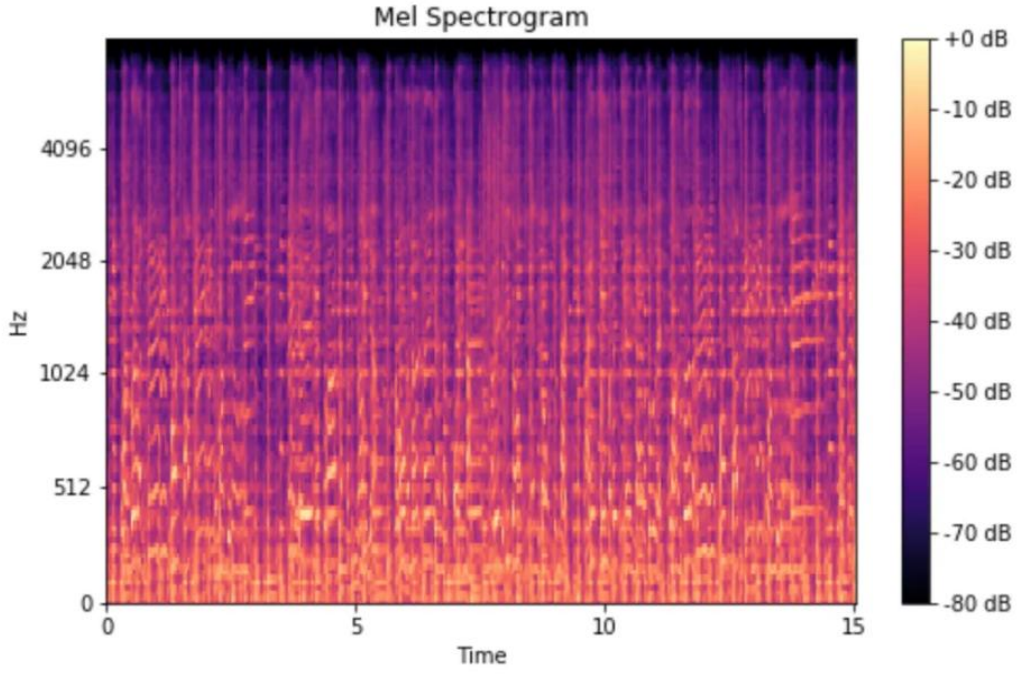

Рисунок.1.2 – Приклад мел-спектрограми [6]

3) Акустичне моделювання

Під час акустичного моделювання створюється модель, яка навчається розпізнавати закономірності та встановлювати зв’язок між підготовленими виділеними ознаками і відповідними символами, фонемами або словами транскрипції. Згорткові шари нейронної мережі обробляють вхідні мел-спектрограми, шляхом поступового зменшення їхніх просторових розмірів та збільшення глибини ознак формують карти ознак, де кожен кадр має набір імовірностей для кожного з можливих символів. В процесі навчання моделі за допомогою функції втрат відбувається оцінка правильності сформованого передбачення послідовності символів відповідно до міток, корегування ваг та оновлення параметрів, доки метрики точності розпізнавання символів не досягнуть оптимального значення.

4) Декодування

Після завершення навчання моделі здійснюється перетворення отриманих вихідних імовірностей символів у кінцевий текстовий рядок, де декодер вибирає найімовірнішу послідовність символів або слів методами пошуку, наприклад, за допомогою пошуку за променями (beam search), який ітерація за ітерацією вибирає декілька найперспективніших комбінацій та поступово звужує вибір до найбільш імовірної з них [7]. Для кращого врахування контексту та структури речення можливе також використання мовної моделі.

5) Постобробка

Постобробка отриманої послідовності стосується нормалізації тексту та пунктуації, фільтрації службових символів та усунення типових помилок розпізнавання мовлення [8].

На діаграмі послідовності, зображеній на рис. 1.3, відображено послідовний процес автоматичного розпізнавання мовлення. Вона відображає повний життєвий цикл обробки аудіо у системі автоматичного розпізнавання мовлення від моменту завантаження медіафайлу користувачем на сторінку вебсайту до повернення його текстової транскрипції.

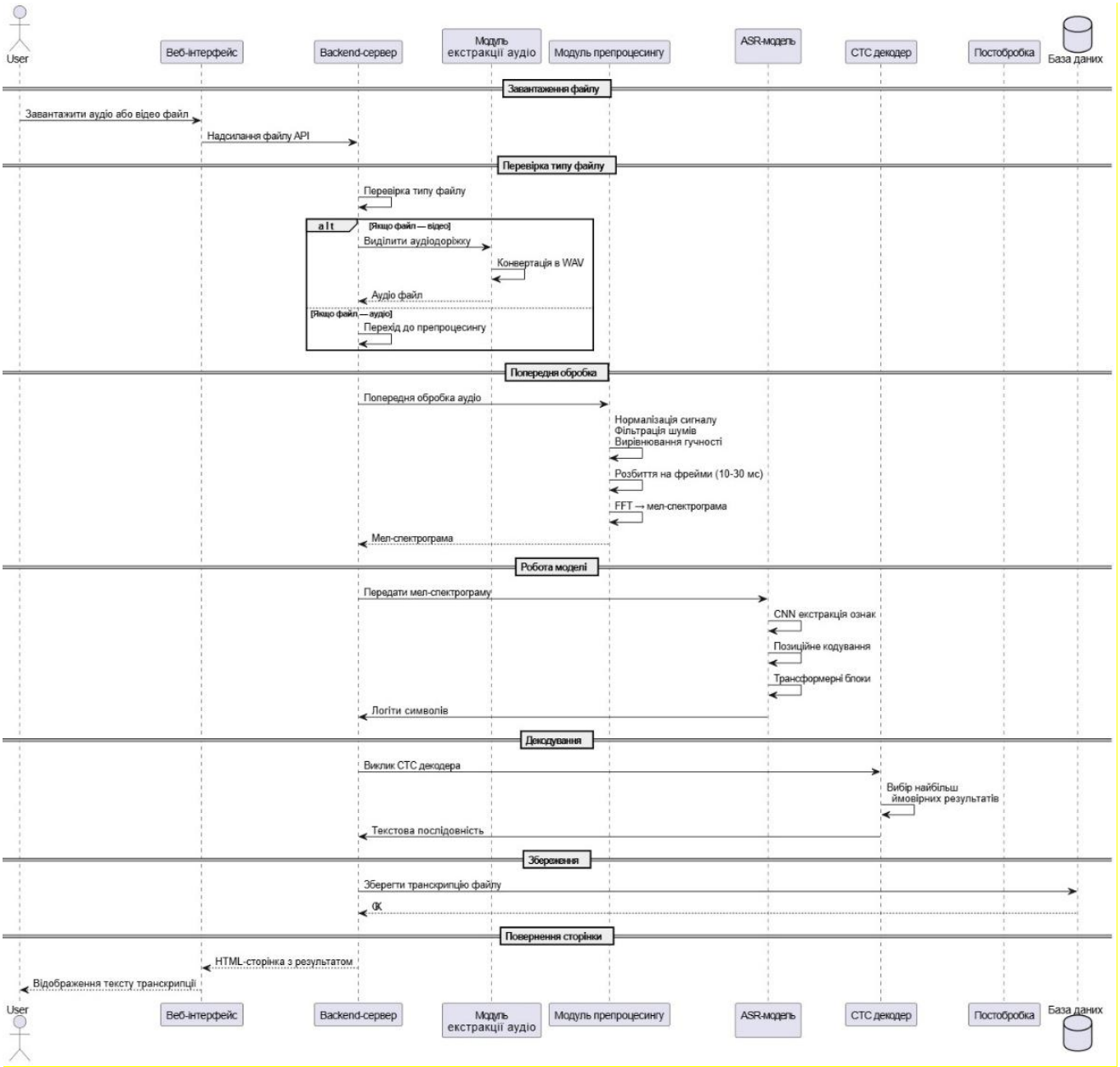

Рисунок 1.3 – Діаграма послідовності процесу автоматичного розпізнавання мовлення

Представлена діаграма послідовності демонструє взаємодію між вебінтерфейсом, бекенд-сервером, модулем вилучення ознак, акустичною моделлю, декодером та базою даних. Послідовність операцій включає перевірку формату, конвертацію аудіо, попередню обробку сигналу, побудову мел-спектрограм, проходження через згорткові та трансформерні блоки моделі, CTC-декодування та формування вихідного тексту. Кожен компонент виконує свою частину обробки, а їх взаємодія забезпечує коректну та послідовну роботу всієї ASR-системи від сирого аудіосигналу до готової транскрипції.

1.2 Особливості задачі розпізнавання мовлення

На відміну від класичних задач класифікації, розпізнавання мовлення має свої характерні особливості. У системах розпізнавання мовлення (ASR-системах) модель має навчитися перетворювати довгу послідовність звукових фреймів у значно коротшу послідовність символів без чіткої відповідності довжин між вхідними та вихідними послідовностями. Щоб вирішити цю проблему, додається алгоритм CTC, який допомагає моделі самостійно навчитися вирівнювати аудіопослідовності змінної довжини з відповідними текстовими транскрипціями змінної довжини без потреби зазначення точних часових проміжків кожного символу [9]. Також під час роботи з великими нейронними мережами виникає ризик перенавчання, коли через недостатню кількість прикладів система запам’ятовує приклади з навчальної вибірки, але не узагальнює закономірність та через це має низький показник точності на тестових даних. Щоб уникнути цього явища, в ASR-системах потрібно застосовувати методи регуляризації, наприклад: Dropout, Weight Decay та Batch Normalization.

Процес навчання моделі розпізнавання мовлення стосується використання великих обсягів даних, що містять тисячі годин аудіозаписів з точними транскрипціями, а якість та репрезентативність цих даних безпосередньо визначають якість майбутньої моделі. Для досягнення високої точності розпізнавання важливо, щоб навчальні дані охоплювали різні варіанти вимови, інтонації, темпу, діалектів, вікових особливостей мовців і умов навколишнього середовища. Таким чином модель отримує здатність ефективно узагальнювати закономірності та розпізнавати різні варіанти людського мовлення. Рівень складності мови, специфічність предметної області, вимоги до різноманітності прикладів у зв’язку з ризиком перенавчання суттєво збільшують складність добору навчальних даних [10]. Оскільки аудіосигнал є неперервним за часом і характеризується значними варіаціями гучності, частоти та спектрального складу, перед подачею в нейронну мережу проводиться підготовка вхідних даних, де сигнал проходить попередню обробку в декілька стадій, лише після чого параметри стають вхідними ознаками для моделі. Також ASR-модель повинна навчатися

працювати з часовими залежностями між сусідніми звуками або словами, отже під час навчання використовуються великі вікна часових фреймів, які дозволяють мережі сприймати мовлення як послідовний сигнал, а не як набір ізольованих звуків, а також залучаються рекурентні чи двонапрямлені мережі для врахування контексту. Через це навчання ASR-моделей вимагає великих обсягів пам’яті та часу, а також складної архітектури реалізації.

В контексті розпізнавання української мови навчання моделі ускладняється обмеженістю мовних ресурсів, адже датасети, які містять записи з низькою якістю або нерівномірним поданням різних діалектів, темпів мовлення та соціальних груп несуть ризик перенавчання моделі на дикторське мовлення і втрати точності розпізнавання мови в реальних умовах. У дослідженні [11] українську мову віднесли до числа малоресурсних мов. Недостатня узгодженість серед датасетів стандартів транскрипції для міток може призвести до неоднозначності між звуковим і текстовим поданням та відсутності стабільності, що впливає на якість процесу навчання моделі. Різноманітність діалектів та суттєва різниця в мовленні, акцентах, словниках представників різних областей України теж є складною задачею для узагальнення навіть сучасними нейронними мережами. Крім того, українська мова має обмежену підтримку в популярних фреймворках, що використовуються для розробки мереж глибинного навчання (TensorFlow, PyTorch та інші), більшість сучасних архітектур систем розпізнавання мовлення оптимізовані для англійської мови та потребують адаптування або навчання для розпізнавання української. Також для англійської мови є великі тестові набори (LibriSpeech, CommonVoice), створені для об’єктивної оцінки та вимірювання точності результатів моделей, аналогів чого для української мови наразі бракує.

Отже, навчання у системах автоматичного розпізнавання мовлення має низку характерних особливостей: воно вимагає великих наборів мічених даних, спеціальних алгоритмів для узгодження послідовностей та врахування контексту, складної попередньої обробки сигналів і потужних обчислювальних ресурсів. Побудова моделі розпізнавання українського мовлення ускладнюється меншою кількістю ресурсів – навчальних датасетів та технічних засобів, адаптованих для

роботи з українською мовою. Відкритий доступ до готових моделей, донавчаних до української мови, також є дуже обмеженим.

1.3 Методи глибинного навчання у задачах розпізнавання мовлення

Переважна більшість наукових статей з обробки природної мови, розпізнавання аудіоданих та їхньої генерації використовують моделі глибинного навчання. Глибинне навчання (Deep Learning, DL) – це підгалузь машинного навчання, яка використовує штучні нейронні мережі (Artificial Neural Networks, ANN) з багатьох шарів нейронів, кожен з яких витягує все більш абстрактні ознаки з вхідних даних для навчання на великих обсягах даних. Глибинне навчання відрізняється від традиційного машинного навчання: своєю здатністю автоматично виявляти складні шаблони та залежності в даних; наявністю багатошарових моделей «чорних скриньок» з прихованими шарами; потребою в більших обсягах навчальних даних та потужніших ресурсах; високою ефективністю в розв’язанні складних задач на неструктурованих вхідних даних. Дослідження [12] підтверджує, що моделі глибинного навчання показують кращі результати з вирішення складних задач розпізнавання мови, порівняно з традиційними системи.

Для розпізнавання аудіоданих застосовують різного роду глибинні нейронні мережі:

1. *Згорткові нейронні мережі (CNN)* використовують згорткові шари для виявлення локальних шаблонів у даних (спектрограм для аудіо). Згорткові шари застосовують фільтри до вхідних даних, щоб виявити локальні ознаки (наприклад, частоти звуку), ефективні для виявлення шаблонів у спектрограмах. Як показує дослідження [13], такі моделі мають вищу точність порівняно з рекурентними нейронними мережами (RNN) у класифікації навколишніх звуків – 85% до 78% у RNN. Також, в порівнянні з традиційними методами фільтрування аудіоданих, застосування CNN зменшує затримку до >10мс та покращує співвідношення «сигнал/шум» при роботі з онлайн-аудіопотоками [14]. А запровадження мультимодальної системи (візуального матеріалу в поєднанні з аудіорозпізнаванням мови) підвищує точність на 10–15% у сценаріях з високим

рівнем фонового шуму або частковою деградацією аудіо. CNN виділяють просторові ознаки з відеокадрів губ із точністю 90% [15].

2. *Рекурентні нейронні мережі (RNN)* обробляють послідовні дані за рахунок циклічних зв'язків між нейронами. Кожен елемент послідовності обробляється по черзі, мережа запам'ятовує попередні елементи через внутрішній стан (hidden state) та враховує попередні елементи послідовності для прийняття рішень. На кожному кроці RNN приймає вхідний елемент та внутрішній стан, який оновлюється на основі поточного входу та попереднього збереженого стану. Через проблему «зникаючого градієнта» RNN моделі мають обмежену здатність обробляти довгі послідовності. Проте поєднання RNN моделі з LTSM усуває проблему градієнта, дозволяє обробляти речення, а метрика відстеження помилок (WER) покращується на 15%, порівняно з традиційними моделями без застосування глибинного навчання [16].

3. *Довга короткострокова пам'ять (LSTM)* – покращена версія RNN мереж, яка вирішує проблему зникаючого градієнта за рахунок спеціальних «воріт», що контролюють, яку інформацію зберігати, ігнорувати, виводити або видаляти з пам'яті. У розпізнаванні мови LSTM може запам'ятовувати контекст попередніх слів, щоб краще розуміти поточне слово. Згідно з даними дослідження [17] гібридні моделі CNN-LSTM зменшують час навчання на 25%, зберігають при цьому точність на рівні 90% у розпізнаванні мови, що перевершує окремі CNN в умовах шуму. Поєднання CNN з трансформерами покращує метрику відстеження помилок на 10% при роботі з багатоакцентними аудіо даними в порівнянні з моделями LSTM [2]. Комбінація CNN-LSTM перевершила окремі CNN (точність 70%) та LSTM (точність 72%), оскільки змогла захопити як локальні шаблони, так і довгострокові залежності в емоційному мовленні за результатами використання в малоресурсному середовищі (племінна мова) [18].

4. *Трансформери (Transformers)* використовують механізми уваги (self attention), щоб аналізувати всі елементи послідовності одночасно, а не по черзі, як RNN. Кожен елемент послідовності взаємодіє з усіма іншими елементами, щоб визначити їхні взаємозв'язки. У розпізнаванні мови трансформери можуть

аналізувати весь контекст речення для точного розпізнавання слів. Результати трансформерів перевершують RNN у задачах автоматичного розпізнавання мови на 15% [19] за показником WER (частки помилкових слів) на зашумлених наборах даних, що пояснюється здатністю механізмів уваги захоплювати довгострокові залежності. Ці результати підтверджує і дослідження [20], де трансформери мають меншу частоту помилок, ніж RNN в довших висловлюваннях, проте потребують на 30% довшого періоду тренування та 40% більше оперативної пам'яті. Застосування трансформерів у формуванні транскрипції для тих, хто вивчає мови, зменшує WER на 15–20% порівняно з традиційними системами, а надання зворотного зв'язку щодо вимови учнів в умовах реального часу сягає 85% точності [21]. Використання трансформерів на діалектах має високі результати [22], проте підіймає проблему пониженої ефективності на портативних пристроях через ускладнену архітектуру та багатошаровість.

Застосування моделей глибинного навчання є актуальним підходом в галузі розпізнавання мови та аудіоданих. Проаналізовані джерела [2 – 22] збігаються на думці, що глибокі нейронні мережі показують кращі результати в порівнянні з традиційними інструментами, які використовувалися до їхньої появи. Також об'єднання декількох підходів, наприклад, поєднання RNN з LSMT [16], застосування CNN з трансформерами [2], CNN-LSMT гібриди [17, 18] має тенденцію отримувати кращі показники точності та частоти помилок, ніж ті самі інструменти окремо. Використання трансформерів самостійно або в поєднанні зі згортковими моделями є популярним підходом до розпізнаванні аудіоданих та показує високі результати якості в дослідженнях [2, 19, 20, 21, 22], проте постають проблеми ускладненої архітектури, багатошаровості, великої кількості необхідних ресурсів та обчислювальних потужностей для тренування моделей.

1.4 Аналіз архітектури сучасних ASR-моделей

Популярними сучасними архітектурами систем автоматичного розпізнавання мовлення, які об'єднують використання згорткових нейронних мереж для вилучення акустичних ознак та трансформерних блоків для

моделювання довготривалих залежностей є Wav2Vec2.0 та HuBERT. Однак їх відрізняє підхід до самонавчання.

Wav2Vec2.0 працює за принципом контрастивного навчання, де модель отримує одні й ті самі аудіодані, проте в двох різних формах: контекстне подання з трансформера та дискретне з блока квантизації. За квантизацію відповідає блок між CNN-енкодером та трансформером, який отримує приховане подання аудіосигналу зі згорткового блоку та перетворює його безперервні (акустичні) ознаки на набір дискретних кодів, що відповідають типовим звуковим патернам. Трансформер паралельно отримує ті самі акустичні ознаки зі згорткового енкодера та перетворює їх на контекстні вектори, які містять подання не тільки про локальний символ, а й ті, що йдуть перед та після нього в контексті слів та цілих речень. Модель навчається визначати, які з цих двох форм відповідають одному фрагменту мовлення, порівнює контекстне подання з трансформера з кількома кандидатами з квантового й вчиться вибирати з них правильне за допомогою функції контрастивної втрати. Це робить модель орієнтованою на акустичну реконструкцію [23]. Після процесу попереднього навчання Wav2Vec 2.0 на великих наборах даних, можна донавчати модель для конкретних задач автоматичного розпізнавання мовлення.

HuBERT поєднує згорткову нейронну мережу з трансформерною архітектурою BERT, проєкційним та CTC-шарами. Після отримання ознак зі згорткового енкодера відбувається їхня кластеризація за допомогою алгоритму k-means в шарі кластеризації, яка перетворює аудіо у дискретні псевдо-мітки (номери кластерів). Також деякі фрейми спектрограм маскуються, а трансформер навчається передбачати ці мітки для замаскованих фрагментів, а не самі спектральні ознаки. В наступних ітераціях кластеризація відбувається вже не на даних CNN-енкодера, а на минулих вихідних даних з трансформера. Такий підхід робить архітектуру більш дискретизованою і зміщує її увагу з низькорівневих акустичних патернів до мовних одиниць, наближаючи модель до розуміння структури мови. Після навчання на великих обсягах немічених аудіоданих, модель донавчається на даних із транскрипціями [24].

Крім розглянутих, мультимовні моделі Whisper вважають одними з найефективніших у галузі розпізнавання мовлення. На відміну від попередніх прикладів зі згортковими складовими, Whisper має два повноцінних трансформера для кодування (encoder) та декодування (decoder) подібно для моделей машинного перекладу, а також працює як генеративна модель. Оскільки моделі були навчені на величезному наборі мічених даних, вони не потребують попереднього самонавчання (маскування, кластеризації) і здатні працювати одразу: не лише розпізнавати мовлення, а й перекладати, визначати мову та транскрибувати аудіо в реальному часі. Така архітектура є значно більшою та важчою за попередньо згадані. Також моделі «sequence-to-sequence», які працюють в авторегресивному режимі, послідовно генерують вихідні токени один за одним, працюють повільніше, особливо з малоресурсними мовами, що потребують генерації більшої кількості токенів на слово і збільшують час обробки, оскільки вони рідше зустрічалися в навчальному наборі даних.

Отже, Wav2Vec2.0 навчається на відмінностях із застосуванням контрастивного навчання, а HuBERT – на відновленні прихованих одиниць через кластеризацію та маскування. Дослідження порівняння ефективності цих моделей за метрикою WER показує, що за наявності великого обсягу навчальних даних, обидві моделі мають високі показники точності, проте HuBERT має перевагу, коли необхідне глибоке контекстне розуміння тексту [25]. За умов обмежених мічених даних HuBERT показує кращу продуктивність після тонкого налаштування, оскільки попереднє тренування з BERT архітектурою формує стабільніші кластерні цілі, корисні при обмежених мічених даних, в той час як wav2vec2.0 потребує більшої кількості прикладів для досягнення таких самих показників точності. На зашумлених аудіозаписах обидві моделі показують високі результати, якщо їхній навчальний датасет мав достатнє різноманіття прикладів, проте wav2vec2.0 може потребувати більший їхній обсяг. Обидві моделі також потребують значних обчислювальних ресурсів для тренування моделей, HuBERT зазвичай навчається повільніше за wav2vec2.0 через наявність кластеризації як додаткового кроку попередньої обробки даних [25].

Порівняльну характеристику різних моделей подано в табл.1.1.

Таблиця 1.1 – Порівняльна таблиця моделей Wav2Vec2.0, HuBERT, Whisper

	Wav2Vec2.0	HuBERT	Whisper
Тип попереднього навчання	Контрастивне навчання: співставлення контекстних та квантованих ознак	Масковане навчання з кластеризацією: передбачення кластерних псевдо-міток	Навчання на величезному мультимовному наборі мічених аудіо
Перший рівень обробки сигналу	Згортковий енкодер та квантова кодова книга	Згортковий енкодер та k-means кластеризація	Повноцінний трансформерний encoder
Тип трансформера	Трансформер для контекстних представлень (encoder-only)	BERT-архітектура (encoder-only)	Encoder-decoder (2 трансформери)
Стійкість до шуму	Висока за достатньої різноманітності аудіо та донавчання з вчителем на великих корпусах даних	Висока за донавчання на малих наборах мічених даних, стабільніша за wav2vec2.0	Дуже висока, модель навчена на багатомовному та шумному датасеті
Продуктивність у малоресурсних мовах	Гірша за малого обсягу мічених даних під час донавчання	Краща завдяки кластерним цілям та швидша завдяки BERT-підходу	Менш точна для мов, яких не було в датасеті, повільніша через авторегресію
Швидкість роботи	Найвижча завдяки меншому розміру архітектури	Висока, але повільніша за Wav2Vec2.0 через кластеризацію	Низька через авторегресивне декодування токенів
Вимоги до ресурсів	Високі, потребує великих датасетів	Високі, модель менш вимоглива до обсягів мічених даних	Дуже високі (великий трансформер з авторегресивним декодером)
Задачі з найбільшою ефективністю	Акустично складні задачі за наявності великих аудіокорпусів	Мови з обмеженими ресурсами, глибоке контекстне розуміння	Мультимовні системи, переклад та транскрибування в режимі реального часу
Недоліки	Потребує багато даних; менш ефективна в малоресурсних умовах	Кластеризація уповільнює процес попередньої обробки вхідних даних, ускладнена архітектура	Повільна генерація, великі вимоги до GPU, повільніша за реальний час на CPU, складна архітектура
Здатність працювати з українською мовою	Потрібне донавчання	Потрібне донавчання	Не потребує донавчання

За умов обмеженості навчальних ресурсів та обчислювальних можливостей, модель HuBERT є більш ефективною з погляду використання ресурсів, оскільки вона потребує меншої кількості даних для досягнення конкурентоспроможної точності. Після аналогічного донавчання HuBERT демонструє подібний рівень WER у задачі розпізнавання малоресурсних мов, як і модель Whisper-large-v3,

проте працює у 10–30 разів швидше й споживає приблизно у 2,5 рази менше ресурсів за даними досліджень [26].

Порівняння архітектур Wav2Vec2.0, HuBERT і Whisper показує, що кожна з них має яскраві сильні та водночас суттєві слабкі сторони. Wav2Vec2.0 вирізняється високою швидкістю роботи та ефективністю за умови наявності великих корпусів аудіо, але значно втрачає якість у малоресурсних мовах і потребує обов'язкового тривалого донавчання для української.

HuBERT демонструє стабільність, високу стійкість до шуму та кращі результати за умов обмежених даних, проте кластеризація та складність архітектури збільшують обчислювальні витрати і роблять процес навчання повільнішим. Whisper забезпечує високу точність, мультимовність, однак є надзвичайно ресурсномісткою моделлю, її авторегресивний декодер сильно сповільнює обробку, а вимоги до GPU роблять її малопридатною для легких або швидкодіючих систем. HuBERT працює швидше за Whisper, оскільки не потребує послідовного декодування, точніше за Wav2Vec2.0 у малоресурсних умовах на менших обсягах мічених даних, є стійкою до шуму, діалектів і варіативності мовлення, що робить її більш оптимальною при виборі для застосування в системах розпізнавання українського мовлення. Проте HuBERT та Wav2Vec2.0 також вимагають додаткового навчання для коректної роботи з українською мовою, що потребує великих обсягів даних, тоді як Whisper є занадто масштабною й важкою для практичної інтеграції у швидкі та ресурсообмежені системи. Це створює реальну потребу у побудові оптимізованої моделі зі спрощеною BERT-подібною архітектурою, навченою саме на українському наборі даних, що дозволяє усунути недоліки наявного підходу і зберегти його сильні риси, забезпечити якісне та швидке автоматичне розпізнавання мовлення за умов обмежених ресурсів.

1.5 Висновки до розділу 1

У розділі здійснено системний огляд теоретичних засад автоматичного розпізнавання мовлення, визначено його поняття, алгоритмічні етапи та ключові особливості. Розкрито процес перетворення аудіосигналу на текстову

транскрипцію, починаючи від попередньої обробки даних і вилучення ознак до акустичного моделювання, декодування та постобробки результатів. Показано специфіку задачі розпізнавання мовлення, яка відрізняється від класичних задач класифікації через потребу узгодження послідовностей змінної довжини та врахування контексту, що зумовлює застосування алгоритму CTC та методів регуляризації для уникнення перенавчання.

Окремо акцентовано на проблемах малоресурсності української мови, що ускладнює створення якісних моделей через обмеженість корпусів, різноманітність діалектів, відсутність стандартизованих тестових наборів та недостатню підтримку у сучасних фреймворках. Якість навчальних даних визначає продуктивність ASR-систем, а для української мови практична реалізація вимагає адаптації наявних моделей, створення якісних датасетів та врахування фонетичних і діалектних особливостей.

Розглянуто методи глибинного навчання, які застосовуються у сучасних системах розпізнавання мовлення, зокрема згорткові, рекурентні та LSTM-мережі. CNN демонструють переваги у вилученні ознак зі спектрограм, проте для української мови потребують додаткового обґрунтування їхньої оптимальності порівняно з іншими архітектурами.

Тим самим розділ сформував теоретичну основу для подальшої розробки системи автоматичного розпізнавання українського мовлення на основі згорткових нейронних мереж, окресливши як загальні принципи роботи ASR-систем, так і специфічні виклики, пов’язані з українською мовою.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ АВТОМАТИЧНОГО РОЗПІЗНАВАННЯ УКРАЇНСЬКОГО МОВЛЕННЯ

2.1 Принципи BERT-архітектури

Архітектура BERT та її принципи, як-от: багатошарове трансформерне кодування, двобічна увага, маскування та ефективне моделювання довготривалих залежностей, успішно використовуються в сучасних рішеннях задач автоматичного розпізнавання мовлення.

BERT-підхід відрізняється від авторегресивної трансформерної архітектури тим, що моделі не генерують вихід послідовно, а створюють контекстні подання для всіх елементів одночасно, що дозволяє охопити повний глобальний контекст. Класичний трансформер складається з двох частин: енкодера і декодера з перехресною увагою, в той час як BERT-архітектура немає декодера взагалі, а енкодер зосереджений на аналізі контексту, а не на генерації послідовностей. Механізм перехресної уваги замінює багатоголова самоувага, що дозволяє моделі паралельно аналізувати декілька аспектів взаємодій у послідовності: локальні зв’язки, далекі залежності, патерни ритму чи інтонації. Це особливо корисно в задачах ASR, де модель має одночасно фокусуватися на дрібних фонетичних деталях і глобальній структурі мовлення, тому незалежно від розміру моделі, механізм багатоголової уваги є ключовим для досягнення точних вихідних послідовностей.

На відміну від рекурентних мереж, врахування контексту яких обмежене за довжиною, BERT паралельно аналізує всю послідовність та формує подання кожного елемента з урахуванням взаємодії з усіма іншими, що є особливо ефективним у задачах, де якість кодування контексту визначає точність моделі.

Через відсутність вбудованого розуміння послідовностей BERT-архітектура використовує позиційне кодування. У задачах розпізнавання мовлення послідовність аудіокадрів має часову структуру, а позиційне кодування допомагає моделі відстежувати переходи між фонемами та темпоральні патерни сигналу.

Шари трансформерного енкодера BERT мають уніфіковану структуру, яка має механізм самоуваги, feed-forward мережу, шари нормалізації та залишкових зв’язків. Наявність таких залишкових зв’язків покращує стабільність глибокого навчання й допомагає моделі зберігати інформацію між шарами та враховувати контекст як нових, так і попередніх вихідних даних. Для збільшення швидкодії кількість шарів ASR-моделі може бути скорочена, проте саме ці ключові елементи допомагають досягти високої точності результату навіть у середовищах з обмеженими ресурсами.

Відсутність декодера та механізму перехресної уваги спрощує архітектуру і робить її більш придатною до задач класифікації, розпізнавання та сегментації. При адаптації до задач автоматичного розпізнавання мовлення, використання BERT-подібної архітектури допомагає отримати високоякісні контекстні подання із мінімальною кількістю необхідних шарів та параметрів [27]. У табл. 2.1 наведено SWOT-аналіз BERT-підходу до архітектури моделі розпізнавання мовлення в поєднанні зі згортковим енкодером для вилучення акустичних ознак.

Таблиця 2.1 – SWOT-аналіз BERT-архітектури в поєднанні CNN-енкодером.

Сильні сторони	**Слабкі сторони**
– механізм двобічної самоуваги дозволяє отримувати глибокі контекстні подання кадрів у послідовності, що покращує точність у шумних умовах чи при швидкому мовленні; – поєднання зі згортковим енкодером забезпечує баланс локальних та глобальних ознак; – висока масштабованість для досягнення оптимального співвідношення швидкодії та точності; – енкодерна природа моделі добре інтегрується з алгоритмом CTC, для спрощення навчання; – висока стійкість до шумів.	– обмеження довжини контексту через класичне позиційне кодування; – квадратична складність уваги у трансформерних шарах може обмежувати роботу з довгими аудіо; – потреба ретельного балансування згорткового та трансформерного енкодерів.
Можливості	**Загрози**
– легка адаптація до малоресурсних мов. – інтеграція з невеликою мовною моделлю для підвищення точності декодування; – можливість мобільної оптимізації; – розширення до мультимодальних даних.	– розвиток LLM-мультимодальних моделей, які поєднують роботу з аудіо-, відеоданими та текстом; – витіснення класичних енкодерних моделей універсальними багатомовними Whisper-подібними архітектурами.

BERT-подібна архітектура є потужним рішенням для систем автоматичного розпізнавання мовлення, особливо коли потрібно досягти балансу між точністю, швидкодією та компактністю моделі. Завдяки двобічній самоувазі й поєднанню локальних і глобальних ознак така архітектура демонструє високу стійкість до шумів та забезпечує якісне моделювання контексту, що робить її ефективною навіть у складних акустичних умовах. Водночас наявні архітектурні обмеження, серед яких класичне позиційне кодування, квадратична складність уваги та потреба тонкого налаштування взаємодії між енкодерами, можуть знижувати ефективність під час обробки довгих аудіофрагментів. Попри обмеження BERT-подібна архітектура зберігає потенціал для адаптації до малоресурсних мов, мобільної оптимізації та розширення до мультимодальних даних. Основною загрозою залишається конкуренція зі сторони універсальних мультимодальних моделей і Whisper-подібних систем, проте в сегменті відкритих та легких архітектур цей підхід залишається конкурентним і практичним.

2.2 Проєктування архітектури моделі

Використання BERT-подібної архітектури у сучасних системах автоматичного розпізнавання мовлення дозволяє моделі не лише розпізнавати послідовність звуків, а й розуміти контекст мовлення, що забезпечує високу точність, стійкість до шумів і природність відтворення мовного потоку у текстовій формі. За цим принципом проєктна модель поєднуватиме шари згорткової нейронної мережі для вилучення акустичних ознак та трансформерний блок для формування контекстних уявлень послідовності сигналу. У CTC-базованих моделях розпізнавання мовлення використовується лише енкодерна частина трансформера, оскільки декодер замінює CTC-алгоритм, який самостійно відновлює цільову послідовність із логітів.

Основні елементи архітектури:

1) *Згортковий енкодер (Feature Extraction)*. Кілька 1D-згорткових шарів приймають на вхід послідовність акустичних ознак в якості мел-спектрограми, зменшують часову розмірність, виділяють абстрактні патерни за допомогою функції активації GELU та створюють вектори подання аудіо, які зберігають найважливішу

фонетичну інформацію. Подібно блоку Feature Encoder у BERT-архітектурі, ці елементи відповідають за первинне вилучення ознак із сирого сигналу.

2) *Позиційне кодування (Positional Encoding).* Оскільки трансформер не має вбудованого поняття послідовності, до векторів додають позиційні кодування, які відображають порядок фреймів у часі. Це допомагає моделі розуміти часові зв'язки між елементами.

3) *Трансформерний контекстний енкодер (Transformer encoder)*. Серія шарів multi-head self-attention та feed-forward перетворює послідовність згорткових ознак у контекстно збагачені вектори. Багатошарова мережа на основі механізму самоуваги є ключовою рисою моделей BERT. Вона дозволяє здійснювати двонапрямлене кодування, коли кожному звуковий фрагмент може сприймати інші частини сигналу та враховувати контекст як зліва, так і справа від нього. Feed-Forward мережа здійснює нелінійне перетворення контекстних ознак, яке допомагає моделі будувати узагальнені подання. Після кожного блока додаються шари з нормалізації для стабільності й регуляризації.

4) *Вихідний шар (Output Dense).* На останньому етапі трансформерні подання подають у повнозв'язний шар, де відбувається перетворення представлень із трансформерного енкодера в набір необроблених імовірностей (логітів) для мовного можливого символу зі словника моделі з введенням додаткового класу blank для подальшого CTC-алгоритму. Кожен фрейм на виході відповідає одному рядку в матриці логітів. Тим самим цей повнозв'язний шар виконує роль класифікатора та надає CTC-декодеру ймовірнісну карту символів по часовій осі аудіосигналу.

5) *CTC модель (CTC decoder).* Метод декодування та функція втрат CTC накладається після останнього шару моделі. На відміну від класичних трансформерів із самостійним модулем декодування, CTC не генерує вихідну послідовність поетапно, а працює з усією послідовністю логітів, які надходять з енкодера та застосовує алгоритм вирівнювання для співставлення аудіопотоку довільної довжини з текстовою транскрипцією без маркування часових меж. З використанням порожнього символу та правила згортання повторів послідовність

символів і пропусків перетворюється у зв’язний текст. У цій архітектурі CTC виконує функцію нелінійного декодера, який дозволяє моделі навчатися без попередньо заданого вирівнювання між аудіо та текстом, що робить її ефективною й спрощеною альтернативою традиційному трансформерному декодеру в задачах автоматичного розпізнавання мовлення. У поєднанні з контекстними ознаками, отриманими з трансформера, алгоритм забезпечує точне і гнучке перетворення мовлення у текст навіть за наявності шуму чи варіацій дикції.

Діаграма компонентів на рис.2.1 показує архітектуру ASR-моделі.

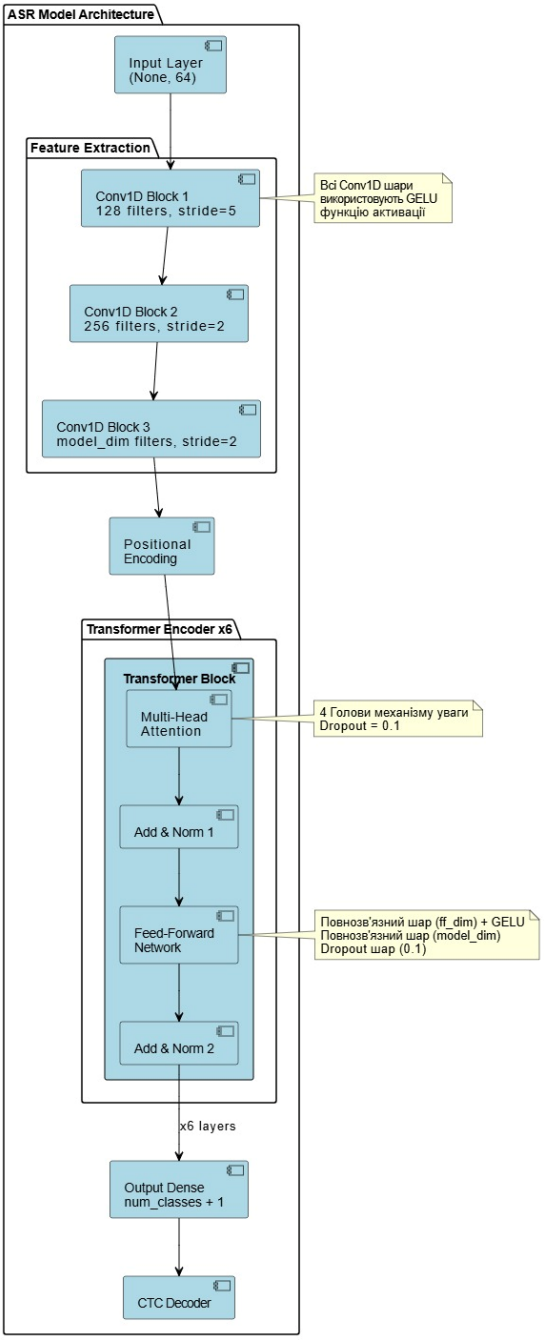

Рисунок 2.1 – Архітектура ASR-моделі

Для доступного обсягу навчальних даних (приблизно 400 годин українського аудіо) така структура забезпечує оптимальну модельну ємність: менша кількість шарів призводила б до недостатнього моделювання довготривалих фонетичних і морфологічних залежностей, властивих українській мові, тоді як значно більша глибина викликала б зниження пропускної здатності та потребу у великих обчислювальних ресурсах. Розмірність в 3 згорткові шари та 6 трансформерних блоків дозволяє моделі ефективно інтегрувати локальні акустичні ознаки, сформовані згортковим енкодером, та контекстні залежності, які обробляються трансформером, з можливістю забезпечення низьких значень метрики похибки WER за збереження швидкодії в режимі реального часу. Такий вибір глибини є оптимальним компромісом між точністю, стійкістю та продуктивністю ASR-системи.

Модель побудована за принципами BERT-архітектури, проте на відміну від систем як HuBERT, вона не проходить попереднього самонавчання з кластеризацією, а навчається на наборах мічених даних за допомогою CTC, тому така модель є простішою в реалізації та ресурсно економною.

2.3 Основні компоненти моделі

2.3.1 Згорткова нейронна мережа

Особливість нейронних мереж полягає в можливості самостійно навчатися, діяти на підставі попередньо набутого досвіду та мінімізувати кількість помилок після кожної ітерації. За зразком структури нервової системи людини, такі мережі складаються з великого числа окремих взаємопов’язаних обчислювальних елементів – нейронів, організованих у шари, які обробляють інформацію послідовно або рекурентно. Нейрони отримують вхідні сигнали від вхідного шару або інших нейронів у вигляді числового вектора, після чого кожен сигнал множиться на вагу, підсумовується та активується за допомогою функції активації. За потреби може застосовуватися зміщення, що зсуває функцію у відповідному

напрямку і дозволяє нейронам активуватися, навіть якщо сума є негативною. Алгоритм функціонування штучного нейрона описано за формулою [28]:

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \quad (2.1)$$

де x_i – множина вхідних сигналів;

w_i – їхні вагові коефіцієнти;

b – зміщення (bias);

$f()$ – функція активації.

Сила та важливість сигналу, який надходить від одного нейрона до іншого визначається вагами, оновлення яких у процесі навчання мережі сприяє формуванню здатності моделі до узагальнення вхідних даних, а налаштування ваг і зміщення відбувається так, щоб мінімізувати помилку моделі на навчальній вибірці. Сукупність нейронів, які працюють одночасно на одному рівні мережі є шаром нейронної мережі, а системи, що містять декілька прихованих шарів, вважаються глибинними.

Згорткові нейронні мережі (Convolutional Neural Networks, CNN) є різновидом глибинних нейронних мереж. Вони розроблені для ефективної обробки двовимірних або тривимірних даних, що мають просторову або часову структуру (зображення, спектрограми тощо). У задачі розпізнавання мовлення згорткові нейронні мережі здатні аналізувати мел-спектрограми шляхом фіксування згортковими шарами їхніх локальних та загальних часових і частотних закономірностей, характерних для певних фонем або слів. Завдяки цьому шари згорткових моделей часто використовуються як складові частини більш складних архітектур в поєднанні з іншими моделями. Вони забезпечують попереднє вилучення, формування ієрархічного уявлення ознак своїми послідовними шарами та зменшення розмірності вхідних даних, що спрощує їхню подальшу обробку.

Такі мережі дотримуються концепцій:

– локального з’єднання нейронів, яке, на відміну від повнорозмірних мереж, дозволяє кожному нейрону опрацьовувати лише окремий фрагмент вхідних

даних та будувати локальні закономірності. Локальні з’єднання дозволяють скоротити кількість параметрів і підвищити здатність моделі до узагальнення;

– спільного використання ваг: застосування одного й того самого фільтру, а отже й групи ваг до всіх позицій, куди він переміщується під час операції згортки, завдяки чому мережа може розпізнати однакову ознаку, незалежно від її розташування у вхідному сигналі, та значно зменшити кількість параметрів;

– зменшення розмірності шляхом здійснення відбору найбільш виражених ознак (max pooling) й обчислення середнього значення (average pooling) у локальних фрагментах карти ознак, що допомагає відсіяти надлишкові обсяги даних та обчислень, підвищити стійкість мережі до варіацій, шумів і дрібних деформацій вхідних даних [29].

Основними типами шарів, які характеризують CNN, є згортковий, пулінговий (шар об’єднання) та повнозв’язний шари, кожен з яких отримує вхідні дані від попереднього та передає результати наступному, доки не буде отримано кінцевий результат.

1) Згортковий шар

Завдання згорткових шарів полягає в перетворенні вхідних даних для вилучення їхніх локальних ознак та зменшення кількості параметрів, зі збереженням при цьому просторових (часових) співвідношень між елементами даних. Ядро (фільтр), яке має вигляд невеликої матриці, пересувається з певною величиною кроку по часовій та частотній осях вхідної мел-спектрограми, яку потрібно згорнути. На кожному кроці обчислюється скалярний добуток між елементами фільтра і відповідним фрагментом мел-спектрограми, добутки підсумовуються в одне значення на виході. В результаті формується карта ознак – нова матриця, яка відображає, наскільки певний заданий фільтром шаблон проявляється у різних ділянках спектрограми. Для вилучення різних типів ознак у мережі використовується кілька фільтрів, кожен з яких навчається розпізнавати власний набір патернів, має власні вагові коефіцієнти та окрему карту активації. Вихідні карти ознак кожного фільтру об’єднуються та утворюють багатоканальне подання вхідних даних. Перший шар виділяє низькорівневі ознаки сигналу, проте

наступні згорткові шари мають можливість комбінувати ці базові ознаки та формувати більш високоабстрактні подання, де нейрони реагують уже не на окремі спектральні лінії, а на комбінації звуків або їхніх динамік та здатні виділяти вже середньо- й високорівневі ознаки вхідних даних фільтрами другого та більше порядків.

Вагові коефіцієнти ядер оновлюються під час навчання через зворотне поширення помилки (backpropagation). Після проходження сигналу через мережу (forward pass) модель порівнює вихідні дані з очікуваними та обчислює величину помилки, яка передається у зворотному напрямку через усі шари мережі. Для кожного нейрона визначається, наскільки його активація вплинула на загальну помилку. На основі цього обчислюються градієнти як часткові похідні функції помилки відносно кожного вагового коефіцієнта, які покажуть, як саме потрібно змінити вагу. Корекція ваг здійснюється за алгоритмом градієнтного спуску – поступового руху у напрямку зменшення значення функції помилки. Таким чином мережа поступово навчається визначати оптимальні фільтри, які найкраще характеризують особливості вхідного сигналу [30].

Перед подальшими операціями застосовується нелінійна функція активації, яка дозволяє мережі виявляти складні залежності. Однією з найбільш поширених є функція ReLU, яка замінює всі від’ємні вхідні значення на нуль та тримає нейрони з вхідним сигналом, меншим нуля, неактивованими для пришвидшення навчання. Проте в сучасних системах все частіше використовують функцію GELU, яка на відміну від стандартної ReLU, не відкидає від’ємні значення повністю, а плавно зменшує їх вплив, залежно від ймовірності, що вхідне значення належить до нормального (гауссівського) розподілу:

$$GELU(x) = x * \Phi(x) \tag{2.2}$$

де Φ(x) – функція розподілу Гаусса [31].

Чим більше вхідне значення, тим вища ймовірність пропустити цей сигнал повністю, а чим менший – тим сильніше він приглушується. Ця функція враховує

статистичну природу даних і дозволяє моделі працювати більш стабільно та точно на великих обсягах даних та враховувати складніші закономірності.

2) Шар об'єднання (пулінговий)

Карта ознак після згортки не зберігає точної позиційної інформації, а отже навіть незначні зміни можуть змінити її вигляд. Шар об'єднання виконує агрегацію ознак із попередніх карт задля зменшення їхньої розмірності шляхом збереження найважливіших ознак із загальної кількості. Цей шар додає підвищення узагальнювальної здатності мережі та суттєво зменшує кількість параметрів. Він використовує ядро для здійснення операції об'єднання. Методи вибору максимуму в локальному фрагменті (максимальний пул) та зглажування ознак обчисленням середнього локального значення (середній пул) є найбільш часто використовуваними для згорткової нейронної мережі. Завдяки операціям об'єднання CNN стають менш чутливими до незначних зсувів або деформацій у даних, а шляхом чергування згорткових і пулінгових шарів мережа поступово переходить від простих локальних патернів до високорівневих абстракцій [28].

3) Нормалізаційні та Dropout-шари

Нормалізаційні і Dropout-шари відіграють важливу роль у стабілізації та підвищенні ефективності навчання глибинних нейронних мереж.

Пакетна нормалізація (Batch Normalization) – метод нормування вихідних значень нейронів усередині пачки (batch) для зменшення «зсуву внутрішніх ковариацій». Такий зсув виникає, коли під час навчання розподіл вхідних даних у шарі постійно змінюється через оновлення ваг попередніх шарів, що змушує цей шар пристосовуватися до змін та вповільнює процес навчання. Пакетна нормалізація дозволяє зменшити цей ефект шляхом забезпечення приблизно однакового розподілу середнього рівня активації та розкиду між шарами моделі. Вона дозволяє масштабувати вхідні дані для передачі їх на наступний шар мережі та зменшувати кількість кроків навчання моделі. Як наслідок, навчання стає стабільнішим і швидшим, а мережа менш чутливою до вибору початкових параметрів.

Dropout – це метод регуляризації, який штучно зменшує взаємозалежність між нейронами для зменшення ризику перенавчання. Під час тренування випадкова частина нейронів тимчасово вимикається, їхні вихідні дані не передаються наступним шарам. Це змушує мережу не покладатися надлишково на окремі шляхи активації, а навчатися узагальненим, стійким ознакам. Під час тестування (інференсу) всі нейрони активні, а їхні виходи масштабуються відповідно до ймовірності виключення під час тренування [32].

4) Повнозв'язний шар

У повнозв'язних (dense) шарах кожен нейрон має зв'язок з усіма нейронами вхідного шару, що дозволяє здійснювати глобальну обробку інформації. Зазвичай це останній шар, який отримує вектор перетворених карт ознак, отриманих після кількох згорткових і пулінгових шарів. Кожен нейрон цього шару відповідає певній цільовій категорії (символу, букві тощо) та обчислює зважену суму всіх вхідних ознак, після чого перетворює її в набір чисел (логітів) з оцінками ймовірностей для кожного моменту часу. Функція активації Softmax перетворює ці логіти у ймовірнісний розподіл між усіма можливими вихідними категоріями. Для визначення остаточної послідовності символів отримані після функції активації імовірності передаються до CTC-декодера.

2.3.2 Алгоритм CTC

Одним із ключових елементів сучасних систем автоматичного розпізнавання мовлення є алгоритм Connectionist Temporal Classification (CTC). Він застосовується, коли вхідні та вихідні послідовності мають різну довжину, а відповідність між ними, кількість фреймів, їх початок та кінець – невідомі. Щоб упоратися з невизначеністю у тривалості звуків, накладанням або повторенням символів, CTC вводить спеціальний порожній псевдосимвол (blank), який позначає відсутність будь-якого символу та дозволяє моделі коректно розмежовувати літери, вирівнювати аудіо і текст та не плутати їх із пробілами між словами.

Під час навчання алгоритм CTC аналізує всі можливі комбінації символів, які можуть призвести до правильного тексту й обчислює загальну ймовірність цих

комбінацій. Кожне передбачення порівнюється з правильною транскрипцією, після чого обчислюється функція втрат (CTC Loss), яка показує, наскільки сильно вихідний результат моделі відрізняється від очікуваного. На основі цієї похибки відбувається зворотне поширення помилки (backpropagation), що коригує ваги моделі у напрямі зменшення помилки. Таким чином, модель поступово вдосконалює свої внутрішні подання звуку та лінгвістичних закономірностей. Під час передбачення (CTC Decoding) модель вибирає найімовірніший символ для кожного фрейму, у тому числі введеного порожнього blank, після чого об’єднує повтори символів, не розділені порожнім, видаляє всі порожні символи та отримує фінальну послідовність [33].

Поєднання згорткових нейронних мереж із алгоритмом CTC є одним із найефективніших підходів у сучасних системах автоматичного розпізнавання мовлення, що дозволяє використовувати сильні сторони обох компонентів Згортковий енкодер забезпечує високу якість вилучення акустичних ознак, зменшення кількості параметрів, стійкість до шумів аудіоданих та оптимізацію обробки великих обсягів даних на GPU завдяки паралельності операцій згортки. CTC в свою чергу дозволяє системі самостійно визначити, на яких часових інтервалах звучить конкретний звук та навчитися динамічно вирівнювати послідовності без ручного втручання. Недоліком такого поєднання є складність у моделюванні довготривалих залежностей, тому CTC часто комбінується з іншими архітектурами для моделювання контексту, наприклад, трансформерами.

2.3.3 Трансформерний енкодер

Архітектура трансформерів замінила рекурентні мережі в задачах обробки послідовностей завдяки механізму уваги, позиційному кодуванню та паралелізації, що забезпечують більшу ефективність обчислень та глибше розуміння контексту. Здатність трансформера обробляти всі елементи послідовності одночасно значно пришвидшує тренування та краще використовує сучасні GPU. Структура енкодерного трансформерного блока зображена на рис. 2.2.

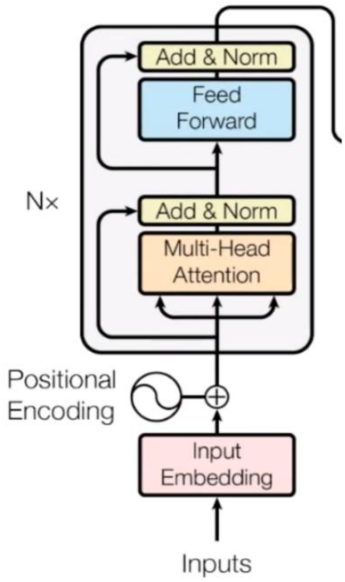

Рисунок 2.2 – Схема трансформерного енкодера [34]

Ключові компоненти трансформерів:

1) *Позиційне кодування*. Для того щоб модель могла розрізняти порядок елементів у послідовності, до вхідних векторів додаються позиційні коди. На відміну від рекурентних мереж, трансформер не має вбудованого механізму пам'яті, тому інформація про послідовність вводиться штучно через позиційні вектори, що визначають місце кожного елемента у послідовності та зберігають її структуру.

2) *Механізм самоуваги.* Шар самоуваги визначає, які частини вхідної послідовності є найважливішими для поточного елемента. Для цього кожен вхідний вектор перетворюється у три різні подання: запит (Query), ключ (Key) та значення (Value). На основі скалярного добутку запиту та ключа обчислюється ступінь подібності між елементами, який після нормалізації перетворюється у вагові коефіцієнти. Нормалізація відбувається за допомогою функції Softmax [35]:

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_i}} \tag{2.3}$$

де z_i – елемент вхідного вектора;

e^{z_i} – експонента вхідного вектора;

$\sum_{j=1}^{n} e^{z_i}$ – сума експонент усіх елементів вектора.

Отримані ваги використовуються для зваженого підсумовування значень та утворюють нове подання, яке відображає контекст усієї послідовності. Такий підхід дозволяє моделі гнучко фокусуватися на релевантних частинах вхідного сигналу, незалежно від їхньої віддаленості у часі. Для підвищення здатності моделі до узагальнення застосовується механізм багатоголової уваги, де декілька незалежних шарів самоуваги працюють паралельно, а результати їхньої роботи об'єднуються для отримання багатогранного подання контексту. Залежно від специфіки задачі трансформер часто має і декодер, який містить власний шар самоуваги та додатковий шар міжмодальної (перехресної) уваги, що дозволяє співвідносити ознаки аудіо з відповідними мовними одиницями.

3) *Feed-forward блок*. Після проходження через механізм уваги вихідні вектори обробляються позиційною мережею прямого поширення (FFN), що складається з двох повнозв'язних шарів з нелінійною активацією. Вона виконує роль перетворювача ознак, що дозволяє моделі узагальнювати отримані ознаки та відокремлювати більш високорівневі закономірності. Кожен блок завершується операціями нормалізації та залишкового зв'язку, що стабілізує навчання і запобігає втраті інформації.

4) *Вихідний шар*. У трансформерній моделі вихідний шар виконує роль, подібну до повнозв'язного шару згорткової мережі, але він працює не з просторовими ознаками, а з послідовними контекстними поданнями, сформованими механізмом уваги. Він перетворює отримані подання у ймовірнісні розподіли для кожного можливого символу, що у системах автоматичного розпізнавання мовлення можуть інтерпретуватися за допомогою алгоритму CTC [35].

Трансформерна архітектура забезпечує гнучке, контекстно-орієнтоване та ефективне моделювання послідовностей, завдяки чому вона стала важливим елементом сучасних моделей розпізнавання мовлення, таких як Wav2Vec2, HuBERT та Whisper.

2.4 Інструменти реалізації

Для розробки сучасних систем розпізнавання мовлення на основі нейронних мереж є широкий спектр інструментів.

TensorFlow – фреймворк, створений компанією Google, має широкий набір інструментів для забезпечення розробки, навчання та розгортання глибинних нейронних мереж, а також підтримує всі ключові компоненти сучасних архітектур: згорткові, рекурентні, трансформерні, CTC-шари тощо. Однією з найважливіших переваг TensorFlow є його модуль Keras, який забезпечує високорівневий інтерфейс для швидкого створення, тестування та оптимізації моделей, дозволяє легко комбінувати різні типи шарів та створювати архітектури будь-якої складності. TensorFlow має вбудовані засоби для навчання з вчителем, без вчителя та з підкріпленням, підтримує функції втрат, такі як CTC Loss та різні алгоритми оптимізації. Система автоматично обчислює градієнти під час зворотного поширення помилки, що значно спрощує процес навчання моделей. Також підтримка багатоплатформності дозволяє розгортати моделі на мобільних пристроях, вебзастосунках або вбудованих системах за допомогою TensorFlow Lite чи TensorFlow.js [36].

TensorFlow має можливість використання графової моделі виконання операцій. На відміну від звичайного імперативного режиму, де обчислення виконуються покроково без зберігання попередніх результатів, така модель спершу будує граф виконання, де кожна операція подана у вигляді вузла, а зв’язки між ними у вигляді ребер, що передають тензори (багатовимірні масиви даних). Кожна функція додається до цього графа та утворює цілісну структуру обчислень, що оптимізується перед запуском, а отже TensorFlow може заздалегідь проаналізувати залежності між операціями, об’єднати або скоротити їх, виконати паралелізацію та використати апаратні ресурси максимально ефективно. Такий підхід прискорює роботу в багато разів, особливо при використанні GPU, оскільки граф дозволяє групувати операції з тензорами однакової форми та розмірності та виконувати векторизовані обчислення великими блоками.

Через величезну кількість матричних операцій у задачах автоматичного розпізнавання мовлення найбільш оптимальним рішенням для оптимізації обробки великих обсягів даних та пришвидшення навчання моделі є перенесення обчислень з центрального процесора, який має обмежений паралелізм, на графічний (GPU) з тисячами паралельних ядер. Для цього бібліотека *TorchCodec* використовує технологію NVIDIA CUDA, де всі низькорівневі обчислення реалізовані через CUDA-ядра, оптимізовані для GPU, а тензори розміщуються на підсистемі CUDA allocator, що оптимізує використання відеопам'яті. Це дозволяє обробляти більші пачки даних паралельно, підвищувати стабільність і швидкість оптимізації моделі.

Бібліотека *Datasets* забезпечує високопродуктивний, стандартизований, зручний спосіб завантаження, попередньої обробки, зберігання та потокової передачі великих аудіо- та текстових наборів даних. Бібліотека автоматично керує метаданими та форматами аудіо, перетворює аудіофайли у тензори для роботи у PyTorch та TensorFlow, дозволяє зчитувати аудіо, нормалізувати частоту дискретизації, отримувати спектрограми. Методи попередньої обробки вхідних даних спрощують їхню підготовку для використання моделями систем розпізнавання мовлення.

Для попередньої обробки аудіоданих та побудови мел-спектрограм для нейронної моделі застосовується бібліотека *Librosa*. За допомогою її інструментів здійснюється перетворення аудіосигналу з часової області у часово-частотну та переведення ознак у мел-шкалу для побудови мел-спектрограм. Використання Librosa дозволяє забезпечити стандартизоване формування та нормалізацію вхідних ознак сигналу для обробки нейронною моделлю.

Бібліотека *Matplotlib* є зручним інструментом для візуалізації хвильової форму звуку, побудови спектрограм під час аналізу роботи акустичної частини моделі, а також подання результатів у вигляді графіків метрик та функції втрат моделі.

Для надання користувачам можливості взаємодіяти з моделлю за допомогою графічного інтерфейсу використовується фреймворк *Django*, що має повний набір інструментів для розробки вебзастосунків. Для демонстрації результатів системи

автоматичного розпізнавання мовлення Django дозволяє створити інтерфейс з функціями завантаження медіафайлів та виведення текстової транскрипції на сторінку в зручному форматі, а обчислення моделі можуть виконуватися у фонових сервісах або безпосередньо в Django-поданнях. Автоматично інтегрована з фреймворком *SQLite* виконує роль простої вбудованої бази даних, оскільки Django ORM дає змогу розробникам працювати з цією базою на рівні Python-класів та самостійно створює файл бази даних, керує створенням таблиць, виконує міграції й генерує SQL-запити у фоновому режимі. У SQLite можна зберігати історію оброблених аудіофайлів, текстові транскрипції, часові мітки, статистику моделі тощо. У поєднанні Django забезпечує реалізацію клієнтської та серверної частини застосунку, а SQLite відповідає за систему зберігання даних, необхідних для повноцінного вебзастосунку.

2.5 Висновки до розділу 2

У розділі здійснено проєктування архітектури системи автоматичного розпізнавання українського мовлення з урахуванням сучасних підходів глибинного навчання. Розглянуто принципи BERT-архітектури та її адаптацію до задач ASR, що дозволяє формувати контекстні подання аудіопослідовностей завдяки механізму двобічної самоуваги та позиційному кодуванню. Показано, що поєднання трансформерного енкодера зі згортковим блоком для вилучення акустичних ознак забезпечує баланс між локальними та глобальними характеристиками сигналу, підвищує стійкість моделі до шумів і варіацій дикції та створює умови для ефективного узгодження аудіо з текстом за допомогою CTC-алгоритму.

У межах проєктування архітектури визначено оптимальну глибину моделі, яка складається з трьох згорткових шарів та шести трансформерних блоків, що дозволяє інтегрувати фонетичні ознаки та контекстні залежності без надмірних обчислювальних витрат. Такий компроміс між точністю та продуктивністю є особливо важливим для роботи з українською мовою, де обсяг доступних даних

обмежений, а фонетичні та діалектні особливості потребують більш гнучкого моделювання.

Окремо розглянуто роль згорткових нейронних мереж у вилученні ознак зі спектрограм, їхню здатність до формування ієрархічних подань та зменшення розмірності даних, що спрощує подальшу обробку трансформерними шарами. Пояснено механізми роботи згорткових шарів, трансформерних блоків, функцій активації та алгоритмів навчання, які забезпечують поступове вдосконалення моделі. Також наведено огляд інструментів, які будуть використовуватися в роботі.

Загалом поєднання CNN-енкодера та BERT-подібного трансформера з CTC-декодуванням є ефективним рішенням для малоресурсних мов, здатним забезпечити високу точність розпізнавання у складних акустичних умовах та водночас залишатися ресурсно економним і придатним для реального застосування.

3 РЕАЛІЗАЦІЯ СИСТЕМИ АВТОМАТИЧНОГО РОЗПІЗНАВАННЯ МОВЛЕННЯ НА ОСНОВІ ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ

3.1 Попередня обробка аудіоданих

Першим етапом під час побудови системи автоматичного розпізнавання мовлення є пошук якісного датасету, який матиме достатню кількість та варіативність навчального набору аудіозаписів і не спричинить перенавчання моделі. Критерії, за якими відбувався вибір ресурсу: безкоштовність, зручний формат аудіо та простота вилучення даних, великий розмір та різноманіття вибірки. Найбільш відповідним до цих вимог є датасет на платформі HuggingFace, що має 50 Гб навчальних даних, тобто близько 400 годин сирих аудіозаписів, розподілених на три групи: train, dev та test для навчання, валідації та оцінки моделі відповідно. Оскільки проводиться обробка великого обсягу даних, було ухвалено рішення не завантажувати його повністю на диск традиційним способом, а користуватися в стримінговому форматі, тобто замість зберігання цілого датасету підвантажувати його частинами (пачками) за потреби. Такий підхід може дещо сповільнювати процес навчання, оскільки потрібен час на підвантаження даних з віддаленого джерела в порівнянні з локальним, але дані отримуються великими пачками безперервним потоком, пачки (batches) підвантажуються заздалегідь, поки модель працює з попередніми даними, отже таке сповільнення є незначним.

Перед початком роботи з аудіоданими, з них вилучаються порожні файли довжиною 0 секунд. Формується список всіх символів, які буде використовувати модель розпізнавання мовлення. В нього входять: пробіл (він має бути першим елементом для коректної роботи подальшого CTC алгоритму), літери українського алфавіту та апостроф. Оскільки нейронна мережа розуміє лише числові вектори, створюється словник, де за допомогою методу `enumerate()` кожному з елементів списку присвоюється порядковий номер для подальшого перетворення тексту на числову послідовність, а також зворотній словник для подальшого декодування результатів моделі назад в літери тексту. Також оголошується розмір пачки (batch size), що відповідає за кількість прикладів, які оброблятимуться одночасно під час

навчання. В даному випадку вибрано розмір у 8 аудіозаписів. Важливо подавати нейронній мережі в роботу дані саме пачками для забезпечення стабільного процесу навчання, адже таким чином модель не залежить від одного випадкового прикладу, що змушує значення ваг різко стрибати при оновленні, а натомість змінює ваги плавно відповідно до усереднених градієнтів по кількох зразках пачки.

Для подальшого перетворення лінійного спектра сигналу на мел-спектрограму створюється матриця ваг з необхідними параметрами: кількістю мел-фільтрів (рядків матриці), кількістю спектральних бінів після перетворення Фур'є (колонок матриці), частоту дискретизації (стандартно 16 кГц), нижню та верхню межі частот матриці (від 80 Гц до половини частоти дискретизації:

```
NUM_MEL_BINS = 64
SAMPLE_RATE = 16000
lower_hz, upper_hz = 80.0, SAMPLE_RATE / 2
NUM_SPECTRO_BINS = FFT_LENGTH // 2 + 1

MEL_WT = tf.signal.linear_to_mel_weight_matrix(
    NUM_MEL_BINS,
    NUM_SPECTRO_BINS,
    SAMPLE_RATE,
    lower_hz,
    upper_hz
)
```

Декоратор `@tf.function` інструктує TensorFlow автоматично компілювати функцію у високопродуктивний граф, що суттєво пришвидшує обробку великих обсягів аудіоданих під час навчання нейронної мережі. В функції з вилучення ознак `extract_features()` на невеликих перекривних вікнах тривалістю 640 семплів (40 мс) обчислюють швидкі та короткочасні перетворення Фур'є, в результаті чого отримують комплексні спектральні коефіцієнти, з модулів яких будується спектрограма. Для її переведення в мел-шкалу, яка краще відповідає особливостям людського сприйняття, попередньо створену матрицю перемножують на спектрограму за допомогою методу `tf.tensordot()`.

До отриманих значень мел-спектрограми застосовують натуральний логарифм для нормалізації даних та покращення їхньої якості через більшу

наближеність до слуху людини. Щоб уникнути логарифма нуля, до спектрограми додають мале число 1e-6. Функція повертає лог-мел-спектрограму:

```
@tf.function
def extract_features(waveform, sample_rate=16000):
    stfts = tf.signal.stft(waveform, frame_length=640, frame_step=320,
fft_length=FFT_LENGTH)
    spectrogram = tf.abs(stfts)

    mel_spectrogram = tf.tensordot(spectrogram, tf.cast(MEL_WT,
spectrogram.dtype), 1)
    return tf.math.log(mel_spectrogram + 1e-6)
```

На рис.3.1 за допомогою `librosa.display.specshow()` відображено мел-спектрограму.

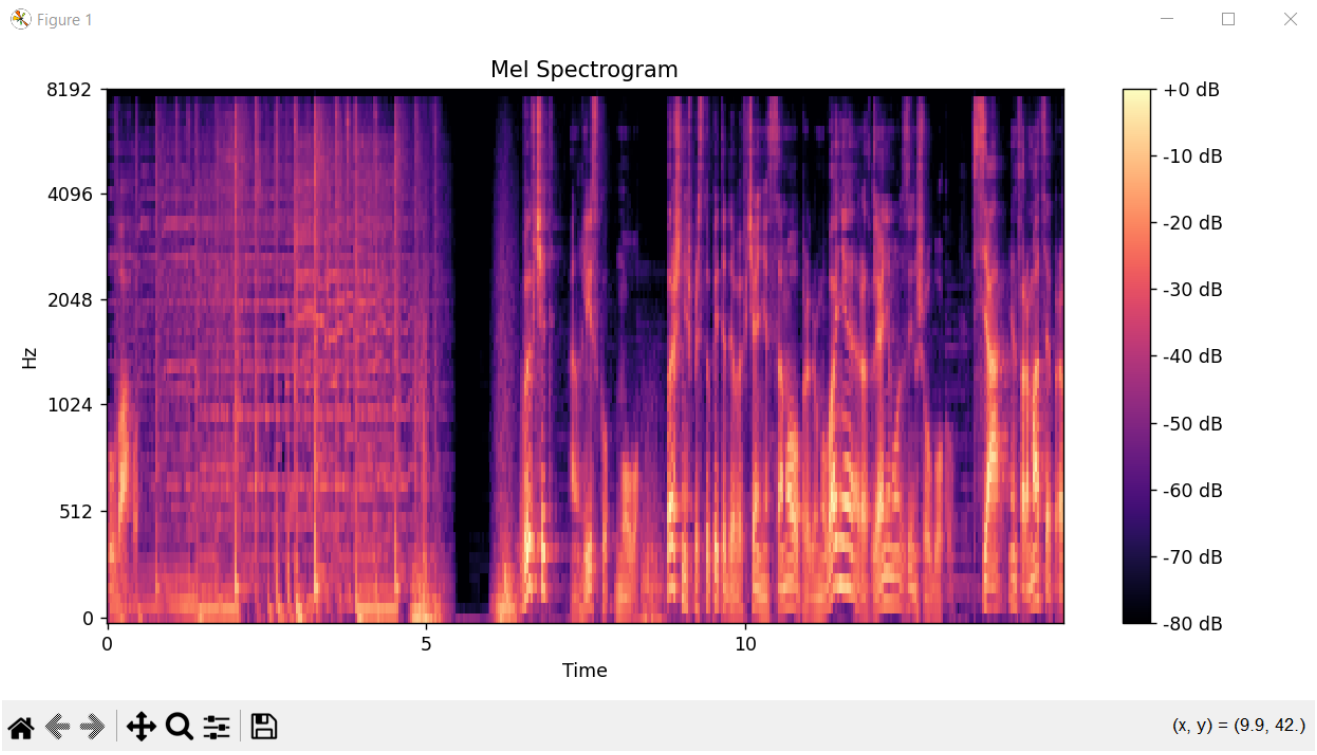

Рисунок 3.1 – Зразок мел-спектрограми

Візуальне подання аудіосигналу у вигляді мел-спектрограми показує, як змінюється енергія звуку в різних частотах упродовж часу. Горизонтальна вісь відображає тривалість аудіо, вертикальна вісь позначає частоти (Гц), перетворені у мел-шкалу для наближення до особливостей людського слуху. Колірна інтенсивність відповідає амплітуді сигналу в децибелах: яскраві жовто-помаранчеві ділянки вказують на високий рівень енергії, тоді як темні синьо-

фіолетові ділянки означають слабкі або відсутні звукові компоненти. Така спектрограма дозволяє наочно оцінити структуру мовлення, наявність пауз, шумів та фонетичних особливостей аудіо.

Функція `data_generator()` нормалізує вигляд транскрипцій для аудіо, мітки переводять до нижнього регістра, очищують від пробілів до та після тексту, перетворюють на послідовність числових індексів за допомогою функції `encode_text()`, яка замінює кожен символ на його індекс у заздалегідь сформованому словнику. Якщо результат виявляється порожнім, такий приклад пропускається. Аудіозаписи перетворюють на тензори TensorFlow, на основі яких обчислюються лог-мел-спектрограми. Через ключове слово `yield` функція послідовно генерує пари лог-мел-спектрограм з числовими послідовностями їхніх транскрипцій, що можуть бути використані в якості вхідних даних моделі розпізнавання мовлення:

```
def data_generator(raw_dataset):
    for example in raw_dataset:
        text = example ["transcription"].strip().lower()
        label = encode_text(text)
        if len(label) == 0:
            continue
        waveform = tf.convert_to_tensor(example ["audio"] ["array"],
dtype=tf.float32)
        log_mel = extract_features(waveform, sample_rate=example ["audio"]
["sampling_rate"])
        label = encode_text(example ["transcription"])
        yield log_mel, label
```

Функція `prepare_dataset()` створює об’єкт `tf.data.Dataset` та визначає структури вихідних даних лог-мел-спектрограм та міток. Така ініціалізація дозволяє потоково формувати кожен приклад під час навчання, замість збереження повного набору даних в пам’яті. Датасет групується у пачки (batches) фіксованого розміру, послідовності автоматично вирівнюються функцією `pad_batch()`: лог-мел-спектрограми доповнюються нулями, а мітки – значеннями -1, що необхідно в подальшому для коректної роботи функції втрат. Спеціальний параметр `drop_remainder` гарантує однаковість розміру кожної пачки. Оптимізований для обробки моделлю датасет повертається у вигляді об’єкта в режимі `prefetch`, що

дозволяє завантажувати наступні пачки наперед і пришвидшувати навчання системи:

```
def prepare_dataset(raw_dataset):
  ds = tf.data.Dataset.from_generator(lambda: data_generator(raw_dataset),
  output_signature=( tf.TensorSpec(shape=(None, 64), dtype=tf.float32),
                     tf.TensorSpec(shape=(None,), dtype=tf.int32)))
  def pad_batch(log_mel, label):
        return {
            "features": log_mel,
            "label": label,
            "input_length": tf.shape(log_mel) [0],
            "label_length": tf.shape(label) [0]
        }
    ds = ds.map(pad_batch)
    ds = ds.padded_batch(
        BATCH_SIZE,
        padded_shapes={
            "features": [None, 64],
            "label": [None],
            "input_length": [],
            "label_length": []
        },
        padding_values={
            "features": 0.0,
            "label": tf.cast(-1, tf.int32),  # ignored by ctc_batch_cost
            "input_length": 0,
            "label_length": 0
        },
        drop_remainder=True
    )
    return ds.prefetch(tf.data.AUTOTUNE)
```

В результаті оброблені пачки вхідних даних приведені до вигляду, що може сприймати нейронна мережа.

3.2 Реалізація ASR-моделі

Функція для створення нейронної мережі з автоматичного розпізнавання мовлення приймає на вхід параметри, що визначають конфігурацію майбутньої моделі: кількість вихідних класів (розмір словника символів, які мають бути передбачені моделлю), показники глибини та розмірності трансформерних блоків, а саме ширина прихованого простору (кількість ознак для обробки шарами), кількість голів багатоголових механізмів уваги, кількість шарів та розмір

внутрішньої Feed-Forward мережі. Таким чином користувач може масштабувати модель залежно від обчислювальних ресурсів.

Вхідний шар моделі очікує послідовність мел-спектрограм довільної довжини, де кожен часовий крок описується 64 мел-коефіцієнтами. Це стандартне подання аудіосигналу для ASR-моделей.

Згортковий енкодер моделі вилучає акустичні ознаки сигналу за допомогою шарів згортки Conv1D:

```
    x = tf.keras.layers.Conv1D(128, 10, strides=5, padding='same', 
activation='gelu')(inputs)
    x = tf.keras.layers.Conv1D(256, 3, strides=2, padding='same', 
activation='gelu')(x)
    x = tf.keras.layers.Conv1D(model_dim, 3, strides=2, padding='same', 
activation='gelu')(x)
```

Перший шар пропускає вхідні дані через 128 фільтрів з часовим вікном в 10 елементів (семплів), вказаний крок руху вікна (strides) допомагає зменшити довжину послідовності в 5 разів, що значно прискорює подальші обчислення. Функція активації GELU на цих шарах дозволяє уникнути різких стрибків в навчанні та плавно пригнічувати негативні значення сигналу. Наступний шар отримує вихідні дані з попереднього, розширює кількість фільтрів з коротшим часовим вікном для вилучення локальних ознак. Послідовність зменшується вдвічі за допомогою вказаного кроку руху, кожна точка описується 256 ознаками. Останній шар узгоджує кількість ознак з вказаним при оголошенні функції моделі, зменшує розмірність та на виході формує подання вхідної лог-мел-спектрограми у вигляді, придатному для обробки механізмом уваги.

Таким чином, згортковий енкодер зменшує розмірність, вилучає низько- та середньорівневі акустичні ознаки та оброблює сигнал для подальшої роботи трансформерного блока.

Для врахування порядку елементів у послідовності додається її позиційне кодування:

```
def add_pos_emb(tensor):
      seq_len = tf.shape(tensor) [1]
      positions = tf.range(seq_len)
```

```
        pos_emb = tf.keras.layers.Embedding(1000, model_dim)(positions)
        pos_emb = pos_emb [tf.newaxis, :, :]  # (1, seq_len, dim)
        return tensor + pos_emb [:, :seq_len, :]
    x = tf.keras.layers.Lambda(add_pos_emb)(x)
```

Позиційні ембеддинги створюють для кожного часового кроку унікальне векторне подання його позиції. Функція `add_pos_emb()` приймає дані від згорткового енкодера (тензор), визначає довжину послідовності та створює вектор позицій, що містить послідовні індекси всіх її часових кроків. Ці індекси подаються в окремий шар, де кожна позиція перетворюється на вектор фіксованої довжини, що відповідає розмірності прихованого простору трансформера. Тензори, з якими працює модель, мають параметри: розмір пачки, довжина, розмірність ознак. Для приведення позиційних векторів до такої структури додається нова вісь (newaxis). Через шар Lambda функція позиційного кодування додає позиційні вектори до тензору, отриманого від згорткового енкодера. Після цього дані передаються в трансформерний блок.

Оголошення трансформерного енкодера моделі:

```
for _ in range(num_layers):
        attn_out = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=model_dim//num_heads)(x, x)
        attn_out = tf.keras.layers.Dropout(0.1)(attn_out)
        x = tf.keras.layers.LayerNormalization(epsilon=1e-6)(x + attn_out)
```

Цикл визначає створення послідовності з шести трансформених блоків. На кожній його ітерації застосовується шар з механізмом багатоголової уваги MultiHeadAttention, який аналізує взаємозв’язки між усіма елементами послідовності та дозволяє моделі враховувати широкий контекст аудіосигналу. Параметри шару визначають кількість голів уваги та розмірність ключей у кожній з них. Вхідні дані шару перетворюються на три матриці: запитів, ключей та значень. Через добуток матриць запитів та ключей відбувається обчислення, наскільки елементи є важливими один для одного, після чого softmax фунція перетворює результати на ваги, що комбінуються з їхніми значеннями. Формується тензор, де кожен елемент містить інформацію про контекст усієї послідовності. Він проходить через Dropout-шар, що вимикає 10% нейронів для кращого абстрагування моделі та усунення ризику перенавчання.

Для того, щоб модель мала змогу зберігати попередню інформацію та доповнювати її новою, в шарі нормалізації застосовується залишковий зв’язок, де вихідні дані попереднього шару об’єднується з новими ознаками, отриманими з механізму самоуваги. Додавання цих двох тензорів дає можливість зберегти вже відому інформацію і накласти на неї корисні зміни, без руйнування базової структури сигналу. Такий підхід стабілізує навчання, запобігає зниканню градієнтів і дозволяє кожному трансформерному шару поступово вдосконалювати подання даних замість того, щоб повністю їх перезаписувати.

Оголошення двошарової Feed-Forward мережі:

```
ffn = tf.keras.Sequential( [
     tf.keras.layers.Dense(ff_dim, activation='gelu'),
     tf.keras.layers.Dense(model_dim),
     tf.keras.layers.Dropout(0.1)
])
x = tf.keras.layers.LayerNormalization(epsilon=1e-6)(x + ffn(x))
x = tf.keras.layers.Dense(num_classes + 1, name="ctc_logits")(x)
```

Мережа складається з двох повнозв’язних шарів та Drop-out шару для регуляризації. Перший шар розширює розмірність (кількість вхідних ознак) в деклька разів та застосовує функцію активації GELU для нелінійного перетворення та можливості захоплення більш складних нелінійних залежностей даних. Другий шар повертає значення розмірності назад для узгодження формату вихідних даних зі структурою попередніх шарів. Отриманий з FFN мережі результат в шарі нормалізації додається до вхідного тензора ще одним залишковим з’єднанням та нормалізується для ефективного навчання трансформера.

Фінальний повнозв’язний шар проєктує сформовані трансформерними блоками високорівневі ознаки у новий простір розміром `num_classes+1`, що відповідає кількості всіх можливих символів у словнику моделі. Додатковий клас відповідає спеціальному символу «blank» (порожній знак), необхідному для роботи алгоритму CTC. Таким чином, трансформерний енкодер на виході формує набір логітів текстової послідовності, які надалі інтерпретуватимуться як ймовірності відповідних символів на кожному часовому кроці.

Функція створення моделі повертає об’єкт `tf.keras.Model`, який визначає повний шлях перетворення від вхідної послідовності лог-мел-спектрограм до вихідних логітів для CTC-алгоритму. Модель готова до навчання на завданні розпізнавання мовлення. Щоб побачити повну структуру моделі застосовується функція `Summary()`, результат роботи якої наведено в додатку А. Структура моделі має інформацію про типи шарів, формат вихідних даних, кількість параметрів, зв’язок з іншими шарами. Модель складається зі згорткового енкодера та трансформерних блоків загальним обсягом в 48 шарів, що робить її моделлю середнього розміру.

3.3 Навчання ASR-моделі

Для навчання моделі необхідна функція втрат, яка через обчислення відмінності між вихідною та цільовою послідовностями сприяє оновленню ваг моделі. Для розрахунку CTC Loss приймаються параметри: тензор індексів символів транскрипцій, тензор вихідних логітів моделі, довжина кожної вхідної послідовності, довжина кожної мітки. Для міток створюється розріджений тензор SparseTensor, що ігнорує попередньо додані значення -1 замість порожніх. Такий формат тензору використовується для обробки даних з великою кількістю відсутніх значень та є необхідним під час обчислень втрат для даних різної довжини. Він має список індексів всіх дійсних елементів, їхні значення та розмірність повного тензора, який відновлюється з цих елементів. Оскільки алгоритм вимагає, щоб перша вісь логітів відповідала часовим крокам, вони транспонуються за допомогою методу `tf.transpose()`.

Функція обчислення втрат приймає як параметри розріджений тензор міток, транспоновані логіти, значення None для довжини міток, щоб алгоритм визначив її автоматично, фактичну довжину вхідних послідовностей та індекс порожнього символу blank. Алгоритм CTC обчислює логарифм ймовірності правильної послідовності міток шляхом врахування всіх можливих вирівнювань між логітами і цільовими символами. Цільова послідовність розширюється шляхом вставки

порожнього символу між усіма символами. Розглядаються всі допустимі послідовності передбачень моделі, які після видалення цих символів і об’єднання повторів відновлюють цільову послідовність. Ймовірність цільової послідовності є сумою добутків ймовірностей символів на відповідних часових кроках. Для зручності навчання як значення втрати береться мінус логарифм цієї ймовірності.

Результатом є тензор втрат для кожного елементу пачки, який показує, наскільки сильно передбачення моделі відрізняється від правильних міток. Ці дані використовуються моделлю для обчислення градієнтів і оновлення ваг під час навчання.

Оголошення функції CTC Loss для кожної вхідної пачки:

```
def ctc_batch_cost(y_true, y_pred, input_lengths, label_lengths):
    """
    y_true: int32 tensor of shape (batch_size, max_label_length)
    y_pred: float32 tensor of shape (batch_size, max_time, num_classes +
1) – logits
    input_lengths: int32 tensor of shape (batch_size,) – actual input
sequence lengths
    label_lengths: int32 tensor of shape (batch_size,) – actual label
lengths
    """
    indices = tf.where(tf.not_equal(y_true, -1))
    values = tf.gather_nd(y_true, indices)
    shape = tf.shape(y_true, out_type=tf.int64)
    sparse_labels = tf.SparseTensor(indices=indices, values=values,
dense_shape=tf.cast(shape, tf.int64))
    y_pred = tf.transpose(y_pred, [1, 0, 2])
    loss = tf.nn.ctc_loss(
        labels=sparse_labels,
        logits=y_pred,
        label_length=None,
        logit_length=input_lengths,
        blank_index=-1
    )
    return loss
```

Для збереження моделі визначається її клас:

```
@keras.saving.register_keras_serializable()
class CTCLossModel(tf.keras.Model):
    def __init__(self, base_model, **kwargs):
        super().__init__(**kwargs)
        self.base = base_model
        self.loss_tracker = tf.keras.metrics.Mean(name="loss")
    def get_config(self):
        config = super().get_config()
```

```
        # Serialize the inner model by config, not the object
        config.update({
            "base_model_config": self.base.get_config(),
            "base_model_class": self.base.__class__.__name__
        })
        return config
```

Створюється спеціальна модель Keras, призначена для коректного збереження, завантаження та подальшої роботи з CTC-моделями. Клас `CTCLossModel` успадковує можливості стандартної моделі `tf.keras.Model`, розширює їх для надання можливості збереження у файл та відновлення як самої моделі, так і вкладеної в неї нейронної мережі. Декоратор `@keras.saving.register_keras_serializable()` реєструє клас у внутрішній системі серіалізації. У конструкторі `__init__` зберігається посилання на нейронну модель, яка виконує основні обчислення, а також створюється метрика `loss_tracker`, що підраховує середнє значення функції втрат під час навчання. Метод `get_config` перевизначається для правильного відтворення моделі під час десеріалізації, він повертає словник конфігурації, у якому зберігаються конфігурація базової моделі, вкладеної моделі та назва її класу. Такий підхід дозволяє відтворювати архітектуру базової моделі через структурний опис замість збереження повного об’єкта, що забезпечує коректне відновлення моделі при завантаженні та робить її переносимою на інші носії або середовища.

Таким чином, цей код реалізує обгортку для нейронної моделі, яка підтримує збереження, відновлення і подальше навчання з втратою CTC Loss.

Описані методи для серіалізації класу:

```
@classmethod
    def from_config(cls, config):
        base_model_config = config.pop("base_model_config")
        base_model_class_name = config.pop("base_model_class")
        # Rebuild the base model using its class and config
        base_model_class = getattr(tf.keras.models, base_model_class_name,
None)
        if base_model_class is None:
            base_model = tf.keras.Model.from_config(base_model_config)
        else:
            base_model = base_model_class.from_config(base_model_config)
        return cls(base_model, **config)
    def get_build_config(self):
        # Keras saves this at save-time to later rebuild layers
```

```
        return {"input_shape": (None, None, 64)}
    def build_from_config(self, config):
        # Called during loading to rebuild the model's variables
        input_shape = config ["input_shape"]
        self.build(input_shape)
```

Метод `from_config()` дозволяє відновлювати модель зі словника та передавати параметри конфігурації.

Метод `get_build_config()` визначає інформацію про форму вхідних даних, яку потрібно зберігати для відтворення необхідних тензорних розмірів.

Метод `build_from_config()` використовується для ініціалізації внутрішніх змінних моделі (ваг, допоміжних структур), необхідних для її функціонування.

У сукупності ці методи забезпечують повноцінну підтримку серіалізації та десеріалізації моделі для її збереження, передачі або завантаження.

Визначення процесу навчання моделі:

```
  @property
    def metrics(self):
        return [self.loss_tracker]

    def train_step(self, data):
        features = data ["features"]
        labels = data ["label"]
        input_lengths = data ["input_length"]
        label_lengths = data ["label_length"]

        with tf.GradientTape() as tape:
            logits = self.base(features, training=True)
            input_lengths = tf.fill( [tf.shape(logits) [0]],
tf.shape(logits) [1])
            loss = ctc_batch_cost(labels, logits, input_lengths,
label_lengths)

        grads = tape.gradient(loss, self.base.trainable_variables)
        self.optimizer.apply_gradients(zip(grads,
self.base.trainable_variables))

        self.loss_tracker.update_state(loss)
        return {"loss": self.loss_tracker.result()}

    def call(self, inputs):
        return self.base(inputs)
```

Властивість metrics повертає метрику функції втрат після кожної епохи, що виводить середнє значення функції втрат CTC Loss вхідних пачек.

Метод `train_step()` визначає навчальний крок для моделі. З вхідної пачки даних зчитуються акустичні ознаки, мітки та їхні довжини послідовностей. У блоці зі стрічкою `tf.GradientTape()` забезпечується відстеження обчислень та навчання моделі шляхом поширення помилки назад по мережі (backpropagation). На основі кількості часових кроків логітів переобчислюється фактична довжина вхідної послідовності. Також обчислюється функція втрат CTC для пачки даних, де порівнюється вихід моделі з правильними мітками без потреби в попередньому вирівнюванні. Після отримання значення втрат шляхом автоматичного диференціювання підраховуються градієнти по всіх тренувальних параметрах базової моделі, відповідно до значень яких оптимізатор застосовує оновлення ваг. Оптимізатор Adam, обраний для цієї моделі, автоматично відстежує обчислені градієнти, моменти яких на початку навчання можуть бути зміщені й потребують корегування, та адаптивно змінює кроки оновлення для кожної з ваг моделі відповідно до значення градієнтів. Використання отимізатора забезпечує швидше та стабільніше навчання. Додатково оновлюється внутрішня метрика середньої втрати, яка відслідковує прогрес навчання, а метод повертає словник із поточним її значенням. Метод `call()` визначає пряме проходження моделі, отримує вхідні дані та передає їх до базової нейронної мережі в режимах навчання та інференсу.

Реалізація індивідуального циклу навчання моделі:

```
def train(model: CTCLossModel, dataset: tf.data.Dataset, epochs: int):
    optimizer = tf.keras.optimizers.Adam()
    model.compile(optimizer=optimizer)

    for epoch in range(epochs):
        print(f"\nEpoch {epoch + 1}/{epochs}")
        model.reset_metrics()

        for step, batch in enumerate(dataset):
            logs = model.train_step(batch)
            if step % 10 == 0:
                print(f"  Step {step}: Loss = {logs ['loss']:.4f}")
                if logs ['loss'] < 5:
                    dummy_input = tf.random.normal( [1, 100, 64])
                    _ = model(dummy_input)
                    model.save('./final_model.keras')
                    return
```

```
        print(f"Epoch {epoch + 1} completed. Avg Loss: {logs
['loss']:.4f}")
    dummy_input = tf.random.normal( [1, 100, 64])
    _ = model(dummy_input)
    model.save('./final_model.keras')
```

Функція приймає в якості параметрів екземпляр моделі, пачки даних зі сформованого датасету та кількість проходів по датасету в обох напрямках (епох). Для оновлення ваг в попередньо описаному методі `train_step()` створюється оптимізатор Adam, після чого модель компілюється з його використанням. Навчання проходить протягом заданої кількості епох, для кожної з яких виконується скидання накопичених метрик, щоб коректно відстежувати якість роботи моделі на кожному новому циклі. Модель послідовно обробляє пачки з датасету, повертає словник із поточним значенням функції втрат. Для зручності спостереження за процесом навчання після кожних десяти кроків виводиться значення втрат поточної пачки. Також передбачено механізм ранньої зупинки: якщо значення втрат падає нижче порогового значення, навчання достроково завершується. Перед збереженням моделі виконується одне додаткове пряме проходження на штучно створених даних для ініціалізації всіх внутрішніх структур моделі, що забезпечує її коректне збереження в файл.

В результаті описані методи реалізують повний цикл навчання моделі з функцією втрат CTC Loss, контроль прогресу навчання та відстеження метрик, оновлення ваг моделі та її збереження в стабільному вигляді.

3.4 Розгортання моделі на вебсайті Django

Інтеграція навченої моделі з вебсайтом на основі Django забезпечує можливість автоматичного розпізнавання мовлення з аудіо- та відеофайлів безпосередньо після їхнього завантаження користувачем. Після збереження моделі у форматі Keras, вона імпортується на серверну частину проєкту та завантажується один раз під час запуску застосунку, що дозволяє уникнути повторне створення моделі для кожного запиту та значно скорочує час обробки.

Транскрибування аудіоданих відбувається в файлі інференсу, з якого викликається функція `transcribe_with_midsize_wav2vec()` для передавання ознак в нейронну модель та CTC-декодер:

```
    def transcribe_with_midsize_wav2vec(audio_path: str) -> str:
        audio, sr = sf.read(audio_path)

        if len(audio.shape) > 1:
            audio = audio.mean(axis=1)
        features        =        extract_features(tf.constant(audio,
dtype=tf.float32))
        features = tf.expand_dims(features, 0)

        logits = wav2vec2_ctc_midsize(features, training=False)
        text = ctc_decode(logits)

        return text
```

У Django реалізується подання View, що приймає файли, завантажені користувачем через форму. Отримані дані тимчасово зберігаються на сервері та проходять етап попередньої обробки. Для аудіофайлів здійснюється нормалізація формату, приведення до єдиного частотного діапазону та сегментація на фрейми, з яких формується мел-спектрограма. Якщо користувач завантажує відеофайл, із нього спочатку виділяється аудіодоріжка, яка далі обробляється аналогічним способом.

Функція `extract_audio_from_video()` виконує автоматичне вилучення аудіодоріжки з відеофайлу та її збереження у форматі WAV з параметрами, сумісними з моделлю розпізнавання мовлення:

```
def extract_audio_from_video(video_path: str, out_wav_path: str) ->
None:
    os.makedirs(os.path.dirname(out_wav_path), exist_ok=True)

    cmd = [
        'ffmpeg', '-y',
        '-i', video_path,
        '-ac', '1',         # mono
        '-ar', '16000',     # 16 kHz
        '-vn',              # no video
        out_wav_path
    ]
    completed = subprocess.run(cmd, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
    if completed.returncode != 0:
```

```
        raise RuntimeError(f"ffmpeg failed:
{completed.stderr.decode('utf-8', errors='ignore')}")
```

За допомогою зовнішнього інструменту для обробки мультимедійних файлів `ffmpeg` визначаються шлях до вхідного відео та бажані властивості вихідного аудіофайлу: монофонічний канал, частота дискретизації 16 кГц та повне відокремлення аудіозапису від відео частини. Це гарантує, що отриманий аудіофайл відповідає формату, необхідному для подальшої акустичної обробки та передавання в модель розпізнавання мовлення.

Функція отримання медіафайлів від користувача та подальшої їх обробки для отримання транскрипцій мовлення:

```
def upload_view(request):
    if request.method == 'POST':
        form = UploadForm(request.POST, request.FILES)
        if form.is_valid():
            f = form.cleaned_data ['video']  #отримання аудіо або відео
            video = Video.objects.create()
            video.file.save(f.name, f)
            video.save()
            out_audio_name = f"{uuid.uuid4().hex}.wav"
            out_audio_path = os.path.join(settings.MEDIA_ROOT, 'audio',
out_audio_name)
            file_ext = os.path.splitext(f.name) [1].lower()
            try:
                if file_ext in ['.mp4', '.mov']:
                    extract_audio_from_video(video.file.path,
out_audio_path)
                elif file_ext in ['.wav', '.mp3']:
                    convert_audio_to_wav_16k(video.file.path,
out_audio_path)
                else:
                    raise ValueError("Непідтримуваний формат файлу.
Завантажте аудіо або відео.")
            except Exception as e:
                video.file.delete(save=False)
                video.delete()
                return render(request, 'transcriber/upload.html', {
                    'form': form,
                    'error': f'Помилка під час обробки файлу: {e}'
                })
            transcript = transcribe_with_midsize_wav2vec(out_audio_path)
            request.session ['transcript'] = transcript
            request.session ['video_pk'] = video.pk
            return redirect('transcriber:result', pk=video.pk)
    else:
        form = UploadForm()
    return render(request, 'transcriber/upload.html', {'form': form})
```

Після надсилання POST-запиту форма перевіряється на валідність, отриманий файл зчитується з поля форми. Програма допускає завантаження як відеофайлів, так і аудіофайлів, оскільки подальша обробка залежить лише від того, чи потрібно вилучати аудіодоріжку. Завантажений файл зберігається як об'єкт моделі, що дозволяє забезпечити єдине місце зберігання всіх медіафайлів незалежно від їхнього типу. До завантажених відеофайлів застосовується функція вилучення аудіо, до аудіофайлів – конвертація в необхідний формат, тому для обох медіа форматів результатом є стандартизований WAV-файл, сумісний з моделлю. У разі виникнення помилки під час обробки відео файл видаляється, щоб уникнути накопичення некоректних даних у сховищі. Функція `transcribe_with_midsize_wav2vec()` передає файл у модель розпізнавання мовлення, що генерує логіти, а CTC-декодер перетворює їх на фінальну текстову транскрипцію. Отриманий текст зберігається у сесії користувача разом з ідентифікатором медіафайлу, що дозволяє показати результат на окремій сторінці після перенаправлення. Здійснюється переадресація користувача до подання `result`, де відображається транскрипція завантаженого аудіо чи відео. Користувач має можливість також завантажити отриманий текст за допомогою функції `download_transcript_view()`, що приймає запит на транскрипт з сесії користувача та передає його як файл для завантаження з розширенням .txt.

Таким чином, вебсайт забезпечує прийом файлів, їх обробку, виклик моделі та повернення результату користувачу з можливістю завантаження. Результати роботи зображені на рис. 3.2 – 3.5.

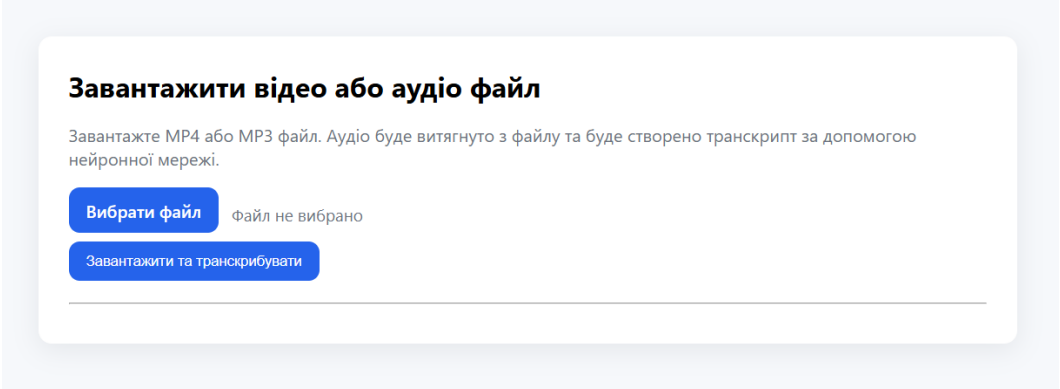

Рисунок 3.2 – Сторінка завантаження медіафайлів.

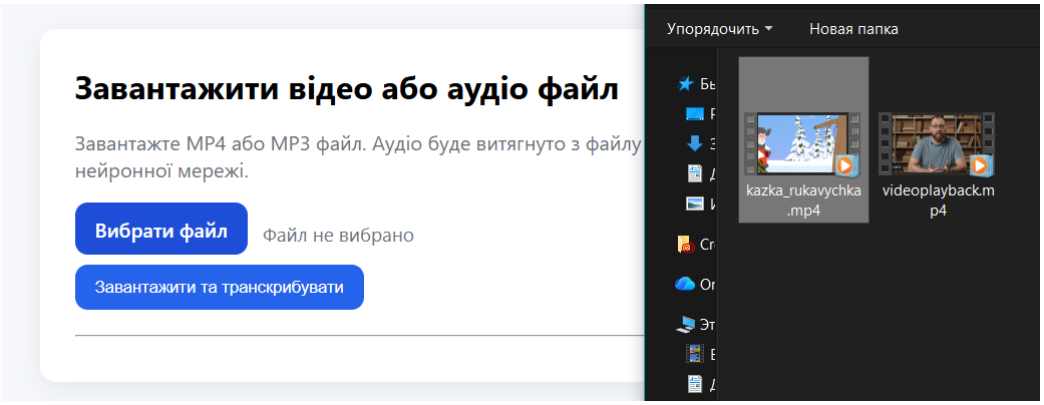

Рисунок 3.3 – Вибір файлу для завантаження

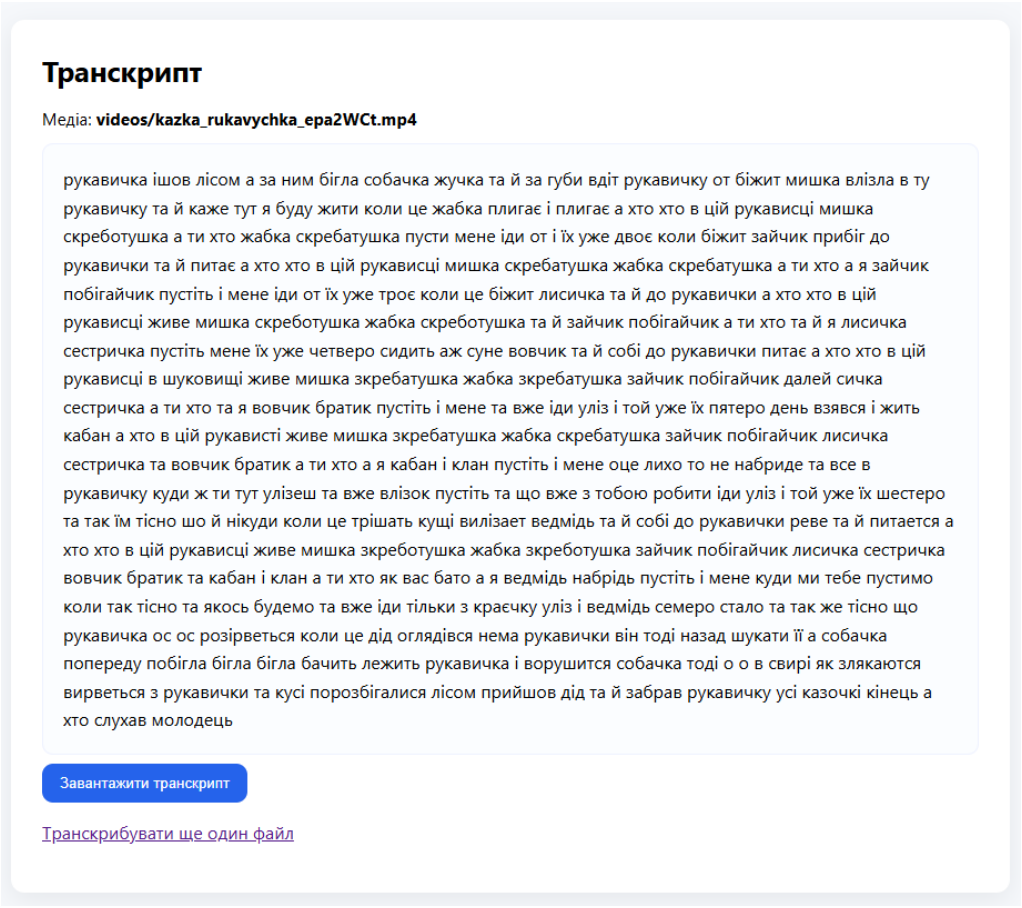

Рисунок 3.4 – Отримання текстової транскрипції відео файлу

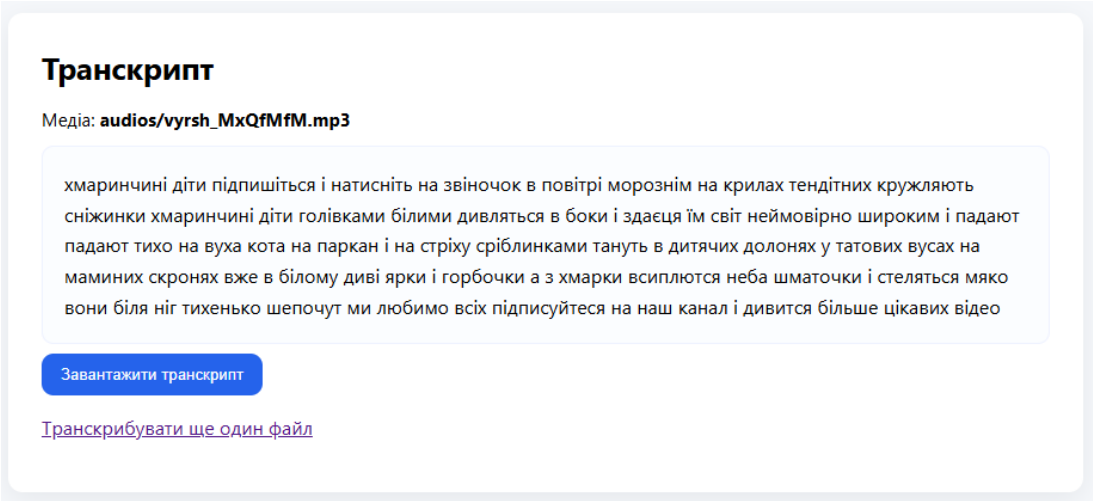

Рисунок 3.5 – Отримання текстової транскрипції аудіофайлу

Наведені результати доводять працездатність моделі, що надає можливість завантаження відео- та аудіофайлів, перегляду та завантаження їхніх транскрипцій. Результати роботи програми можуть застосовуватися для накладення субтитрів на відео чи отримання тексту аудіофайлів. Можна побачити, що до типових помилок транскрипцій моделі належать пропущення апострофів, подвоєнь літер та неузгодження закінчень слів, тому створення граматично впорядкованих текстів, придатних для використання та публікації, а також розставлення пунктуації можливе за подальшої інтеграції з мовною моделлю.

3.5 Метрики оцінки якості

Під час інференсу CTC-декодер виконує усунення повторюваних символів і спеціального «blank»-токена та формує найбільш правдоподібну послідовність вихідного тексту. Для коректного вимірювання якості такого перетворення використовують метрики WER, CER, SER та RTF, оскільки вони всебічно відображають різні аспекти продуктивності ASR. WER є базовим показником точності ASR-моделей на рівні слів і демонструє, наскільки змінився зміст фрази після розпізнавання. CER додатково дозволяє оцінити похибки моделі на рівні символів, що особливо важливо для української мови з її морфологічною варіативністю та великою кількістю форм слів. SER підкреслює, скільки речень

містять помилки, що дає якісну оцінку розуміння висловлювань. RTF використовується як показник реальної швидкодії моделі, адже він порівнює час інференсу з тривалістю аудіо та показує, чи придатна система для роботи в режимі реального часу. Разом ці метрики забезпечують комплексну оцінку точності, стабільності та практичної застосовності ASR-моделі.

Метрика WER вимірює частоту похибки в словах за формулою [37]:

$$WER = \frac{S+D+I}{N} \tag{3.1}$$

де S – кількість замін (substitutions) слів (слово передбачене неправильно);

D – кількість видалених (deletions) слів (слово пропущено);

I – кількість вставок (insertions) слів (додаткове слово передбачене);

N – загальна кількість слів у еталонному (правильному) тексті.

Метрика CER вимірює частоту похибки на рівні символів за формулою:

$$CER = \frac{S_{char}+D_{char}+I_{char}}{N_{char}} \tag{3.2}$$

де S, D, I – кількість замін, видалень і вставок символів;

N – загальна кількість символів у еталонному тексті.

Метрика SER вимірює, у скількох реченнях є хоча б одна помилка за формулою:

$$SER = \frac{N_{err}}{N_{total}} \tag{3.3}$$

де N_{err} – кількість речень, в яких є хоча б одна помилка;

N_{total} – загальна кількість речень тексту.

Для даної моделі були розраховані метрики:

```
def compute_cer(refs, hyps):
    total_dist = 0
    total_chars = 0
    for r, h in zip(refs, hyps):
        total_dist += levenshtein(r, h)
        total_chars += len(r)
    return total_dist / total_chars if total_chars > 0 else 0
```

```
def compute_wer(refs, hyps):
    total_dist = 0
    total_words = 0
    for r, h in zip(refs, hyps):
        r_words = r.split()
        h_words = h.split()
        total_dist += levenshtein(r_words, h_words)
        total_words += len(r_words)
    return total_dist / total_words if total_words > 0 else 0

def compute_ser(refs, hyps):
    sentence_errors = 0
    total_sentences = len(refs)
    for r, h in zip(refs, hyps):
        if r.strip() == "":
            continue
        if levenshtein(r.split(), h.split()) != 0:
            sentence_errors += 1
    return sentence_errors / total_sentences if total_sentences > 0 else 0
```

Таблиця 3.1 – Метрики моделі wav2vec2-ctc-midsize

Метрика	Значення
CER	4.1%
WER	12.3%
SER	24%

Отримані результати метрик демонструють збалансований рівень точності моделі з архітектурою середнього розміру. Для тестового набору даних модель досягла значення WER = 12.3%, що є реалістичним і конкурентним показником для української мови, особливо з огляду на її фонетичну та морфологічну складність. Значення CER = 4.1%, свідчить про те, що більшість помилок пов’язані з незначними відхиленнями у вимові окремих символів, а не з повним неправильним розпізнаванням слова, отже модель достатньо добре захоплює фонетичну структуру мови. Метрика SER = 24% відображає частку речень, в яких присутня принаймні одна помилка. Отже, попри відносно невелику кількість символьних та словникових похибок, значна частина речень містить хоча б одну неточність, що є типовою ситуацією для моделей без додаткової мовної корекції. У сукупності ці результати свідчать про те, що модель забезпечує достатньо високу точність на рівні символів і слів, але все ще допускає помилки на рівні повних речень. Отримані

значення метрик є цілком очікуваними для середньої за розміром CTC-моделі розпізнавання українського мовлення й демонструють, що система вже придатна для практичного використання, а подальше зниження SER і WER можливе за рахунок донавчання на більших обсягах даних або інтеграції мовної моделі.

Порівняння ключового показника WER з показниками сучасних ASR-моделей у відкритому доступі приведено в табл. 3.2.

Отримані результати показують, що промислові та комерційні моделі, такі як FastConformer Hybrid Large, Citrinet-1024 чи ElevenLabs Scribe, демонструють найнижчі показники WER завдяки використанню складних багаторівневих архітектур, масштабних мовних моделей та великих обсягів попереднього навчання. Проте ці системи зазвичай є важкими, ресурсоємними та недоступними для повного відтворення, що ускладнює їх адаптацію та розгортання на обмежених обчислювальних потужностях.

Таблиця 3.2 – Порівняння WER моделей.

Модель	Показник WER	Архітектура
wav2vec2-ctc-midsize (створена)	12.3	Спрощена BERT-подібна архітектура + CTC
uk-pods Conformer	6.75	Conformer-CTC
Yehor/w2v-bert-uk	17.34	Wav2Vec2 + BERT-Transformer
NVIDIA Citrinet-1024	4.32	Citrinet (QuartzNet v2)
HuBERT-uk	37.07	HuBERT + CTC
FastConformer Hybrid Large (UA)	4.0–4.5	FastConformer (CTC + Transducer)
arampacha/wav2vec2-xls-r-1b	16.52	XLS-R 1B
Whisper Large v2	13.72	Whisper
XLS-R 1B + LM (Barkovska 2022)	14.62	XLS-R + KenLM
data2vec-large-uk	31.17	data2vec
Squeezeformer-ML	5.91	Squeezeformer-CTC
ElevenLabs Scribe	5.5	Архітектура не розкривається компанією

На цьому фоні створена wav2vec2-ctc-midsize модель з WER 12.3% посідає оптимальну середню нішу: вона суттєво швидша під час інференсу, має меншу архітектурну складність і зберігає відкритість реалізації, дозволяючи гнучко модифікувати шари, параметри та стратегії декодування. Попри компактність, її точність перевищує низку великих відкритих моделей (Yehor/w2v-bert-uk, XLS-R 1B, data2vec-uk), що підкреслює вдалі архітектурні рішення на основі поєднання CNN-енкодера та BERT-подібних трансформерних блоків. Отже, у порівнянні з

важкими промисловими системами модель демонструє раціональний баланс між швидкістю, точністю та ресурсною економністю, що робить її доречною для практичного застосування та обґрунтовано ефективною.

Показник RTF (Real-Time Factor) був виміряний як відношення часу повної обробки аудіо (від моменту завантаження файлу до отримання текстового результату) до тривалості самого аудіозапису. Такий підхід відображає реальну швидкодію сервісу в умовах кінцевого користувача. Розрахунок відбувається за формулою[38]:

$$RTF = \frac{T_{\text{обробки}}}{T_{\text{аудіо}}} \tag{3.4}$$

де $T_{\text{обробки}}$ – час обробки аудіо;

$T_{\text{аудіо}}$ – тривалість аудіо.

У роботі здійснено розрахунок показника RTF для створеного вебсайту та порівняння з 5 конкурентами – вебсервісами, які безкоштовно надають повний чи обмежений функціонал з транскрибування відео- та аудіофайлів українською мовою. Результати розрахунку на прикладі відеофайлу довжиною в 9 хв 34 с. подані в табл. 3.3.

Таблиця 3.3 – Порівняння з іншими сервісами транскрибування медіа

Вебсайт	Тривалість часу обробки	Показник RTF
Створений вебсайт	1 хв 19 с (79 с)	0.14
elevenLabs	40.3 с	0.07
Notta ai	4 хв 50с (290 с)	0.50
Turboscribe	1 хв 38с (98 с)	0.17
Any2text	1 хв 19с (79 с)	0.14
speechmatics	1 хв 44с (104 с)	0.18

На основі розрахованих значень часу обробки та RTF видно, що створена модель демонструє високу швидкість серед порівнюваних сервісів. Показник RTF ≈ 0.14 означає, що система витрачає лише 14% реального часу аудіо на обробку, що є конкуретноспроможним показником продуктивності для українських ASR-моделей з відкритою архітектурою. Більшість сервісів мають вищі значення RTF

(0.17–0.50), що вказує на повільнішу роботу та більшу тривалість обробки. Тим самим модель демонструє конкурентну, стабільно високу швидкість, показує кращі результати за більшість сервісів, поступається лише комерційній індустріальній платформі ElevenLabs, яка використовує закриту оптимізовану інфраструктуру. Це підкреслює ефективність архітектури та доцільність її використання.

Також важливою метрикою для систем автоматичного розпізнавання мовлення є показник втрат, що вимірюється під час навчання. Реалізовано візуалізацію графіків спадання втрат за отриманими значеннями:

```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))
    # --- Left plot ---
    axes[0].plot(steps, loss_values, marker='o', linestyle='-',
color='b', linewidth=2)
    axes[0].set_title('Training Loss first 100 steps', fontsize=14)
    axes[0].set_xlabel('Step', fontsize=12)
    axes[0].set_ylabel('Loss', fontsize=12)
    axes[0].grid(True, linestyle='--', alpha=0.6)
    for i, loss in enumerate(loss_values_end):
        axes[0].text(steps_end[i], loss + 0.02, f"{loss:.2f}",
ha='center', fontsize=12)
    # --- Right plot ---
    axes[1].plot(steps_end, loss_values_end, marker='o', linestyle='-',
color='b', linewidth=2)
    axes[1].set_title('Training Loss last 100 steps', fontsize=14)
    axes[1].set_xlabel('Step', fontsize=12)
    axes[1].set_ylabel('Loss', fontsize=12)
    axes[1].grid(True, linestyle='--', alpha=0.6)
    fig.tight_layout()
    plt.show()
```

Прогрес навчання моделі можна спостерігати на графіках спадання втрат за перші 100 кроків (рис. 3.6) та останні 100 кроків (рис. 3.7). Ці два графіки відображають динаміку змін функції втрат моделі автоматичного розпізнавання мовлення з CTC-алгоритмом впродовж усього циклу навчання від перших кроків оптимізації до фінальної стадії збіжності. Перший графік демонструє поведінку втрат на початкових 100 ітераціях і характеризується різким стрімким спадом значення втрат майже в два рази. Така форма кривої є типовою для нейронних мереж, які на старті мають випадково ініціалізовані ваги та формують хаотичні передбачення, що призводить до великих початкових втрат.

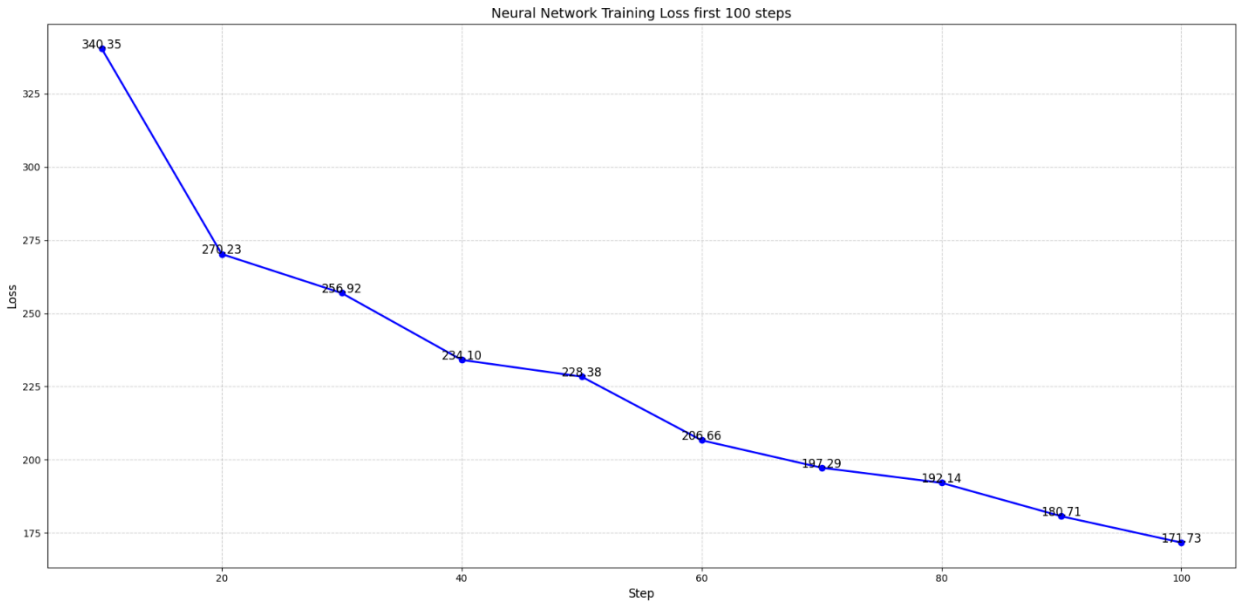

Рисунок 3.6 – Графік спадання втрат за перші 100 кроків

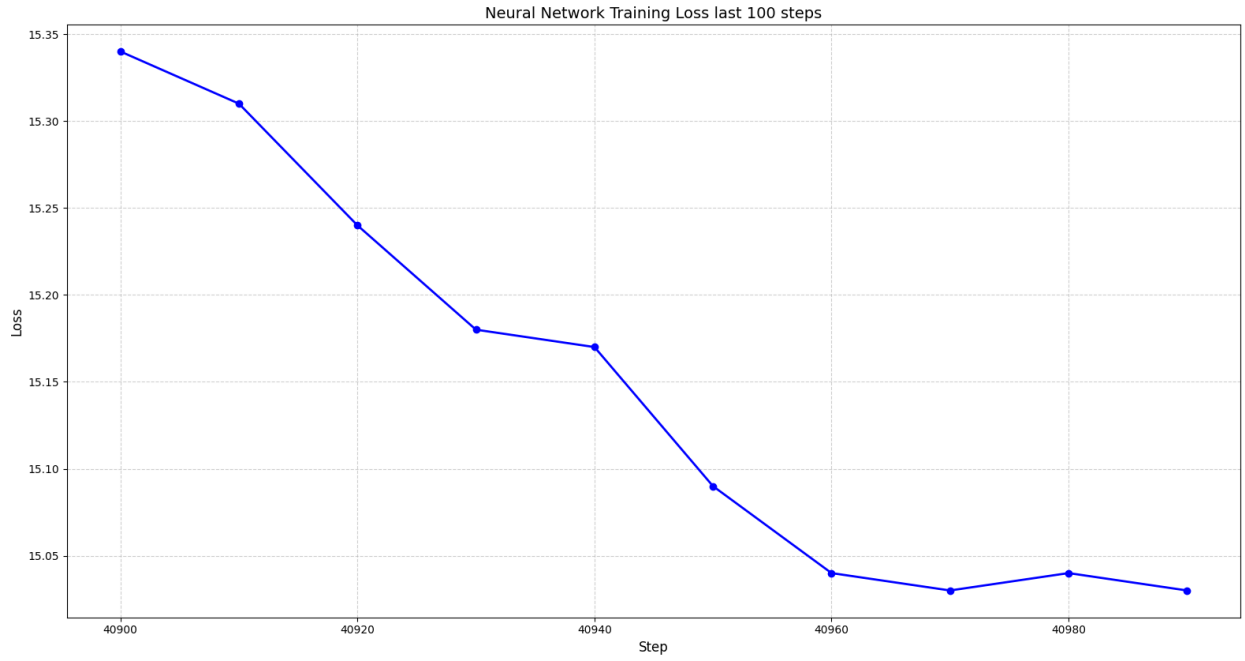

Рисунок 3.7 – Графік спадання втрат за останні 100 кроків

В цьому випадку оптимізатор отримує сильні градієнти та швидко зсуває ваги моделі у напрямку правильних співвідношень між акустичними патернами та текстовими послідовностями. Із застосуванням функції втрати CTC-loss модель починає навчатись базового вирівнювання аудіо і транскрипції, формувати структуру тимчасових відповідностей, виділяти паузи та загальні контури фонем,

що забезпечує швидке зменшення помилки. Різке спадання втрат у перші десятки кроків свідчить про те, що модель швидко переходить від повної невизначеності до стану, коли вона вже здатна розрізняти базові структурні елементи сигналу.

На другому графіку наведено поведінку втрат на останніх ста ітераціях навчання. Крива втрат втрачає стрімкість і демонструє повільне та дуже плавне зниження значення втрат. Це показує, що модель перебуває на етапі, коли більшість глобальних закономірностей вже засвоєно, а подальше вдосконалення можливе лише через мінімальні, локальні зміни параметрів мережі. Градієнти стають дуже малими, кроки оптимізації короткі, а зменшення втрат уповільнюється. Мережа уточнює межі фонем, краще опрацьовує складні слова та зменшує кількість локальних помилок. Відсутність помітних стрибків та мінімальність коливань втрат свідчать про відсутність перенавчання або нестабільності, коректну роботу оптимізатора та добре налаштований процес навчання.

Разом два графіки ілюструють завершений цикл навчання моделі ASR: від швидкого переходу з хаотичних прогнозів до структурованого подання даних, поступове вдосконалення й стабілізацію на останніх кроках навчання. Така форма кривої є ознакою якісного тренувального процесу для моделей із CTC-loss, де спочатку модель вчиться вирівнювати сигнал з транскрипцією, а потім оптимізує відповідності між окремими часовими фреймами та символами. Стабільна поведінка кривої наприкінці свідчить про досягнення моделлю приблизно оптимальної точки, де подальше істотне зниження втрат вимагає або додаткових даних, або зміни архітектури чи гіперпараметрів. Значення ≈15% – це допустимий рівень втрат для середньої моделі, вона є придатною для практичного використання в задачах автоматичного транскрибування мовлення. Для зменшення цього показника в подальшому можна провести аугментацію даних, збільшення навчального датасету та тонке налаштування гіперпараметрів.

3.6 Висновки до розділу 3

У розділі реалізовано практичну частину системи автоматичного розпізнавання українського мовлення на основі згорткових нейронних мереж у поєднанні з трансформерними блоками та алгоритмом CTC. Детально описано процес попередньої обробки аудіоданих, включно з формуванням словника символів, створенням лог-мел-спектрограм та організацією потокового завантаження даних у форматі batches, що забезпечує стабільність і ефективність навчання моделі. Продемонстровано побудову архітектури ASR-моделі, яка складається зі згорткового енкодера для вилучення акустичних ознак, позиційного кодування для відображення часової структури сигналу, трансформерних блоків із багатоголовою самоувагою та feed-forward шарами, а також вихідного шару з логітами для подальшого CTC-декодування.

Наведено список шарів моделі з їхніми параметрами. Для зручної взаємодії користувачів з моделлю розроблено вебсайт в якості графічного інтерфейсу, де користувач може завантажувати медіафайли та здійснювати їхнє транскрибування. Під час інференсу обчислено метрики оцінки якості моделі: коефіцієнт помилок слів (WER), символів (CER) та речень (SER). Метрикою з найнижчим показником є CER, а з найвищим – SER, адже слова та речення мають більше інформації, що потрібно передбачити, ніж окремі літери тексту, отже зростає складність задачі та ймовірність помилки. Показник швидкості роботи моделі RTF є меншим за 1, що показує здатність нейронної мережі розпізнавати аудіозапис швидше, ніж він відтворюється та вказує на підтримку роботи моделі в режимі реального часу. Отримані результати метрик показують, що модель поступається за точністю промисловим та комерційним моделям зі складною архітектурою, проте є швидшою та витрачає менше реального часу аудіо на процес обробки. Таким чином, вона забезпечує збалансоване поєднання точності, швидкодії та ресурсної ефективності. Також під час навчання отримано графіки спадання втрат за перші та останні 100 кроків. Вони показують стрімке спадання втрат на початку навчання та мінімальне наприкінці, що показує на оптимально налаштований процес навчання та готовність моделі до використання.

Окремо розглянуто механізм навчання моделі із застосуванням функції втрат CTC Loss, що дозволяє узгоджувати аудіопослідовності довільної довжини з текстовими транскрипціями без необхідності попереднього вирівнювання. Такий підхід забезпечує гнучкість і точність моделі навіть за умов варіативності дикції та наявності шумів. Практична реалізація підтвердила можливість ефективної інтеграції CNN-шарів для вилучення локальних ознак та трансформерних енкодерів для моделювання глобального контексту, що створює оптимальний баланс між точністю, стійкістю та продуктивністю системи.

Реалізована модель є ресурсно економною, здатною працювати з обмеженими наборами даних української мови та забезпечує перспективи для розширення системи у напрямку підвищення точності розпізнавання, адаптації до діалектів та оптимізації для реального застосування.

ВИСНОВКИ

У кваліфікаційній роботі виконано комплексне дослідження, проєктування та реалізацію системи автоматичного розпізнавання українського мовлення, що поєднує теоретичні засади, архітектурні рішення та практичну реалізацію. Робота має актуальність у контексті розвитку технологій обробки природної мови, адже українська мова належить до малоресурсних, і створення ефективних систем розпізнавання мовлення для неї є важливим завданням як для науки, так і для практики.

У першому розділі здійснено системний огляд теоретичних основ автоматичного розпізнавання мовлення, визначено його ключові особливості та виклики. Показано, що на відміну від класичних задач класифікації, розпізнавання мовлення потребує спеціальних алгоритмів для узгодження послідовностей змінної довжини, врахування контексту та роботи з великими обсягами даних. Окремо акцентовано на проблемах української мови як малоресурсної: обмеженість корпусів, різноманітність діалектів, відсутність стандартизованих тестових наборів. Це підкреслює актуальність роботи, адже створення якісних моделей для української мови є складним і водночас необхідним завданням.

У другому розділі спроєктовано архітектуру моделі, яка поєднує згортковий енкодер для вилучення локальних акустичних ознак, трансформерні блоки для моделювання глобального контексту та CTC-алгоритм для узгодження аудіо з текстом. Така структура є оптимальною для умов обмежених ресурсів, оскільки дозволяє уникнути надмірної складності промислових моделей і водночас забезпечує високу точність. Новизна полягає у спрощенні BERT-подібної архітектури шляхом використання навчання з учителем без попереднього самонавчання та маскування, що робить модель більш доступною для практичної реалізації. SWOT-аналіз підтвердив сильні сторони такого підходу: стійкість до шумів, масштабованість, можливість адаптації до малоресурсних мов.

У третьому розділі здійснено програмну реалізацію системи: описано етапи попередньої обробки аудіо, побудову моделі та процес її навчання. Реалізована

модель складається з 48 шарів і належить до середнього класу за розміром, що забезпечує оптимальний компроміс між точністю та швидкодією. Показник WER у 12.3% підтверджує практичну придатність моделі, а здатність працювати швидше за реальний час відтворення аудіо відкриває перспективи використання системи у режимі онлайн-транскрибування. Стабільність навчання та відсутність ознак перенавчання свідчать про коректність вибраної архітектури та налаштувань. Важливою особливістю є реалізація графічного інтерфейсу, що забезпечує зручність взаємодії користувачів із системою та доводить її прикладну цінність.

Тим самим робота має як теоретичну, так і практичну значущість. Теоретично вона узагальнює сучасні підходи до автоматичного розпізнавання мовлення та адаптує їх до української мови. Практично – створено систему, яка поєднує простоту реалізації з достатнім рівнем точності та високою швидкодією. Оптимальність розробленої архітектури полягає у здатності працювати з обмеженими ресурсами, забезпечуючи при цьому якісні результати. Новизна роботи полягає у застосуванні CNN+Transformer+CTC для української мови, що раніше не мало широкої реалізації. Актуальність підтверджується потребою у створенні систем розпізнавання мовлення для малоресурсних мов, які можуть використовуватися у сферах освіти, бізнесу, медицини та державного управління.

Подальші дослідження можуть бути спрямовані на інтеграцію мовних моделей для зниження WER, розширення навчальних корпусів із зашумленими прикладами, адаптацію до діалектів та використання мультимодальних підходів. Це дозволить підвищити точність і універсальність системи, зробити її конкурентною з промисловими рішеннями та забезпечити ще ширше застосування у реальних умовах.

Отже, поставлені мета і завдання роботи виконані повністю: створено систему автоматичного розпізнавання українського мовлення, яка демонструє оптимальний баланс між точністю, швидкодією та простотою реалізації, а також відкриває перспективи для подальшого розвитку технологій у сфері обробки природної мови.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Урсатьєв О.А., Волков О.Є., Ткаля В.Г. Автоматизоване машинне навчання. Стан та перспективи розвитку. *Інформаційні технології та системи*. 2025. № 2 (2). С. 3–33. DOI: https://doi.org/10.15407/intechsys.2025.02.003

2. Mehrish A., Majumder N., Bharadwaj R., Mihalcea R., Poria S. A review of deep learning techniques for speech processing. *Information Fusion*. 2023. Vol. 99. DOI: https://doi.org/10.1016/j.inffus.2023.101869

3. Belcic I., Stryker C. What is supervised learning? URL: https://www.ibm.com/think/topics/supervised-learning

4. Training vs Testing vs Validation Sets URL: https://www.geeksforgeeks.org/machine-learning/training-vs-testing-vs-validation-sets

5. Doshi K. Audio Deep Learning Made Simple: Sound Classification, step-by-step. URL: https://towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-cebc936bbe5

6. Leland R. Understanding the Mel Spectrogram. URL: https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53

7. Doshi K. Foundations of NLP Explained Visually: Beam Search, How it Works. 01.04.2021. URL: https://towardsdatascience.com/foundations-of-nlp-explained-visually-beam-search-how-it-works-1586b9849a24

8. Doshi K. Audio Deep Learning Made Simple: Automatic Speech Recognition (ASR), How it Works. URL: https://medium.com/data-science/audio-deep-learning-made-simple-automatic-speech-recognition-asr-how-it-works-716cfce4c706

9. Automatic Speech Recognition using CTC. URL: https://www.geeksforgeeks.org/nlp/automatic-speech-recognition-using-ctc

10. Wang C., Wu Y., Liu S., Li J., Qian Y., Kumatani K., Wei F. UniSpeech at scale: An Empirical Study of Pre-training Method on Large-Scale Speech Recognition Dataset. *arXiv*. 2021. DOI: https://doi.org/10.48550/arXiv.2107.05233

11. Silovsky J., Deng L., Argueta A., Arvizo T., Hsiao R., Kuznietsov S., Lin Y., Xiao X., Zhang Y. Cross-lingual Knowledge Transfer and Iterative Pseudo-labeling for

Low-Resource Speech Recognition with Transducers. *arXiv*. 2023. DOI: https://doi.org/10.48550/arXiv.2305.13652

12. Sirora L. W., Mutandavari M. A Deep Learning Automatic Speech Recognition Model for Shona Language. *International Journal of Innovative Research in Computer and Communication Engineering*. 2024. Vol. 12, №. 09. С. 1–14. DOI: https://doi.org/10.15680/ijircce.2024.1209019

13. Purwins H., Li B., Virtanen T., Schlüter J., Chang S.-Y., Sainath T. Deep Learning for Audio Signal Processing. *IEEE Journal on Selected Topics in Signal Processing*. 2019. Vol. 13, №2. С. 206–219. DOI: https://doi.org/10.1109/JSTSP.2019.2908700

14. Rascon C. Characterization of Deep Learning-Based Speech-Enhancement Techniques in Online Audio Processing Applications. *Sensors.* 2023. Vol. 23, №9. С. 4394. DOI: https://doi.org/10.3390/s23094394

15. He Y., Seng K. P., Ang L. M. Multimodal Sensor-Input Architecture with Deep Learning for Audio-Visual Speech Recognition in Wild. *Sensors.* 2023. Vol. 23, №. 4. С.1834. DOI: https://doi.org/10.3390/s23041834

16. Abbas M A., Haider A., Nawroz H. Using RNN Language Model Effect on the Development of Speech Recognition System: A Review. *YMER.* 2023. Vol. 22, №5. С.1486-1504. URL: https://ymerdigital.com/uploads/YMER2205CM.pdf

17. Syed A. A. Q., Taiba M. W. Review of Deep Learning Architectures for Speech and Audio Processing. *International Journal of Scientific and Engineering Research.* 2020. Vol. 11, №10 С.63. URL: https://ijisrt.com/assets/upload/submitted_files/1577686157.pdf

18. Kumar N. S., Kumar N. A., Mishra S., Mohanty P., Tripathy N., Surjeet C. K. Exploring Speech Emotion Recognition in Tribal Language with Deep Learning Techniques. *International journal of electrical and computer engineering systems.* 2025. Vol. 16, №. 1. С. 53–64. DOI: http://dx.doi.org/10.32985/ijeces.16.1.6

19. Nassif A. B., Shahin I., Attili I., Azzeh M., Shaalan K., Speech Recognition Using Deep Neural Networks: A Systematic Review. *IEEE Access*. 2019. Vol. 7. С. 19143-19165. DOI: https://doi.org/10.1109/ACCESS.2019.2896880

20. Liao Z. Comparative analysis between application of transformer and recurrent neural network in speech recognition. *Applied and Computational Engineering.* 2023. Vol. 6, № 1. С. 629–634. DOI: https://doi.org/10.54254/2755-2721/6/20230879

21. Li L. Application of Deep Learning Technology in Speech Recognition and Language Teaching. *Lecture Notes in Education Psychology and Public Media*. 2024. Vol. 59, № 1. DOI: https://doi.org/10.54254/2753-7048/59/20241721

22. Wang A. Speech recognition for different dialects and accents. *ITM Web of Conferences*. 2025. Vol. 73, № 02011. DOI: https://doi.org/10.1051/itmconf/20257302011

23. Sus Ł. Wav2Vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. 24.06.2021. URL: https://towardsdatascience.com/wav2vec-2-0-a-framework-for-self-supervised-learning-of-speech-representations-7d3728688cae/

24. HuBERT Model. URL: https://www.geeksforgeeks.org/nlp/hubert-model

25. Chaudhuri S., Dr. Athilakshami R. Comparison of Wav2vec2 2.0 and Hubert Models based on word error rate. *Tijer – International Research Journal*. 2023. Vol.10, №12, С. 977-979. URL: https://tijer.org/tijer/papers/TIJER2312131.pdf

26. Radford A., Kim J. W., Xu T., Brockman G., McLeavey C., Sutskever I. Robust Speech Recognition via Large-Scale Weak Supervision. *arXiv.* 2022. DOI: https://doi.org/10.48550/arXiv.2212.04356

27. Devlin J., Chang M-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv*. 2018. DOI: https://doi.org/10.48550/arXiv.1810.0480528.

28. Ткаліченко С.В. Штучні нейронні мережі: навч. посібник. Кривий Ріг, 2023. 150 с. URL: https://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058677.pdf

29. Нейронні мережі: навчання з використанням алгоритму зворотного поширення помилки. URL: https://developers.google.com/machine-learning/crash-course/neural-networks/backpropagation

30. Lim A. Convolutional Neural Network CNN: How Do Shared Weights Impact CNNs and RNNs in Deep Learning? URL: https://pupuweb.com/convolutional-neural-network-cnn-how-do-shared-weights-impact-cnns-and-rnns-in-deep-learning

31. Hendrycks D., Gimpel K. Gaussian Error Linear Units (GELUs). *arXiv.* 2018. DOI: https://doi.org/10.48550/arXiv.2212.04356

32. Holbrook R. Dropout and Batch Normalization. URL: https://www.kaggle.com/code/ryanholbrook/dropout-and-batch-normalization

33. Lam E. Understanding CTC loss for speech recognition. URL: https://voidful.medium.com/understanding-ctc-loss-for-speech-recognition-a16a3ef4da92

34. Ankit U. Transformer Neural Networks: A Step-by-Step Breakdown. URL: https://builtin.com/artificial-intelligence/transformer-neural-network

35. Функції активації: Ступінчаста, лінійна, сигмоїда, ReLU та Tanh. URL: https://robotdreams.cc/uk/blog/327-funkciji-aktivaciji-stupinchasta-liniyna-sigmojida-relu-ta-tanh

36. Introduction to graphs and tf.function. *TensorFlow.* URL: https://www.tensorflow.org/guide/intro_to_graphs

37. Metrics for ASR Performance: WER and CER. ULR: https://apxml.com/courses/applied-speech-recognition/chapter-6-evaluating-deploying-asr-systems/asr-performance-metrics-wer-cer

38. Real-Time Transcription Error Rates. URL: https://www.emergentmind.com/topics/real-time-transcription-error-rates

ДОДАТКИ

Додаток А. Структура моделі wav2vec_ctc_midzise

Model: "wav2vec2_ctc_midsize"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, None, 64)	0	-
conv1d (Conv1D)	(None, None, 128)	82,048	input_layer[0][0]
conv1d_1 (Conv1D)	(None, None, 256)	98,560	conv1d[0][0]
conv1d_2 (Conv1D)	(None, None, 256)	196,864	conv1d_1[0][0]
lambda (Lambda)	(None, None, 256)	0	conv1d_2[0][0]
multi_head_attention (MultiHeadAttention)	(None, None, 256)	263,168	lambda[0][0], lambda[0][0]
dropout_1 (Dropout)	(None, None, 256)	0	multi_head_attention[0][0]
add (Add)	(None, None, 256)	0	lambda[0][0], dropout_1[0][0]
layer_normalization (LayerNormalization)	(None, None, 256)	512	add[0][0]
sequential (Sequential)	(None, None, 256)	525,568	layer_normalization[0][0]
add_1 (Add)	(None, None, 256)	0	layer_normalization[0][0], sequential[0][0]
layer_normalization_1 (LayerNormalization)	(None, None, 256)	512	add_1[0][0]
multi_head_attention_1 (MultiHeadAttention)	(None, None, 256)	263,168	layer_normalization_1[0][… layer_normalization_1[0][…
dropout_4 (Dropout)	(None, None, 256)	0	multi_head_attention_1[0]…
add_2 (Add)	(None, None, 256)	0	layer_normalization_1[0][… dropout_4[0][0]
layer_normalization_2 (LayerNormalization)	(None, None, 256)	512	add_2[0][0]
sequential_1 (Sequential)	(None, None, 256)	525,568	layer_normalization_2[0][…
add_3 (Add)	(None, None, 256)	0	layer_normalization_2[0][… sequential_1[0][0]
layer_normalization_3 (LayerNormalization)	(None, None, 256)	512	add_3[0][0]
multi_head_attention_2 (MultiHeadAttention)	(None, None, 256)	263,168	layer_normalization_3[0][… layer_normalization_3[0][…

dropout_7 (Dropout)	(None, None, 256)	0	multi_head_attention_2[0]…
add_4 (Add)	(None, None, 256)	0	layer_normalization_3[0][… dropout_7[0][0]
layer_normalization_4 (LayerNormalization)	(None, None, 256)	512	add_4[0][0]
sequential_2 (Sequential)	(None, None, 256)	525,568	layer_normalization_4[0][…
add_5 (Add)	(None, None, 256)	0	layer_normalization_4[0][… sequential_2[0][0]
layer_normalization_5 (LayerNormalization)	(None, None, 256)	512	add_5[0][0]
multi_head_attention_3 (MultiHeadAttention)	(None, None, 256)	263,168	layer_normalization_5[0][… layer_normalization_5[0][…
dropout_10 (Dropout)	(None, None, 256)	0	multi_head_attention_3[0]…
add_6 (Add)	(None, None, 256)	0	layer_normalization_5[0][… dropout_10[0][0]
layer_normalization_6 (LayerNormalization)	(None, None, 256)	512	add_6[0][0]
sequential_3 (Sequential)	(None, None, 256)	525,568	layer_normalization_6[0][…
add_7 (Add)	(None, None, 256)	0	layer_normalization_6[0][… sequential_3[0][0]
layer_normalization_7 (LayerNormalization)	(None, None, 256)	512	add_7[0][0]
multi_head_attention_4 (MultiHeadAttention)	(None, None, 256)	263,168	layer_normalization_7[0][… layer_normalization_7[0][…
dropout_13 (Dropout)	(None, None, 256)	0	multi_head_attention_4[0]…
add_8 (Add)	(None, None, 256)	0	layer_normalization_7[0][… dropout_13[0][0]
layer_normalization_8 (LayerNormalization)	(None, None, 256)	512	add_8[0][0]
sequential_4 (Sequential)	(None, None, 256)	525,568	layer_normalization_8[0][…
add_9 (Add)	(None, None, 256)	0	layer_normalization_8[0][… sequential_4[0][0]
layer_normalization_9 (LayerNormalization)	(None, None, 256)	512	add_9[0][0]
layer_normalization_9 (LayerNormalization)	(None, None, 256)	512	add_9[0][0]
multi_head_attention_5 (MultiHeadAttention)	(None, None, 256)	263,168	layer_normalization_9[0][… layer_normalization_9[0][…
dropout_16 (Dropout)	(None, None, 256)	0	multi_head_attention_5[0]…
add_10 (Add)	(None, None, 256)	0	layer_normalization_9[0][… dropout_16[0][0]
layer_normalization_10 (LayerNormalization)	(None, None, 256)	512	add_10[0][0]
sequential_5 (Sequential)	(None, None, 256)	525,568	layer_normalization_10[0]…
add_11 (Add)	(None, None, 256)	0	layer_normalization_10[0]… sequential_5[0][0]
layer_normalization_11 (LayerNormalization)	(None, None, 256)	512	add_11[0][0]
ctc_logits (Dense)	(None, None, 33)	8,481	layer_normalization_11[0]…

Додаток Б. Фрагменти з лістингу програмного коду

data_preprocesing

```
import tensorflow as tf


VOCAB = list(" " + "абвгґдеєжзиіїйклмнопрстуфхцчшщьюя" + "'")  # space
must be index 0 in CTC
char2idx = {c: i for i, c in enumerate(VOCAB)}
idx2char = {i: c for c, i in char2idx.items()}

BATCH_SIZE = 8


FFT_LENGTH = 1024
NUM_MEL_BINS = 64
SAMPLE_RATE = 16000
lower_hz, upper_hz = 80.0, SAMPLE_RATE / 2
NUM_SPECTRO_BINS = FFT_LENGTH // 2 + 1

MEL_WT = tf.signal.linear_to_mel_weight_matrix(
    NUM_MEL_BINS,
    NUM_SPECTRO_BINS,
    SAMPLE_RATE,
    lower_hz,
    upper_hz
)


@tf.function
def extract_features(waveform, sample_rate=16000):
    stfts = tf.signal.stft(waveform, frame_length=640, frame_step=320,
fft_length=FFT_LENGTH)
    spectrogram = tf.abs(stfts)

    mel_spectrogram = tf.tensordot(spectrogram, tf.cast(MEL_WT,
spectrogram.dtype), 1)
    return tf.math.log(mel_spectrogram + 1e-6)


def encode_text(text):
    return [char2idx [c] for c in text.lower() if c in char2idx]


def decode_text(encoded: list [int]) -> str:
    return "".join(idx2char [i] for i in encoded)


def data_generator(raw_dataset):
    for example in raw_dataset:
        text = example ["transcription"].strip().lower()
```

```
        label = encode_text(text)
        if len(label) == 0:
            continue
        waveform = tf.convert_to_tensor(example ["audio"] ["array"],
dtype=tf.float32)
        log_mel = extract_features(waveform, sample_rate=example ["audio"]
["sampling_rate"])
        label = encode_text(example ["transcription"])
        yield log_mel, label


def prepare_dataset(raw_dataset):
    ds = tf.data.Dataset.from_generator(lambda:
data_generator(raw_dataset),
                                        output_signature=(
                                            tf.TensorSpec(shape=(None,
64), dtype=tf.float32),
                                            tf.TensorSpec(shape=(None,),
dtype=tf.int32)))

    def pad_batch(log_mel, label):
        return {
            "features": log_mel,
            "label": label,
            "input_length": tf.shape(log_mel) [0],
            "label_length": tf.shape(label) [0]
        }

    ds = ds.map(pad_batch)
    ds = ds.padded_batch(
        BATCH_SIZE,
        padded_shapes={
            "features": [None, 64],
            "label": [None],
            "input_length": [],
            "label_length": []
        },
        padding_values={
            "features": 0.0,
            "label": tf.cast(-1, tf.int32),  # ignored by ctc_batch_cost
            "input_length": 0,
            "label_length": 0
        },
        drop_remainder=True
    )

    return ds.prefetch(tf.data.AUTOTUNE)
```

models

```
def build_wav2vec2_ctc_model(num_classes, model_dim=256, num_layers=6,
num_heads=4, ff_dim=1024):
```

```
    inputs = tf.keras.Input(shape=(None, 64))

    x = tf.keras.layers.Conv1D(128, 10, strides=5, padding='same',
activation='gelu')(inputs)
    x = tf.keras.layers.Conv1D(256, 3, strides=2, padding='same',
activation='gelu')(x)
    x = tf.keras.layers.Conv1D(model_dim, 3, strides=2, padding='same',
activation='gelu')(x)

    def add_pos_emb(tensor):
        seq_len = tf.shape(tensor) [1]
        positions = tf.range(seq_len)
        pos_emb = tf.keras.layers.Embedding(1000, model_dim)(positions)
        pos_emb = pos_emb [tf.newaxis, :, :]  # (1, seq_len, dim)
        return tensor + pos_emb [:, :seq_len, :]

    x = tf.keras.layers.Lambda(add_pos_emb)(x)

    for _ in range(num_layers):
        attn_out = tf.keras.layers.MultiHeadAttention(num_heads=num_heads,
key_dim=model_dim//num_heads)(x, x)
        attn_out = tf.keras.layers.Dropout(0.1)(attn_out)
        x = tf.keras.layers.LayerNormalization(epsilon=1e-6)(x + attn_out)

        ffn = tf.keras.Sequential( [
            tf.keras.layers.Dense(ff_dim, activation='gelu'),
            tf.keras.layers.Dense(model_dim),
            tf.keras.layers.Dropout(0.1)
        ])
        x = tf.keras.layers.LayerNormalization(epsilon=1e-6)(x + ffn(x))

    x = tf.keras.layers.Dense(num_classes + 1, name="ctc_logits")(x)

    return tf.keras.Model(inputs, x, name="wav2vec2_ctc_midsize")
```

main

```
def ctc_batch_cost(y_true, y_pred, input_lengths, label_lengths):
    """
    y_true: int32 tensor of shape (batch_size, max_label_length)
    y_pred: float32 tensor of shape (batch_size, max_time, num_classes +
1) - logits
    input_lengths: int32 tensor of shape (batch_size,) - actual input
sequence lengths
    label_lengths: int32 tensor of shape (batch_size,) - actual label
lengths
    """
    indices = tf.where(tf.not_equal(y_true, -1))
    values = tf.gather_nd(y_true, indices)
    shape = tf.shape(y_true, out_type=tf.int64)

    sparse_labels = tf.SparseTensor(indices=indices, values=values,
dense_shape=tf.cast(shape, tf.int64))
```

```
    y_pred = tf.transpose(y_pred, [1, 0, 2])

    loss = tf.nn.ctc_loss(
        labels=sparse_labels,
        logits=y_pred,
        label_length=None,
        logit_length=input_lengths,
        blank_index=-1
    )

    return loss


@keras.saving.register_keras_serializable()
class CTCLossModel(tf.keras.Model):
    def __init__(self, base_model, **kwargs):
        super().__init__(**kwargs)
        self.base = base_model
        self.loss_tracker = tf.keras.metrics.Mean(name="loss")

    def get_config(self):
        config = super().get_config()
        # Serialize the inner model by config, not the object
        config.update({
            "base_model_config": self.base.get_config(),
            "base_model_class": self.base.__class__.__name__
        })
        return config

    @classmethod
    def from_config(cls, config):
        base_model_config = config.pop("base_model_config")
        base_model_class_name = config.pop("base_model_class")

        # Rebuild the base model using its class and config
        base_model_class = getattr(tf.keras.models, base_model_class_name,
None)
        if base_model_class is None:
            base_model = tf.keras.Model.from_config(base_model_config)
        else:
            base_model = base_model_class.from_config(base_model_config)

        return cls(base_model, **config)

    def get_build_config(self):
        # Keras saves this at save-time to later rebuild layers
        return {"input_shape": (None, None, 64)}

    def build_from_config(self, config):
        # Called during loading to rebuild the model's variables
        input_shape = config ["input_shape"]
        self.build(input_shape)
```

```
    @property
    def metrics(self):
        return [self.loss_tracker]

    def train_step(self, data):
        features = data ["features"]
        labels = data ["label"]
        input_lengths = data ["input_length"]
        label_lengths = data ["label_length"]

        with tf.GradientTape() as tape:
            logits = self.base(features, training=True)
            input_lengths = tf.fill( [tf.shape(logits) [0]],
tf.shape(logits) [1])
            loss = ctc_batch_cost(labels, logits, input_lengths,
label_lengths)

        grads = tape.gradient(loss, self.base.trainable_variables)
        self.optimizer.apply_gradients(zip(grads,
self.base.trainable_variables))

        self.loss_tracker.update_state(loss)
        return {"loss": self.loss_tracker.result()}

    def call(self, inputs):
        return self.base(inputs)


def train(model: CTCLossModel, dataset: tf.data.Dataset, epochs: int):
    optimizer = tf.keras.optimizers.Adam()
    model.compile(optimizer=optimizer)

    for epoch in range(epochs):
        print(f"\nEpoch {epoch + 1}/{epochs}")
        model.reset_metrics()

        for step, batch in enumerate(dataset):
            logs = model.train_step(batch)
            if step % 10 == 0:
                print(f"  Step {step}: Loss = {logs ['loss']:.4f}")
                if logs ['loss'] < 5:
                    dummy_input = tf.random.normal( [1, 100, 64])
                    _ = model(dummy_input)
                    model.save('./final_model.keras')
                    return
        print(f"Epoch {epoch + 1} completed. Avg Loss: {logs
['loss']:.4f}")
    dummy_input = tf.random.normal( [1, 100, 64])
    _ = model(dummy_input)
    model.save('./final_model.keras')
```

```
"""
Dataset element structure:
{
    audio: {
        path: str,
        array: np.ndarray,
        sampling_rate: int,
    },
    duration: float,
    transcription: str,
}
"""

if __name__ == '__main__':
    # dataset = load_dataset("speech-uk/voice-of-america", split='train',
streaming=True)

    prepared_ds = prepare_dataset(dataset)
    dummy_input = tf.random.normal([1, 100, 64])
    base_model = build_wav2vec2_ctc_model(num_classes=len(VOCAB))
    ctc_model = CTCLossModel(base_model)
    train(ctc_model, prepared_ds, epochs=2)
    base_model(dummy_input)
    base_model.summary()
    loaded_model = keras.models.load_model('./final_model.keras',
custom_objects={"CTCLossModel": CTCLossModel})

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # --- Left plot ---
    axes[0].plot(steps, loss_values, marker='o', linestyle='-', color='b',
linewidth=2)
    axes[0].set_title('Training Loss first 100 steps', fontsize=14)
    axes[0].set_xlabel('Step', fontsize=12)
    axes[0].set_ylabel('Loss', fontsize=12)
    axes[0].grid(True, linestyle='--', alpha=0.6)

    for i, loss in enumerate(loss_values_end):
        axes[0].text(steps_end[i], loss + 0.02, f"{loss:.2f}",
ha='center', fontsize=12)

    # --- Right plot ---
    axes[1].plot(steps_end, loss_values_end, marker='o', linestyle='-',
color='b', linewidth=2)
    axes[1].set_title('Training Loss last 100 steps', fontsize=14)
    axes[1].set_xlabel('Step', fontsize=12)
    axes[1].set_ylabel('Loss', fontsize=12)
    axes[1].grid(True, linestyle='--', alpha=0.6)

    fig.tight_layout()
    plt.show()

y, sr = librosa.load(wav_path, sr=None)
```

```
S = librosa.feature.melspectrogram(
    y=y,
    sr=sr,
    n_fft=1024,
    hop_length=512,
    n_mels=64,
    power=2.0
)

S_dB = librosa.power_to_db(S, ref=np.max)

plt.figure(figsize=(10, 5))
librosa.display.specshow(
    S_dB,
    sr=sr,
    hop_length=512,
    x_axis='time',
    y_axis='mel',
    cmap='magma'
)
plt.colorbar(format='%+2.0f dB')
plt.title("Mel Spectrogram")
plt.tight_layout()
plt.show()

inference
midsize_wav2vec = keras.models.load_model(
    './final_model.keras',
    custom_objects={"CTCLossModel": CTCLossModel}
)

def greedy_decode(logits, vocab=VOCAB, blank_id=0):
    probs = tf.nn.softmax(logits, axis=-1)
    token_ids = tf.argmax(probs, axis=-1).numpy()

    decoded = []
    prev = blank_id
    for t in token_ids:
        if t != blank_id and t != prev:
            decoded.append(vocab[t])
        prev = t

    return "".join(decoded).strip()


def transcribe_with_midsize_wav2vec(audio_path: str) -> str:
    audio, sr = sf.read(audio_path)

    if len(audio.shape) > 1:
        audio = audio.mean(axis=1)
    features = extract_features(tf.constant(audio, dtype=tf.float32))
    features = tf.expand_dims(features, 0)
```

```
    logits = wav2vec2_ctc_midsize(features, training=False)
    text = ctc_decode(logits)

    return text

compute metrics

def levenshtein(a, b):
    n, m = len(a), len(b)
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    for i in range(n + 1):
        dp[i][0] = i
    for j in range(m + 1):
        dp[0][j] = j

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            cost = 0 if a[i - 1] == b[j - 1] else 1
            dp[i][j] = min(
                dp[i - 1][j] + 1,  # deletion
                dp[i][j - 1] + 1,  # insertion
                dp[i - 1][j - 1] + cost  # substitution
            )

    return dp[n][m]


def compute_cer(refs, hyps):
    total_dist = 0
    total_chars = 0

    for r, h in zip(refs, hyps):
        total_dist += levenshtein(r, h)
        total_chars += len(r)

    return total_dist / total_chars if total_chars > 0 else 0


def compute_wer(refs, hyps):
    total_dist = 0
    total_words = 0

    for r, h in zip(refs, hyps):
        r_words = r.split()
        h_words = h.split()

        total_dist += levenshtein(r_words, h_words)
        total_words += len(r_words)

    return total_dist / total_words if total_words > 0 else 0
```

```
def compute_ser(refs, hyps):
    sentence_errors = 0
    total_sentences = len(refs)

    for r, h in zip(refs, hyps):
        if r.strip() == "":
            continue
        if levenshtein(r.split(), h.split()) != 0:
            sentence_errors += 1

    return sentence_errors / total_sentences if total_sentences > 0 else 0


if __name__ == "__main__":
    cers = []
    wers = []
    sers = []
    rtfs = []
    for element in dataset:
        ref_text = element['transcription']
        start_time = time.perf_counter()
        hyp_text = transcribe_using_midsize_wav2vec(midsize_wav2vec,
element['audio']['array'])
        end_time = time.perf_counter()
        elapsed_time = end_time - start_time

        cers.append(compute_cer(ref_text, hyp_text))
        wers.append(compute_wer(ref_text, hyp_text))
        sers.append(compute_ser(ref_text, hyp_text))
        rtfs.append(elapsed_time / element['duration'])

    print("CER:", sum(cers) / len(cers))
    print("WER:", sum(wers) / len(wers))
    print("SER:", sum(sers) / len(sers))
    print("RTF:", sum(rtfs) / len(rtfs))
```

forms

```
class UploadForm(forms.Form):
    video = forms.FileField(label='Виберіть медіа файл')

    def clean_video(self):
        f = self.cleaned_data ['video']

        allowed_content_types = (
            'video/mp4',
            'video/x-matroska',
            'audio/mpeg',
            'application/octet-stream',
        )
```

```
        allowed_extensions = ('.mp4', '.mkv', '.mp3', '.wav')

        # MIME type check
        if f.content_type not in allowed_content_types:
            # Extension check as backup
            if not any(f.name.lower().endswith(ext) for ext in
allowed_extensions):
                raise forms.ValidationError(
                    'Будь ласка, завантажте файл у форматі MP4 або MP3.'
                )

        return f


def extract_audio_from_video(video_path: str, out_wav_path: str) -> None:
    """
    Uses ffmpeg to extract audio to a WAV file with 16 kHz mono PCM —
typical for many ASR models.
    Requires `ffmpeg` on PATH.
    """
    os.makedirs(os.path.dirname(out_wav_path), exist_ok=True)

    cmd = [
        'ffmpeg', '-y',
        '-i', video_path,
        '-ac', '1',        # mono
        '-ar', '16000',    # 16 kHz
        '-vn',             # no video
        out_wav_path
    ]
    completed = subprocess.run(cmd, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
    if completed.returncode != 0:
        raise RuntimeError(f"ffmpeg failed: {completed.stderr.decode('utf-
8', errors='ignore')}")

def download_transcript_view(request, pk):
    video = get_object_or_404(Video, pk=pk)
    transcript = request.session.get('transcript', '')

    response = HttpResponse(transcript, content_type='text/plain;
charset=utf-8')
    response['Content-Disposition'] = f'attachment;
filename="transcript_{pk}.txt"'

    return response
```

base

```
<!doctype html>
<html lang="en">
<head>
     <meta charset="utf-8" />
```

```
     <meta name="viewport" content="width=device-width,initial-scale=1"
/>
    {% load static %}
    <link rel="stylesheet" href="{% static 'styles/styles.css' %}">

     <title>Розпізнавання мови з медіа</title>
    <script>
        document.addEventListener('DOMContentLoaded', () => {
            const fileInput = document.querySelector('#id_video');
            const fileChosen = document.querySelector('#file-chosen');
            const fileLabel = document.querySelector('label
[for=id_video]');
            if (fileInput) {
                    fileInput.addEventListener('change', function() {
                        fileChosen.textContent = this.files.length > 0 ?
this.files [0].name : 'Файл не вибрано';
                });
            }
        });
    </script>
</head>
<body>
     <div class="container">
          <div class="card">
               {% block content %}{% endblock %}
          </div>
     </div>
</body>
</html>
```

result

```
{% extends 'transcriber/base.html' %}
{% block content %}
     <h1>Транскрипт</h1>
     <div class="meta">
          <div class="file-info">Медіа: <strong>{{ video.file.name
}}</strong></div>
          <!--  <div class="muted">Дата і час завантаження: {{
video.uploaded_at }}</div> -->
     </div>

     {% if transcript %}
          <div class="transcript">{{ transcript }}</div>
          <button class="download-btn" type="submit">Завантажити
транскрипт</button>
     {% else %}
          <p class="muted">Розшифровку не знайдено. Можливо, сеанс
закінчився — спробуйте завантажити ще раз.</p>
     {% endif %}
```

```
     <p style="margin-top:16px"><a href="{% url 'transcriber:upload' 
%}">Транскрибувати ще один файл</a></p>
{% endblock %}
```

upload

```
{% extends 'transcriber/base.html' %}
{% block content %}
     <h1>Завантажити відео або аудіофайл</h1>
     <p class="muted">Завантажте MP4 або MP3 файл. Аудіо буде витягнуто з 
файлу та буде створено транскрипт за допомогою нейронної мережі.</p>

     {% if error %}
          <p style="color:crimson;">{{ error }}</p>
     {% endif %}

     <form method="post" enctype="multipart/form-data">
          {% csrf_token %}
          <div class="file-upload-wrapper">
           <label for="id_video" class="file-label">Вибрати файл</label>
               {{ form.video }}
           <span id="file-chosen" class="file-chosen">Файл не 
вибрано</span>
        </div>
           <button class="btn" type="submit">Завантажити та 
транскрибувати</button>
     </form>
     <hr style="margin-top:20px" />
{% endblock %}
```