МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ГОЛОСОВОЇ СИСТЕМИ АКТИВАЦІЇ З ВИКОРИСТАННЯМ НЕЙРОННИХ МЕРЕЖ»**

Виконав: студент 2 курсу, групи 8.1213-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
Д.Є. Пугач
(ініціали та прізвище)

Керівник завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра прикладної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Пугачу Денису Євгенійовичу
(прізвище, ім’я та по батькові)

1. Тема роботи Розробка голосової системи активації з використанням нейронних мереж

керівник роботи Гребенюк Сергій Миколайович, д.т.н., професор
(прізвище, ім’я та по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від «01» березня 2024 року № 385-с

2. Строк подання студентом роботи 25.11.2024 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка голосової системи активації.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 07.03.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	01.04.2024	
2.	Збір вихідних даних.	13.05.2024	
3.	Обробка методичних та теоретичних	28.06.2024	
	джерел.		
4.	Розробка першого та другого розділу.	09.09.2024	
5.	Розробка третього розділу.	28.10.2024	
6.	Оформлення та нормоконтроль	20.11.2024	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	13.12.2024	

Студент _____ Д.Є. Пугач
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.М. Гребенюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка голосової системи активації з використанням нейронних мереж»: 54 с., 22 рис., 18 джерел, 3 додатки.

ГОЛОСОВА СИСТЕМА АКТИВАЦІЇ, НЕЙРОННІ МЕРЕЖІ, РЕКУРЕНТНІ МЕРЕЖІ, GRU, MFCC.

Об’єкт дослідження – процес розпізнавання голосових команд на базі рекурентних нейронних мереж.

Мета роботи:розробка та аналіз голосової системи активації з використанням рекурентних нейронних мереж, а також дослідження її ефективності та можливостей застосування.

Методи дослідження – моделювання, аналіз, експеримент, синтез.

У роботі досліджено предметну область – розробка голосової системи активації за допомогою рекурентних нейронних мереж з використанням бібліотеки PyTorch мови програмування Python.

У ході розробки даної системи особливу увагу приділено складанню оптимального набору даних для навчання моделі, що забезпечило стабільну роботу моделі у різних умовах незалежно від навколишніх обставин.

Розроблена система демонструє високу точність у розпізнаванні ключової фрази, ефективно фільтруючи сторонні голосові команди. Результати роботи підтверджують ефективність нейронних мереж із GRU у задачах розпізнавання голосу та можливість їхнього застосування у реальних системах голосового керування.

Практична значущість розробленої системи полягає в її можливості бути інтегрованою у розумні пристрої, системи автоматизації та інтерфейси голосового керування. Запропонований підхід забезпечує високу точність і адаптивність до різних умов використання, що робить його перспективним для подальшого впровадження та вдосконалення.

SUMMARY

Master’s qualifying paper “Development of a Voice Activation System Using Neural Networks”: 54 p., 22 figures, 18 references, 3 supplements.

VOICE ACTIVATION SYSTEM, NEURAL NETWORKS, RECURRENT NETWORKS, GRU, MFCC.

The object of the study is the process of recognizing voice commands based on recurrent neural networks.

The aim of the study is the development and analysis of a voice activation system using recurrent neural networks, as well as studying its efficiency and application possibilities.

The method of research is modeling, analysis, experiment, synthesis.

The work investigates the subject area – the development of a voice activation system using recurrent neural networks with the PyTorch library in the Python programming language.

During the development of this system, special attention was paid to creating an optimal dataset for training the model, which ensured the stable performance of the model under various conditions, regardless of the surrounding environment.

The developed system demonstrates high accuracy in recognizing the key phrase while effectively filtering out extraneous voice commands. The results confirm the effectiveness of GRU-based neural networks in voice recognition tasks and their potential application in real-world voice control systems.

The practical significance of the developed system lies in its ability to be integrated into smart devices, automation systems, and voice control interfaces. The proposed approach ensures high accuracy and adaptability to various usage conditions, making it promising for further implementation and improvement.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 7
1 Аналіз сучасних систем розпізнавання мови 9
1.1 Аналіз систем голосового керування 9
1.2 Аналіз методу розпізнавання мови 10
1.3 Нейронні мережі 16
2 Технології голосових систем активації 19
2.1 Рекурентні нейронні мережі 19
2.2 Мел-частотні кепстральні коефіцієнти 22
3 Програмна реалізація голосової системи активації 25
3.1 Вибір засобів реалізації 25
3.2 Збір датасету 26
3.3 Розробка програми 28
3.4 Тестування й оцінка роботи моделі 35
Висновки 39
Перелік посилань 41
Додаток А Програмна реалізація голосової системи активації з використанням нейронних мереж 43
Додаток Б Скрипт для пошуку файлів, що містять ключову фразу, у розміченому Text-to-Speech датасеті 51
Додаток В Програмна реалізація тестування голосом розробленої системи активації 52

ВСТУП

В сучасному цифровому світі, де технології стрімко розвиваються, зручні та інтуїтивні інтерфейси взаємодії з обчислювальними пристроями стають ключовими для забезпечення ефективності та зручності користувачів. Серед різноманітних інтерфейсів виокремлюються голосові системи активації, які дозволяють взаємодіяти з пристроями за допомогою голосових команд. Розробка таких систем є актуальною та перспективною завдяки своєму потенціалу в удосконаленні способів взаємодії людини з технікою.

Ця робота спрямована на розробку голосової системи активації, що базується на використанні нейронних мереж. На відміну від традиційних систем, що використовують правила та шаблони, глибокі нейронні мережі можуть самостійно вивчати та адаптуватися до різних голосових команд, що робить їх більш гнучкими та ефективними в розпізнаванні мови.

Метою дослідження є розробка та аналіз голосової системи активації з використанням нейронних мереж, а також дослідження її ефективності та можливостей застосування. Ця робота має важливе практичне значення, оскільки вона відкриває нові можливості для розробки інноваційних інтерфейсів взаємодії з технологією, що може сприяти покращенню продуктивності та комфорту користувачів.

Об’єктом дослідження є процес розпізнавання голосових команд, який охоплює розуміння та інтерпретацію звукових сигналів.

Предметом дослідження є розробка голосової системи активації з використанням нейронних мереж, включаючи її алгоритми, методи та програмну реалізацію.

Завдання роботи:

– проведення аналізу сучасних систем розпізнавання мови, включаючи системи голосового керування та методи розпізнавання мови;

- ознайомлення з сучасними технологіями голосових систем активації, зокрема рекурентних нейронних мереж і мел-кепстральних коефіцієнтів;
- розробка програмної реалізації голосової системи активації з використанням нейронних мереж;
- проведення тестування та оцінка роботи розробленої моделі, включаючи перевірку точності розпізнавання голосових команд та їх швидкість;
- визначення практичної застосовності розробленої системи та її потенціалу для подальшого вдосконалення та розвитку голосових інтерфейсів.

Новизна даної роботи полягає у використанні рекурентних нейронних мереж для розробки голосової системи активації. Цей підхід відрізняється від традиційних методів і може покращити точність та ефективність розпізнавання мови. Інтеграція рекурентних нейронних мереж та мел-кепстральних коефіцієнтів забезпечує більш гнучке та динамічне розпізнавання голосових команд. Робота також включає програмну реалізацію та тестування розробленої системи, що дозволяє практично перевірити її ефективність.

Враховуючи практичну застосовність системи у різних галузях, ця робота має потенціал у вдосконаленні голосових інтерфейсів та покращенні способів взаємодії з обчислювальною технікою.

1 АНАЛІЗ СУЧАСНИХ СИСТЕМ РОЗПІЗНАВАННЯ МОВИ

Розробка голосової системи активації з використанням нейронних мереж є важливою і складною задачею у сфері штучного інтелекту та машинного навчання. Ця технологія знаходиться на перетині обробки природної мови (NLP), розпізнавання мови та глибокого навчання.

1.1 Аналіз систем голосового керування

Технології систем голосового керування значно просунулися вперед завдяки розвитку штучного інтелекту (AI) та машинного навчання (ML). Сучасні системи голосового керування широко використовуються у різних галузях, від побутових пристроїв до промислового застосування.

Моделі глибокого навчання, такі як BERT і GPT-3, забезпечують високоточне розуміння та генерацію тексту, що покращує взаємодію користувачів з голосовими системами. Сучасні системи здатні розуміти контекст і сенс сказаного, що робить взаємодію більш природною та ефективною. Використання глибоких нейронних мереж, таких як CNN (Convolutional Neural Networks), RNN (Recurrent Neural Networks) та LSTM (Long Short-Term Memory), значно покращило точність розпізнавання мови, а гібридні системи, які поєднують традиційні методи обробки мови з нейронними мережами, дозволяють досягти кращих результатів у складних умовах [1].

Віртуальні асистенти, такі як Siri, Google Assistant та Alexa, стали звичними у смартфонах, розумних колонках, автомобілях та інших пристроях. Їх функціонал розширився, і тепер вони можуть виконувати широкий спектр завдань, від встановлення будильників до керування розумним будинком. Інтеграція голосового керування з Інтернетом речей (IoT) дозволяє користувачам керувати освітленням, опаленням, системами

безпеки та іншими пристроями в розумних будинках, а також навігацією та мультимедіа в автомобілях.

У бізнесі та промисловості голосове керування використовується для автоматизації процесів в офісах, на виробничих лініях та у логістичних центрах. Голосові боти покращують обслуговування клієнтів, забезпечуючи швидку та ефективну підтримку. Проте, незважаючи на значні досягнення, залишаються виклики, пов’язані з точністю та надійністю систем. Голосові системи можуть мати проблеми з розпізнаванням мови в шумних середовищах, а також із розумінням акцентів і діалектів.

Захист даних та конфіденційність також є важливими питаннями, оскільки збір та обробка голосових даних викликає занепокоєння щодо безпеки. Голосові системи можуть бути вразливі до фальсифікацій та атак з імітацією голосу. Високоточні моделі потребують значних обчислювальних ресурсів та енергії, що може бути проблемою для мобільних та вбудованих систем.

У майбутньому очікується покращення моделей для кращого розуміння контексту і багатозначності мови, а також створення систем, які можуть адаптуватися до нових користувачів і умов. Інтеграція голосового керування з іншими технологіями, такими як розширена реальність (AR) та швидкісні мережі 5G, обіцяє нові можливості для взаємодії. Удосконалення безпеки шляхом поліпшення біометричної аутентифікації та використання передових методів шифрування для захисту голосових даних також буде пріоритетом.

Розвиток технологій голосового керування продовжує робити взаємодію з пристроями та системами більш зручною та інтуїтивно зрозумілою для користувачів.

1.2 Аналіз методу розпізнавання мови

Програмні методи розпізнавання мови являють собою комплекс технологій та алгоритмів, спрямованих на перетворення усного мовлення в

текст. Ідентифікація мови є складним процесом розпізнавання патернів, під час якого акустичні сигнали досліджуються та впорядковуються у багаторівневу систему. Ця структура охоплює фонеми, окремі слова, фрази та цілі речення. На кожному з цих рівнів можуть існувати певні часові обмеження.

Одним із ключових етапів розпізнавання мови є попередня обробка сигналу. Сюди входять процеси очищення аудіо від шумів та інших перешкод, а також нормалізація рівня гучності. Після цього аудіо розбивається на невеликі фрагменти (фрейми), які піддаються аналізу для виділення важливих ознак, таких як мел-кепстральні коефіцієнти (MFCC) або спектрограми. Ці ознаки є основою для подальшого аналізу і розпізнавання мови [2].

Стандартна система розпізнавання мовних сигналів складається з кількох ключових етапів:

– цифровізація мовлення: початковий аудіосигнал перетворюється у цифрову форму шляхом дискретизації на високій частоті – це створює часову послідовність амплітуд, які представляють акустичний сигнал;
– обробка сигналу: на цьому етапі проводиться попередня обробка, яка включає перетворення та компресію мовних даних (мета цієї обробки – спростити подальший аналіз без суттєвих втрат інформації);
– методи вилучення характеристик.

Для ефективного аналізу сигналів і компресії даних використовуються такі підходи:

– аналіз Фур’є дозволяє визначати частоти в часовому масштабі, забезпечуючи візуальне представлення спектра;
– перцептивне лінійне прогнозування (PLP) орієнтоване на моделювання слухового сприйняття мовлення;
– лінійне прогностичне кодування (LPC) використовується для визначення спектральних характеристик мовлення;

– кепстральний аналіз застосовується для вилучення параметрів мовлення, що характеризують акустичну природу сигналу [3].

Після аналізу сигналів отримується послідовність кадрів, які можуть бути доповнені даними про динаміку мовлення. Це підвищує точність і ефективність роботи системи.

На рисунку 1.1 представлено загальну конструкцію типової системи розпізнавання голосових команд, а на рисунку 1.2 – акустичний аналіз отриманих кадрів мовлення. Формування таких кадрів є критично важливим для точного моделювання мовних сигналів і подальшого розпізнавання.

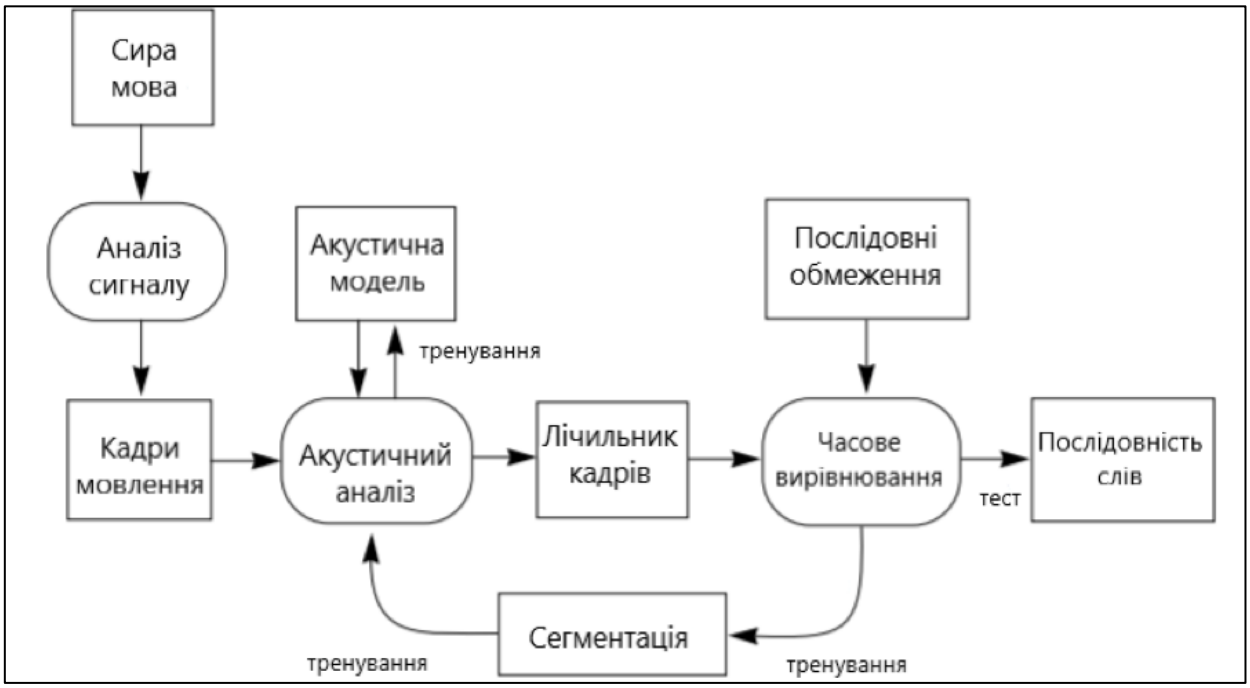


Рисунок 1.1 – Типова схема роботи системи розпізнавання голосових команд

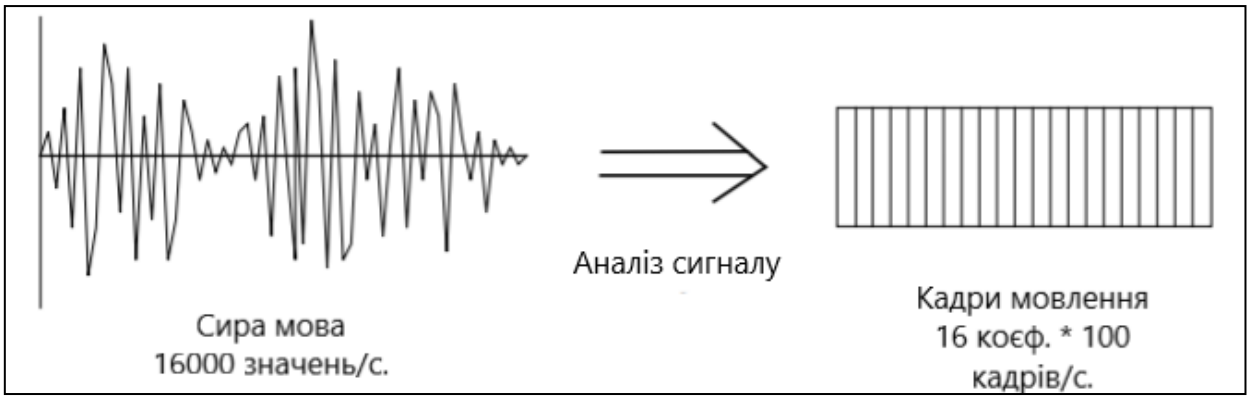


Рисунок 1.2 – Акустичний аналіз кадрів мовлення

Аналіз мовних кадрів, зосереджений на їхньому акустичному змісті, потребує використання спеціалізованих акустичних моделей. Ці моделі розрізняються за ступенем деталізації, рівнем контекстної залежності, методом представлення інформації та іншими характеристиками, що впливають на їхню ефективність у різних системах.

Одним із найпростіших підходів є використання шаблонів для розпізнавання окремих слів. У цьому випадку невідоме слово ідентифікується шляхом порівняння його акустичного сигналу із заздалегідь створеними зразками. Хоча цей метод є інтуїтивно зрозумілим і відносно легким у реалізації, його ефективність значно обмежується. Він не здатний моделювати широкий спектр акустичних варіацій, якщо для кожного слова не створити велику кількість шаблонів. Також цей підхід ускладнюється у випадках, коли необхідно працювати із короткими фрагментами мовлення, що не досягають рівня слова. Через це метод шаблонів є придатним лише для простих систем із фіксованим і невеликим набором слів [4].

На відміну від статичних шаблонів, сучасні системи розпізнавання мовлення використовують більш гнучкі підходи, засновані на навчанні акустичних моделей. У таких моделях кожне слово представляється як послідовність станів, де кожен стан відповідає певному звуку в межах слова. Ймовірнісні розподіли, які описують ці звуки, часто моделюються параметрично, наприклад, за допомогою гаусівського розподілу. Навчання моделі спрямоване на визначення оптимальних параметрів, які найкраще описують характерні ознаки сигналу.

У ситуаціях, коли зробити припущення про форму розподілу даних виявляється складно або недоцільно, перевага надається непараметричним методам. Такі підходи дозволяють працювати без необхідності задавати попередню форму розподілу й забезпечують більшу гнучкість у моделюванні складних залежностей. Найбільш поширеним прикладом використання непараметричних методів є нейронні мережі, оскільки завдяки своїй архітектурі вони можуть ефективно навчатися на великих обсягах даних та адаптуватися до різноманітних структур і властивостей сигналів.

Важливим аспектом ефективності моделей є їхня контекстна чутливість. Моделі, здатні враховувати контекст, демонструють вищу точність у розпізнаванні, однак для цього необхідні значні обсяги даних для навчання. В умовах великих словникових запасів часто використовуються фонемні моделі, які забезпечують оптимальний баланс між обсягом необхідної інформації та точністю розпізнавання.

Розпізнавання мови є складним процесом, який залежить від багатьох факторів. Коли система працює з невеликим словниковим запасом, ідентифікація окремих слів зазвичай не викликає труднощів. Однак із розширенням кількості слів зростає ймовірність помилок. Також важливе значення має залежність системи від конкретного спікера, що може обмежувати універсальність розпізнавання. Тип мовлення (ізольоване, з паузами або безперервне) відіграє значну роль у виборі відповідного методу обробки сигналу [4].

Системи, орієнтовані на конкретного користувача, створити простіше, адже вони адаптовані до індивідуальних особливостей мовлення певної особи [5].

Системи, незалежні від конкретного спікера, є набагато складнішими, оскільки мають враховувати унікальні риси мовлення кожної людини, такі як інтонація, швидкість і чіткість. Як наслідок, рівень помилок у таких системах зазвичай у декілька разів вищий [5].

Сучасні технології пропонують альтернативні підходи, зокрема мультиакустичні системи, які використовуються обмеженим колом осіб, і адаптивні системи, здатні налаштовуватися на різних користувачів. Водночас ефективність цих систем залежить від типу мовлення. Наприклад, ізольоване мовлення, що складається з окремих слів або звуків, та мовлення з паузами між словами є відносно простими для обробки. У таких випадках слова чітко розмежовані, і їх межі легко визначити.

Неперервне мовлення, навпаки, становить серйозний виклик, оскільки слова не мають явних меж і можуть бути вимовлені невиразно або

супроводжуватися сторонніми шумами. Крім того, навіть при використанні фіксованого словника ефективність розпізнавання залежить від порядку слів у реченні, особливостей вимови, а також семантичних і граматичних обмежень.

Граматичні обмеження базуються на очікуваних послідовностях слів, що дозволяє системі передбачати ймовірний розвиток речення. Однак складність системи краще оцінювати не за обсягом словника, а за мірою її непередбачуваності.

Ефективність також значною мірою залежить від умов експлуатації, зокрема рівня навколишнього шуму, акустичних спотворень, типу використовуваних мікрофонів та обмежень частотного діапазону, наприклад, у разі передачі сигналу через телефон. Усе це разом зі стилем мовлення може суттєво впливати на точність розпізнавання.

Однією з головних проблем у розпізнаванні мови є її варіативність, яка має два основних аспекти: акустичний і часовий. Акустична варіативність включає відмінності у вимові, акценти, висоту голосу та гучність, тоді як часова варіативність стосується змін швидкості мовлення. Ці два аспекти взаємопов’язані, оскільки зміна швидкості мовлення впливає на акустичні параметри.

Для обробки часової варіативності раніше застосовувалася лінійна деформація, проте вона виявилася недостатньо ефективною через те, що швидкість мовлення може змінюватися нерівномірно. Тому був розроблений алгоритм динамічної деформації часу, який нині є одним із найефективніших інструментів у системах розпізнавання мови [5].

Отже, розробка високоефективних систем розпізнавання мови вимагає врахування численних чинників, серед яких варіативність мовлення, акустичні умови та граматичні обмеження. Попри значний прогрес у цій галузі, залишаються виклики, подолання яких потребує подальшого вдосконалення алгоритмів і технологій.

1.3 Нейронні мережі

Нейронні мережі становлять фундамент для розвитку систем штучного інтелекту та машинного навчання в сучасному світі. Їхня структура та принципи роботи імітують біологічні нервові системи, що дозволяє ефективно аналізувати масиви даних та ідентифікувати складні патерни. Однією з ключових характеристик нейронних мереж є здатність до самонавчання через ітеративну корекцію ваг на основі помилок, що виявляються під час розв’язання задач.

Навчання нейронної мережі здійснюється за допомогою алгоритму зворотного розповсюдження помилок, який дозволяє адаптувати ваги зв’язків між нейронами з метою мінімізації розбіжності між актуальним та очікуваним результатами мережі. Процес навчання включає застосування різних функцій втрат, які визначають точність виходу мережі для кожного вхідного набору даних. Серед найбільш вживаних функцій втрат виділяють середньоквадратичну помилку (MSE) для регресійних задач та крос-ентропію для задач класифікації.

Залежно від специфіки задачі, нейронні мережі можуть бути реалізовані у різноманітних конфігураціях. Згорткові нейронні мережі (CNN) оптимізовані для аналізу зображень та відео, в той час як рекурентні нейронні мережі (RNN) та їх розвинені версії, такі як LSTM (Long Short-Term Memory) та GRU (Gated Recurrent Unit), застосовуються для обробки послідовних даних, включно з текстом та аудіо [6].

Трансформерні моделі, такі як BERT та GPT, стали проривом у задачах обробки природної мови, оскільки дозволяють ефективно моделювати довгострокові залежності в текстах.

Сучасні дослідження в області нейронних мереж акцентують на використанні механізмів уваги, які дозволяють системі концентруватися на ключових елементах вхідних даних. Це особливо важливо для комплексних задач, де дані мають великий обсяг, а значущі характеристики розподілені по всьому датасету.

Процес обчислень у нейронних мережах є фундаментальним етапом їх роботи, і його оптимізація визначає загальну продуктивність системи. Активація оброблювальних одиниць, або нейронів, є центральним елементом цього процесу. Залежно від архітектури, активація може бути синхронною, де всі блоки активуються одночасно, або асинхронною, коли нейрони оновлюються послідовно або випадковим чином. У багатошарових мережах процес активації має чітко визначений напрямок і називається поширенням вперед, оскільки сигнал рухається від вхідного шару до вихідного, забезпечуючи обробку даних [7].

У двоетапному процесі оновлення кожної оброблювальної одиниці спочатку обчислюється чистий вхід, або внутрішня активація, яка є сумою сигналів, отриманих через зв’язки з іншими нейронами, помножених на відповідні вагові коефіцієнти. Після цього застосовується функція активації для визначення вихідного сигналу.

Функції активації відіграють ключову роль у здатності нейронних мереж моделювати складні залежності. Лінійні функції є найпростішими, але вони обмежують здатність мережі до нелінійного розділення даних. Порогові функції дозволяють реалізувати дискретні переходи, тоді як сигмоїдальні функції, завдяки своїй безперервності та диференційованості, забезпечують ефективність багатошарових мереж і підтримують алгоритм зворотного поширення помилки. Сучасні нейронні мережі, однак, використовують і більш просунуті функції активації, такі як ReLU (Rectified Linear Unit), яка значно прискорює навчання глибоких мереж за рахунок усунення проблеми затухання градієнта [8].

Процес навчання нейронної мережі базується на ітеративному оновленні вагових коефіцієнтів, що визначають силу з’єднань між нейронами. Для цього найчастіше використовується алгоритм градієнтного спуску, який дозволяє мінімізувати функцію втрат шляхом поступового коригування ваг у напрямку зменшення помилки. Для складних багатошарових мереж застосовується його модифікована версія – алгоритм

зворотного поширення помилки, який розраховує похідні для всіх вагових коефіцієнтів через весь шлях поширення сигналу.

Швидкість навчання є важливим параметром, який контролює величину змін ваг на кожному етапі. Якщо швидкість надто висока, це може призвести до нестабільності та навіть до того, що мережа пропускатиме глобальний мінімум. Занадто низька швидкість робить процес навчання надто повільним і може призвести до застрягання у локальних мінімумах. Для оптимізації цього процесу часто застосовуються адаптивні алгоритми, такі як Adam, Adagrad або RMSprop, які автоматично змінюють швидкість навчання залежно від характеристик даних.

Окремої уваги заслуговує використання регуляризації для запобігання перенавчанню. Метод Dropout, наприклад, тимчасово вимикає випадкові нейрони під час навчання, що змушує мережу розподіляти знання між всіма вагами та знижує залежність від окремих нейронів. L2-регуляризація обмежує розмір вагових коефіцієнтів, щоб запобігти їхньому надмірному зростанню.

Навчання мережі також може включати зміну її структури, наприклад, додавання або видалення шарів чи нейронів. Це дозволяє моделі адаптуватися до складності задачі, але такі зміни зазвичай виконуються вручну або з використанням автоматизованих підходів, таких як пошук архітектури нейронної мережі (NAS).

Ефективність роботи нейронних мереж залежить від багатьох факторів, серед яких вибір функції активації, алгоритму оптимізації, стратегії регуляризації та параметрів навчання. Їх здатність адаптуватися до складних даних і завдань робить нейронні мережі надзвичайно потужним інструментом у задачах обробки природної мови, хоча їх навчання потребує значних обчислювальних ресурсів та ретельного налаштування.

2 ТЕХНОЛОГІЇ ГОЛОСОВИХ СИСТЕМ АКТИВАЦІЇ

2.1 Рекурентні нейронні мережі

Рекурентні нейронні мережі (РНМ, recurrent neural networks, RNN) являють собою унікальний клас штучних нейронних мереж, які здійснюють моделювання послідовних залежностей у датасетах за допомогою застосування орієнтованого у часі графа зв'язків між елементами. Ця архітектура надає мережі внутрішній стан, що виконує роль пам'яті, дозволяючи враховувати інформацію з попередніх входів при обробці нових сигналів. Така особливість робить RNN особливо придатними для аналізу послідовностей даних будь-якої довжини, наприклад, у задачах розпізнавання мовлення або рукописного тексту, що є складним для реалізації з використанням нейронних мереж прямого розповсюдження [9].

Рекурентні нейронні мережі унікальні тим, що здатні обробляти послідовності даних, такі як тексти, часові ряди та мовлення. На відміну від згорткових нейронних мереж, які ідеально підходять для обробки зображень, RNN мають здатність «пам'ятати» попередню інформацію і використовувати її для обробки поточних даних.

Основною проблемою при роботі з RNN є згасання та вибух градієнта. Ці явища виникають у процесі навчання мережі, коли ваги моделі коригуються за допомогою зворотного поширення похибки. Загасання градієнта відбувається, коли ваги коригуються настільки слабко, що навчання стає неефективним. З іншого боку, вибух градієнта відбувається, коли ці коригування дуже великі, що також веде до нестабільності навчання. Обидві проблеми можуть серйозно ускладнити навчання RNN.

На щастя, на допомогу приходять LSTM і GRU – два різновиди RNN, розроблені спеціально для боротьби з цими проблемами.

LSTM. Рекурентні мережі, здатні обробляти послідовності даних та зберігати інформацію про попередні стани, були представлені у 1980-х роках.

Однак вони зіткнулися з серйозними проблемами при навчанні через загасання та вибух градієнта, що обмежувало їхню здатність зберігати інформацію на тривалі періоди часу.

LSTM була запропонована в 1997 Йоханнесом Шмідхубером і Зеппом Хохрайтером. Ця концепція була революційною: LSTM призначалася для збереження інформації на тривалі періоди та її ефективного вилучення, коли це необхідно. Основною ідеєю було впровадження «воріт» у структуру нейронної мережі, які контролюють потік інформації, дозволяючи уникнути проблеми згасання градієнта.

Основною інновацією в LSTM є три типи воріт: вхідні (input gates), забування (forget gates) та вихідні (output gates). Ці ворота дозволяють моделі вирішувати, яку інформацію зберегти, яку оновити та яку передати далі. Така структура дозволяє LSTM обробляти дані із довгостроковими залежностями, що було недосяжно для традиційних RNN.

LSTM почала застосовуватися у різних галузях, де необхідно було враховувати довгострокові залежності даних. Одним із перших значних застосувань LSTM стало розпізнавання рукописного тексту. У наступні роки LSTM знайшла широке застосування в обробці природної мови, прогнозуванні часових рядів, генерації музики та багатьох інших областях.

Згодом LSTM піддавалася різним модифікаціям та покращенням. Наприклад, були запропоновані варіації, що спрощують оригінальну модель, такі як GRU, яка має меншу кількість параметрів, але при цьому зберігає багато ключових особливостей LSTM.

Gated Recurrent Unit (GRU). GRU, або Gated Recurrent Unit, є одним із видів рекурентних нейронних мереж (RNN), аналогічним LSTM (Long Short-Term Memory). Розроблена у 2014 році, ця архітектура була призначена для спрощення та прискорення навчання в порівнянні з LSTM, зберігаючи при цьому більшу частину її ефективності, особливо у завданнях обробки послідовностей даних.

На відміну від LSTM, у GRU немає окремого довгострокового стану комірки. Стан комірки в GRU є комбінацією минулого стану та нових

вхідних даних, що модулюються через оновлювальні ворота та ворота забування. Цей стан оновлюється на кожному кроці та переносить інформацію всією мережею.

Ворота оновлення GRU контролюють, наскільки нова інформація впливатиме на стан осередку. Вони визначають, яку частину минулого стану слід зберегти та яку частину поточного введення слід врахувати.

Ворота скидання допомагають моделі вирішити, яка з минулого інформація необхідна для поточного виходу. Якщо ворота скидання близькі до нуля, це означає, що інформація не буде враховуватися, що дозволяє моделі відкинути нерелевантні дані.

У GRU немає окремого осередку пам’яті, як у LSTM. Вона оновлює прихований стан безпосередньо. Ворота оновлення допомагають моделі вирішити, яка частина попереднього прихованого стану має бути збережена. Це робиться шляхом комбінування минулого стану та нової інформації.

Ворота скидання визначають, яка частина минулого стану важлива для поточного введення, а ворота оновлення використовують цю інформацію для формування прихованого нового стану. Це дозволяє GRU гнучкіше регулювати потік інформації.

GRU виконує серію математичних операцій для оновлення свого стану, включаючи використання сигмоїдної функції для воріт та гіперболічного тангенса для створення нових кандидатів на стан. Ці операції допомагають GRU динамічно регулювати потік інформації. Схему роботи GRU представлено на рисунку 2.1.

Якщо важлива швидкість навчання та доступні обчислювальні ресурси обмежені, GRU може бути кращим через свою простішу структуру та меншу кількість параметрів порівняно з LSTM.

Для простіших або менших наборів даних GRU може бути достатньою, оскільки вона може ефективно навчатися на таких даних, не втрачаючи в продуктивності.

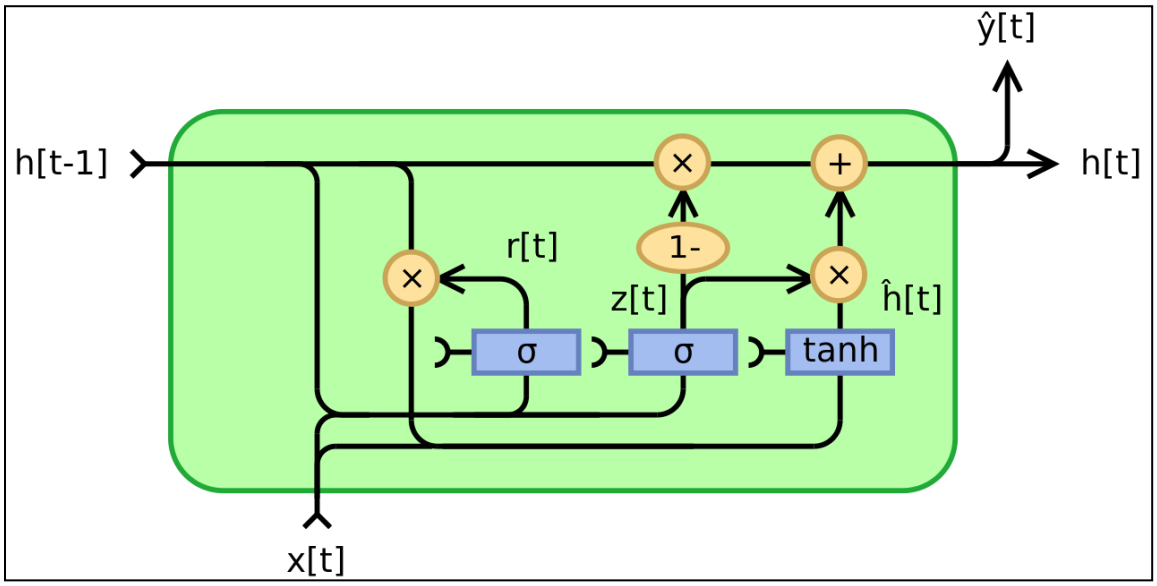


Рисунок 2.1 – Схема роботи GRU

Якщо завдання потребує більш детального управління інформацією та довгострокової пам’яті (наприклад, у складних завданнях обробки природної мови з довгими залежностями), LSTM може бути кращим вибором завдяки додатковій складності та контролю над інформацією [10].

2.2 Мел-частотні кепстральні коефіцієнти

Коли мова йде про порівняння мовних послідовностей, то що саме потрібно порівнювати? Безпосередньо звукові сигнали у часовій області – довго та не дуже ефективно. Спектрограми – вже швидше, але не набагато ефективніше. Пошуки найбільш раціонального рішення приводять до MFCC або мел-частотних кепстральних коефіцієнтів, які часто використовуються як характеристики мовних сигналів.

Мел-частотні кепстральні коефіцієнти (Mel-frequency cepstral coefficients, MFCCs) є числовими характеристики, які отримуються шляхом аналізу мел-частотного кепстру. Ці коефіцієнти широко використовуються для визначення акустичних характеристик і є основним інструментом у задачах розпізнавання мовлення та обробки звуку.

У обробці звуку кепстр mel-frequency (MFC) – це представлення короткочасного спектра потужності звуку на основі лінійного косинусного перетворення логарифмічного спектра потужності на нелінійній шкалі мел-частоти.

Мел визначається як одиниця вимірювання висоти звуку, що базується на його сприйнятті слуховою системою людини. Відомо, що частотна характеристика слуху не є лінійною, а амплітуда звуку не завжди точно відображає його гучність. З цієї причини були розроблені емпіричні одиниці для вимірювання гучності, такі як фон [3].

Так само, висота звуку, яку сприймає людина, не має прямої залежності від його частоти (див. рис. 2.2) [11].

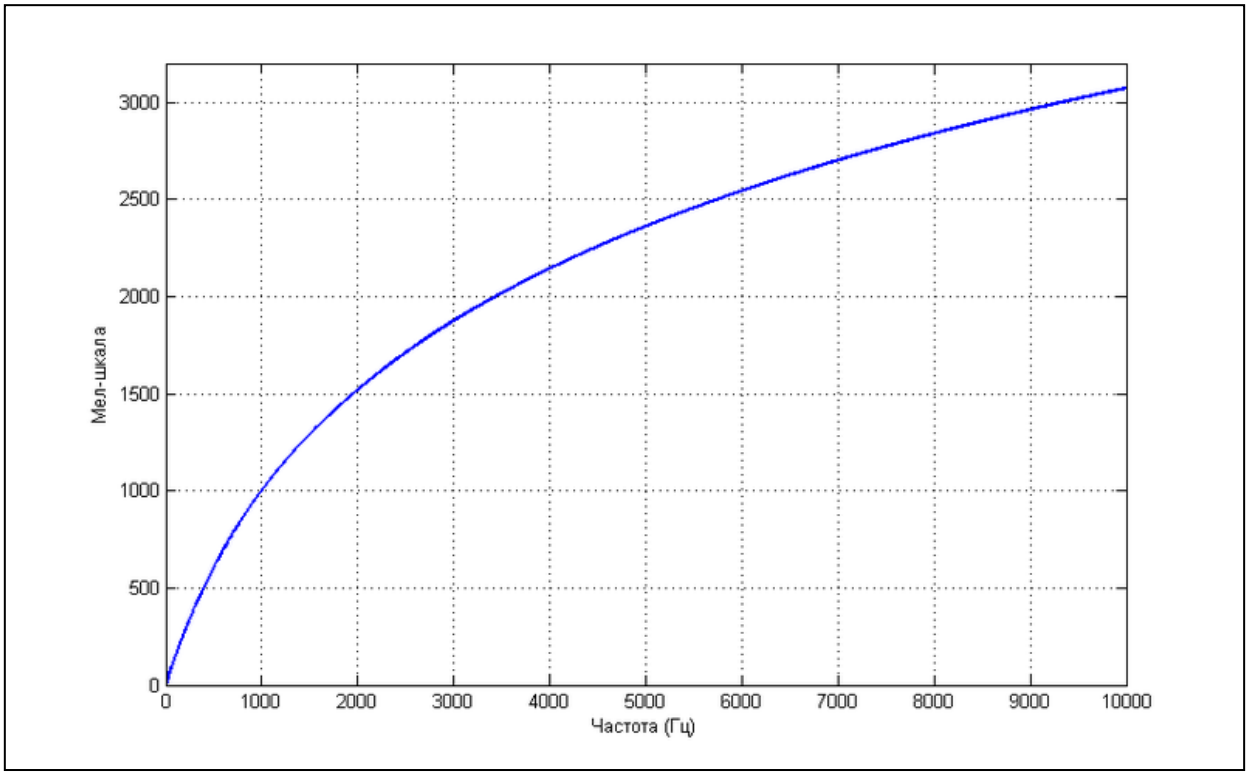


Рисунок 2.2 – Залежність висоти звуку, що сприймається людським вухом, від його частоти

Схожі одиниці виміру широко використовуються для ідентифікації, оскільки наближають обробку до людського сприйняття, що забезпечує вищу ефективність порівняно з іншими системами розпізнавання мови.

Mel-частотні кепстральні коефіцієнти (MFCC) – це коефіцієнти, які утворюють MFC. Вони виводяться з особливого типу кепстрального аналізу аудіозапису. Відмінність між звичайним кепстром та mel-частотним кепстром полягає у тому, що у MFC частотні смуги розподілені рівномірно на mel-шкалі, що дозволяє точніше імітувати відгук слухової системи людини, ніж лінійний розподіл смуг частот, який застосовується у звичайному спектральному аналізі. Таке перетворення частоти може забезпечити більш точне представлення звуків, наприклад, у процесах стиснення аудіо, що може знизити потреби у пропускній здатності для передачі та вимоги до зберігання аудіосигналів.

MFCC зазвичай отримують таким чином:

– візьміть перетворення Фур’є (віконного уривка) сигналу;
– відобразіть потужності спектра, отриманого вище, на шкалу мел, використовуючи трикутні вікна, що перекриваються, або, альтернативно, вікна косинусного перекриття;
– візьміть значення потужностей на кожній частоті mel;
– візьміть дискретне косинусне перетворення списку логарифмічних ступенів мел, ніби це сигнал.

MFCC є амплітудами результуючого спектру.

Можуть існувати варіації цього процесу, наприклад: відмінності у формі чи відстані вікон, що використовуються для відображення масштабу, або додавання динамічних функцій.

Європейський інститут стандартів телекомунікацій на початку 2000-х визначив стандартизований алгоритм MFCC для використання в мобільних телефонах.MFCC зазвичай використовуються як функції в системах розпізнавання мовлення, таких як системи, які можуть автоматично розпізнавати номери, промовлені в телефон.

MFCC також все частіше знаходять застосування в програмах пошуку музичної інформації, таких як класифікація жанрів, вимірювання подібності звуку тощо.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ГОЛОСОВОЇ СИСТЕМИ АКТИВАЦІЇ

3.1 Вибір засобів реалізації

Оскільки основною проблемою для виконання нашого завдання є обмеження в обчислюваних ресурсах, а розпізнавати будемо одну ключову фразу, кращим вибором архітектури нейронної мережі буде GRU (Gated Recurrent Units). Відомо, що застосування даного методу для розв’язання задач моделювання музичних та мовних сигналів демонструє ефективність, аналогічну до тієї, що забезпечується за допомогою LSTM, але при цьому відзначається значно вища продуктивність [10].

Програму було вирішено реалізовувати мовою програмування Python, оскільки її перевагою є здатність вирішувати широке коло завдань завдяки своїй розширюваності та гнучкості.

Основною бібліотекою для роботи з нейронними мережами буде *PyTorch* [12]. Основною структурою даних у *PyTorch* є n-вимірний тензор, що забезпечує зручне представлення багатовимірних даних. Цей тензор підтримується набором базових математичних і статистичних функцій, а також підпрограмами лінійної алгебри, реалізованими на Сі, що гарантує високу продуктивність.

Бібліотека включає кілька пакетів, кожен з яких відповідає за специфічний набір функціональних можливостей. Наприклад, пакет *optim* реалізує основні оптимізаційні алгоритми, такі як метод стохастичного градієнтного спуску. Пакет *nn* є ядром роботи з нейронними мережами. У ньому містяться модулі, які визначають структуру мережі та включають функції активації, такі як *Tanh* і *Sigmoid*, а також базові компоненти, як *Linear, Reshape* і *Max*. Контейнерні модулі, такі як *Sequential*, *Parallel* і *Concat*, дозволяють створювати складні архітектури, включаючи паралельні та послідовні структури. Клас *torch.nn.GRU* є основним у роботі з GRU.

PyTorch також підтримує розширення можливостей через пакет *nngraph*, який дозволяє створювати складні архітектури нейронних мереж. Окрім цього, бібліотека має низку інших корисних модулів, включаючи засоби для інтеграції зі сторонніми бібліотеками, об’єктно-орієнтованого програмування, попередньої перевірки аргументів, а також для роботи з хешуванням, раціональними числами та потоками.

Для роботи з аудіофайлами, включаючи завантаження, зміну частоти дискретизації та розрахунок мел-частотних кепстральних коефіцієнтів (MFCC), будемо використовувати бібліотеку *torchaudio* [13]. Вона забезпечує потужні інструменти для роботи з аудіо, дозволяючи легко інтегрувати обробку аудіоданих у процес створення та навчання нейронних мереж.

Для зручної та наочної візуалізації результатів навчання нейронної мережі будемо користуватися *TensorBoard* [14], а для створення зручних прогрес-барів, що показували б поточний рівень епохи тренування – бібліотеку *tqdm* [15].

3.2 Збір датасету

Якісні дані є ключовим елементом у систем голосової активації, оскільки саме якість і різноманітність даних безпосередньо впливають на ефективність і точність роботи моделі. Датасет забезпечує основу для навчання нейронної мережі, дозволяючи їй адаптуватися до різних сценаріїв використання та зовнішніх умов.

Насамперед, якісний датасет має містити велику кількість записів ключової фрази, яка розпізнається системою, а також негативні приклади, щоб модель могла відрізняти цільові сигнали від шуму та сторонніх команд. Включення різних голосів, акцентів, швидкості вимови та навколишніх звуків (фоновий шум, перешкоди) підвищує здатність моделі до узагальнення та робить її стійкою до реальних умов експлуатації. Наприклад, модель,

навчена лише на аудіофайлах без шуму, може показувати низьку ефективність у складних акустичних середовищах.

Створення добре збалансованого датасету також важливе для запобігання зміщення моделі. Якщо у даних буде значна перевага одного типу записів (наприклад, фраз від одного мовця або лише за певних умов), це може призвести до обмеженості моделі в реальних сценаріях. Тому важливо включати різноманітні дані, що представляють якомога ширший спектр ситуацій.

Важливість датасету полягає і в його впливі на тестування моделі. Розподіл даних на навчальні, валідаційні та тестові набори дозволяє оцінювати точність, узагальненість і стабільність роботи системи, забезпечуючи її готовність до реального використання.

Отже, правильно підготовлений датасет із врахуванням різноманітності та якості даних є визначальним фактором для успіху таких систем. Він забезпечує навчання моделі, адаптацію до різних умов та високу точність роботи в реальних сценаріях.

Основу датасету для тренувань нейронної мережі, що використовується у даній роботі, взято з відкритих джерел – це датасет Ukrainian Open Speech To Text Dataset 4.2 [16].

Даний набір даних складається з приблизно 1200 годин розмічених аудіоданих українською мовою у вигляді коротких аудіофайлів довжиною зазвичай від 1 до 10 секунд та загальним обсягом близько 190 ГБ. Метою цього репозиторію є збір інформації та наборів даних для розпізнавання мовлення українською. Даний набір даних вже містить у своєму складі широко відомий датасет Mozila Common Voice uk [17].

На етапі планування даної роботи вважалося, що кількості позитивних прикладів, відібраних з відкритих україномовних Speech-to-Text датасетів, вистачить для тренування моделі. Було написано скрипт для пошуку ключової фрази серед розмічених файлів датасету.

Код даного скрипту представлено у Додатку Б.

За результатами сканування понад 1000 годин мовлення, що містяться у цьому наборі даних, виявилося лише 26 збігів з ключовою фразою, що ясно дало зрозуміти, що позитивні приклади треба збирати іншими методами.

Оскільки вказаний вище датасет містить велику кількість україномовних даних, не пов’язаних із ключовою фразою, в основному він буде використовуватись як набір негативних зразків для тренування, моделюючи випадкові фрази та різні види фонового шуму.

Більшу частину позитивних прикладів було записано та приведено до потрібного формату файлів вручну. У роботі використовуються записи голосу суб’єктів різного віку та статі, вимовлені з різної відстані від мікрофона та з різною тривалістю пауз між словами.

Частина записів зроблена з фоновим шумом, що повинно покращити якість розпізнавання ключової фрази незалежно від навколишніх обставин. Під час збирання даних у фоновому режимі відтворюються різні шуми (типові шуми домашнього середовища, такі як музика та телевізор) із різними рівнями звукового тиску.

Щоб зменшити кількість хибних спрацювань на поодинокі слова, що разом становлять ключову фразу, позитивні приклади були розбиті на окремі слова і додані до негативних.

Оскільки кількість негативних прикладів значно перевищує кількість позитивних, було прийнято рішення збільшити вибірку, пришвидшивши та сповільнивши на 10% оригінальні позитивні приклади, тим самим адаптуючи модель до різної швидкості вимови фрази.

Набір даних розбито на тренувальний та валідаційний датасети приблизно у пропорції 80:20.

3.3 Розробка програми

Відкриємо середовище розробки та імпортуємо потрібні бібліотеки (див. рис 3.1).

```
import argparse
import torch
import torchaudio
import json
import os
import time
import random
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm
▶ Launch TensorBoard Session
from torch.utils.tensorboard import SummaryWriter
```

Рисунок 3.1 – Імпорт бібліотек

На рисунку 3.2 представлено конструктор класу *AudioDataset*, де:

– *sample_rate* – частота дискретизації аудіофайлу;
– *n_mfcc* – кількість коефіцієнтів MFCC;
– *melkwargs* – словник з параметрами для створення мел-спектрограми;
– *n_fft* – кількість точок для швидкого перетворення Фур'є;
– *win_length* – довжина вікна;
– *hop_lengt* – довжина кроку;
– *n_mels* – кількість мел-фільтрів.

```
class AudioDataset(Dataset):
    def __init__(
            self,
            paths,
            positive_index,
            audio_path,
            sample_rate=16000,
            n_mfcc=13,
            n_fft=400,
            win_length=400,
            hop_length=160,
            n_mels=20
        ):
```

Рисунок 3.2 – Клас AudioDataset

Завантажуємо дані з файлу, створюємо списки усіх прикладів та позитивних прикладів, а також списку з усіма шляхами до аудіофайлів (див. рис. 3.3).

```
super().__init__()

all_examples = []

for path in paths:
    with open(path, "r") as f:
        data = json.load(f)
    all_examples.extend(data)

self.positive_examples = [example["keyword_id"] == positive_index for example in all_examples]

self.all_paths = [
    example["audio_path"]
    for example in all_examples
]
```

Рисунок 3.3 – Завантаження даних

Створюємо об’єкт *mfcc_transform* класу MFCC (Mel-frequency cepstral coefficients) для розрахунку мел-частотних кепстральних коефіцієнтів (рис. 3.4).

```
self.mfcc_transform = torchaudio.transforms.MFCC(
    sample_rate=sample_rate,
    n_mfcc=n_mfcc,
    melkwargs={"n_fft": n_fft, "win_length": win_length, "hop_length": hop_length, "n_mels": n_mels}
)

self.sample_rate = sample_rate
```

Рисунок 3.4 – MFCC

Перетворюємо аудіофайл у MFCC, об’єднаємо списки *mfccs* та правильних відповідей у кортеж, розраховуємо максимальну довжину MFCC, після чого доповнюємо їх нулями до максимального розміру та об’єднуємо в один тензор (рис. 3.5).

```
def collate_fn(batch):
    mfccs, labels = zip(*batch)
    # mfccs have variable length, so we need to pad them
    # Shape of single mfcc is (1, n_mfccs, time)

    mfcc_sizes = torch.tensor([mfcc.shape[2] for mfcc in mfccs])

    max_len = mfcc_sizes.max().item()
    padded_mfccs = [torch.nn.functional.pad(mfcc, (0, max_len - mfcc.shape[2])) for mfcc in mfccs]
    padded_mfccs = torch.cat(padded_mfccs, dim=0)

    # Permute from (batch_size, n_mfccs, time) to (time, batch_size, n_mfccs)
    padded_mfccs = padded_mfccs.permute(2, 0, 1)

    labels = torch.tensor(labels)
    return padded_mfccs, labels, mfcc_sizes
```

Рисунок 3.5 – Розрахунок MFCC

Створюємо клас нашої моделі *WakeUpWordModel*, що наслідується від класу *torch.nn.Module* – базового класу для всіх модулей нейронної мережі (рис. 3.6). Задаємо наступні параметри:

– *n_mfcc* – кількість коефіцієнтів MFCC;
– *hidden_size* – розмір прихованого шару GRU;
– *num_gru_layers* – кількість шарів GRU;
– *dropout* – ймовірність дропауту (метод регуляризації шляхом занулення значення нейрона під час тренування);
– *torch.nn.GRU* – клас, що реалізує шар GRU;
– *GRU* – Gated Recurrent Unit.

Обробляємо *mfccs* через нейронну мережу, знаходимо останнє виведення для кожної послідовності та використовуємо для нього дропаут з певною імовірністю для запобігання перенавчанню.

```
class WakeUpWordModel(torch.nn.Module):
    def __init__(
        self,
        n_mfcc=13,
        hidden_size=64,
        num_gru_layers=1,
        dropout=0.2
    ):
        super().__init__()

        self.gru = torch.nn.GRU(
            input_size=n_mfcc,
            hidden_size=hidden_size,
            num_layers=num_gru_layers
        )

        self.dropout = dropout

        self.proj = torch.nn.Linear(hidden_size, 1)
```

Рисунок 3.6 – Конструктор класу *WakeUpWordModel*

Використовуємо повнозв’язний шар до останнього виведення, перетворюючи його в одне число (-∞;∞) (див. рис. 3.7).

```python
def forward(self, mfccs, mfcc_sizes):
    # mfccs: (time, batch_size, n_mfccs)
    # mfcc_sizes: (batch_size,)
    output = self.gru(mfccs)[0]

    # output: (time, batch_size, hidden_size)
    # We want to take the last output for each sequence
    last_output = output[mfcc_sizes - 1, torch.arange(mfcc_sizes.size(0))]

    last_output = torch.nn.functional.dropout(last_output, p=self.dropout, training=self.training)

    return self.proj(last_output).squeeze()
```

Рисунок 3.7 – Метод *forward* (прогін MFCC через GRU)

Створюємо парсер аргументів, що містить шляхи до датасетів тренування та оцінки, індекс потрібного ключового слова, шляхи до аудіофайлів та логів роботи, ймовірність дропауту, розмір батчу (кількість аудіофайлів для одночасного прогону), кількість епох тренування нейронної мережі, необхідність перемішування файлів датасету, кількість потоків для завантаження файлів у фоновому режимі та швидкість навчання мережі за замовченням (рис. 3.8).

```python
parser.add_argument(
    "--dropout", type=float, default=0.2,
    help="Dropout probability"
)

parser.add_argument(
    "--batch_size", type=int, default=256,
    help="Batch size"
)
parser.add_argument(
    "--num_epochs", type=int, default=10,
    help="Number of epochs"
)
parser.add_argument(
    "--shuffle", type=bool, default=True,
    help="Shuffle dataset"
)
parser.add_argument(
    "--num_workers", type=int, default=2,
    help="Number of workers for DataLoader"
)
parser.add_argument(
    "--lr", type=float, default=1e-3,
    help="Learning rate"
)
```

Рисунок 3.8 – Парсер аргументів

Створюємо *Dataset* та *DataLoader* для тренування та оцінки, створюємо модель *WakeUpWordModel* та переносимо її на пристрій, розраховуємо відношення позитивних прикладів до їх загальної кількості для корекції функції похибок. Створюємо функцію похибок та оптимізатор (рис. 3.9).

```
eval_dataloader = DataLoader(
    eval_dataset,
    batch_size=args.batch_size,
    shuffle=args.shuffle,
    collate_fn=collate_fn,
    num_workers=args.num_workers
)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

writer = SummaryWriter(
    log_dir=args.log_dir
)

positive_to_total_ratio = train_dataset.get_positive_to_total_ratio()
negative_to_total_ratio = 1 - positive_to_total_ratio

model = WakeUpWordModel(
    n_mfcc=args.num_mfcc,
    hidden_size=args.hidden_size,
    num_gru_layers=args.num_gru_layers,
    dropout=args.dropout
).to(device)
loss_fn = torch.nn.BCEWithLogitsLoss(reduction="none")
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
```

Рисунок 3.9 – Створення *Dataset, DataLoader* та моделі *WakeUpWordModel*

Torch.optim.Adam – клас, що реалізує алгоритм оптимізації Adam.

Adam – адаптивний метод оптимізації, що комбінує у собі методи адаптивного градієнта і моментів [18].

Для кожної епохи по тренувальному датасету переносимо дані на пристрій, проганяємо дані через модель, підраховуємо функцію похибок і усереднюємо втрати з урахуванням відношення кількості позитивних і негативних прикладів. Якщо негативних прикладів більше, то збільшуємо втрати з позитивних прикладів. Зануляємо значення градієнтів і використовуємо метод зворотного поширення помилки (backpropagation) для розрахунку градієнтів, після чого оновлюємо ваги моделі та записуємо логи роботи у TensorBoard (див. рис. 3.10).

```
global_step = 0

for epoch in range(args.num_epochs):
    for mfccs, labels, mfcc_sizes in tqdm(train_dataloader, desc=f"Train epoch {epoch}"):
        mfccs = mfccs.to(device)
        labels = labels.to(device).float()
        mfcc_sizes = mfcc_sizes.to(device)

        output = model(mfccs, mfcc_sizes)

        loss = loss_fn(output, labels)
        loss = (loss * (labels * negative_to_total_ratio + (1 - labels) * positive_to_total_ratio)).mean()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        writer.add_scalar("train/loss", loss.item(), global_step)

        global_step += 1
```

Рисунок 3.10 – Тренування моделі

У режимі валідації проганяємо модель по датасету для оцінки роботи, підраховуючи кількість правильних відповідей та суму похибок для визначення середньої похибки, не розраховуючи градієнти. Записуємо логи валідації та переводимо модель назад у режим тренування (рис. 3.11).

```
# Evaluation
model.eval()
with torch.no_grad():
    total = 0
    correct = 0
    total_loss = 0
    for mfccs, labels, mfcc_sizes in tqdm(eval_dataloader, desc=f"Eval epoch {epoch}"):
        mfccs = mfccs.to(device)
        labels = labels.to(device)
        mfcc_sizes = mfcc_sizes.to(device)

        output = model(mfccs, mfcc_sizes)

        loss = loss_fn(output, labels.float())
        loss = (loss * (labels.float() * negative_to_total_ratio + (1 - labels.float()) * positive_to_total
        total_loss += loss.item()

        predictions = output > 0
        correct += (predictions == labels).sum().item()
        total += labels.size(0)

    accuracy = correct / total
    writer.add_scalar("eval/accuracy", accuracy, epoch)

    avg_loss = total_loss / len(eval_dataloader)
    writer.add_scalar("eval/loss", avg_loss, epoch)
model.train()
```

Рисунок 3.11 – Оцінка роботи моделі

Зберігаємо стан моделі (словник з параметрами моделі та оптимізатора) у чекпоінт з номером епохи (див. рис. 3.12).

```
# Save model and optimizer state
torch.save({
    "model": model.state_dict(),
    "optimizer": optimizer.state_dict(),
    "epoch": epoch,
    "global_step": global_step,
    "params": params_dict
}, os.path.join(args.log_dir, f"checkpoint_{epoch}.pth"))
```

Рисунок 3.12 – Збереження стану моделі

Підбираємо налаштування розміру батчу та кількість потоків для завантаження файлів у фоновому режимі відповідно до наявного технічного обладнання і переходимо до тренування моделі (рис. 3.13).

Повний код проєкту представлено у Додатку А.



Рисунок 3.13 – Тренування моделі

3.4 Тестування й оцінка роботи моделі

Після завершення тренування десяти епох на обраному датасеті обираємо потрібний файл логів та переходимо у TensorBoard для більш зручного перегляду результатів тренування моделі (рис. 3.14 – 3.17).

Як видно з результатів (рис. 3.14), похибка під час тренування стрімко падає вниз з 0,3 до <0,1 за перші 1000 батчів, що свідчить про правильну роботу моделі.

На рисунку 3.15 зображено цей графік зі згладженням значень та у логарифмічній шкалі по осі похибки для наочності роботи моделі.

На рисунку 3.16 зображено середню точність розпізнавання ключової фрази на валідаційному датасеті (звичайно, що тренувальний та валідаційний датасети не перетинаються).

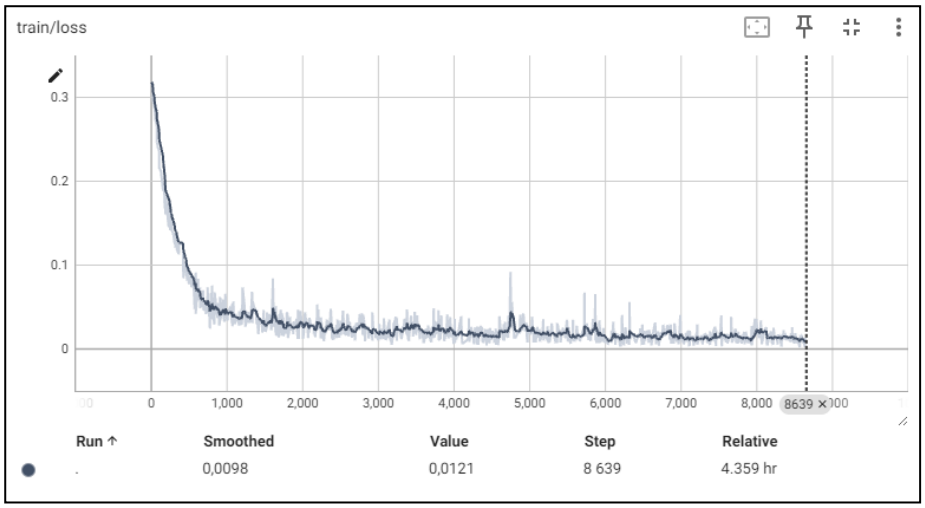


Рисунок 3.14 – Середня похибка для кожного батчу під час тренування

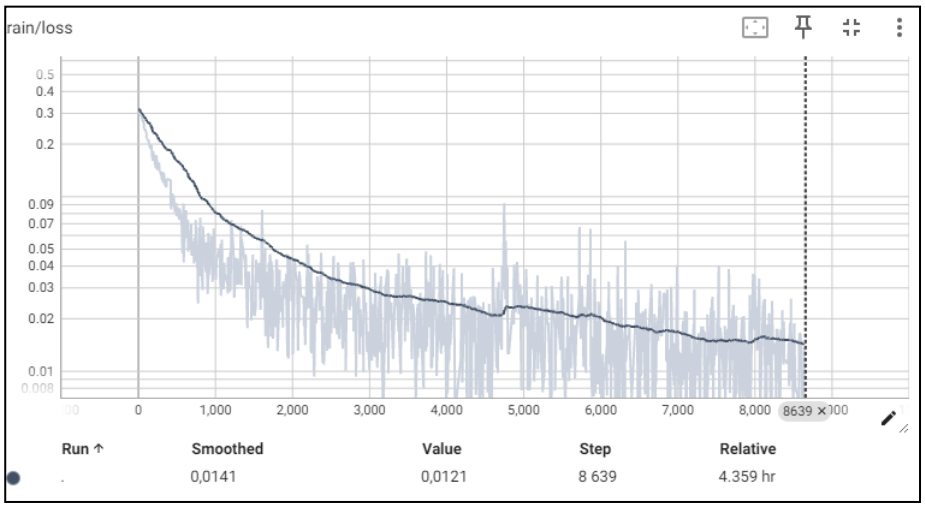


Рисунок 3.15 – Згладжена середня похибка для кожного батчу під час тренування у логарифмічному масштабі по осі Y

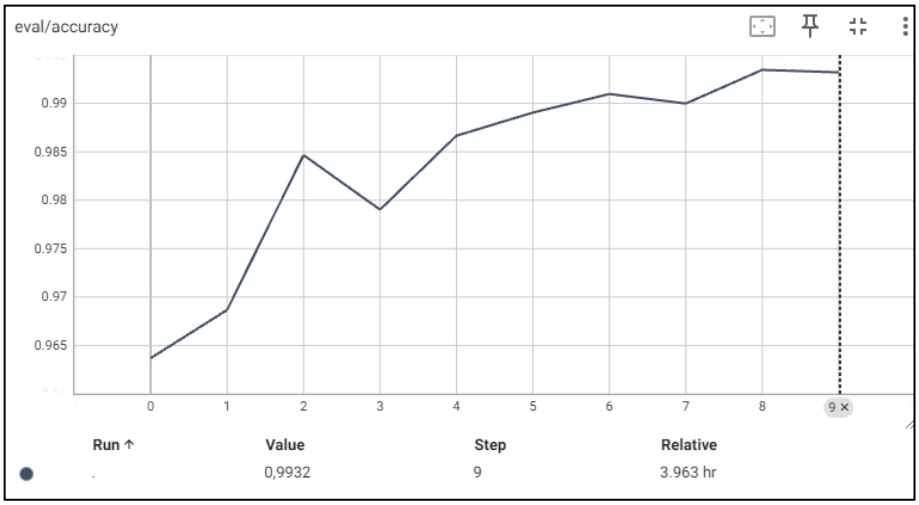


Рисунок 3.16 – Середня точність розпізнавання ключової фрази на валідаційному датасеті для кожної епохи

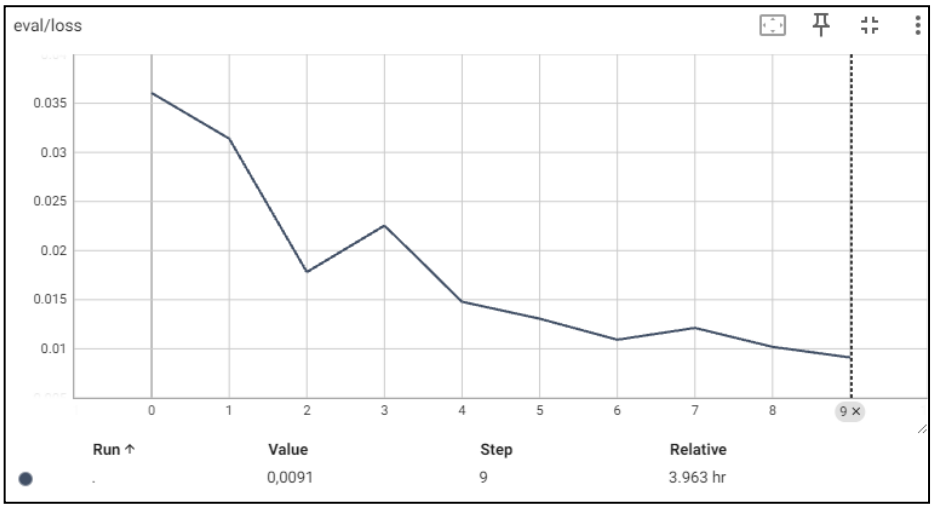


Рисунок 3.17 – Середня похибка на валідаційному датасеті для кожної епохи

З результатів видно, що точність роботи моделі перевищує 96% після першої епохи та 99% у кінці тренування, що є дуже хорошою точністю розпізнавання.Отже, дана модель точно розпізнає ключову фразу і можна переходити до тестування її власним голосом.

Як видно з рисунка 3.18, модель успішно активується ключовою фразою при фоновому шумі.

```
PS S:\Wake_up> python test.py
################################################################################
press Ctrl+C to stop the recording
################################################################################
0 0.17042508721351624
1 0.10093618184328079
2 0.0006932878750376403
3 0.0002124793390976265
4 0.00016499699268024415
5 0.03671959415078163
6 0.9974421262741089
7 0.996484637260437
8 0.00020697173022199422
9 0.00019311315554659814
10 0.00016882442287169397
11 0.0001703604793874547
12 0.00016713145305402577
13 0.00015856180107221007
14 0.0001639503170736134
15 0.0007120835362002254
16 0.7210971713066101
17 0.999225378036499
18 0.0013156565837562084
19 0.0002058234385913238
20 0.00017089526227209717
21 0.00016748401685617864
22 0.00017849769210442 9
23 0.00016175388009287417
24 0.0001758779981173575
25 0.2945602834224701
26 0.9977020621299744
27 0.013083782978355885
28 0.00021810703037772328
29 0.0001712009106995538
30 0.00016618514200672507
31 0.0001645829470362514 3
32 0.00012563678319565952
```

Рисунок 3.18 – Тестування моделі з мікрофона

Напишемо невелику програму, що записуватиме аудіо з мікрофона і буде подавати його невеликими уривками з перекриттям до моделі для оцінки наявності ключової фрази.

Код програми для голосового тестування представлено у Додатку В.

ВИСНОВКИ

У ході виконання даної роботи було успішно розроблено голосову систему активації на основі нейронних мереж. Основною метою роботи було створення системи, здатної точно та ефективно розпізнавати голосові команди, що було досягнуто шляхом використання сучасних методів машинного навчання – рекурентних нейронних мереж (RNN) з вентильним рекурентним вузлом (GRU).

Проведений аналіз сучасних систем розпізнавання мови показав, що традиційні підходи, засновані на правилах і шаблонах, мають суттєві обмеження щодо гнучкості та адаптивності. Використання нейронних мереж дозволяє подолати ці обмеження завдяки здатності моделей до самонавчання та адаптації до нових умов. Це значно підвищує ефективність і точність розпізнавання мови з меншою залежністю від особистих характеристик мовлення людини та фонових шумів.

У процесі роботи було проведено детальний аналіз сучасних технологій і методів у галузі розпізнавання мови та голосового керування; розроблено програмну реалізацію голосової системи активації, що включає інтеграцію рекурентних нейронних мереж та мел-кепстральних коефіцієнтів для аналізу звукових сигналів; проведено тестування розробленої системи, що продемонструвало високу точність і швидкість розпізнавання голосових команд; оцінено практичну застосовність системи та її потенціал для подальшого вдосконалення.

Отримані результати підтверджують, що використання нейронних мереж у розробці голосових систем активації є перспективним напрямком, який може значно покращити взаємодію користувачів з обчислювальною технікою. Розроблена система показала високу ефективність у розпізнаванні голосових команд, що свідчить про її практичну значущість та можливість інтеграції в різноманітні технічні рішення.

Таким чином, дана робота робить важливий внесок у розвиток технологій голосового керування та підтверджує доцільність використання нейронних мереж у цій галузі. Подальші дослідження можуть бути спрямовані на вдосконалення алгоритмів розпізнавання, розширення функціональних можливостей системи та її адаптацію до різних мовних середовищ та умов використання.

ПЕРЕЛІК ПОСИЛАНЬ

1. Бондаренко М. Ф., Работягов А. В., Щепковський С. В. Розпізнавання мови: етапи розвитку, сучасні технології і перспективи їх застосування. *Біоніка інтелекту*. 2010. № 2 (73). С. 164–168.

2. Huang X., Acero A., Hon H.-W. Spoken Language Processing: A Guide to Theory, Algorithm, and System Development. Prentice Hall, 2001. 1032 p.

3. Kamath U., Lui J., Whitaker J. Deep Learning for NLP and Speech Recognition. Springer, 2019. 621 p.

4. Homayoon B. Fundamentals of Speaker Recognition. Springer, 2011. 942 p.

5. Delcroix S., Metze M., Hershey F. New Era for Robust Speech Recognition: Exploiting Deep Learning. Springer, 2017. 436 p.

6. Yu D., Li D. Automac Speech Recognition: A Deep Learning Approach. Springer, 2015. 300 p.

7. Soumya S., Dutta A., Dey N. Audio Processing and Speech Recognition: Concepts, Techniques and Research Overviews. Springer, 2019. 96 p.

8. Jurafsky D., Martin J. Speech and Language Processing. Prentice Hall, 2008. 1032 p.

9. Sak H., Senior A. W., Beaufays F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. *Proc. Interspeech*. 2014. № 1. P. 338–342.

10. Chung J., Gulcehre C., Cho K., Bengio Y. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *ArXiv*. URL: https://doi.org/10.48550/arXiv.1412.3555 (дата звернення: 20.09.2024).

11. Balentine B., Morgan D. P. How to build a speech recognition application. Enterprise Integration Group, 2001. 414 p.

12. Бібліотека Python “Pytorch”. *GitHub*. URL: https://github.com/pytorch/pytorch (дата звернення: 12.10.2024).

13. Бібліотека Python “Torchaudio”. *GitHub*. URL: https://github.com/pytorch/audio (дата звернення: 12.10.2024).

14. Tensorboard. *GitHub*. URL: https://github.com/tensorflow/tensorboard (дата звернення: 12.10.2024).

15. Бібліотека Python “tqdm”. *GitHub*. URL: https://github.com/tqdm/tqdm (дата звернення: 12.10.2024).

16. Ukrainian Open Speech To Text Dataset 4.2. *Kaggle*. URL: https://www.kaggle.com/datasets/aikhmelnytskyy/ukrainian-open-speech-to-text-dataset-42/data (дата звернення: 14.09.2024).

17. Mozilla Common Voice Dataset. *Mozilla Common Voice*. URL: https://commonvoice.mozilla.org/uk/datasets (дата звернення: 14.09.2024).

18. Офіційна документація оптимізатора Adam бібліотеки “Pytorch”. *Pytorch*. URL: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html (дата звернення: 17.10.2024).

ДОДАТОК А

Програмна реалізація голосової системи активації з використанням нейронних мереж

```
import argparse
import torch
import torchaudio
import json
import os
import time
import random
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm
from torch.utils.tensorboard import SummaryWriter

classAudioDataset(Dataset):
    def__init__(
            self,
            paths,
            positive_index,
            audio_path,
            sample_rate=16000,
            n_mfcc=13,
            n_fft=400,
            win_length=400,
            hop_length=160,
            n_mels=20
        ):
        super().__init__()

        all_examples = []

        for path in paths:
            withopen(path, "r") as f:
                data = json.load(f)
            all_examples.extend(data)
        self.positive_examples = [example["keyword_id"] == positive_index
for example in all_examples]

        self.all_paths = [
            example["audio_path"]
            for example in all_examples
```

```
        ]

        self.mfcc_transform = torchaudio.transforms.MFCC(
            sample_rate=sample_rate,
            n_mfcc=n_mfcc,
            melkwargs={"n_fft": n_fft, "win_length": win_length,
"hop_length": hop_length, "n_mels": n_mels}
        )

        self.sample_rate = sample_rate

    def__len__(self):
        returnlen(self.all_paths)

    def__getitem__(self, idx):
        audio_path =self.all_paths[idx]
        label =self.positive_examples[idx]

        audio, sr = torchaudio.load(audio_path)

        if sr !=self.sample_rate:
            audio = torchaudio.transforms.Resample(sr,
self.sample_rate)(audio)

        mfcc =self.mfcc_transform(audio)

        return mfcc, label

    defget_positive_to_total_ratio(self):
        returnsum(self.positive_examples) /len(self.positive_examples)

defcollate_fn(batch):
    mfccs, labels =zip(*batch)

    # mfccs have variable length, so we need to pad them
    # Shape of single mfcc is (1, n_mfccs, time)

    mfcc_sizes = torch.tensor([mfcc.shape[2] for mfcc in mfccs])

    max_len = mfcc_sizes.max().item()
    padded_mfccs = [torch.nn.functional.pad(mfcc, (0, max_len -
mfcc.shape[2])) for mfcc in mfccs]
    padded_mfccs = torch.cat(padded_mfccs, dim=0)
```

```
    # Permute from (batch_size, n_mfccs, time) to (time, batch_size,
n_mfccs)
    padded_mfccs = padded_mfccs.permute(2, 0, 1)

    labels = torch.tensor(labels)
    return padded_mfccs, labels, mfcc_sizes

classWakeUpWordModel(torch.nn.Module):
    def__init__(
        self,
        n_mfcc=13,
        hidden_size=64,
        num_gru_layers=1,
        dropout=0.2
    ):
        super().__init__()

        self.gru = torch.nn.GRU(
            input_size=n_mfcc,
            hidden_size=hidden_size,
            num_layers=num_gru_layers
        )

        self.dropout = dropout

        self.proj = torch.nn.Linear(hidden_size, 1)

    defforward(self, mfccs, mfcc_sizes):
        # mfccs: (time, batch_size, n_mfccs)
        # mfcc_sizes: (batch_size,)
        output =self.gru(mfccs)[0]

        # output: (time, batch_size, hidden_size)
        # We want to take the last output for each sequence
        last_output = output[mfcc_sizes -1,
torch.arange(mfcc_sizes.size(0))]

        last_output = torch.nn.functional.dropout(last_output,
p=self.dropout, training=self.training)

        returnself.proj(last_output).squeeze()

defmain():
    parser = argparse.ArgumentParser()
```

```
parser.add_argument(
    "--train_paths", type=str, nargs="+", default=[
        "slava_dataset_resources/p_train.json",
        "slava_hotword_dataset_resources/p_speed_up.json",
        "slava_hotword_dataset_resources/p_speed_down.json",
        "slava_hotword_dataset_resources/n_train.json"
    ] + ([
        "slava_hotword_dataset_resources/splitted_16k.json",
        "slava_dataset_resources/splitted_16k_speed_down.json",
        "slava_hotword_dataset_resources/splitted_16k_speed_up.json"
    ] *1) + ([
       "slava_hotword_dataset_resources/splitted_pc.json",
       "slava_hotword_dataset_resources/splitted_pc_speed_down.json",
       "slava_hotword_dataset_resources/splitted_pc_speed_up.json"
    ] *10),
    help="Paths to training examples"
)
parser.add_argument(
    "--eval_paths", type=str, nargs="+", default=[
        "slava_hotword_dataset_resources/p_dev.json",
        "slava_hotword_dataset_resources/n_dev.json"
    ],
    help="Paths to eval examples"
)
parser.add_argument(
    "--positive_index", type=int, default=0,
    help="Index of correct keyword"
)
parser.add_argument(
    "--audio_path", type=str, default="slava_dataset",
    help="Path to audio files"
)
parser.add_argument(
    "--log_dir", type=str, default=None,
    help="Directory for TensorBoard logs"
)

parser.add_argument(
    "--sample_rate", type=int, default=16000,
    help="Sample rate of audio files"
)
parser.add_argument(
    "--num_mfcc", type=int, default=13,
    help="Number of MFCC features"
)
```

```
parser.add_argument(
    "--n_fft", type=int, default=400,
    help="Number of FFT points"
)
parser.add_argument(
    "--win_length", type=int, default=400,
    help="Window length"
)
parser.add_argument(
    "--hop_length", type=int, default=160,
    help="Hop length"
)
parser.add_argument(
    "--n_mels", type=int, default=20,
    help="Number of mel bins"
)

parser.add_argument(
    "--hidden_size", type=int, default=20,
    help="Hidden size of GRU"
)
parser.add_argument(
    "--num_gru_layers", type=int, default=1,
    help="Number of GRU layers"
)
parser.add_argument(
    "--dropout", type=float, default=0.2,
    help="Dropout probability"
)

parser.add_argument(
    "--batch_size", type=int, default=256,
    help="Batch size"
)
parser.add_argument(
    "--num_epochs", type=int, default=10,
    help="Number of epochs"
)
parser.add_argument(
    "--shuffle", type=bool, default=True,
    help="Shuffle dataset"
)
parser.add_argument(
    "--num_workers", type=int, default=2,
    help="Number of workers for DataLoader"
```

```
    )
    parser.add_argument(
        "--lr", type=float, default=1e-3,
        help="Learning rate"
    )

    args = parser.parse_args()

    if args.log_dir isNone:
        args.log_dir = os.path.join("logs", time.strftime("%Y-%m-%d_%H-
%M-%S"))

    params_dict =vars(args)

    train_dataset = AudioDataset(
        paths=args.train_paths,
        positive_index=args.positive_index,
        audio_path=args.audio_path,
        sample_rate=args.sample_rate,
        n_mfcc=args.num_mfcc,
        n_fft=args.n_fft,
        win_length=args.win_length,
        hop_length=args.hop_length,
        n_mels=args.n_mels
    )

    train_dataloader = DataLoader(
        train_dataset,
        batch_size=args.batch_size,
        shuffle=args.shuffle,
        collate_fn=collate_fn,
        num_workers=args.num_workers
    )

    eval_dataset = AudioDataset(
        paths=args.eval_paths,
        positive_index=args.positive_index,
        audio_path=args.audio_path,
        sample_rate=args.sample_rate,
        n_mfcc=args.num_mfcc,
        n_fft=args.n_fft,
        win_length=args.win_length,
        hop_length=args.hop_length,
        n_mels=args.n_mels
    )
```

```
    eval_dataloader = DataLoader(
        eval_dataset,
        batch_size=args.batch_size,
        shuffle=args.shuffle,
        collate_fn=collate_fn,
        num_workers=args.num_workers
    )

    device = torch.device("cuda"if torch.cuda.is_available() else"cpu")

    writer = SummaryWriter(
        log_dir=args.log_dir
    )

    positive_to_total_ratio = train_dataset.get_positive_to_total_ratio()
    negative_to_total_ratio =1- positive_to_total_ratio

    model = WakeUpWordModel(
        n_mfcc=args.num_mfcc,
        hidden_size=args.hidden_size,
        num_gru_layers=args.num_gru_layers,
        dropout=args.dropout
    ).to(device)
    loss_fn = torch.nn.BCEWithLogitsLoss(reduction="none")
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)

    global_step =0

    for epoch inrange(args.num_epochs):
        for mfccs, labels, mfcc_sizes in tqdm(train_dataloader,
desc=f"Train epoch {epoch}"):
            mfccs = mfccs.to(device)
            labels = labels.to(device).float()
            mfcc_sizes = mfcc_sizes.to(device)

            output = model(mfccs, mfcc_sizes)

            loss = loss_fn(output, labels)
            loss = (loss * (labels * negative_to_total_ratio + (1-
labels) * positive_to_total_ratio)).mean()

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
            writer.add_scalar("train/loss", loss.item(), global_step)
            global_step +=1

        # Evaluation
        model.eval()
        with torch.no_grad():
            total =0
            correct =0
            total_loss =0
            for mfccs, labels, mfcc_sizes in tqdm(eval_dataloader,
desc=f"Eval epoch {epoch}"):
                mfccs = mfccs.to(device)
                labels = labels.to(device)
                mfcc_sizes = mfcc_sizes.to(device)

                output = model(mfccs, mfcc_sizes)

                loss = loss_fn(output, labels.float())
                loss = (loss * (labels.float() * negative_to_total_ratio
+ (1- labels.float()) * positive_to_total_ratio)).mean()
                total_loss += loss.item()

                predictions = output >0
                correct += (predictions == labels).sum().item()
                total += labels.size(0)

            accuracy = correct / total
            writer.add_scalar("eval/accuracy", accuracy, epoch)

            avg_loss = total_loss /len(eval_dataloader)
            writer.add_scalar("eval/loss", avg_loss, epoch)
        model.train()

        # Save model and optimizer state
        torch.save({
            "model": model.state_dict(),
            "optimizer": optimizer.state_dict(),
            "epoch": epoch,
            "global_step": global_step,
            "params": params_dict
        }, os.path.join(args.log_dir, f"checkpoint_{epoch}.pth"))

if__name__=="__main__":
    main()
```

ДОДАТОК Б

Скрипт для пошуку файлів, що містять ключову фразу, у розміченому Text-to-Speechдатасеті

```
fromglobimportglob
fromtqdmimporttqdm

text_files=glob('**/*.done.data', recursive=True)

search_strings= ["слава україні", "слава україні"]

withopen("output.txt", "w", encoding="utf-8") aso:
    fortext_fileintqdm(text_files):
        root=text_file.split('\\')[0]

        withopen(text_file, "r", encoding="utf-8") asf:
            forlineinf:
                forsearch_stringinsearch_strings:
                    ifsearch_stringinline.lower():
                        o.write(f"/{root}{line.strip()}\n")
                        break
```

ДОДАТОК В

Програмна реалізаціятестуванняголосом розробленої системи активації

```
import sounddevice assd
importtorch
importtorchaudio

classWakeUpWordModel(torch.nn.Module):
    def__init__(
        self,
        n_mfcc=13,
        hidden_size=64,
        num_gru_layers=1
    ):
        super().__init__()

        self.gru=torch.nn.GRU(
            input_size=n_mfcc,
            hidden_size=hidden_size,
            num_layers=num_gru_layers
        )

        self.proj=torch.nn.Linear(hidden_size, 1)

    defforward(self, mfccs, mfcc_sizes):
        # mfccs: (time, batch_size, n_mfccs)
        # mfcc_sizes: (batch_size,)
        output=self.gru(mfccs)[0]

        # output: (time, batch_size, hidden_size)
        # We want to take the last output for each sequence
        last_output=output[mfcc_sizes-1,
torch.arange(mfcc_sizes.size(0))]

        returnself.proj(last_output).squeeze()

    @torch.no_grad()
    definference(self, mfccs, mfcc_sizes):
        output, _=self.gru(mfccs)
        returnself.proj(output)
```

```
device=torch.device("cpu"iftorch.cuda.is_available() else"cpu")

checkpoint_path=r"logs\2024-11-06_01-33-06\checkpoint_6.pth"
checkpoint=torch.load(checkpoint_path, map_location=device)
model=WakeUpWordModel(
    n_mfcc=checkpoint["params"]["num_mfcc"],
    hidden_size=checkpoint["params"]["hidden_size"],
    num_gru_layers=checkpoint["params"]["num_gru_layers"]
)

model.load_state_dict(checkpoint["model"])
model.to(device)
model.eval()

mfcc_transform=torchaudio.transforms.MFCC(
    sample_rate=checkpoint["params"]["sample_rate"],
    n_mfcc=checkpoint["params"]["num_mfcc"],
    melkwargs={
        "n_fft": checkpoint["params"]["n_fft"],
        "win_length": checkpoint["params"]["win_length"],
        "hop_length": checkpoint["params"]["hop_length"],
        "n_mels": checkpoint["params"]["n_mels"]
    }
)

prev_audio=torch.zeros(8000)
idx=0

defcallback(indata, frames, time, status):
    globalprev_audio
    globalidx

    ifstatus:
        print(status)

    audio=torch.tensor(indata.flatten(), dtype=torch.float32)
    audio=torch.cat([prev_audio, audio])

    mfcc=mfcc_transform(audio)
    prev_audio=audio[-8000:]

    # mfcc has shape (n_mfccs, time), but we need (time, 1, n_mfccs)
    mfcc=mfcc.permute(1, 0).unsqueeze(1).to(device)

    #output = model.inference(mfcc, torch.tensor([mfcc.shape[0]]))
```

```
    withtorch.no_grad():
        output=model(mfcc, torch.tensor([mfcc.shape[0]]))
    output=torch.sigmoid(output.squeeze()).max()

    print(idx, output.item())
    idx+=1

withsd.InputStream(
    samplerate=16000, blocksize=8000, channels=1, callback=callback,
latency=1
) ass:
    print('#'*80)
    print('press Ctrl+C to stop the recording')
    print('#'*80)

    whileTrue:
        sd.sleep(1000)
```