Одеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра оптимального керування та економічної кібернетики

Кваліфікаційна робота

на здобуття ступеня вищої освіти «магістр»

Генеративно-змагальна нейронна мережа для перетворення тексту в мову

Generative adversarial network for text to speach

Виконав: здобувач денної форми навчання
спеціальності 113 Прикладна математика
Освітня програма «Прикладна математика»
Григорян Костянтин Ашотович

Керівник: канд. техн. наук, доц. Мазурок І. Є.
Рецензент: канд. техн. наук, доц. Мороз В. В.

Рекомендовано до захисту:

Протокол засідання кафедри

№ ___ від _____ 2022 р.

Завідувач кафедри

_____ _____

Захищено на засіданні ЕК № _____

Протокол № ___ від _____ 2022 р.

Оцінка _____ / _____ / _____

Голова ЕК

_____ _____

Одеса — 2022 р.

ЗМІСТ

ВСТУП

Актуальність

Останнім часом інтелектуальні інформаційни системи все більше і більше проникають в життя людини та стають невід'ємною частиною її повсякденного життя. Окремою галузю таких систем є голосові помічники, які служать для пошуку необхідної інформації в Інтернеті, коригування маршруту тощо. Незважаючи на розповсюдженність, більшість голосових помічників має суттевий недолік: їхні голоси занадто роботизовані, що робить їх використання не дуже зручним.

Проблема роботизованності машиного голосу актуальна не тільки для голосових помічників. Саме це заважає реалізовувати автоматичну озвучку фільмів та мультфільмів: голоси не схожі на реальних персонажів, не передають необхідну інтонацію тощо.

Не менш актуальним є застосування не роботизованого машиного голосу для вивчення іноземних мов: такий підхід дозволяє імітувати спілкування з різними носіями мови.

Рішення, наявні на ринку, на даний час в більшості своєї або не мають необхідного функціоналу, надаючи, наприклад невелекий вибір заздалегідь готових голосів, що значно звужує можливі сфери використання, або платні та в закритому доступі.

Не менш важливою є проблема шквидкості донавчання існуючих рішень. Воно вимагає підготовку начального датасету дуже великого обсягу, що значно звужує можливі сфери використання.

Актуальність проблематика TTS та використання GAN для вирішення цієї задачі підтверджується великою кількістю досліджень на цю тематику. Наприклад, стаття Glow-TTS[1] вивчає можливість застосування генеративно-змагальної мережі для перетворення текста в мову. Також застосування ГЗМ для цієї задачі можна зустріти у High fidelity speech synthesis with adversarial networks[9]. Застосування досить нових мереж трансформерів досліджені у FastSpeech[2] і FastSpeech 2[3]. Duration informed attention network for multimodal synthesis[4] надає опис стратегії багатоголосної пара-

лельної генерації поверх моделі WaveRNN.

Мета

Розробка системи перекладу тексту в мову певним голосом для озвучки медіафайлів.

Для досягненя цієї мети, потрібно вирішити такі задачі: розробка системи перекладу тексту в мову певним голосом, аналіз існуючих рішень: виявленяя їх сильних сторін та недоліків, порівняння метрик якості роботи, швидкості навчання та процесингу, аналіз попереднього дослідження.

Об'єкт дослідження

Перетворення тексту до мови за допомогою генеративно-змагальних мереж. Застосування технології Transfer Learning (донавчання моделі).

Предмет дослідження

Нейроні мережі. Генеративно-змагальнихі мережи.

РОЗДІЛ 1

АНАЛІЗ ТА ПОРІВНЯННЯ ІСНУЮЧИХ РІШЕНЬ

TTS (text-to-speech) - технологія, що дозволяє комп'ютеру говорити як людина. Текстова інформація є входом, а відповідна мовна waveform – виходом. Існує багато механізмів реалізації TTS. Розглянемо один із найпопулярніших підходів.

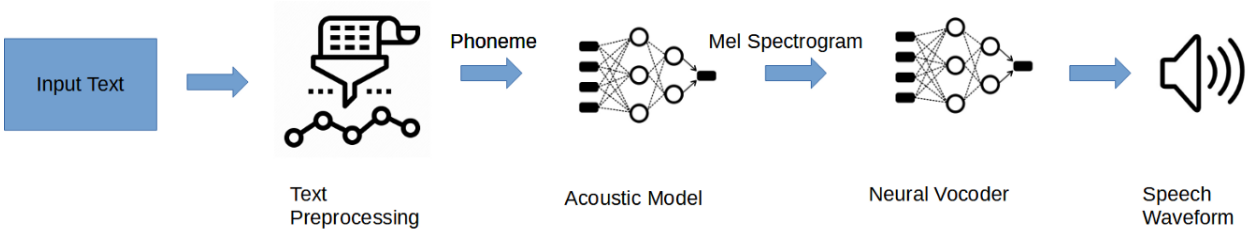

Рис. 1.1. TTS scheme.

1.1. Text preprocessing

Text preprocessing – ціль цього модуля - конвертація вхідного тексту у формат, який зможе бути розпізнаний нашою подальшою моделлю. Однією з ключових дій у цьому модулі є нормалізація тексту.

1.2. Acoustic model

Acoustic model – модуль, що конвертує лінгвістичні «фічі» в акустичні. Акустичні «фічі» включають в себе довжину і тональну якість голосу. І тут нам допоможе Mel-спектограмма. Mel-спектограма – конвертує звуковий спектр у шкалу, яка підходить для людського слуху, яка називається mel шкала.

1.2.1. Mel-спектограма

Дослідження показали, що люди не сприймають частоти в лінійному масштабі. Ми краще виявляємо відмінності на більш низьких частотах, ніж на більш високих частотах. Наприклад, ми легко можемо визначити різницю між 500 і 1000 Гц, але навряд чи зможемо відрізнити між 10 000 і 10 500 Гц, навіть якщо відстань між двома парами однакова.

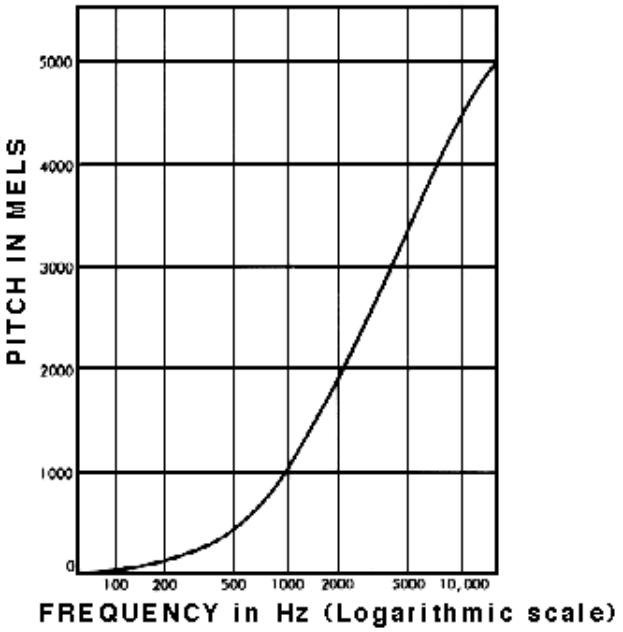

Рис. 1.2. Мел перетворення.

У 1937 році Стівенс, Фолькманна і Ньюманн запропонували таку одиницю висоти звуку, при якій рівні по висоті звуки звучали однаково далеко для слухача. Це називається шкалою мела. Ми виконуємо математичну операцію з частотами, щоб перетворити їх в мел-шкалу.

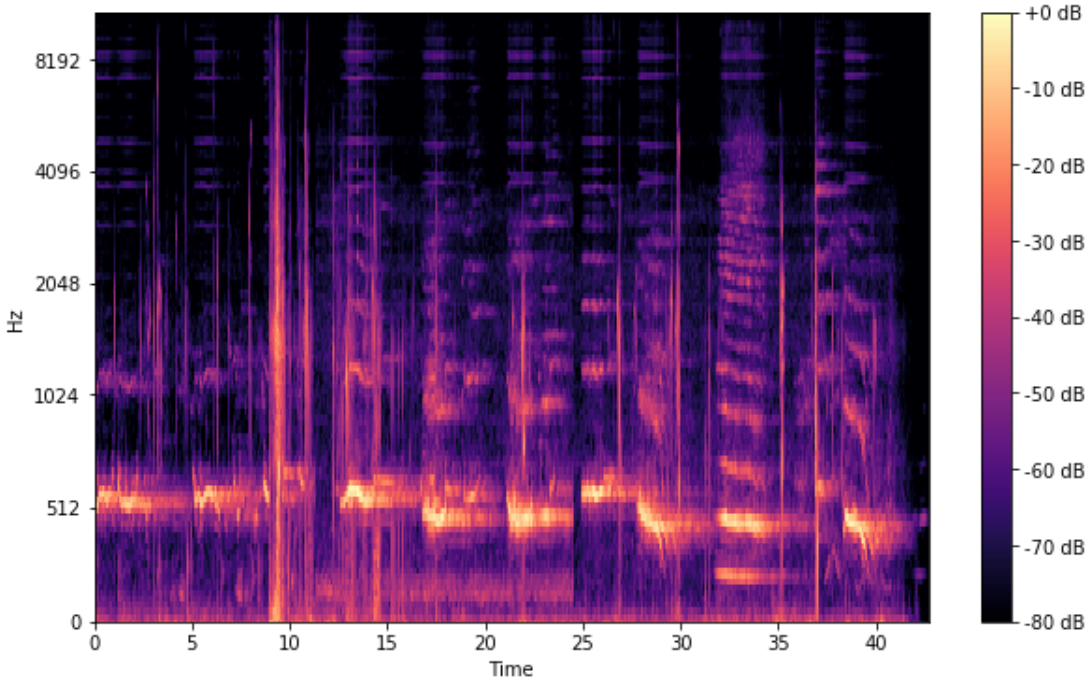

Рис. 1.3. Mel-спектограма.

1.2.2. Tacotron 2

Одним із популярних прикладів перетворення тексту в Mel-спектограму є Tacotron 2 [11].

Tacotron 2 має sequence to sequence архітектуру. Він складається з енкодера, який створює внутрішнє представлення вхідного сигналу (символічні маркери), і декодера, який перетворює це подання на Мел-спектрограму.

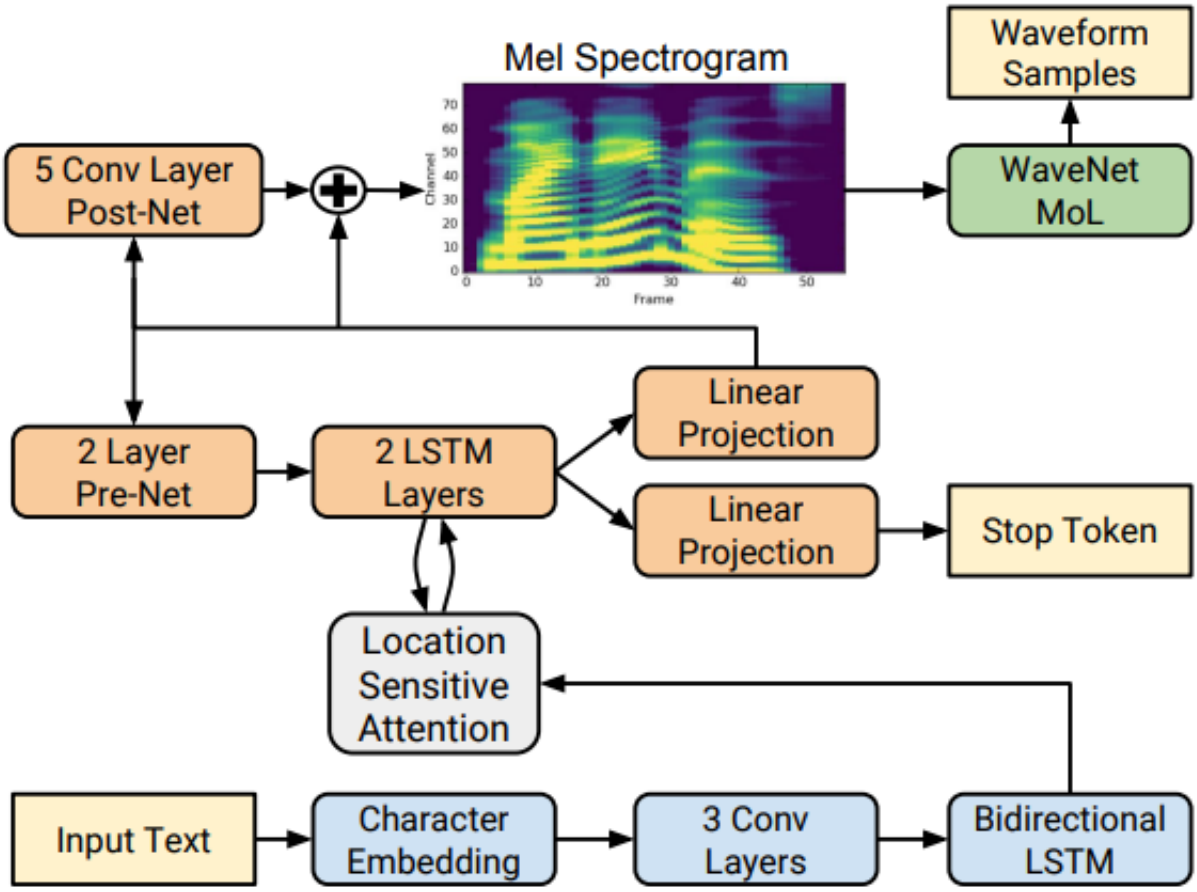

Рис. 1.4. Архітектура мережі Tacotron 2.

1.3. Neural Vocoder

Neural Vocoder – цей модуль приймає акустичні «фічі» на вхід і видає мовну waveform. У сучасних дослідженнях вокодери переважно використовують нейронні мережі. Саме тому цей модуль має назву нейронний вокодер. При створенні системи синтезу мовлення однією з найбільших проблем є цей модуль. Він сильно впливає як на якість, так і швидкість роботи. Розглянемо існуючі вокодери та їх недоліки.

1.3.1. WaveNet [5]

Розроблений google у 2016. Це авторегресивна модель, в якій кожен наступний звук залежить від усіх попередніх. Тобто модель генерує мову послідовно від початку. WaveNet показала дуже високий результат якості

генерованого звуку (4.21 за шкалою MOS – Mean opinion score). Проте швидкість роботи цієї моделі виявилася низькою порівняно з її конкурентами. Генерація однієї секунди мови займає у цієї моделі 180 секунд.

1.3.2. WaveGlow [6]

Це генеративная модель, яка генерує звук шляхом вибірки з деякого розподілу. Щоб використовувати нейронну мережу як генеративної моделі, беруться зразки з простого розподілу, в цьому випадку сферичний Гауссіан з нульовим середнім і з тієї ж розмірністю що і бажаний вихід і поміщаються ці зразки через серію слоїв, яка перетворює простий розподіл в таке, яке має бажане розподіл. В цьому випадку, моделюється розподіл аудіосемплам, обумовлених мел-спектрограм. WaveGlow має гарну швидкість навчання та процесингу, проте потребує чималого обсягу даних для навчання та має середню якість генерації звуку (3.9 за шкалою MOS).

1.3.3. Parallel WaveNet (PWN) [7]

Це неавторегресивна модель, яка генерує мову паралельно та одночасно для всіх звуків. Ця модель дозволяє використовувати GPU для паралелізації обчислень, що призводить до того, що генерація однієї секунди мови у Parallel WaveNet займає лише 30 мілісекунд. Однак мінусом цієї моделі є якість мови, що генерується, і швидкість навчання. PWN вимагає навчання двох нейронних мереж – учень та вчитель, що призводить до складності навчання.

1.3.4. Parallel WaveGAN [8]

Дана модель має схожу на WaveNet структуру, що складається з 30 слоїв згорткової нейронної мережі. Різниця полягає у вході та виході моделі. Вхід складається із зашумленого вектора такої ж довжини, як і мова на виході. Цей вектор паралельно процесується нейронною мережею, для генерації мовної waveform одночасно. У Parallel WaveGAN використовується технологія звана GAN (Generative adversarial network) [10].

Розглянемо докладніше принципи роботи GAN. Він складається з двох нейронних мереж – генератор та дискримінатор, які конкурують один з одним. Генератор приймає шум на вхід та генерує підроблений об'єкт, а дискримінатор приймає справжній та підроблений об'єкти та вчиться правильно їх розрізняти. Відповідно, генератор вчиться генерувати більш реалістичні об'єкти, щоб змогти обманути дискримінатор. При множинному повторенні цього процесу, генератор навчиться генерувати об'єкти, які не відрізняються від реальних. GAN має широке застосування в області фотографій та відеозаписів. Одним із прикладів використання GAN є генерація картинок з високою роздільною здатністю.

Parallel WaveGAN – показала високий результат якості мови (4.16 за шкалою MOS). За швидкістю генерації мови ця модель схожа з Parallel WaveNet. Так само дана модель показала хорошу швидкість навчання, не дивлячись на те, що вона також, як і попередня модель, містить дві нейронні мережі. Проте Parallel WaveGAN також має свої мінуси. Один з головних мінусів є кількість необхідних даних для навчання моделі

1.3.5. GAN-TTS [9]

У даній моделі також використовуються технологія GAN. Особливістю GAN-TTS є дискримінатори. Замість одного модель використовує набір дискримінаторів, кожен з яких має випадковий розмір. Ці дискримінатори працюють із випадково вибраними фрагментами реальних або згенерованих зразків. Вони оцінюють звучання як з погляду загального реалізму, так й з погляду вимови.

GAN-TTS показав гарну якість роботи (4.2 за шкалою MOS). Однак дана модель також вимагає великої кількості даних для навчання.

РОЗДІЛ 2

АНАЛІЗ ПОПЕРЕДНЬО ОТРИМАНИХ РЕЗУЛЬТАТІВ

У нашому попередньому дослідженні [15] [16] було отримано досить гарний результат. Наша система синтезу мовлення (TTS) являла собою комбінацію двох моделей нейронних мереж:

1) модифікована модель Tacotron 2
2) модель нейронної мережі WaveGlow

2.1. Tacotron 2 і WaveGlow

Задача перетворення тексту в мову складається з двох підзадач: перетворення тексту в мел, перетворення мелу в аудіозапис. Для рішеня першої задачи застосовувався Tacotron 2, а для другої - WaveGlow.

Моделі Tacotron 2 і WaveGlow утворюють систему текст-в-мову, яка дозволяє користувачам синтезувати природну звукову мову з необроблених транскриптів без додаткової інформації, такої як шаблони і / або ритми мови.

Ми використовували модель Tacotron 2, яка відрізняється від оригінальної. Оригінальна система перетворення тексту в мову використовує модель WaveNet для синтезу сигналів. Ми для цього використовуємо генеративну модель WaveGlow, яка набагато швидша.

2.2. Тренінг

Тренінг нейронної мережі це процес пошуку оптимальних значень ваг для мінімізації функції втрат. Зазвичай виконуються методом зворонього поширення помилки на основі розмічених даних.

Transfer Learning - це метод в машинному навчанні (ML), яка фокусу-

ється на збереженні знань, отриманих при вирішенні однієї проблеми, та застосуванні їх до іншої, але пов'язаної проблеми [14].

Скориставшись технологією Transfer Learning, ми донавчили вже навчену модель Tacatron 2, на обраному нами голосі персонажа і отримували mel спекторграму, яку відтворювали стандартною нейронною мережею WaveGlow в аудіо.

Таким чином ми отримали рішення, яке не потребує багатьох часів даних для навчаняя, має швидку скорость процессінгу та гарну якість генерованних аудиосемплів.

2.3. Мінуси попереднього рішення

Але в цьому рішенні були свої недоліки. Для поліпшення якості аудіосемплів нам доводилося вдаватися до хитрощів в плані написання самих фраз. Наприклад замість просто фрази Hello world, ми писали фразу Hellllooo worlldd!!, так як в такому випадку вимова виходила краща і зрозуміліша. Так само доводилося деякі великі літери переводити в маленькі і навпаки. Додавалися також знаки пунктуації. Були спроби автоматизувати цей процес, але робити завжди ці перетворення виявилося неправильним. У деяких випадках результат дійсно виявлявся краще, а в деяких навпаки - виходив гірше ніж початковий текст. Це все ускладнювало процес користування такою моделлю.

Також якщо ми подивимося на таблицю метрик з попереднюго нашого дослідження, ми побачемо, що схожість голоса гарна, але якість вимови все ж таки повина буду більша.

Кількість епох	Схожість голоса %	Якість вимови %
500	30	10
1500	33	18
2500	51	43
4500	91	85
6000	89	81
8000	83	50
10000	85	30

Табл. 2.1. Середня оцінка людей якости (за двома параметрами) аудиосеплів згенерованих на різних вагах для Tacatron 2.

2.4. Мінуси розмітки даних

Розмітка даних - це процес співставлення даних що подаються до нейроної мережі з очікуваним аутпутом.

Ще одним досить незручним моментом була розмітка даних. Для кожного аудіозапису треба було зіставити текст із вмістом у ньому. У нашому випадку для 579 записів було надано текст із вмістом, залишилося лише правильно скласти датасет. Однак часто трапляються випадки, коли є аудіозапис, але немає до нього тексту, що ускладнює підготовку даних у сотні разів. Можна скористатися автоматичним парсером слів, проте він також працює не ідеально, а наш датасет використовується як ground truth, тому в ньому повинні бути точні дані.

РОЗДІЛ 3

ЗАСТОСУВАННЯ ГЕНЕРАТИВНО-ЗМАГАЛЬНИХ МЕРЕЖ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ ПЕРЕТВОРЕННЯ ТЕКСТУ В МОВУ

GAN - генеративно-змагальні мережі, які знайшли широке застосування в сучасних задачах [10]. Ми вирішили застосувати для цієї задачі цю технологію, бо це новий, і водночас ефективний підхід. Ще однією важливою причиною є обмежена кількість даних для тренінгу моделей. Багато нейронних мереж потребують великої кількості даних, яких не завжди може бути в запасі. Так само генеративно-замагальні мережі є зручним у навчанні, так як не треба розмічати датасети, тобто вони практично на вхід приймають сирі дані. Аналогічно GAN оцінює лінгвістичні фічі і за підсумком показує набагато кращу якість вимови порівняно з його конкурентами.

Як було зазначено вище, GAN складається з двох нейронних мереж – генератора та дискримінатора, які конкурують один з одним. Генератор вчиться генерувати більш реалістичні об'єкти, щоб змогти обманути дискримінатор, а дискримінатор вчиться розрізняти реальні об'єкти від згенерованих.

Мінусом же GAN є імовірно довший час навчання через те, що доводиться навчати дві мережі - генератор і дискримінатор. Однак зрештою ми переконалися, що нам знадобилося часу на навчання навіть трохи менше, ніж у попередньому нашому досліджені.

3.1. Підготовка даних

Всього у нашому розпорядженні виявилося 579 фраз по $1-2$ речення і аудіодоріжка для кожної фрази. Разом 26 хвилин аудіо. Як і в попереньому досліженні, вибравши акса на сайті Dota2Wiki, ми розпарсили сторінку спеціальним чином, щоб автоматизувати процес викачування аудіозаписів і текстів до них. Вручну це зайняло б досить багато часу. Ці аудіозаписи були у

форматі mp3, тож ми написали свій власний конвертер, щоб змінити формат на wav. Будь який wav нам не підійшов, тому що сітки для тренування ГАНу працюють з певним sample rate, а саме 24000Hz.

3.2. Архітектура

Розглянемо детальніше архітектуру генератора да дискримінатора [9].

3.2.1. Генератор

Архітектуру генератора G наведено нижче в таблиці. Опишемо формат вхідних даних, вихідних даних та прихованих шарів. G на вході приймає послідовність лінгвістичних та звукових характеристик на частоті 200Hz. Віхідними даними будуть сирі waveforms на частоті 24000Hz. Генератор складається з семи GBlocks блоків, кожний з яких - з двох резідуал блоків. Використовується розширена згортка, щоб гарантувати що рецептивне поле генератора достатньо велике для того, щоб коректно опрацьовувати довготривалі залеждності через те, що генератор створює сирі аудіозаписи. Чотири розміри ядра у кожному GBlock мають зростаючі коефіцієнти розширення: 1, 2, 4, 8.

Для нормалізації даних застосовується Conditional Batch Normalisation після згортки. GBlock містить два скіп зв'язків. Перший зв'язок виконує апсемлінг, вхідні дані мають нижчу частоту, ніж вихідні. Також містить додаткову згортку, якщо кількість каналів відрізняється від вхідної. GBlocks 3–7 поступово апсемплять тимчасову розмірність приховних представлень на множники 2, 2, 2, 3, 5, а третій, шостий та сьомий GBlocks зменьшує кількість каналів в два рази. Одноканальний аудіо waveform генерується за допомогою останнього згорткового шару з гіперболічним тангенсом в якості функції активації.

layer/input	*t*	*freq.*	*ch*
linguistic features, *z*	400	200Hz	567
conv, *kernel size 3*	400	200Hz	768
GBlock	400	200Hz	768
GBlock	400	200Hz	768
GBlock, *upsample* $\times 2$	800	400Hz	384
GBlock, *upsample* $\times 2$	1600	800Hz	384
GBlock, *upsample* $\times 2$	3200	1600Hz	384
GBlock, *upsample* $\times 3$	9600	4800Hz	192
GBlock, *upsample* $\times 5$	48000	24kHz	96
conv, *kernel size 3*	48000	24kHz	1
Tanh			

Рис. 3.1. Generator.

3.2.2. Дискримінатор

Розглянемо архітектуру дискрмінатора, завданням якого є навчитися з високою точністю відріздняти справжні аудіозаписи від штучно-згенерованих. Блоки дискримінатора, які називаются DBlocks подібні до Gblocks, що застосовуються при побудові генератора. Головною відмінністю є відсутність batch normalization. Архітектури стандартних та умовних Dblocks. Їхньою єдиною відмінністю є додавання ембедінгів лінгвістичних характеристик після першої згортки в умовних DBLocks. Перший та останній Dblock не змінюють розмірність даних, що вони оброблюють. Крім цього в середині додається що найменш два блоки, що зменшують розмірність. Безумовні RWD повністю складаються з Dblocks. В умовних розмірність вхідної wasveform поступово зменшується за допомогою Dblocks до тих пір, доки розмірність не дорівнює розмірності умови, після чого використовуються умовні Dblocks. Об‘єднана інформація передається Dblocks, що залишилился, які нарешті генеруються скаляр. Множники розширення відповідають моделі 1-2-1-2 — на відміну від генератора, дискримінатор працює з відносно малим вікном.

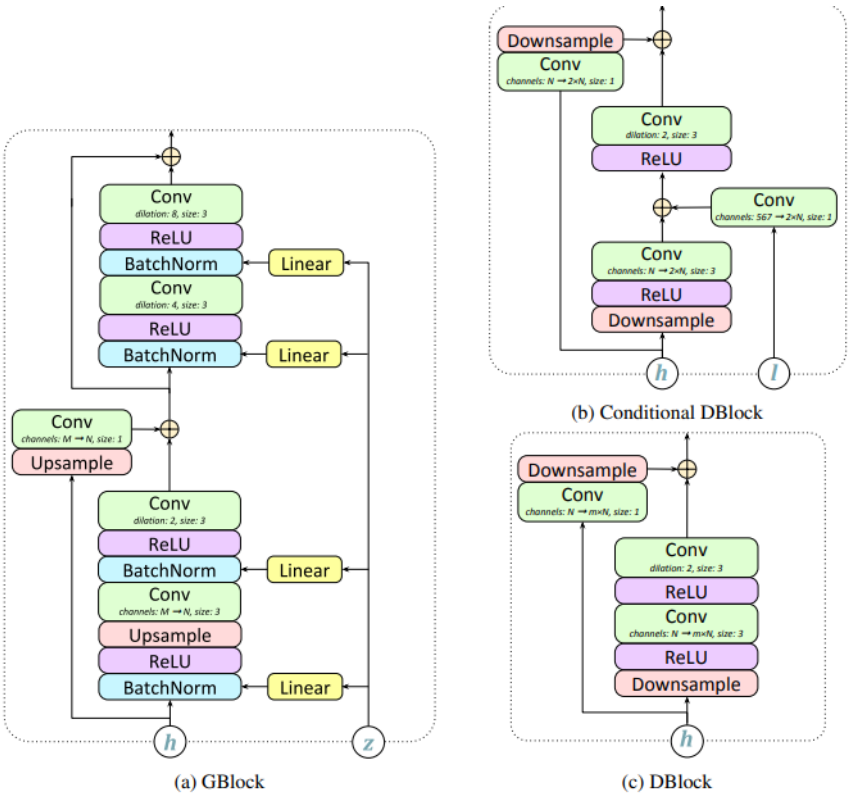

Рис. 3.2. Discriminator.

3.3. Треннінг

Розглянемо докладніше процес треннінгу генератора та дискримінатора [9].

3.3.1. Технічні дані

Тренінг проводився на локальному пристрої з CPU 11th Gen Intel (R) Core (TM) i7-11370H, з оперативною пам'яттю 16 гігабайт, і з GPU NVIDIA GeForce RTX 3060. Також, для порівняння часу навчання, були проведені експерименти на інших пристроях. Результати зведени у таблицю.

Усі треннінги були проведені на GPU.

Відеокарта	RAM	Час роботи
GeForce RTX 3060 6Gb (local)	16 Gb	16h
GeForce RTX2070 Super 8Gb (local)	24 Gb	14h
GeForce GeForce GTX1080 TI 11Gb (local)	24 Gb	15h
GeForce GTX1060 6Gb (local)	8 Gb	30h
Tesla T4 16 Gb (Google)	32 Gb	10h
Tesla K80 12 Gb (Google)	32 Gb	11h

Табл. 3.1. Дані експериментів з різними відеокартами.

3.3.2. Гіперпараметри генератора та дискримінатора

Для нашої моделі одним з важливійших факторів була кількість епох, на яких генератор навчається один, без допомоги дискримінатора. Після закінчення цієї кількості епох, у треннінг підключався дискримінатора, і відбувалося оновлення ваг вже для двох сіток.

Також важливим чинником виявилася довжина вікна дискримінатора. Були спроби узагальнити довжину вікна і використовувати константну довжину протягом усього тренінгу. Але це не призвело до бажаного результату. Тому було обрано рандомні довжини вікон дискримінатора.

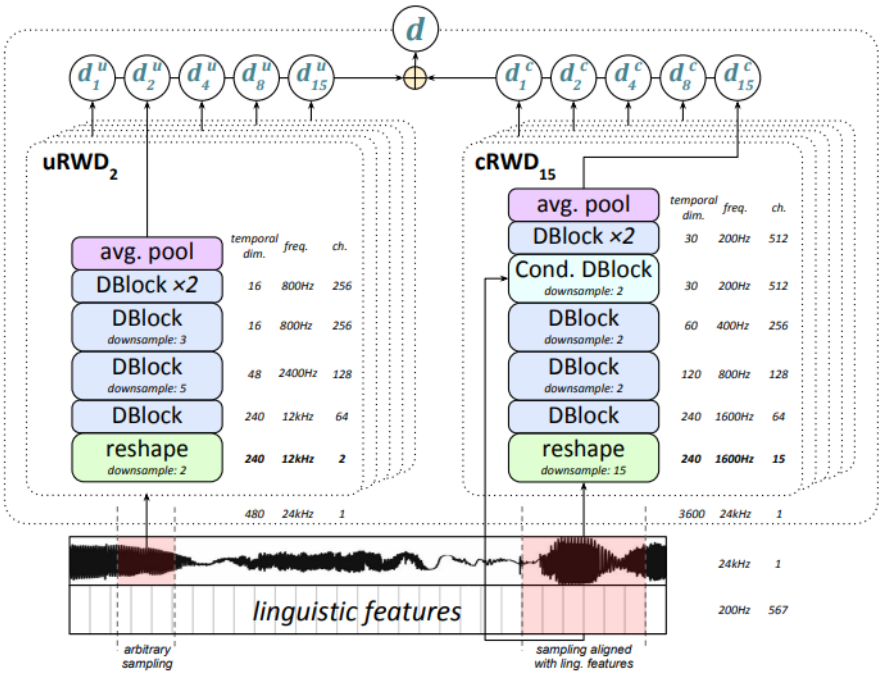

Рис. 3.3. Random windows.

Всі значення важливих гіперпараметрів ми вписали у таблицю:

Назва гіперпараметра	Значення
epochs	90000
discriminator train start steps	15000
batch size	25
sample rate	24000
n fft	2048
num mels	80
hop length	120
win length	240
fmin	40
min level db	-100
ref level db	20

Табл. 3.2. Гіперпараметри для GAN

Ітерації тренінгу виглядали так:

```
Step: 59996 --adv_loss: 0.254 --real_loss: 0.252 --fake_loss: 0.245 --sc_loss: 0.353 --mag_loss: 0.704 --Time: 0.31 seconds
Step: 59997 --adv_loss: 0.249 --real_loss: 0.252 --fake_loss: 0.245 --sc_loss: 0.340 --mag_loss: 0.711 --Time: 0.31 seconds
Step: 59998 --adv_loss: 0.244 --real_loss: 0.246 --fake_loss: 0.256 --sc_loss: 0.340 --mag_loss: 0.711 --Time: 0.31 seconds
Step: 59999 --adv_loss: 0.250 --real_loss: 0.255 --fake_loss: 0.248 --sc_loss: 0.438 --mag_loss: 0.891 --Time: 0.11 seconds
Saved checkpoint: logdir\model.ckpt-60000.pt
Step: 60000 --adv_loss: 0.247 --real_loss: 0.240 --fake_loss: 0.258 --sc_loss: 0.345 --mag_loss: 0.706 --Time: 0.57 seconds
Step: 60001 --adv_loss: 0.253 --real_loss: 0.245 --fake_loss: 0.256 --sc_loss: 0.336 --mag_loss: 0.715 --Time: 0.30 seconds
Step: 60002 --adv_loss: 0.259 --real_loss: 0.249 --fake_loss: 0.245 --sc_loss: 0.355 --mag_loss: 0.730 --Time: 0.31 seconds
Step: 60003 --adv_loss: 0.254 --real_loss: 0.258 --fake_loss: 0.243 --sc_loss: 0.343 --mag_loss: 0.710 --Time: 0.31 seconds
```

Рис. 3.4. Iterations of training.

3.3.3. Losses in GAN

Розглянемо losses, які були використанні у нашій моделі.

Log magnitude loss має вигляд:

$$\|log(|STFT(y)| + \epsilon) - log(|STFT(\hat{y})| + \epsilon)\|_l$$

Spectral convergence loss має вигляд:

$$\frac{\|\,|STFT(y)| - |STFT(\hat{y})|\,\|_E}{\|\,|STFT(\hat{y})|\,\|_E}$$

Графіки загальних losses.

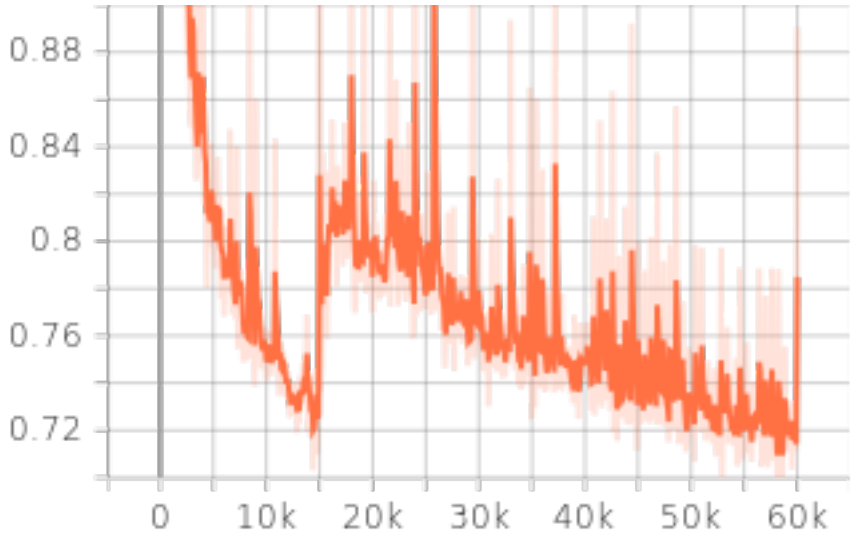

Рис. 3.5. Mag loss.

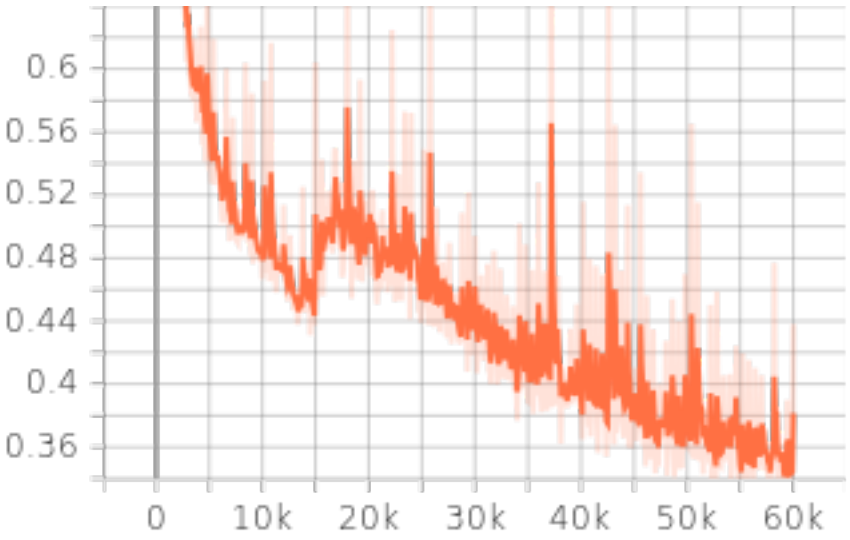

Рис. 3.6. Sc loss.

Коли дискримінатор почав навчатись, з'явилися три нових losses: fake loss, real loss, d loss. Real loss показує середню квадратичну помилку дискриминатора для реальних семплів, взятих з датасета. Аналогічно і fake loss, але для даних, які згенеровані за допомогою генератора. D loss це сума fake loss та real loss. На графіку ми бачимо, що ці значення дорівнюють нулю, доки дискримінатор не використовувався (до 15000 епохи).

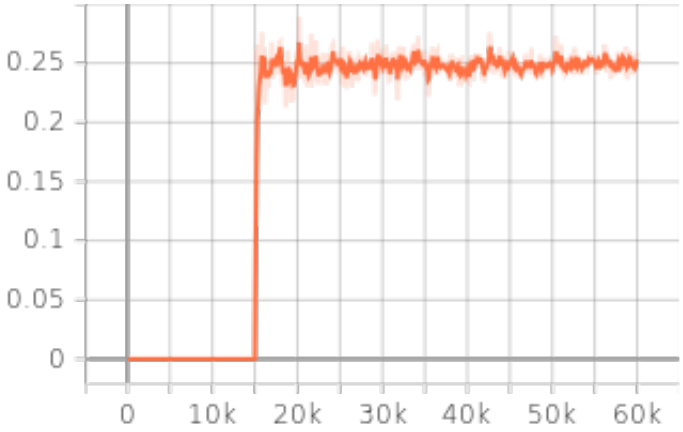

Рис. 3.7. Real loss.

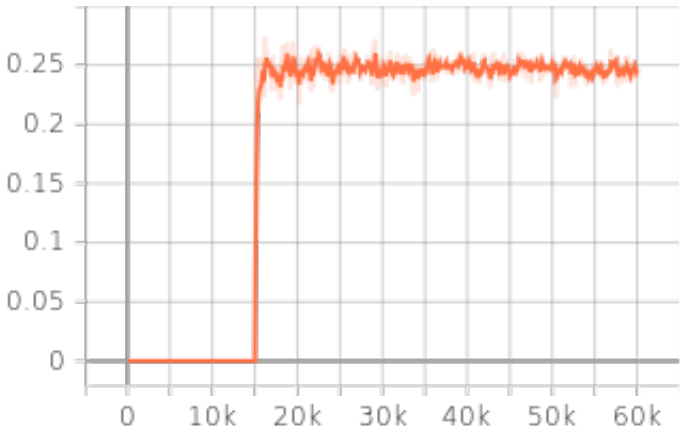

Рис. 3.8. Fake loss.

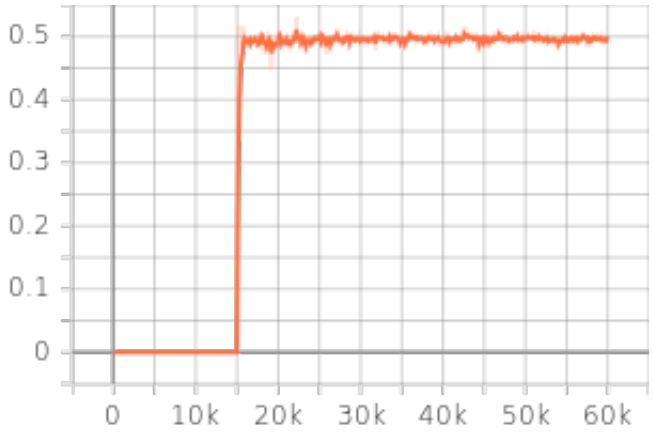

Рис. 3.9. D loss.

3.4. GAN та Tacotron 2

GAN котрий ми використали генерує waveform вибраним нами голосом з mel спектограми. Щоб отримати цю спекторгаму, ми вирішили скористатися згаданим раніше Tacatron 2, який з тексту вміє генерувати mel. Ми взяли той самий набір даних, який використовувався для тренінгу GAN, і донавчили преднавчанну модель, використовуючи технологію Transfer Learning [14].

Tacatron 2 донавчався 34000 ітерацій, що зайняло приблизно 15 годин на локальному пристрою. Кожні 1000 ітерациій робилися checkpoint, щоб можно було дали тестити не тільки на останніх вагах моделі, але й на проміжних.

Тепер Tacatron 2 вміє з тексту генерувати mel спектограму заданим

нами голосом, а наша генеративно-змагальна мережа озвучувати отриману спектограму. Однак це не дало нам гарного результату. Mel Tacatron 2 генерувався в іншій системі, відмінної від вхідних даних GAN. Були спроби конвертувати mel у mel, проте ці спроби не увінчалися успіхом через малу кількість даних.

Також були спроби навчити Tacatron 2 на різних форматах wav, що мало вплинути на подальшу якість звуку. Однак модель перенавчалася і так само не давала позитивних результатів.

3.5. Mel to audio GAN

Так як згадувалося раніше, наш обученний GAN вміє генерувати аудіо з mel спектограми. Однак генерувати ці спекторгами саме голосом нашого обраного героя виявилося не дуже гарною ідеєю.

Тому було вирішено брати будь-які mel спектограми, а не тільки навчені нами, з потрібним нам текстом, і перетворювати їх на аудіо, що показало дуже хороший результат. Спектограми ми генерували стандартними бібліотекою librosa на python. Вона використовує стандартне перетворення без використання нейронних мереж.

У результаті ми отримуємо гарну модель, яка вміє відтворювати потрібним нам голосом аудіо. Таким чином, ми досягли не тільки технології TTS, але й технології end-to-end, тобто звук у звук. Тобто нам вдалося отримати універсальний інструмент, який можна застосовувати у двох великих задачах.

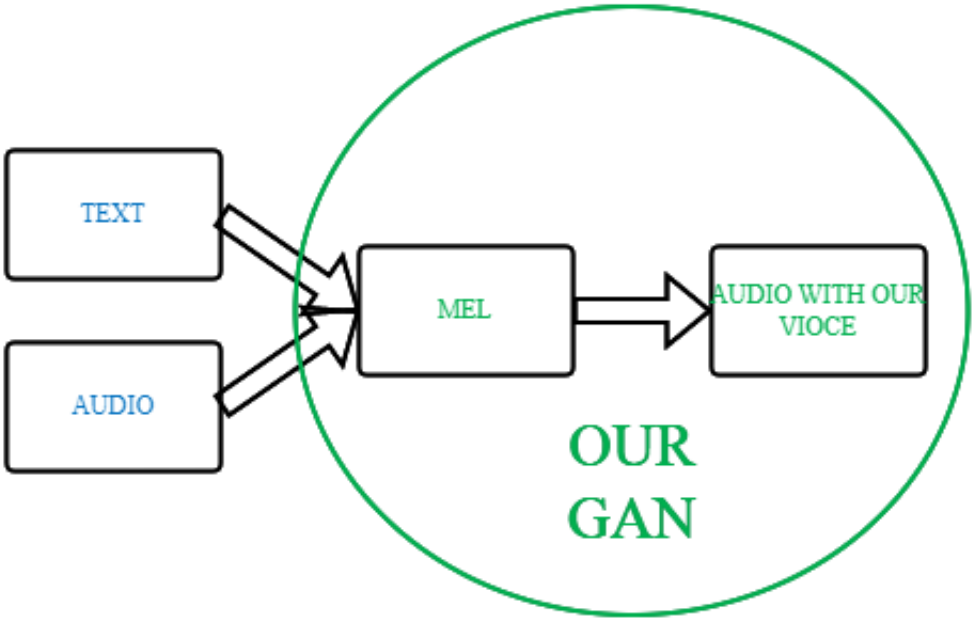

Рис. 3.10. General scheme.

3.6. GAN у реальному часі

Генерація аудіосемплів за допомогою GAN здійснюється дуже швидко. Наприклад, щоб згенерувати звук довжиною 3-5 секунд, программа втрачає меньш ніж секунду. Це показує, що цю технологію ми можемо використовувати у реальному часі.

Ми маємо, що в порівнянні з нашим минулим дослідженням, швидкість генерації аудіосемплів значно прискорилася. Минуле рішення могло працювати досить швидко, але тільки в режимі офлайн, тому його можна було застосувати в меншій кількості задач, ніж поточне.

3.7. Оцінка якості аудіосемпла

Існує чимала кількість метрік оцінки якості аудіосемплів, деякі з яких використовують нейронні мережі. Однією з найпопулярніших метрик є метрика MOS – mean opinion score. Так само існують такі метрики, як безумовна і умовна Frechet DeepSpeech відстань (FDSD, cFDSD) і Kernel DeepSpeech відстань (KDSD, cKDSD) [9].

Але як і в попередньому нашому дослідженні, ми використали дві міри кращості, які є модифікованою метрикою mean opinion score, для згенерованних нами аудіосемплів. А саме:

1) на скільки відсотків голос схожий на обраний нами (голос Axe),
2) на скільки відсотків вимовний текст можна розібрати.

В нашому експерименті взяло участь 37 осіб, кожному було дано прослухати по 6 фраз отриманих застосуванням різних ваг з різних епох. Ми врахували середнє їх оцінок та отримали наступні результати:

Кількість епох	Схожість голоса %	Якість вимови %
1000	5	40
5000	30	43
10000	51	64
20000	55	78
60000	87	98
75000	70	83
90000	70	81

Табл. 3.3. Середня оцінка людей якости (за двома параметрами) аудиосеплів згенерованих на різних вагах.

Найкращі ваги були отримані на 60000 епосі. Якщо порівнювати с попереднім дослідженням, де найкраший результат був:

1) Схожість голоса: 91 %
2) Якість вимови: 85 %

ми отримаємо, що схожість голоса трошки погіршилась, але стала значно краще якість вимови та час генерації аудіосемплів.

ВИСНОВКИ

Ця робота присвячена рішеню проблеми відтворення тексту в мову певним голосом. Це може допомогти сліпим людям в читанні книг або в користуванні комп'ютером, де голосовий помічник може говорити голосом, обраним цією людиною.

В першому розділі розглянути більшість популярних рішень, деякі з яких використовують технолгию GAN. Проведено аналіз та виявлено сильні сторони та недоліки кожного з рішень.

Другий розділ присвячений аналізу попереднього нашого рішення та підкреслення недоліків, які заважали зручному користуванню моделлю.

Третій розділ пояснює нове рішення, описується процес треннінгу, а саме гіперпараметри для генератора та дискримінатора, loss-функції. Проведені експерименти з різними відеокартами і різною кількістю оперативної пам'яті. Також розглянуто деякі підхіди, які не дали бажанного результату. За допомогою експериментів проведено аналіз якості звуку створеного з написаного тексту. Введено спеціальні метрики, такі як схожість голоса та якість вимови. Показано універсальність рішення, яке вирішує не одну, а дві задачі.

В ході дослідження були розглянуті та вирішені задачі для виконання поставленної цілі.

Проведено аналіз таких рішень: WaveNet, WaveGlow, Parallel WaveNet (PWN), Parallel WaveGAN, GAN-TTS, виявлено їх сильні сторони та недолікі, порівняно метрикі якості роботи, швидкості навчання та процесингу.

Проведено аналіз попереднього дослідження, в основі якого були Tacotron 2 та WaveGlow. Виявлення мінусів цього підходу, основними з яких були якість вимови тексту, незручність тренінгу моделі та відсутність робастності.

Розроблено систему перекладу тексту в мову певним голосом за допомогою технології GAN. Описано архітектури моделей та процес тренінгу. Описано спроби об'єднання GAN та Tacatron 2. Була натренована модель, а саме генератор та дискримінатор. Проведено оцінка якості згенерованого

аудіосемплу.

Таким чином ми маємо рішення перетворення тексту в мову певним голосом, яке не вимагає багатьох годин трейн аудіосеплів і дає якісну аудіозапис, та також може використовуватися у реальному часі, а не в офлайн режимі.

СПИСОК ЛІТЕРАТУРИ

1. Kim J., Glow-TTS: A Generative Flow for Text-to-Speech viaMonotonic Alignment Search / J. Kim, S. Kim, J. Kong, S. Yoon – 2020. – Resource access mode: https://arxiv.org/pdf/2005.11129.pdf
2. Ren Y. FastSpeech: Fast, Robust and ControllableText to Speech / Y. Ren, Y. Ruan, X. Tan, T. Qin, S. Zhao, Z. Zhao, Tie-Yan Liu – 2019. – Resource access mode: https://arxiv.org/pdf/1905.09263.pdf.
3. Ren Y. FastSpeech 2: Fast and hight-quality end-to-end text to speech / Y. Ren, C. Hu, X. Tan, T. Qin, S. Zhao, Z. Zhao, Tie-Yan Liu – 2021. – Resource access mode: https://arxiv.org/pdf/2006.04558.pdf.
4. Yu C. Durian: duration informed attention network for multimodal synthesis / C. Yu, H. Lu, N. Hu, M. Yu, C. Weng, K. Xu, P. Liu, D. Tuo, S. Kang, G. Lei, D. Su, D. Yu – 2019. – Resource access mode: https://arxiv.org/pdf/1909.01700.pdf.
5. Aaron van den Oord: WaveNet: A generative model for raw audio / A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuoglu - 2016 - Resource access mode: https://arxiv.org/pdf/1609.03499.pdf.
6. Prenger R. WaveGlow: a flow-based generative network for speech synthesis / R. Prenger, R. Valle, B. Catanzaro. – 2018. – Resource access mode: https://arxiv.org/pdf/1811.00002.pdf.
7. Aaron van den Oord: Parallel WaveNet: Fast High-Fidelity Speech Synthesis / A. van den Oord, Y. Li, I. Babuschkin, K. Simonyan, O. Vinyals, K. Kavukcuoglu - 2017 - Resource access mode: https://arxiv.org/pdf/1711.10433.pdf.
8. Ryuichi Yamamoto: Parallel WaveGAN: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram / R. Yamamoto, E. Song, J. Kim - 2020 - Resource access mode: https://arxiv.org/pdf/1910.11480.pdf.
9. Donahue J. High fidelity speech synthesis with adversarial networks / J. Donahue, S. Dieleman, A. Clark, E. Elsen, N. Casagrande, L. C. Cobo, K. Simonyan – 2019. – Resource access mode:

https://arxiv.org/pdf/1909.11646.pdf.

10. Ian J. Goodfellow: Generative Adversarial Nets / I. Goodfellow, J. Pouget-Abadie, M. Mirza, B, Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio - 2014 - Resource access mode: https://arxiv.org/pdf/1406.2661v1.pdf.
11. Shen J. Natural TTS synthesis by conditioning wavenet on mel spectrogram predictions / [J. Shen, R. Pang, W. Ron]. – 2018. – Resource access mode: https://arxiv.org/pdf/1712.05884.pdf.
12. Abadi M. TensorFlow: A system for large-scale machine learning / M. Abadi, P. Barham, J. Chen – (2016), pp. 265-283 – Resource access mode: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.
13. Paszke A. PyTorch: An Imperative Style, High-Performance Deep Learning Library / A. Paszke, S. Gross, F., A. Lerer, J. Bradbury, Massa – 2018. – Resource access mode: https://arxiv.org/abs/1912.01703.
14. Chandeep S. Transfer Learning and its application in Computer Vision / S. Chandeep, P. Sayonee // Electrical and Computer Engineering University of Waterloo Waterloo – 2022
15. Hryhorian K., Volkov K., Mazurok I. Tacotron 2 and WaveGlow for text-to-speech for PC game characters // Інформатика, інформаційні системи та технології: тези доповідей вісімнадцятої всеукраїнської конференції студентів і молодих науковців. – Одеса. – 2021. – 62-63 с.
16. Григорян К.А., Мазурок І. Є., Волков К.С., Масальський Р.О. Tacotron 2 І WaveGlow для перетворення тексту до речи для персонажів комп'ютерних ігор // Стан, досягнення та перспективи інформаційних систем і технологій / Матеріали XXI Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів. - Одеса, Видавництво ОНАХТ. – 2021. – 207-208 с.
17. Григорян К., Майдан А., Масальський Р., Мазурок І. Моделювальня системи для навчання нейронних мереж // Інформатика, інформаційні системи та технології: тези доповідей дев'ятнадцятої всеукраїнської конференції студентів і молодих науковців. – Одеса. – 2022. – 65-67 с.
18. Hryhorian K., Maidan A., Masalskyi R., Mazurok I. Simulating systems for training neural networks // Стан, досягнення та перспективи інформаційних систем і технологій / Матеріали XXII Всеукраїнської

науково-технічної конференції молодих вчених, аспірантів та студентів. - Одеса, Видавництво ОНАХТ. – 2022. – 187-189 с.

19. Hryhorian Kostiantyn. Simulating system for training neural networks // 1 st Student Scientific Conference of Joint Research Cooperation between Odesa I.I. Mechnikov National University and Huaiyin Institute of Technology. – 2022. – 157-159 с.

ДОДАТОК А. КОДИ

```
from pydub import AudioSegment
import os
import librosa
import scipy

# path = "C:\\Users\\Asus\\Desktop"
path = "mp3_files"
for filename in os.listdir(path):
    f = os.path.join(path, filename)
    if os.path.isfile(f):
        print(f)
    if f.endswith(".mp3"):
        sound = AudioSegment.from_mp3(f)
        audio_path = 'C:\\Users\\Asus\\Desktop\\' \
                     'wavs_22050\\' \
                    + filename[:-4] + '.wav'
        sound.export(audio_path, format="wav")
        x, sr = librosa.load(audio_path)
        scipy.io.wavfile.write(audio_path, 22050, x)


from pydub import AudioSegment

start = 60 * 1000
end = 12 * 60 * 1000
step = 5 * 1000
newAudio = AudioSegment.from_wav(
    "C:\\Users\\Asus\\Desktop\\internet_speech\\"
    "zel1.wav")
# for i in range(0, len(newAudio), step):
for i in range(start, end, step):
```

```
        newAudio_split = newAudio[i:i + step]
        print(i)
        newAudio_split.export(
            'C:\\Users\\Asus\\Desktop\\train_data_zel\\'
            'zel2_{}.wav'.format(
                i),
            format="wav")


# import required modules
from pydub import AudioSegment
import librosa
import soundfile as sf
import os

# convert mp3 file to wav file

path = "C:\\Users\\Asus\\Desktop\\mp3_for_pr"
output_file = "C:\\Users\\Asus\\Desktop\\wav_for_pr\\"
for filename in os.listdir(path):
    f = os.path.join(path, filename)
    if os.path.isfile(f):
        print(f)
    if f.endswith(".mp3"):
        y, s = librosa.load(f)
        sf.write(
            output_file + filename[:-4] + '.wav',
            y, 24000, 'PCM_16')


import torch
from pydub import AudioSegment
import sys
```

```
sys.path.append('waveglow/')
from denoiser import Denoiser
from layers import TacotronSTFT, STFT
from hparams import create_hparams
import IPython.display as ipd

hparams = create_hparams()

# Load mels
from utils import load_wav_to_torch
import os
import scipy

stft = TacotronSTFT(hparams.filter_length,
                    hparams.hop_length,
                    hparams.win_length,
                    hparams.n_mel_channels,
                    hparams.sampling_rate,
                    hparams.mel_fmin,
                    hparams.mel_fmax)


def load_mel(path):
    audio, sampling_rate = load_wav_to_torch(path)
    if sampling_rate != stft.sampling_rate:
        raise ValueError(
            "{} {} SR doesn't match target {} SR".format(
                sampling_rate,
                stft.sampling_rate))
    audio_norm = audio / hparams.max_wav_value
    audio_norm = audio_norm.unsqueeze(0)
    audio_norm = torch.autograd.Variable(
        audio_norm, requires_grad=False)
    melspec = stft.mel_spectrogram(audio_norm)
```

```
    return melspec


# Load WaveGlow
# waveglow_path = 'WAVEGLOW PATH HERE'
# waveglow = torch.load(waveglow_path)['model']
waveglow = torch.hub.load(
    'nvidia/DeepLearningExamples:torchhub',
    'nvidia_waveglow')
waveglow.cuda().eval().half()
for k in waveglow.convinv:
    k.float()
denoiser = Denoiser(waveglow)

mel_outputs_postnet = load_mel(
    file_path).cuda().half()

MAX_WAV_VALUE = 32768.0

mel = torch.autograd.Variable(
    mel_outputs_postnet.cuda())
# mel = torch.unsqueeze(mel, 0)
mel = mel.half()
with torch.no_grad():
    audio = waveglow.infer(mel, sigma=0.666)
    denoiser_strength = 0.1
    if denoiser_strength > 0:
        audio = denoiser(audio, denoiser_strength)
    audio = audio * MAX_WAV_VALUE
audio = audio.squeeze()
audio = audio.cpu().numpy()
audio = audio.astype('int16')
audio_path = os.path.join(
    "audios/test/",
```

```
    "{}_synthesis.wav".format("inf_test3"))
scipy.io.wavfile.write(audio_path,
                       hparams.sampling_rate,
                       audio)
print(audio_path)

import matplotlib
import matplotlib.pylab as plt

import IPython.display as ipd
from pydub import AudioSegment

import sys

sys.path.append('waveglow/')
import numpy as np
import torch

from hparams import create_hparams
from model import Tacotron2
from layers import TacotronSTFT, STFT
from audio_processing import griffin_lim
from train import load_model
from text import text_to_sequence

# from mel2samp import files_to_list, MAX_WAV_VALUE
MAX_WAV_VALUE = 32768.0

from denoiser import Denoiser
import scipy
import librosa
import scipy
# import audio
import librosa.display
```

```
def plot_data(data, figsize=(16, 4)):
    fig, axes = plt.subplots(1, len(data),
                             figsize=figsize)
    np.save('mels/mel.npy', data[1].transpose(),
            allow_pickle=False)
    for i in range(len(data)):
        axes[i].imshow(data[i], aspect='auto',
                       origin='lower',
                       interpolation='none')
    fig.savefig('plots/full_figure.png')


hparams = create_hparams()
# hparams.sampling_rate = 22050

checkpoint_path = "outdir_24000_custom_params/checkpoint_5000"
# checkpoint_path = "C:\\Users\\Asus\\Downloads\\tacotron2-mas
model = load_model(hparams)
model.load_state_dict(
    torch.load(checkpoint_path)['state_dict'])
_ = model.cuda().eval().half()

# waveglow_path = 'waveglow_256channels_universal_v5.pt'
# waveglow = torch.load(waveglow_path)['model']
waveglow = torch.hub.load(
    'nvidia/DeepLearningExamples:torchhub',
    'nvidia_waveglow')

waveglow.cuda().eval().half()
for k in waveglow.convinv:
    k.float()
denoiser = Denoiser(waveglow)
```

```
text = "Hello everybody"
sequence = np.array(
    text_to_sequence(text, ['english_cleaners']))[
            None, :]
sequence = torch.autograd.Variable(
    torch.from_numpy(sequence)).cuda().long()

mel_outputs, mel_outputs_postnet, _, alignments = model.inferen
    sequence)
print(np.max(np.array(
    mel_outputs_postnet.cpu().detach().numpy())))
print(np.min(np.array(
    mel_outputs_postnet.cpu().detach().numpy())))
plot_data(
    (mel_outputs.float().data.cpu().numpy()[0],
     mel_outputs_postnet.float().data.cpu().numpy()[
         0],
     alignments.float().data.cpu().numpy()[0].T))

n_mels=80, fmax=8000, pad_mode='reflect')
with torch.no_grad():
    audio = MAX_WAV_VALUE * \
            waveglow.infer(mel_outputs_postnet,
                           sigma=0.666)[0]

audio = audio.cpu().numpy()
audio = audio.astype('int16')
scipy.io.wavfile.write(
    'audios/24000_custom_params.wav',
    hparams.sampling_rate, audio)
```

```
import sys
import copy
import torch


def _check_model_old_version(model):
    if hasattr(model.WN[0], 'res_layers') or \
            hasattr(model.WN[0], 'cond_layers'):
        return True
    else:
        return False


def _update_model_res_skip(old_model, new_model):
    for idx in range(0, len(new_model.WN)):
        wavenet = new_model.WN[idx]
        n_channels = wavenet.n_channels
        n_layers = wavenet.n_layers
        wavenet.res_skip_layers = torch.nn.ModuleList()
        for i in range(0, n_layers):
            if i < n_layers - 1:
                res_skip_channels = 2 * n_channels
            else:
                res_skip_channels = n_channels
            res_skip_layer = torch.nn.Conv1d(
                n_channels, res_skip_channels, 1)
            skip_layer = torch.nn.utils.remove_weight_norm(
                wavenet.skip_layers[i])
            if i < n_layers - 1:
                res_layer = torch.nn.utils.remove_weight_norm(
                    wavenet.res_layers[i])
                res_skip_layer.weight = torch.nn.Parameter(
                    torch.cat(
                    [res_layer.weight, skip_layer.weight]))
```

```
                res_skip_layer.bias = torch.nn.Parameter(
                    torch.cat(
                    [res_layer.bias, skip_layer.bias]))
            else:
                res_skip_layer.weight = torch.nn.Parameter(
                    skip_layer.weight)
                res_skip_layer.bias = torch.nn.Parameter(
                    skip_layer.bias)
            res_skip_layer = torch.nn.utils.weight_norm(
                res_skip_layer, name='weight')
            wavenet.res_skip_layers.append(res_skip_layer)
        del wavenet.res_layers
        del wavenet.skip_layers


def _update_model_cond(old_model, new_model):
    for idx in range(0, len(new_model.WN)):
        wavenet = new_model.WN[idx]
        n_channels = wavenet.n_channels
        n_layers = wavenet.n_layers
        n_mel_channels = \
            wavenet.cond_layers[0].weight.shape[1]
        cond_layer = torch.nn.Conv1d(
            n_mel_channels, 2 * n_channels *
                            n_layers, 1)
        cond_layer_weight = []
        cond_layer_bias = []
        for i in range(0, n_layers):
            _cond_layer = torch.nn.utils.remove_weight_norm(
                wavenet.cond_layers[i])
            cond_layer_weight.append(_cond_layer.weight)
            cond_layer_bias.append(_cond_layer.bias)
        cond_layer.weight = torch.nn.Parameter(torch.cat(
            cond_layer_weight))
```

```
            cond_layer.bias = torch.nn.Parameter(torch.cat(
                cond_layer_bias))
            cond_layer = torch.nn.utils.weight_norm(
                cond_layer, name='weight')
            wavenet.cond_layer = cond_layer
            del wavenet.cond_layers


def update_model(old_model):
    if not _check_model_old_version(old_model):
        return old_model
    new_model = copy.deepcopy(old_model)
    if hasattr(old_model.WN[0], 'res_layers'):
        _update_model_res_skip(old_model, new_model)
    if hasattr(old_model.WN[0], 'cond_layers'):
        _update_model_cond(old_model, new_model)
    return new_model


def convert(o, n):
    old_model_path = o
    new_model_path = n
    model = torch.load(old_model_path)
    model['model'] = update_model(model['model'])
    torch.save(model, new_model_path)

from google.colab import drive

drive.mount('/content/gdrive')

!ln -s /content/gdrive/My\ Drive/ /mydrive
!ls /mydrive

!git clone https://github.com/NVIDIA/tacotron2.git
```

```
!git submodule init
!git submodule update

sys.path.append('/content/tacotron2/waveglow/')

convert("/content/waveglow_256channels_ljs_v2.pt",
        "/content/waveglow_256channels.pt")

!unzip / content / LJSpeech - 1.1.zip -
d / content / tacotron2 / data /

!sed - i -- 's,DUMMY,/content/tacotron2/Hach_wav,g'
filelists / *.txt

!pip install - r requirements.txt

!ls / mydrive

!pip install - U numpy == 1.16.4

!python train.py - -output_directory = /
mydrive / cybersport_hackaton / debug
/ --log_directory = / mydrive / cybersport_hackaton /
debug / -c tacotron2_statedict.pt - -warm_start

!pip uninstall matplotlib
!pip install matplotlib

import matplotlib

import IPython.display as ipd

import sys
```

```
sys.path.append('/content/tacotron2/waveglow/')
import numpy as np
import torch

from hparams import create_hparams

from layers import TacotronSTFT, STFT
from audio_processing import griffin_lim
from train import load_model
from text import text_to_sequence
from denoiser import Denoiser


def plot_data(data, figsize=(16, 4)):
    fig, axes = plt.subplots(1, len(data), figsize=figsize)
    for i in range(len(data)):
        axes[i].imshow(data[i], aspect='auto',
                       origin='bottom',
                       interpolation='none')


hparams = create_hparams()
hparams.sampling_rate = 22050

checkpoint_path = "checkpoint_4500"
model = load_model(hparams)
model.load_state_dict(
    torch.load(checkpoint_path)['state_dict'])

_ = model.cuda().eval()

len(list(model.parameters()))
```

```
print(list(model.parameters()))

print(_)

waveglow = torch.hub.load(
    'nvidia/DeepLearningExamples:torchhub',
    'nvidia_waveglow')
waveglow = waveglow.remove_weightnorm(waveglow)
waveglow = waveglow.to('cuda')
waveglow.eval()

text = "the idea."
sequence = np.array(text_to_sequence(
    text, ['english_cleaners']))[None, :]
print(sequence)

sequence = torch.autograd.Variable(
    torch.from_numpy(sequence)).cuda().long()

with torch.no_grad():
    _, mel, _, _ = model.inference(sequence)
    audio = waveglow.infer(mel)
audio_numpy = audio[0].data.cpu().numpy()
rate = 22050

import numpy as np
from scipy.io.wavfile import write

write("audio.wav", rate, audio_numpy)

from IPython.display import Audio

Audio(audio_numpy, rate=rate)
```

```
mel_outputs, mel_outputs_postnet, _, alignments = \
    model.inference(sequence)
plot_data((mel_outputs.float().data.cpu().numpy()[0],
           mel_outputs_postnet.float().data.cpu().numpy()[0],
           alignments.float().data.cpu().numpy()[0].T))

with torch.no_grad():
    audio = waveglow.infer(mel_outputs_postnet, sigma=0.666)
ipd.Audio(audio[0].data.cpu().numpy(),
          rate=hparams.sampling_rate)

audio_denoised = denoiser(audio, strength=0.01)[:, 0]
ipd.Audio(audio_denoised.cpu().numpy(),
          rate=hparams.sampling_rate)
```