

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МО ЭВМ**

**ОТЧЕТ**  
**по учебной практике**  
**Тема: Генерация голоса конкретного человека по записи его голоса**

Студентка гр. 3384

Копасова К. А.

Студентка гр. 3384

Самойлова Е. М.


Руководитель, к.т.н

Филатов А. Ю.

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_



Санкт-Петербург

2025

## ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студентка Копасова К. А. группы 3384

Студентка Самойлова Е. М. группы 3384

Тема практики: Генерация голоса конкретного человека по записи его голоса


Задание на практику:

Командная итеративная разработка систем синтеза речи TTS (Text-to-speech) для генерации голоса на основе образцов речи основного пользователя и их сравнение.

Сроки прохождения практики: 25.06.2024 – 08.07.2024

Дата сдачи отчета: 07.07.2024

Дата защиты отчета: 07.07.2024

Студентка		Копасова К. А.
Студентка		Самойлова Е. М.
Руководитель		Филатов А. Ю.

## **АННОТАЦИЯ**

Данная работа представляет собой комплексное исследование современных нейросетевых подходов к синтезу речи на примере моделей FastSpeech и FastSpeech 2. В рамках исследования проводится детальный анализ архитектурных особенностей обеих моделей, их вычислительной эффективности и качества синтезированной речи. Особое внимание уделяется практическим аспектам реализации: подготовке и предобработке данных, настройке гиперпараметров, процессу обучения и методам оценки результатов. На основании анализа моделей формулируются практические рекомендации по выбору оптимальной архитектуры в зависимости от конкретных требований проекта.

## **SUMMARY**

This study presents a comprehensive investigation of modern neural network approaches to speech synthesis, using the FastSpeech and FastSpeech 2 models as examples. The research includes a detailed analysis of the architectural features of both models, their computational efficiency, and the quality of synthesized speech. Special attention is given to practical implementation aspects: data preparation and preprocessing, hyperparameter tuning, the training process, and evaluation methods. Based on the model analysis, practical recommendations are provided for selecting the optimal architecture depending on specific project requirements.

## СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ.....</b>	<b>5</b>
<b>ТРЕБОВАНИЯ К ПРОГРАММЕ.....</b>	<b>5</b>
Исходные требования к программе.....	6
<b>РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ .....</b>	<b>7</b>
<b>ТЕОРЕТИЧЕСКАЯ СПРАВКА .....</b>	<b>8</b>
<b>ОСОБЕННОСТИ РЕАЛИЗАЦИИ.....</b>	<b>10</b>
Предобработка данных для модели FastSpeech .....	10
Предобработка данных для модели FastSpeech 2 .....	11
Реализация и обучение модели FastSpeech .....	13
Реализация и обучение модели FastSpeech 2 .....	14
Методы оценивания моделей.....	16
Анализ моделей .....	16
<b>ТЕСТИРОВАНИЕ.....</b>	<b>17</b>
Тестирование модели FastSpeech .....	17
Тестирование модели FastSpeech 2 .....	17
<b>ЗАКЛЮЧЕНИЕ .....</b>	<b>18</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....</b>	<b>19</b>
<b>ПРИЛОЖЕНИЕ А.....</b>	<b>20</b>

## ВВЕДЕНИЕ

**Проблема:** несмотря на значительные достижения в области синтеза речи (TTS), вопрос выбора оптимальной модели, обеспечивающей баланс между скоростью обучения, качеством синтезированного голоса и вычислительной эффективностью, остается открытым. Особенно это актуально для задачи клонирования голоса конкретного человека, где каждый из этих параметров имеет важное значение.

**Цель:** провести сравнительный анализ двух современных TTS-моделей - FastSpeech и FastSpeech 2 - для выявления их преимуществ и ограничений в контексте различных прикладных задач, с акцентом на три ключевых аспекта: скорость обучения, качество синтеза и требования к вычислительным ресурсам.

### **Задачи:**

- 1) Провести теоретическое исследование архитектур FastSpeech, выявив их особенности, преимущества и ограничения;
- 2) Найти датасет аудиозаписей и предобработать его для TTS моделей;
- 3) Адаптировать TTS модели и обучить их;
- 4) Провести тестирование FastSpeech моделей по объективной оценке времени обучения, времени и ресурсов, затраченных на обучение и реализацию и субъективной оценке качеством синтеза голоса,;
- 5) Выполнить сравнительный анализ полученных результатов, сформулировав рекомендации по выбору модели в зависимости от конкретных прикладных требований.

## **ТРЕБОВАНИЯ К ПРОГРАММЕ**

### **Исходные требования к программе**

1. Требования к вводу данных
  - a. Формат текста - поддержка UTF-8 (кириллица, латиница и символы пунктуации);
  - b. Предобработка текста - токенизация и нормализация.
2. Требование к обработке данных
  - a. Формат мел-спектрограмм: определенная размерность и нормализация;
  - b. Словарь фонем должен соответствовать словарю MFA для корректной работы MFA.

## РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

Таблица 1 - Распределение ролей в бригаде.

Задача	Исполнитель
Предобработка данных для FastSpeech и FastSpeech2	Копасова Ксения, Самойлова Екатерина
Реализация и обучение FastSpeech	Самойлова Екатерина
Реализация и обучение FastSpeech2	Копасова Ксения
Анализ моделей	Копасова Ксения, Самойлова Екатерина
Написание отчета	Копасова Ксения, Самойлова Екатерина

## ТЕОРЕТИЧЕСКАЯ СПРАВКА

Начнем с того, что модели синтеза речи называют Text-to-speech (TTS) - это технология, позволяющая обработать письменный текст в естественную человеческую речь. Современные TTS-системы используют нейросетевые архитектуры (трансформенные модели с механизмами внимания; генеративные состязательные сети; диффузионные модели; многоуровневые кодеры). Первым прорывом в TTS были модели WaveNet [1] - первая глубокая генеративная модель, синтезирующая речь на уровне сэмплов и Tacotron [2] - архитектура на основе seq2seq [3] (архитектура нейронных сетей, которая преобразует последовательности одного типа в другую) с механизмом внимания, генерирующая мел-спектрограммы (временно-частотное представление звука) из текста. Но эти модели имели недостаток - генерация речи происходила последовательно что замедляло синтез.

В 2019 году Microsoft Research представила FastSpeech [4] - модель, которая генерировала речь сразу целиком, а не по частям. Основу системы составила трансформерная архитектура с механизмами внимания (self-attention) [5]. Также был создан специальный модуль для предсказания длительности фонем: он обучался на данных, полученных от Tacotron, что позволило сохранить естественный ритм и плавность речи. С помощью регулирования предсказанных длительностей стало возможным гибкое управление скоростью синтеза у FastSpeech. В экспериментах модель показала 38-кратное ускорение по сравнению с прошлыми моделями, при этом сохранив высокое качество звучания.

В 2020 году вышла усовершенствованная версия - FastSpeech 2 [6], значительно улучшила качество и скорость синтеза речи. Модель устранила зависимость от учительской модели Tacotron за счет прямого использования выравнивания текста и аудио для предсказания длительностей и извлечения высоты тона и энергии из эталонных аудиозаписей. Также был введен адаптор вариативности, объединяющий три независимых предсказателя: для



длительности фонем, высоты тона и энергии. Это позволило избежать ошибок выравнивания, характерных для механизмов влияния в FastSpeech 1.

## ОСОБЕННОСТИ РЕАЛИЗАЦИИ

### Предобработка данных для модели FastSpeech

Предобработка данных — это фундаментальный этап, от которого зависит качество синтезированной речи. FastSpeech, в отличие от авторегрессивных моделей (Tacotron), требует точного выравнивания текста и аудио, чтобы правильно предсказывать длительности фонем и генерировать плавную, естественную речь.

*Рассмотрим ключевые этапы обработки данных:*

#### 1. Подготовка датасета:

FastSpeech обучается на парных данных: текст - аудио. Скачиваем русский датасет (текст и аудио) для дальнейшего обучения на нем.

#### 2. Текстовая обработка:

Сначала извлекаем все уникальные русские слова из датасета и сохраняем их в нижнем регистре, чтобы избежать дублирования.

Затем код загружает заранее подготовленный фонетический словарь MFA, где каждому слову соответствует его транскрипция в фонемах. Словарь читается построчно, каждая запись разбивается на слово и соответствующую ему транскрипцию.

На последнем этапе происходит сопоставление: для каждого уникального слова из датасета проверяется его наличие в фонетическом словаре. Если слово найдено, оно вместе со своей транскрипцией записывается в итоговый файл. В результате получается очищенный набор слов с правильными фонетическими транскрипциями, готовый для использования в модели синтеза речи.

#### 3. Обработка аудио:

Исходные записи — это волны звуковых колебаний, которые необходимо привести в форму, понятную нейросети. Для этого каждый аудиофайл преобразуется в мел-спектрограмму - особый вид частотно-временного графика, который отражает, как энергия звука распределяется по разным частотам.

Особенность мел-спектрограмм в том, что они используют мел-шкалу — нелинейную шкалу частот, которая приближена к тому, как звук воспринимает человеческое ухо. Это позволяет модели фокусироваться на самых важных для понимания речи частотах.

#### 4. Выравнивание текста и аудио:

FastSpeech требует точных длительностей фонем — сколько кадров аудио соответствует каждому символу/фонеме. Для этого используем Montreal Forced Aligner (MFA) [7] — инструмент для автоматического выравнивания, который анализирует волновую форму и «привязывает» каждый звук к конкретной букве или фонеме в тексте. На выходе получаем точные временные метки для всех звуков.

#### 5. Разделение данных и сохранение:

Данные разделяются на три набора: обучающий (90%), валидационный (5%) и тестовый (5%). Для хранения используются оптимальные бинарные форматы: мел-спектрограммы и длительности фонем сохраняются в .npy (NumPy), текстовые токены - в .txt с числовыми идентификаторами.

В директории content/ размещаются:

- 1) mel/ - предобработанные мел-спектрограммы;
- 2) durations/ - данные о длительностях фонем
- 3) data\_for\_MFA/ - токенизированные текстовые представления и аудио для обучения MFA;
- 4) metadata.json - файл соответствий между элементами;

Такая организация обеспечивает быстрый доступ к данным при обучении и удобную навигацию по файлам

## **Предобработка данных для модели FastSpeech 2**

FastSpeech 2, в отличие от FastSpeech, требует выровненных данных для трех ключевых компонентов:

- 1) Точных длительностей фонем (чтобы знать, сколько кадров аудио соответствует каждой звуковой единице);

- 2) Pitch-контуров (для естественной интонации);
- 3) Энергии (для правильных акцентов и ритма).

Без тщательной предобработки модель будет спотыкаться на границах слов, потеряет естественную мелодику речи и может генерировать разрывы или шумы.

Рассмотрим ключевые этапы обработки данных:

1. Подготовка датасета: аналогично FastSpeech;
2. Текстовая обработка: аналогично FastSpeech;
3. Обработка аудио: аналогично FastSpeech;
4. Выравнивание текста и аудио: аналогично FastSpeech;
5. Вычисление значений основного тона (pitch):

Pitch (частота основного тона, F0) — это базовая частота колебаний голосовых связок при произнесении звука. Она определяет высоту голоса и измеряется в герцах (Hz). PyWorld [8] извлекает питч (частоту основного тона F0) с помощью двухэтапного алгоритма. Сначала применяется метод DIO [9], который использует динамическое программирование для анализа резонансных частот голосового тракта - он быстро находит грубую оценку F0 через автокорреляцию сигнала, вычисляя период повторения звуковой волны. Затем алгоритм Harvest [10] уточняет эту оценку, особенно эффективно работая с нестабильным или дрожащим голосом, анализируя мелкозернистые изменения в сигнале.

Полученные значения F0 проходят нормализацию: сначала логарифмирование ( $\log(F0)$ ), чтобы уменьшить разброс значений между низкими и высокими тонами, а затем масштабирование.

#### 6. Вычисление энергии:

Энергия сигнала показывает громкость звука в каждом временном отрезке записи. Она вычисляется как сумма квадратов амплитуд звуковой волны в пределах короткого кадра (обычно 256 сэмплов). Для нормализации энергию делят на максимальное значение в датасете (получая диапазон 0-1). Эти

нормализованные значения помогают модели правильно расставлять акценты и регулировать громкость при синтезе речи.

#### 7. Разделение данных и сохранение:

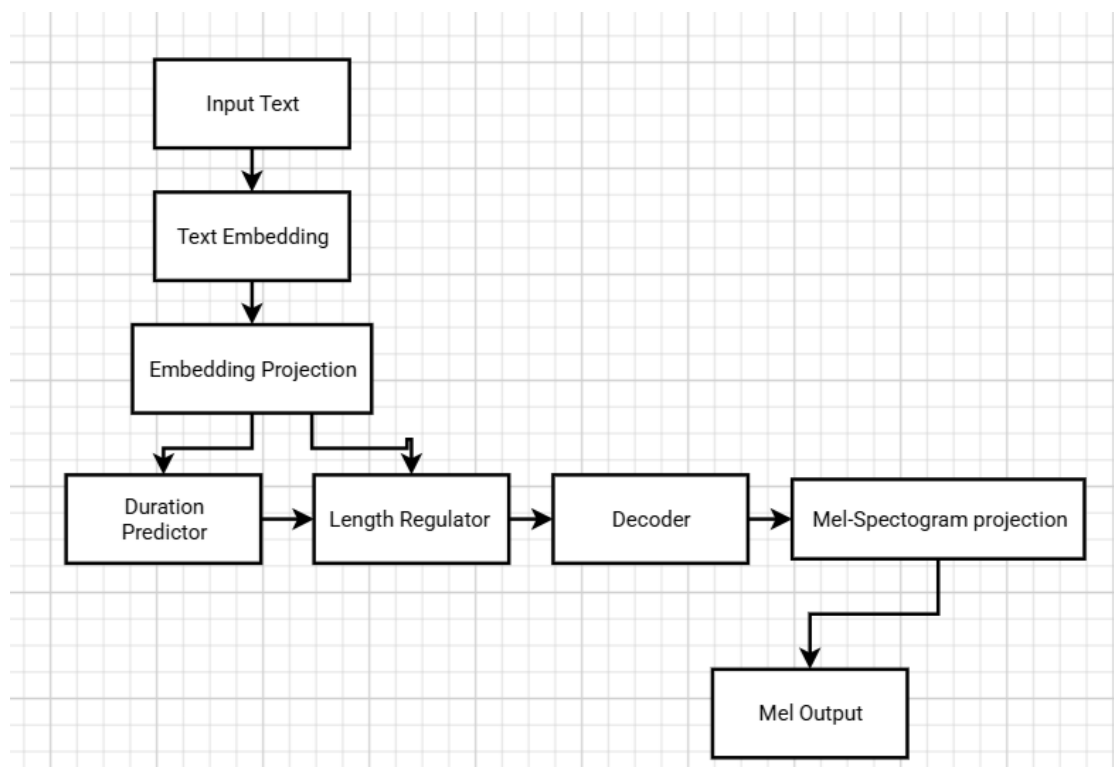
Данные разделяются и сохраняются аналогично, только в директории content/ размещаются еще две позиции:

- 1) pitch/ - вычисленные питчи;
- 2) energy/ - вычисленная энергия.

### **Реализация и обучение модели FastSpeech**

Модель FastSpeech1 представляет собой нейронную систему синтеза речи, которая использует трансформеры и предсказание длительности фонем для генерации мел-спектрограмм. В отличие от автогрессивных моделей, таких как Tacotron, она работает быстрее и стабильнее за счёт отказа от attention-механизма. В оригинальной реализации FastSpeech используются трансформеры с слоях энкодера и декодера, однако в нашей реализации используется лишь идея, то есть слои в данной модели дают схожее качество, что позволяет уменьшить количество вычислений и ускорить обучение.

Рисунок 1 - модель FastSpeech1



На вход модель получает последовательность фонем, которые преобразуются в эмбединги и подаются в энкодер. Энкодер состоит из нескольких подряд идущих блоков и каждый блок включает: Conv1d — обрабатывает последовательность (по сути, заменяет self-attention), BatchNorm1d — стабилизирует и ускоряет обучение, ReLU — добавляет нелинейность, Dropout — предотвращает переобучение.

После получения текстовых токенов специальный модуль DurationPredictor предсказывает, сколько временных фреймов (примерно: сколько кадров мел-спектрограммы) должна занимать каждая фонема. Одна фонема соответствует нескольким спектрограммам. С помощью регулятора длины можно корректировать длительность фонемы, чтобы изменять скорость голоса и паузы между словами.

Далее работает модуль LengthRegulator, который расширяет скрытое представление последовательности: эмбединги дублируются в соответствии с предсказанными длительностями. Это позволяет согласовать длину текстовой и аудиопоследовательности без применения attention-механизмов.

Полученная последовательность обрабатывается простым декодером на основе полносвязных слоёв, после чего результат проходит через линейный слой, формируя мел-спектрограмму — выход модели.

Во время обучения модель оптимизируется с помощью составной функции потерь FastSpeech1Loss, которая одновременно минимизирует абсолютную ошибку между предсказанными и целевыми мел-спектрограммами (таблица значений, отображающая интервалы частот, временные фреймы и амплитуды) и среднеквадратичную ошибку (MSE) между логарифмами предсказанных и реальных длительностей фонем, при этом паддинговые позиции исключаются через соответствующую маску. Чтобы предотвратить переобучение, используется механизм EarlyStopping, прерывающий тренировку при отсутствии улучшения валидационной ошибки в течение нескольких эпох, а при выходе на плато валидационной кривой автоматически снижается скорость обучения через ReduceLROnPlateau.

На последнем этапе она подаётся на vocoder (в нашей реализации используется HiFi-GAN), который преобразует её в аудиосигнал.

FastSpeech1 обеспечивает более стабильный и быстрый синтез речи в сравнении, например, с Tacotron, а также позволяет управлять скоростью воспроизведения путём масштабирования длительностей.

## **Реализация и обучение модели FastSpeech 2**

Особенности FastSpeech 2 были рассмотрены в теоретической справке. Для удобства реализуем его упрощенную версию без трансформеров.

Модель начинается со слоя эмбединга, который преобразует символы текста в векторные представления, после чего линейная проекция адаптирует их размерность для энкодера. Сверточный энкодер состоит из последовательности блоков, каждый из которых включает 1D-свертку с симметричным padding, Batch Normalization, активацию ReLU и Dropout.

Для управления просодикой (совокупность характеристик звучания) в модели используются три независимых предиктора дисперсии: для длительностей фонем, значений основного тона (pitch) и энергии. Каждый из них представляет собой компактную сеть с двукратным уменьшением размерности. Предсказанные pitch и energy встраиваются в скрытое пространство через линейные преобразования и аддитивно объединяются с основным потоком данных.

Регулятор длины расширяет временную последовательность в соответствии с предсказанными длительностями фонем. Декодер состоит из четырех линейных слоев с ReLU и Dropout, сохраняя постоянную размерность на всех этапах. Завершающий слой проецирует скрытые представления в мел-спектрограмму, которая является выходом модели.

Для обучения модели был подготовлен датасет, содержащий:

- 1) Нормализованные мел-спектрограммы (mel);
- 2) Выровненные по времени pitch-контуры (pitch);
- 3) Рассчитанные значения энергии для каждого речевого сегмента (energy);

- 4) Токенизированные текстовые транскрипции в фонемном представлении (text);
- 5) Длительности фонем (duration).

Процесс обучения осуществлялся с применением оптимизатора Adam с начальным learning rate  $1e-4$  в течение первых 5000 шагов. В качестве функции потерь использовалась комбинация нескольких компонент:

- 1) Средняя абсолютная ошибка (MAE) для мел-спектрограмм;
- 2) Функция потерь Хьюбера (SmoothL1Loss) для pitch-значений;
- 3) Функция потерь Хьюбера (SmoothL1Loss) для энергетических характеристик;
- 4) MSE для длительностей фонем.

Также в течении обучения был контроль переобучения.

### **Методы оценивания моделей**

Акустические модели FastSpeech 1 и FastSpeech 2 оценивают по нескольким ключевым метрикам. В первую очередь измеряют ошибку между синтезированными и оригинальными мел-спектрограммами (например, L1 или L2 loss, Mel Cepstral Distortion). Также оценивают точность предсказания длительностей фонем — обычно через среднеквадратичную ошибку или корреляцию.

FastSpeech 2 дополнительно предсказывает частоту основного тона (F0) и энергию, что улучшает выразительность. Для этих параметров тоже рассчитывают ошибки, например RMSE для F0.

Помимо объективных метрик, важна субъективная оценка — MOS (Mean Opinion Score), где люди оценивают качество речи, и ABX-тесты для сравнения моделей. FastSpeech 2 обычно показывает лучшие результаты по качеству звучания и точности предсказаний за счет выделения дополнительных характеристик и более сложной модели.



## **Анализ моделей**

Обучение прототипов моделей FastSpeech 1 и FastSpeech 2 проводилось на 15 и 10 эпохах соответственно, поскольку целью было получить общее представление о работе этих моделей. Несмотря на упрощенную тренировку, уже на этом этапе можно провести сравнение представленных моделей.

Обучение FastSpeech 1 занимает примерно 1 минуту за эпоху, тогда как FastSpeech 2 — около 20 минут. Это связано с более сложной архитектурой и дополнительными параметрами в FastSpeech 2, из чего следует, что по времени и ресурсам FastSpeech 1 гораздо экономичнее.

Что касается качества синтеза, FastSpeech 2 генерирует больше звуковых характеристик, что улучшает качество аудио по сравнению с первой версией.

## ТЕСТИРОВАНИЕ

### Тестирование модели FastSpeech

Таблица 2 - Результаты обучения модели FastSpeech

Epoch	Loss	Mel Loss	Duration Loss	Validation Loss	Time
0	1.26	0.164	1.1	0.965108	01:24
1	1.01	0.0965	0.913	0.937216	01:22
2	1.49	0.102	1.39	0.906427	01:23
3	1.36	0.149	1.21	0.911387	01:23
4	0.984	0.127	0.857	0.893791	01:23
5	0.934	0.108	0.826	0.883095	01:22
6	1.32	0.124	1.2	0.883301	01:23
7	0.642	0.146	0.496	0.880247	01:24
8	0.799	0.11	0.689	0.887387	01:22
9	1.53	0.147	1.38	0.888857	01:22
10	0.852	0.0769	0.775	0.884705	01:22
11	0.858	0.104	0.753	0.881782	01:24
12	1.43	0.135	1.3	0.891582	01:21
13	1.07	0.139	0.929	0.885919	01:23
14	1.12	0.0871	1.03	0.89529	01:22

#### [Пример аудио по модели FastSpeech1.](#)

Текст: привет это генерация текста.

Субъективная оценка качества аудио по шкале от 1 до 5: 1.33 - набор звуков, отсутствует сходство с человеческой речью, однако звук немонотонный, в аудио присутствуют паузы соответствующие предложению.

## Тестирование модели FastSpeech 2

Таблица 2 - Результаты обучения модели FastSpeech2

Epoch	Loss	Mel Loss	Duration Loss	Validation Loss	Pitch Loss	Energy Loss	Time
0	1.26	0.0664	0.936	1.36118	0.133	0.128	15:07
1	1.36	0.0786	0.796	1.35275	0.27	0.212	14:56
2	1.13	0.0695	0.765	1.35314	0.16	0.137	14:41
3	1.23	0.073	0.754	1.35046	0.236	0.169	14:59

### [Пример аудио по модели FastSpeech2.](#)

Текст: привет это генерация текста.

Субъективная оценка качества аудио по шкале от 1 до 5: 1.66 - набор звуков, сходство с человеческой речью практически отсутствует, однако присутствуют паузы, некоторые звуки более высокие, появляется подобие тембра.

## ЗАКЛЮЧЕНИЕ

В ходе работы был проведён сравнительный анализ двух TTS-моделей - FastSpeech и FastSpeech 2 - для выявления их преимуществ и ограничений в контексте различных прикладных задач, с акцентом на три ключевых аспекта: скорость обучения, качество синтеза и требования к вычислительным ресурсам.

Были решены следующие задачи:

1. Проведено теоретическое исследование моделей:

Выявлены их различия в подходах к предсказанию длительностей и просодических характеристик.

2. Подобран и предобработан датасет аудиозаписей:

Использован русский датасет, над которым была проведена нормализация, разметка текста, извлечения мел-спектрограмм и forced alignment (для FastSpeech 2).

3. Адаптированы и обучены TTS-модели:

FastSpeech 1 обучен в среднем за 20 минут при 15 эпохах, средняя ошибка – 1,111, средняя ошибка на валидационных данных – 0,898.

FastSpeech 2 потребовал 59 минут при 4 эпохах, средняя ошибка – 1,245, средняя ошибка на валидационных данных – 1,354.

4. Выполнено тестирование моделей:

Были созданы программы, которые реализовали синтез речи через FastSpeech и FastSpeech 2 и HiFi-GAN [11]. В результате получены сгенерированные аудиофайлы.

5. Проведен сравнительный результат:

Выбор между этими моделями определяется конкретными требованиями: FastSpeech — для задач, которым важна быстрая скорость обучения, работы и малые затраты на ресурсы, FastSpeech 2 — для задач, где приоритетом является качество голосового синтеза.

В дальнейшем планируется оптимизировать предобработку; изменить структуру моделей и использовать в них трансформеры также, как в

оригинальных моделях; обучить их на большем количестве данных с увеличением числа эпох и получить на выходе аудиозапись с оценкой 5-7.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Van den Oord A., Dieleman S., Zen H., Simonyan K., Vinyals O., Graves A. и др. WaveNet: A Generative Model for Raw Audio // arXiv preprint arXiv:1609.03499. — 2016. — Режим доступа: <https://arxiv.org/abs/1609.03499>
- [2] Wang Y., Skerry-Ryan R., Stanton D., Wu Y., Weiss R. J., Jaitly N. и др. Tacotron: Towards End-to-End Speech Synthesis // arXiv preprint arXiv:1703.10135. — 2017. — Режим доступа: <https://arxiv.org/abs/1703.10135>
- [3] Sutskever I., Vinyals O., Le Q. V. Sequence to Sequence Learning with Neural Networks // arXiv preprint arXiv:1409.3215. — 2014. — Режим доступа: <https://arxiv.org/abs/1409.3215>
- [4] Wang Y., Skerry-Ryan R., Stanton D., Wu Y., Weiss R. J., Jaitly N. и др. FastSpeech: Fast, Robust and Controllable Text to Speech // arXiv preprint arXiv:1905.09263. — 2019. — Режим доступа: <https://arxiv.org/abs/1905.09263>
- [5] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N. и др. Attention Is All You Need // arXiv preprint arXiv:1706.03762. — 2017. — Режим доступа: <https://arxiv.org/abs/1706.03762>
- [6] Ren Y., Hu C., Tan X., Qin T., Zhao S., Zhao Z., et al. FastSpeech 2: Fast and High-Quality End-to-End Text to Speech // arXiv preprint arXiv:2006.04558. — 2020. — Режим доступа: <https://arxiv.org/abs/2006.04558>
- [7] McAuliffe M., Socolof M., Mihuc S., Wagner M., Sonderegger M. Montreal Forced Aligner: Trainable Text-Speech Alignment Using Kaldi // Proceedings of Interspeech. — 2017. — Режим доступа: [https://www.isca-archive.org/interspeech\\_2017/mcauliffe17\\_interspeech.html](https://www.isca-archive.org/interspeech_2017/mcauliffe17_interspeech.html)
- [8] Morise M. PyWorldVocoder: Python wrapper for WORLD vocoder [Компьютерная программа]. — Версия 0.3.2. — 2021. Режим доступа: <https://github.com/JeremyCCHsu/Python-Wrapper-for-World-Vocoder>
- [9] Morise M., Kawahara H. DIO: A fundamental frequency estimation algorithm for speech signals based on temporal domain analysis

// IEICE Transactions on Information and Systems. — 2012. — Vol. E95-D, № 4. —  
 P. 1301–1308. — Режим доступа:  
[https://www.jstage.jst.go.jp/article/transinf/E95.D/5/E95.D\\_5\\_1301/\\_article](https://www.jstage.jst.go.jp/article/transinf/E95.D/5/E95.D_5_1301/_article)

[10] Morise M. Harvest: A high-performance F0 estimator for speech analysis  
 // Acoustical Science and Technology. — 2016. — Vol. 37, № 7. — P. 399–406. DOI:  
 10.1250/ast.37.399

[11] Kong J., Kim J., Bae J. HiFi-GAN: Generative Adversarial Networks for Efficient  
 and High Fidelity Speech Synthesis // arXiv preprint arXiv:2010.05646. — 2020. —  
 Режим доступа: <https://arxiv.org/abs/2010.05646>

## ПРИЛОЖЕНИЕ А

### ПРОГРАММНЫЙ КОД

FastSpeech1:  
[https://colab.research.google.com/drive/1WwXVevb78-HKS1q-Qc\\_hSwibjytmxqUw#scrollTo=HguS2KNE1j9U](https://colab.research.google.com/drive/1WwXVevb78-HKS1q-Qc_hSwibjytmxqUw#scrollTo=HguS2KNE1j9U)

[https://colab.research.google.com/drive/1WwXVevb78-HKS1q-Qc\\_hSwibjytmxqUw#scrollTo=HguS2KNE1j9U](https://colab.research.google.com/drive/1WwXVevb78-HKS1q-Qc_hSwibjytmxqUw#scrollTo=HguS2KNE1j9U)

FastSpeech2:  
<https://colab.research.google.com/drive/1EZR6n1IHvTLpAxus17p93HZngXSQmd36?usp=sharing>

Файл test\_FS.py:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import yaml
from hifi_gan.models import Generator
import json
import soundfile as sf
import librosa
import numpy as np
import sys
from huggingface_hub import hf_hub_download

class LengthRegulator(nn.Module):
    """Length Regulator"""
    def __init__(self):
        super(LengthRegulator, self).__init__()

    def forward(self, x, duration, max_len=None):
        output = []
        for x_i, d_i in zip(x, duration):
            expanded = self.expand(x_i, d_i)
            output.append(expanded)

        if max_len is not None:
            output = [self.pad_sequence(o, max_len) for o in output]

        output = torch.stack(output)
        return output

    def expand(self, x, d):
        if d.sum() == 0:
            print("Warning: All durations are zero! Setting to minimum duration of 1")
            d = torch.ones_like(d)

        d = d.long().clamp(min=1) # Ensure durations are at least 1
        d_cumsum = torch.cumsum(d, dim=0)
        d_total = d_cumsum[-1].item()

        expanded = torch.zeros(
            (int(d_total), x.size(1)),
            dtype=x.dtype,
            device=x.device
        )

        for i in range(x.size(0)):
            t_s = d_cumsum[i-1] if i > 0 else 0
            t_e = d_cumsum[i]
            t_len = t_e - t_s
```



```

        expanded[t_s:t_e] = x[i].unsqueeze(0).expand(t_len, -1)

    return expanded

def pad_sequence(self, x, max_len):
    if x.size(0) >= max_len:
        return x[:max_len]
    return torch.cat([x, torch.zeros((max_len - x.size(0), x.size(1)),
device=x.device)])

class FastSpeech1(nn.Module):
    def __init__(self, config):
        super(FastSpeech1, self).__init__()
        model_config = config["model"]

        # Text embedding
        self.embedding = nn.Embedding(
            model_config["n_symbols"],
            model_config["symbols_embedding_dim"],
            padding_idx=0
        )

        # Projection to encoder dimension
        self.embedding_proj = nn.Linear(
            model_config["symbols_embedding_dim"],
            model_config["encoder_embedding_dim"]
        )

        # Encoder
        self.encoder_convs = nn.ModuleList([
            nn.Sequential(
                nn.Conv1d(
                    model_config["encoder_embedding_dim"],
                    model_config["encoder_embedding_dim"],
                    model_config["encoder_kernel_size"],
                    padding=(model_config["encoder_kernel_size"]-1)//2,
                    dilation=1
                ),
                nn.BatchNorm1d(model_config["encoder_embedding_dim"]),
                nn.ReLU(),
                nn.Dropout(model_config["dropout"])
            )
            for _ in range(model_config["encoder_n_convolutions"])
        ])

        # Duration Predictor
        self.duration_predictor = nn.Sequential(
            nn.Linear(model_config["encoder_embedding_dim"],
                model_config["duration_predictor_filters"]),
            nn.ReLU(),
            nn.Dropout(model_config["dropout"]),
            nn.Linear(model_config["duration_predictor_filters"], 1)
        )

        # Length regulator
        self.length_regulator = LengthRegulator()

        # Decoder
        self.decoder_layers = nn.ModuleList([
            nn.Sequential(
                nn.Linear(model_config["encoder_embedding_dim"],
                    model_config["encoder_embedding_dim"]),
                nn.ReLU(),
                nn.Dropout(model_config["dropout"])
            )
        ])

```

```

        for _ in range(4)
    ])

    # Final projection to mel-spectrogram
    self.mel_proj = nn.Linear(
        model_config["encoder_embedding_dim"],
        model_config["n_mel_channels"]
    )

def forward(self, text, src_mask=None, mel_mask=None,
            duration_target=None, max_len=None):
    # Input text: [batch_size, text_seq_len]

    # 1. Text embedding
    x = self.embedding(text)
    x = self.embedding_proj(x)

    # 2. Encoder processing
    for conv in self.encoder_convs:
        x = conv(x.transpose(1, 2)).transpose(1, 2)

    # 3. Duration prediction
    log_duration = self.duration_predictor(x).squeeze(-1)

    if duration_target is not None:
        duration = duration_target
    else:
        duration = torch.exp(log_duration).clamp(min=1.0) # Ensure minimum
duration of 1

        # Apply src_mask if available
        if src_mask is not None:
            duration = duration * src_mask.squeeze(1).float()

    # 4. Length regulation
    x = self.length_regulator(x, duration, max_len)

    # 5. Decoder processing
    for layer in self.decoder_layers:
        x = layer(x)

    # 6. Mel-spectrogram projection
    mel_output = self.mel_proj(x)

    return {
        "mel_output": mel_output,
        "duration_predicted": log_duration
    }

class AttrDict(dict):
    def __init__(self, *args, **kwargs):
        super(AttrDict, self).__init__(*args, **kwargs)
        self.__dict__ = self

def audio_to_mel(audio_path, sr=22050, n_mels=80):
    # Загружаем аудио
    y, sr = librosa.load(audio_path, sr=sr) # y - аудиосигнал, sr - частота

    # Вычисляем мел-спектрограмму
    mel = librosa.feature.melspectrogram(
        y=y,
        sr=sr,
        n_fft=1024,
        hop_length=256,

```

```

        win_length=1024,
        n_mels=n_mels,
        fmin=0,
        fmax=8000,
    )

    # Переводим в логарифмическую шкалу (dB)
    mel_db = librosa.power_to_db(mel, ref=np.max)
    # Нормализуем
    mel_db = (mel_db - mel_db.mean()) / mel_db.std()
    # Преобразуем в тензор [1, n_mels, T]
    mel_tensor = torch.from_numpy(mel_db).unsqueeze(0).float()
    return mel_tensor

class FastSpeechTester:
    def __init__(self, model_path, config_path, symbols_path="symbols.json"):
        self.device = 'cpu'

        # Load config
        with open(config_path, "r") as f:
            self.config = yaml.safe_load(f)

        # Load symbols
        with open(symbols_path, 'r', encoding='utf-8') as f:
            self.symbol_to_id = json.load(f)

        # Initialize FastSpeech1
        self.model = FastSpeech1(self.config).to(self.device)
        checkpoint = torch.load(model_path, map_location=self.device)
        self.model.load_state_dict(checkpoint['model'])
        self.model.eval()

        # Initialize HiFi-GAN
        with open('config_v1.json') as f:
            vocoder_config = AttrDict(json.load(f))

        self.vocoder = Generator(vocoder_config).to(self.device)
        self.vocoder.eval()
        self.vocoder.remove_weight_norm()

    def text_to_ids(self, text):
        # Add start and end tokens if needed
        ids = [self.symbol_to_id.get(c, self.symbol_to_id.get(' ', 0)) for c in
            text.lower()]
        return torch.LongTensor(ids).unsqueeze(0).to(self.device)

    def synthesize(self, text, speed=1.0):
        input_ids = self.text_to_ids(text)
        src_mask = (input_ids != 0).unsqueeze(1)

        with torch.no_grad():
            # Generate mel-spectrogram
            output = self.model(input_ids, src_mask=src_mask)
            # mel = output["mel_output"]
            mel = audio_to_mel("sample_10.wav", sr=22050, n_mels=80) # [B, T,
n_mels]

            # Check mel output
            if mel.size(1) == 0:
                raise ValueError("Empty mel-spectrogram generated. Check duration
prediction.")

            print(f"Mel shape: {mel.shape}")

```

```

        print(f"Mel stats - min: {mel.min().item():.2f}, max: {mel.max().item():.2f}, mean: {mel.mean().item():.2f}")

        # Transpose for HiFi-GAN [B, n_mels, T]
        # mel = mel.transpose(1, 2)

        # Normalize
        mel = (mel - mel.mean()) / (mel.std() + 1e-8)

        # Synthesize audio
        audio = self.vocoder(mel).squeeze()
        audio = audio / torch.max(torch.abs(audio)) * 0.9
        audio = audio.cpu().numpy()

    return audio

tester = FastSpeechTester(
    model_path="final_model.pth",
    config_path="config_FS.yaml",
    symbols_path="symbols.json"
)

# Test synthesis
text = "привет это генерация текста"
audio = tester.synthesize(text)

# Save output
sf.write('output_FS.wav', audio, 22050)

Файл test_FS2.py:
import torch
import torch.nn as nn
import torch.nn.functional as F
import yaml
from hifi_gan.models import Generator
import json
import soundfile as sf
import librosa
import numpy as np
import os

# Загрузка конфигурации
with open("config.yaml", "r") as f:
    config = yaml.safe_load(f)

class LengthRegulator(nn.Module):
    """Length Regulator"""
    def __init__(self):
        super(LengthRegulator, self).__init__()

    def forward(self, x, duration, max_len=None):
        output = []
        for x_i, d_i in zip(x, duration):
            expanded = self.expand(x_i, d_i)
            output.append(expanded)

        if max_len is not None:
            output = [self.pad_sequence(o, max_len) for o in output]

        output = torch.stack(output)
        return output

    def expand(self, x, duration):

```

```

# Преобразуем duration в список целых чисел
if isinstance(duration, torch.Tensor):
    duration = duration.int().tolist() # Например, [2, 3, 1, ...]

d_total = sum(duration)
feat_dim = x.size(1)

# Создаём нулевой тензор
expanded = torch.zeros((d_total, feat_dim), device=x.device)

t_s = 0
for i, d_i in enumerate(duration):
    t_e = t_s + d_i
    t_len = d_i # Длина текущего сегмента

    # Если t_len – тензор, преобразуем в int
    if isinstance(t_len, torch.Tensor):
        t_len = t_len.item() # Важно: делаем int!

    # Расширяем x[i] и копируем в expanded
    expanded[t_s:t_e] = x[i].unsqueeze(0).expand(t_len, -1)
    t_s = t_e

return expanded

def pad_sequence(self, x, max_len):
    if x.size(0) >= max_len:
        return x[:max_len]
    return torch.cat([x, torch.zeros((max_len - x.size(0), x.size(1)),
device=x.device)])

class VariancePredictor(nn.Module):
    """Variance Predictor for pitch and energy"""
    def __init__(self, input_dim, filter_size, kernel_size, dropout):
        super(VariancePredictor, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv1d(input_dim, filter_size, kernel_size,
padding=kernel_size//2),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv1d(filter_size, filter_size, kernel_size,
padding=kernel_size//2),
            nn.ReLU()
        )
        self.linear = nn.Linear(filter_size, 1)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.conv1(x.transpose(1, 2))
        x = self.dropout(x)
        x = self.conv2(x)
        x = self.dropout(x)
        x = self.linear(x.transpose(1, 2))
        return x.squeeze(-1)

class FastSpeech2(nn.Module):
    """FastSpeech 2 Model with fixed dimension handling"""
    def __init__(self, config):
        super(FastSpeech2, self).__init__()

        model_config = config["model"]

```

```

# Эмбединги текста
self.embedding = nn.Embedding(
    model_config["n_symbols"],
    model_config["symbols_embedding_dim"],
    padding_idx=0
)

# Проекция на размер кодера
self.embedding_proj = nn.Linear(
    model_config["symbols_embedding_dim"],
    model_config["encoder_embedding_dim"]
)

# Энкодер
self.encoder_convs = nn.ModuleList([
    nn.Sequential(
        nn.Conv1d(
            model_config["encoder_embedding_dim"],
            model_config["encoder_embedding_dim"],
            model_config["encoder_kernel_size"],
            padding=(model_config["encoder_kernel_size"]-1)//2,
            dilation=1
        ),
        nn.BatchNorm1d(model_config["encoder_embedding_dim"]),
        nn.ReLU(),
        nn.Dropout(0.1)
    )
    for _ in range(model_config["encoder_n_convolutions"])
])

# Предикторы дисперсии
self.duration_predictor = VariancePredictor(
    model_config["encoder_embedding_dim"],
    model_config["encoder_embedding_dim"] // 2,
    3, 0.1
)
self.pitch_predictor = VariancePredictor(
    model_config["encoder_embedding_dim"],
    model_config["encoder_embedding_dim"] // 2,
    3, 0.1
)
self.energy_predictor = VariancePredictor(
    model_config["encoder_embedding_dim"],
    model_config["encoder_embedding_dim"] // 2,
    3, 0.1
)

# Pitch and energy эмбединги
self.pitch_embedding = nn.Linear(1,
model_config["encoder_embedding_dim"])
self.energy_embedding = nn.Linear(1,
model_config["encoder_embedding_dim"])

# Регулятор длины
self.length_regulator = LengthRegulator()

# Декодер с постоянными размерами
self.decoder_layers = nn.ModuleList([
    nn.Sequential(
        nn.Linear(model_config["encoder_embedding_dim"],
            model_config["encoder_embedding_dim"]),
        nn.ReLU(),
        nn.Dropout(0.1))
    for _ in range(4)
])

```

```

    ])

    # Финальная проекция на мел-спектрограмму
    self.mel_proj = nn.Linear(
        model_config["encoder_embedding_dim"],
        model_config["n_mel_channels"]
    )

    def forward(self, text, src_mask=None, mel_mask=None,
                duration_target=None, pitch_target=None, energy_target=None,
                max_len=None):
        # Входящий текст: [batch_size, text_seq_len]

        # 1. Эмбединги текста
        x = self.embedding(text) # [batch, text_seq_len, symbol_embed_dim]
        x = self.embedding_proj(x) # [batch, text_seq_len, encoder_dim]

        # 2. Обработка энкодера
        for conv in self.encoder_convs:
            x = conv(x.transpose(1, 2)).transpose(1, 2) # [batch, text_seq_len,
encoder_dim]

        # 3. Прогнозирование продолжительности
        log_duration = self.duration_predictor(x) # [batch, text_seq_len]

        if duration_target is not None:
            duration = duration_target
        else:
            duration = torch.exp(log_duration) - 1

        # 4. Регулирование длины
        x = self.length_regulator(x, duration, max_len) # [batch, mel_seq_len,
encoder_dim]

        # 5. Прогнозирование питча и встраивание
        pitch_pred = self.pitch_predictor(x) # [batch, mel_seq_len]
        if pitch_target is not None:
            pitch_emb = self.pitch_embedding(pitch_target.unsqueeze(-1))
        else:
            pitch_emb = self.pitch_embedding(pitch_pred.unsqueeze(-1))
        x = x + pitch_emb # [batch, mel_seq_len, encoder_dim]

        # 6. Прогнозирование энергии и встраивание
        energy_pred = self.energy_predictor(x) # [batch, mel_seq_len]
        if energy_target is not None:
            energy_emb = self.energy_embedding(energy_target.unsqueeze(-1))
        else:
            energy_emb = self.energy_embedding(energy_pred.unsqueeze(-1))
        x = x + energy_emb # [batch, mel_seq_len, encoder_dim]

        # 7. Обработка декодера
        for layer in self.decoder_layers:
            x = layer(x) # [batch, mel_seq_len, encoder_dim]

        # 8. Проекция мел-спектрограммы
        mel_output = self.mel_proj(x) # [batch, mel_seq_len, n_mels]

        return {
            "mel_output": mel_output,
            "duration_predicted": log_duration,
            "pitch_predicted": pitch_pred,
            "energy_predicted": energy_pred
        }

```

```

class AttrDict(dict):
    def __init__(self, *args, **kwargs):
        super(AttrDict, self).__init__(*args, **kwargs)
        self.__dict__ = self

def audio_to_mel(audio_path, sr=22050, n_mels=80):
    # Загружаем аудио
    y, sr = librosa.load(audio_path, sr=sr) # y - аудиосигнал, sr - частота

    # Вычисляем мел-спектрограмму
    mel = librosa.feature.melspectrogram(
        y=y,
        sr=sr,
        n_fft=1024,
        hop_length=256,
        win_length=1024,
        n_mels=n_mels,
        fmin=0,
        fmax=8000,
    )

    # Переводим в логарифмическую шкалу (dB)
    mel_db = librosa.power_to_db(mel, ref=np.max)
    # Нормализуем
    mel_db = (mel_db - mel_db.mean()) / mel_db.std()
    # Преобразуем в тензор [1, n_mels, T]
    mel_tensor = torch.from_numpy(mel_db).unsqueeze(0).float()
    return mel_tensor

class FastSpeech2Tester:
    def __init__(self, model_path, config_path="config.yaml",
symbols_path="symbols.json"):
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'

        # Загрузка конфигурации
        with open(config_path, "r") as f:
            self.config = yaml.safe_load(f)

        # Загрузка символов
        # Загрузка символов с учётом структуры файла
        with open(symbols_path, 'r', encoding='utf-8') as f:
            symbols_data = json.load(f)
            self.symbol_to_id = symbols_data["symbol_to_id"] # Получаем только
нужный словарь
            self.id_to_symbol = symbols_data["id_to_symbol"] # Сохраняем
обратное отображение, если нужно

        # Инициализация FastSpeech2
        self.model = FastSpeech2(self.config).to(self.device)
        checkpoint = torch.load(model_path, map_location=self.device)
        self.model.load_state_dict(checkpoint['model'])
        self.model.eval()

        # Инициализация HiFi-GAN
        with open('config_v1.json') as f:
            vocoder_config = AttrDict(json.load(f))
        self.vocoder = Generator(vocoder_config).to(self.device)
        self.vocoder.eval()

    def text_to_ids(self, text):

```



```

# Проверяем наличие пробела в словаре
if ' ' not in self.symbol_to_id:
    raise ValueError("Словарь символов должен содержать пробел (' ')")

return torch.LongTensor([
    self.symbol_to_id.get(c, self.symbol_to_id[' ']) # Заменяем
    неизвестные символы на пробел
    for c in text.lower()
]).unsqueeze(0).to(self.device)

def synthesize(self, text, speed=1.0):
    input_ids = self.text_to_ids(text)
    src_mask = (input_ids != 0).unsqueeze(1)

    with torch.no_grad():
        # Генерация Mel-спектрограммы FastSpeech2
        output = self.model(input_ids, src_mask=src_mask)
        mel = audio_to_mel("sample.wav", sr=22050, n_mels=80) # [B, T, n_mels]

        # Транспонируем для HiFi-GAN: [B, n_mels, T]
        # mel = mel.transpose(1, 2)

        # Нормализация (если не делается в модели)
        mel = (mel - mel.mean()) / mel.std()

        # Преобразование в аудио
        audio = self.vocoder(mel).squeeze()
        audio = audio / torch.max(torch.abs(audio)) * 0.9
        audio = audio.cpu().numpy()

    return audio

# Использование:
tester = FastSpeech2Tester(
    model_path="best_model.pth",
    config_path="config.yaml",
    symbols_path="symbols.json"
)

# Синтез аудио
text = "привет это генерация текста"
audio = tester.synthesize(text)

# Сохранение
sf.write('output_FS2.wav', audio, 22050) # Частота должна соответствовать vocoder

```

**ОТЗЫВ**  
**о прохождении учебной практики**

Копасова Ксения Андреевна и Самойлова Екатерина Михайловна, студенты группы 3384 второго курса бакалавриата Санкт-Петербургского электротехнического университета “ЛЭТИ” им В.И. Ульянова, проходили учебную практику по теме “Генерация голоса конкретного человека по записи его голоса” с 25.06.2025 по 08.07.2025 на кафедре МО ЭВМ.

Во время практики Копасова К. А. и Самойлова Е. М. провели анализ двух версий нейронной сети FastSpeech, поставили собственный эксперимент для получения количественных характеристик, а также смогли получить конкретные аудиофайлы с синтезированным голосом.

По результатам учебной практики Копасова К. А. и Самойлова Е. М. заслуживают оценки **ОТЛИЧНО**.

Подпись руководителя практики:

к.т.н., доцент кафедры МОЭВМ

08.07.2025



Филатов А. Ю.