МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»
Кафедра информатики и программирования

Исследование проблемы стилистического переноса текста между разговорным и художественным стилями с помощью моделей машинного обучения

КУРСОВАЯ РАБОТА

студента 2 курса, 241 группы
направления 02.03.03 «Математическое обеспечение и администрирование информационных систем»
факультета Компьютерных наук и информационных технологий
Петрова Егора Дмитриевича

Научный руководитель (руководитель)
зав. кафедрой _____ Огнёва М. В.
(должность) (подпись, дата) (ФИО)

Зав. кафедрой
канд. физ.-мат. наук, доцент _____ Огнёва М. В.
(звание) (подпись, дата) (ФИО)

Саратов 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Возможность преобразовать текст из разговорного стиля речи в художественный востребована в литературном творчестве, журналистике, маркетинге, а также в образовательной и культурной деятельности, что подтверждается растущим использованием ИИ технологий в этих сферах [1]. Преобразование текстов из разговорного стиля в художественный позволяет создавать контент, который лучше воспринимается аудиторией, повышает вовлечённость и качество взаимодействия. Автоматизация этого процесса позволит людям с недостаточным уровнем грамотности создавать тексты, которые будут легче восприниматься, а тем, кто уверенно владеет письменной речью, — значительно ускорить написание текстов.

Целью курсовой работы является исследование проблемы стилистического переноса текста между разговорным и художественным стилями с помощью моделей машинного обучения, включая анализ существующих подходов, разработку и экспериментальную проверку различных архитектур.

Задачи курсовой работы:

1. Изучить существующие подходы к стилистическому переносу текста, включая методы, успешно применяемые для переноса стиля в других областях, например, в обработке изображений.
2. Изучить теоретические основы генеративно-состязательных сетей (GAN), архитектуры CycleGAN и моделей глубокого обучения, релевантных для задачи обработки и генерации текста.
3. Определить форматы данных и методы их предобработки, подходящие для обучения моделей стилистического переноса текста.
4. Собрать и подготовить датасет, содержащий тексты разговорного и художественного стилей, для обучения и оценки моделей.
5. Исследовать применение модели CycleGAN на основе статических векторных представлений слов (Navec) для задачи переноса стиля.

6. Разработать и обучить модель для декодирования [CLS] эмбеддингов BERT в текст, с целью оценки возможности их использования в качестве промежуточного представления для генеративных задач.
7. Разработать и обучить бинарный классификатор для определения стилистической принадлежности текста (разговорный/художественный) на основе модели DistilRuBERT, как вспомогательный инструмент для оценки качества переноса стиля и потенциального компонента функции потерь.
8. Исследовать применение модели на основе архитектуры mBART, адаптированной по принципу CycleGAN, для стилистического переноса текста.
9. Провести анализ полученных результатов, выявить ограничения использованных подходов и сформулировать выводы о решенных и нерешенных аспектах проблемы стилистического переноса текста в рамках проведенного исследования.

1 Методы преобразования изображений и текстов

В одной из первых и наиболее известных работ, где используется CycleGAN, она используется для перевода изображений между двумя доменами [2]. Такая модель была предложена как подход к решению проблемы отсутствия параллельных данных для некоторых задач по компьютерному зрению и переводов одних изображений в другие. Основная идея заключается в использовании двух генераторов и двух дискриминаторов, а также циклических потерь для сохранения исходных характеристик при преобразовании. Этот подход можно легко адаптировать и для работы с текстом.

Авторы следующей статьи предложили метод переноса стиля текста, который не требует параллельных данных для обучения [3]. Он основан на использовании генератора, который реализован с архитектурой encoder-decoder, и дискриминатора, оценивающего стиль генерируемого текста. Было использовано обучение с подкреплением, где дискриминатор играет роль оценщика стиля, помогая модели генерировать текст, который сохраняет исходное содержание, но меняет стиль. В качестве награды в процессе обучения используется совместная оценка дискриминатором стиля и генерации текстов с корректным содержанием.

Авторы следующей статьи адаптировали классическую архитектуру CycleGAN для работы с текстами [4]. В статье приводятся детализированные эксперименты, демонстрирующие, как использование архитектуры CycleGAN позволяет эффективно обучаться на непараллельных корпусах текстов. Основное внимание уделено проблеме разрыва между стилевыми и содержательными аспектами текста и тому, как CycleGAN помогает преодолеть этот барьер.

Отличного результата добились авторы нового метода, который основан на адаптивном извлечении паттернов переноса стилей из данных и контрастном обучении для создания более точных представлений текстов [5]. Этот метод впервые вводит концепцию паттернов переноса стилей в текстах

и может быть легко интегрирован с другими методами для повышения их производительности. Эксперименты показывают, что предложенный подход эффективен и универсален, что делает его полезным инструментом для дальнейших исследований в данной области.

2 Нейросетевые модели типа sequence-to-sequence для стилистического переноса

Одним из ключевых направлений в разработке систем стилистического переноса текста являются модели, основанные на архитектуре sequence-to-sequence (seq2seq) [6]. Этот класс моделей позволяет решать задачи преобразования одной последовательности в другую, сохраняя их логическую взаимосвязь, и применим как к машинному переводу, так и к задаче трансформации текста из одного стилевого регистра в другой.

2.1 Архитектурные принципы seq2seq

Базовая структура seq2seq-модели включает два компонента:

- Энкодер (encoder) — преобразует входную последовательность токенов $X = \{x_1, x_2, \dots, x_T\}$ в скрытое векторное представление h, которое агрегирует семантику всей последовательности;
- Декодер (decoder) — на основе этого скрытого состояния поэтапно генерирует выходную последовательность $Y = \{y_1, y_2, \dots, y_{T'}\}$, соответствующую целевому стилю.

В простейшем случае (модель с рекуррентными сетями) скрытое состояние энкодера h может быть определено как:

$$h = RNN(x_1, x_2, \dots, x_T)$$

Где RNN — рекуррентная нейросеть (обычно LSTM или GRU). Однако современные модели, как правило, используют трансформерную архитектуру, полностью основанную на механизме внимания (self-attention), предложенном в работе Vaswani et al. [7].

2.2 Механизм внимания и архитектура Трансформер

Ключевым элементом трансформеров является механизм внимания, который позволяет модели при генерации учитывать контекст всей входной последовательности. Формально, внимание между элементами входа вычисляется как:

$$Attention(Q,\ K,\ V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Где:

- $Q,\ K,\ V$ — соответственно запросы (queries), ключи (keys) и значения (values), полученные линейными преобразованиями входных токенов;
- d_k — размерность ключей (нормализующий множитель);
- softmax — преобразует веса внимания в вероятностное распределение.

Этот механизм позволяет каждому токену на выходе быть чувствительным ко всем токенам входа, а не только к последним, как в классических RNN-моделях.

2.3 Предобученные seq2seq-модели для стиля

Одной из наиболее известных и эффективных моделей, применяемых для задач стилистического переноса, является mBART (Multilingual BART) — мультиязычная seq2seq-модель, обученная в режиме восстановления текста с маскировкой и перестановкой сегментов [8].

mBART обучается на задаче:

$$argmax_{\theta} \sum_{i=1}^{N} logP_{\theta}\left(x_i \mid Corrupt\left(x_i\right)\right)$$

Где:

- x_i — исходное предложение;
- $Corrupt\left(x_i\right)$ — зашумлённая версия предложения (например, с удалёнными словами);
- θ — параметры модели.

Обученная таким образом модель способна эффективно решать задачи восстановления и перефразирования, включая перенос стиля. Управление

стилем осуществляется с помощью вставки специальных токенов (например, `[FORMAL]`, `[INFORMAL]`, `[TO_LIT]`, `[TO_CONV]`), которые сигнализируют декодеру о желаемом стиле результата.

Подобные подходы использовались и в других моделях, таких как T5 (Text-To-Text Transfer Transformer) [9] и PEGASUS [10], которые обобщают идею обучения на задаче восстановления смыслового содержания и адаптируют её под широкий спектр задач, включая стиль-трансфер.

2.4 Преимущества и ограничения seq2seq для переноса стиля

К преимуществам подхода на основе seq2seq можно отнести:

- Поддержку контекстно-зависимого генеративного вывода;
- Гибкость в адаптации под многоязычные задачи;
- Высокое качество синтаксической и семантической корректности сгенерированных текстов.

Основные ограничения:

- Требуется значительный объём вычислительных ресурсов;
- При слабом стиле в обучающих данных модель может выдавать тексты нейтрального характера;
- Перенос может оказаться “поверхностным”, если стилистические различия слабо выражены в корпусе.

Таким образом, seq2seq-модели с трансформерами являются мощным инструментом для задачи стилистического переноса текста и находят широкое применение в современных системах генерации.

3 Генеративные модели и адаптация CycleGAN к задаче стилистического переноса

Когда для обучения недоступны параллельные данные (одни и те же тексты в разных стилях), seq2seq-подходы становятся менее эффективными. В таких случаях применяются генеративно-состязательные сети (Generative Adversarial Networks, GAN), способные обучаться на непарных данных.

3.1 Общая структура GAN

Стандартная GAN состоит из двух компонентов:

- Генератор G, преобразующий случайный шум $z \sim p_z(z)$ в сгенерированные данные $G(z)$;
- Дискриминатор D, пытающийся отличить реальные данные $x \sim p_{data}(x)$ от сгенерированных $G(z)$.

Игра между ними описывается функцией потерь:

$$\min_G \max_D E_{x \sim p_{data}(x)}[log\ D(x)] + E_{z \sim p_z(z)}[log\ (1 - D(G(z)))]$$

Здесь:

- $D(x)$ — вероятность, что образец x реален;
- $G(z)$ — результат генератора;
- цель G — “обмануть” дискриминатор.

3.2 CycleGAN для непараллельного переноса

CycleGAN [2] расширяет GAN, чтобы осуществлять перенос между двумя стилями A и B без парных данных. Для этого вводятся:

- Генератор $G_{A \to B}$: стиль A в стиль B;
- Генератор $G_{B \to A}$: стиль B в стиль A;
- Дискриминаторы D_A, D_B: различают, принадлежит ли образец стилю A или B.

Добавляется циклическая потеря согласованности:

$$\varsigma_{cyc}(G,\ F) = E_{x \sim A}\left[||F(G(x)) - x||_1\right] + E_{y \sim B}\left[||G(F(y)) - y||_1\right]$$

Где:

- $G,\ F$ — генераторы в обоих направлениях;
- $||\ \cdot\ ||_1$ — L1-норма: мера различия между исходным и восстановленным текстом;
- эта потеря заставляет генераторы «сохранять смысл» при преобразованиях.

CycleGAN показал высокую эффективность в задачах переноса изображений без парных примеров (например, «зима-лето», «фото-картина»). Для текста требуются адаптации: вместо изображения — вектор (эмбеддинг) текста, и вместо пиксельных дискриминаторов — дискриминаторы, обученные различать стили на основе текстовых признаков.

Такие адаптации реализованы в работах [4, 11], где CycleGAN применялся к стилю текста — например, в задачах смены эмоционального окраса, стилистической формальности, литературности и др.

Проблемы применения CycleGAN к тексту:

- Текст дискретен, в отличие от изображений (непрерывные пиксели);
- Реконструкция текста из эмбеддингов требует дополнительных моделей-декодеров;ц
- GAN плохо обучаются, если генераторы и дискриминаторы несбалансированы.

Тем не менее, CycleGAN — один из немногих подходов, способных эффективно решать задачу стиля в условиях непараллельных корпусов, что делает его особенно ценным для русского языка и редких стилей, где разметка затруднена.

4. Эмбеддинги текста: представление векторов слов и предложений

Машинное обучение не работает напрямую с текстом — ему нужны числовые представления. Такие векторы называются эмбеддингами. Они позволяют схожим по смыслу словам или выражениям быть близкими в пространстве.

4.1 Статические эмбеддинги

Статические эмбеддинги (Word2Vec [12], GloVe [13], FastText [14]) присваивают каждому слову фиксированный вектор, одинаковый вне зависимости от контекста.

FastText, в частности, представляет слово w как сумму векторов его n-грамм:

$$\upsilon_{\omega} = \sum_{g \in G_{\omega}} \upsilon_{g}$$

Где:

- G_{ω} — множество всех n-грамм слова;
- υ_{g} — вектор каждой из них;
- таким образом, даже незнакомое слово можно векторизовать через его морфологические компоненты.

Недостаток — одно и то же слово всегда имеет один и тот же вектор, независимо от контекста. Для русского языка предобученные эмбеддинги FastText доступны в библиотеках Navec и RusVectōrēs.

Сжатие эмбеддингов часто используется для экономии памяти. Пример — метод главных компонент (PCA) проецирует векторы в пространство меньшей размерности:

$$X = U \sum V^{T}$$

Где:

- X — матрица эмбеддингов;
- U, V — матрицы сингулярных векторов;
- $\sum$ — сингулярные значения;
- результат — новые координаты векторов в пространстве наименьших потерь.

4.2 Контекстуальные эмбеддинги

Контекстуальные модели, такие как BERT (Bidirectional Encoder Representations from Transformers) [15], выдают разные эмбеддинги одного и того же слова в зависимости от окружения.

Пример:

- “банк” в “банк сидит у реки” и “я пошёл в банк” будет иметь разные векторы.

В BERT эмбеддинг токена t_i определяется как:

$$h_i = Transformer\left(x_i, \{x_j\}\right)$$

Где:

- x_i — токен на входе;
- h_i — выходной эмбеддинг, учитывающий весь контекст.

Для представления всего предложения используется специальный токен `[CLS]` — его эмбеддинг служит агрегатом всей последовательности.

Для русского языка существуют специализированные модели BERT: RuBERT, DeepPavlov/rubert-base-cased, SlavicBERT и др.

5 Визуализация векторных представлений текста

Чтобы понять, как различаются стили текстов в эмбеддинговом пространстве, применяются методы снижения размерности. Один из наиболее мощных — UMAP (Uniform Manifold Approximation and Projection) [16].

UMAP минимизирует функцию кросс-энтропии между расстояниями в исходном пространстве и проекции:

$$\varsigma_{UMAP} = \sum_{i,j} \omega_{ij} log \frac{\omega_{ij}}{\upsilon_{ij}} + \left(1 - \omega_{ij}\right) log \frac{1-\omega_{ij}}{1-\upsilon_{ij}}$$

Где:

- ω_{ij} — вес ребра между точками i и j в высокоразмерном пространстве;
- υ_{ij} — аналогичный вес после проекции;
- цель — сохранить топологическую структуру данных.

UMAP используется для визуализации векторов предложений в 2D или 3D, чтобы:

- оценить кластеризацию по стилю;
- выявить аномалии или перекрытия;
- оценить качество эмбеддингов.

6 Сбор данных

6.1 Предложения разговорного стиля

Для обучения модели, способной преобразовывать текст из разговорного стиля речи в художественный, нужно много данных обоих стилей. Подходящих размеченных датасетов на русском языке нет, а размечать большое количество данных вручную — слишком долго, поэтому данные нужно собрать самостоятельно, а сам датасет будет непараллельным: две колонки, в каждой находятся предложения, представляющие свой стиль, при этом предложения, находящиеся в одной строке могут быть никак не связаны между собой.

Чтобы собрать предложения, отражающие разговорный стиль, был проведён поиск возможно уже существующих датасетов, которые содержали бы подходящие данные. В процессе встречались датасеты, связанные с публикациями в различных социальных сетях, но всё же это не те данные, которые ожидалось найти, поэтому поиск продолжался. Довольно быстро нашёлся датасет, содержащий сообщения на русском языке из 177 самых популярных чатов мессенджера Telegram в формате json-файлов [17]. Структура файлов json представлена на рисунке 1. Несмотря на то, что данные из Telegram-чатов могут содержать элементы, характерные для спонтанной неформальной коммуникации, такие как сокращения, сленг и разговорные обороты, было решено использовать их. Эти особенности представляют собой яркие проявления современного разговорного стиля, который модель будет учиться преобразовывать в художественный.

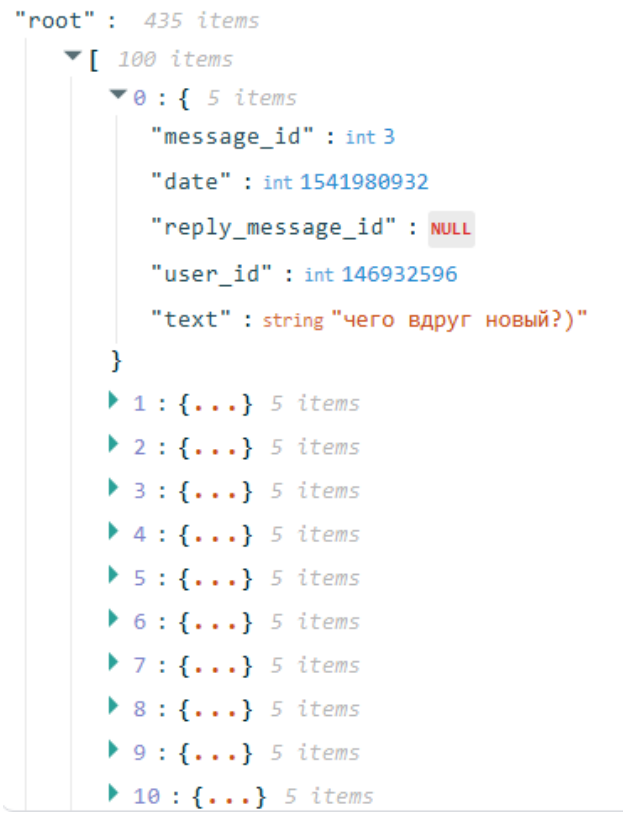


Рисунок 1 — Структура json-файлов

Найденный датасет распространён под лицензией Creative Common Zeros: Public Domain. Используя такую лицензию при публикации датасета, автор отказывается от авторского права на него, поэтому его можно использовать без указания автора даже в коммерческих целях без каких-либо ограничений [18]. Общий вес этого датасета составлял 11 ГБ, а в заархивированном виде — около 3 ГБ. Было решено, что этого вполне достаточно и нужно перейти к сбору предложений художественного стиля.

6.2 Предложения художественного стиля

Источником таких предложений выступили различные литературные произведения, а точнее — проза. Был найден сайт, в одном из разделов которого можно скачать классическую прозу в формате fb2 [19]. Этот формат файлов часто используется для создания и распространения художественной, научной и другой литературы в электронном виде. Обычно именно в этом формате устройства и приложения для чтения электронных книг хранят и обрабатывают их. Некоторые литературные произведения были доступны для

скачивания в txt формате, но было решено не тратить на это время и скачать все книги в формате fb2, а после просто перенести тексты книг в файлы формата txt.

С помощью BeautifulSoup были собраны все ссылки на страницы книг, а с помощью Selenium они были скачаны.

После скачивания книг, их нужно было перевести в формат txt. Формат fb2 основан на XML-разметке, благодаря структурированности которой легко получилось перенести тексты книг в файлы txt формата с помощью BeautifulSoup.

Подробнее с кодом, который использовался для произведения всех описанных действий при сборе данных, можно ознакомиться в приложении А.

7 Небольшая предобработка данных и их сбор в датасет

Все файлы хранились на Google Диске в zip-архивах, поэтому для начала их нужно было разархивировать. Вот как выглядела разархивация txt-файлов, содержащих тексты художественной литературы:

```
with zipfile.ZipFile(books_path, “r”) as zip_ref:
    for filename in zip_ref.namelist():
        zip_ref.extract(filename, “/content/books”)
```

То же самое было проделано с архивом, содержащим json-файлы с сообщениями из чатов Telegram.

После разархивирования нужных файлов начался парсинг предложений из художественной литературы. Каждый текстовый файл считывался в одну строку, которая с помощью функции `sent_tokenize` из библиотеки nltk разбивалась на предложения. После этого начинался перебор полученных предложений. Каждое очищалось от пробельных символов в начале и конце строки методом `strip`. После этого происходило удаление всех символов, кроме букв русского алфавита, пробелов и некоторых знаков пунктуации с помощью не очень сложного регулярного выражения:

```
sentence = re.sub(“[^а-яА-Я---—,.!?:«» ]”, ””, sentence)
```

Это было необходимо, потому что многие произведения содержали, например, вставки на иностранных языках, таких как французский. Следующим этапом необходимо было очистить начало каждой строки от знаков препинания и пробельных символов. Такая очистка требовалась в связи с тем, что после удаления нерелевантных символов (включая иноязычные вставки) знаки препинания могли оказаться в начале строки, что нарушало бы корректность её структуры для дальнейшей обработки:

```
while len(sentence) > 0 and sentence[0] in “---—,.!?:«» ”:
```

```
            sentence = sentence[1:]
```

После такой небольшой предобработки в множество сохранялись предложения, длина которых попадала в следующий интервал: $[\,9;\,199\,]$. Получившиеся множество предложений, преобразованное в список, было сохранено в виде Pandas Series. Всего было получено 1,382,549 предложений художественного стиля..

Точно такие же действия производились с сообщениями из чатов Telegram. Чтобы датасет был сбалансированным, то есть чтобы количество предложений обоих стилей было одинаковым, нужно было прервать перебор сообщений при достижении количества предложений художественного стиля.

Множество предложений разговорного стиля точно также были сохранены в Pandas Series, после чего оба Pandas Series были объединены в один Pandas DataFrame:

```
data = pd.DataFrame({“lit_text”: lit, “tg_text”: com})
```

В переменных `lit` и `com` сохранены Pandas Series с предложениями художественного и разговорного стилей соответственно. Датасет был сохранён в формате csv-таблицы на Google Диск.

Подробнее с кодом, который использовался для произведения всех описанных действий в этом разделе, можно ознакомиться в приложении Б.

8 Представление предложений через эмбеддинги Navec

8.1 Подготовка векторных представлений для слов

Чтобы модель машинного обучения могла обучиться на собранных текстовых данных, необходимо их преобразовать в векторы действительных чисел. Было решено использовать вектора Navec — коллекцию предобученных эмбеддингов для русского языка [20].

При первоначальных экспериментах с использованием полных эмбеддингов Navec (содержащих 300 32-битных чисел плавающей запятой каждый) в моделях с LSTM-слоями возникли проблемы с нехваткой оперативной памяти на имеющихся вычислительных ресурсах при работе с большим объемом данных.

В связи с этим было принято решение модифицировать исходные эмбеддинги Navec с целью оптимизации и уменьшения нагрузки на оперативную память. Модификация включала два основных шага: сжатие размерности векторов с 300 до 10 и изменение типа данных с float32 на float16. Для начала векторы Navec были сохранены в массив NumPy по порядку:

```
path = 'navec_hudlit_v1_12B_500K_300d_100q.tar'
navec = Navec.load(path)
embeddings = list()
for word in navec.vocab.words:
        embeddings.append(navec[word])
embeddings = np.array(embeddings)
```

Сжать полученный Numpy Array помог PCA из библиотеки scikit-learn:

```
target_dim = 10
pca = PCA(n_components=target_dim)
reduced_embeddings = pca.fit_transform(embeddings)
```

После чего был сформирован словарь, содержащий сжатые эмбеддинги Navec. При его формировании, был изменён и тип данных на менее точный:

```
reduced_navec = {word: np.array(reduced_embeddings[i],
dtype=np.float16) for i, word in enumerate(navec.vocab.words)}
```

8.2 Подготовка векторных представлений для знаков пунктуации

Чтобы сохранить как можно больше информации о предложениях, нужно было векторизовать не только слова, но и знаки пунктуации. Для этого был сделан отдельный словарь, в котором хранились постоянные вектора для различных знак пунктуации. Они точно такие же, как и сжатые вектора Navec — десять 16-битных чисел с плавающей запятой, среди которых лишь последнее отличается от нуля. В предложениях встречались разные дефисы и тире, поэтому их вектора было решено сделать идентичными. Вот как выглядит словарь:

```
punkt_vectors = {
    ".": np.array([0.0] * (target_dim - 1) + [1.0], dtype=np.float16),
    "!": np.array([0.0] * (target_dim - 1) + [0.9], dtype=np.float16),
    "?": np.array([0.0] * (target_dim - 1) + [0.8], dtype=np.float16),
    ",": np.array([0.0] * (target_dim - 1) + [0.5], dtype=np.float16),
    ":": np.array([0.0] * (target_dim - 1) + [0.6], dtype=np.float16),
    "–": np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    "-": np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    "—": np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    "«": np.array([0.0] * (target_dim - 1) + [0.2], dtype=np.float16),
    "»": np.array([0.0] * (target_dim - 1) + [0.3], dtype=np.float16),
}
```

Константа target_dim была определена при подборе итогового размера сжатого вектора Navec и равна 10.

8.3 Подготовка к токенизации предложений

Токенами, для которых впоследствии брались соответствующие вектора из подготовленных словарей, выступали слова и знаки пунктуации. Для токенизации предложений был использован метод `word_tokenize` из библиотеки nltk. Чтобы как-то ограничить максимальное количество токенов в предложении в целях оптимизации и уменьшении нагрузки на оперативную память, был проведён небольшой анализ распределения количества токенов в предложениях. С помощью `Counter` из модуля collections было подсчитано, сколько раз встречается то или иное количество токенов в предложениях:

```
vector_lengths = Counter(len(word_tokenize(line)) for line in
df.lit_text)
```

Построив и проанализировав график по полученным данным, который представлен на рисунке 2, число 40 было выбрано как максимальное количество токенов в предложении.

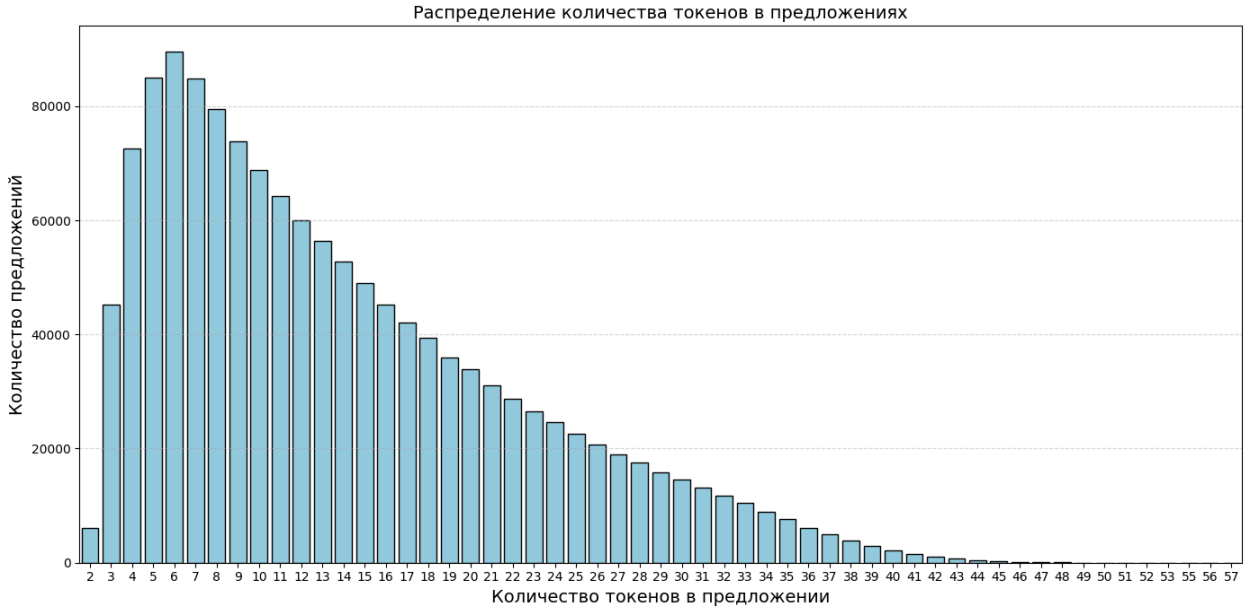


Рисунок 2 — Распределение количества токенов в предложениях

При таком ограничении отсекается очень малая часть предложений и остаётся самая значимая.

8.4 Токенизация и векторизация предложений

Сначала были получены токены для всех предложений художественного стиля, количество токенов в которых меньше выбранного ранее ограничения:

```
lit_tokens = []
for sentence in df.lit_text:
    tokens = word_tokenize(sentence)
    if len(tokens) > max_vector_length:
        continue
    else:
        lit_tokens.append(tokens)
```

Далее была произведена векторизация предложений по токенам. Размерность векторного представления любого предложения должна быть постоянна, независимо от количества токенов в нём, в нашем случае это `max_vector_length * target_dim`, где `max_vector_length` — количество токенов в предложении, а `target_dim` — длина вектора каждого токена. Если токенов в предложении было меньше максимально возможного количества, то начало вектора заполнялось нулями следующим образом:

```
padding_for = max_vector_length - len(tokens)
vector = [np.zeros(target_dim)] * padding_for
```

В Navec есть соответствующий вектор, доступный по ключу “<pad>”, но во время сжатия размерности векторов Navec он перестал быть нулевым, что может создать ненужные шумы при обучении модели, поэтому было решено заполнять начало векторов именно таким образом.

После этого, для каждого токена в предложении выполнялась следующая процедура: сначала предпринималась попытка найти и добавить соответствующий сжатый вектор Navec. Если токен отсутствовал в словаре Navec, предпринималась попытка найти его в словаре векторов знаков

пунктуации. В случае, если токен не был найден и там, добавлялся специальный сжатый вектор “<unk>”, предварительно созданный для представления всех неизвестных токенов. Таким образом, для каждого предложения формировался список векторов его токенов. Далее этот список преобразовывался в массив NumPy, а затем его размерность изменялась таким образом, чтобы каждое предложение было представлено единым одномерным вектором:

```
vector = np.array(vector, dtype=np.float16).reshape(max_vector_length
* target_dim)
```

Аналогичные процедуры векторизации были выполнены и для предложений разговорного стиля. В результате первичной обработки и фильтрации по максимальному количеству токенов (40 токенов на предложение), количество предложений, успешно преобразованных в векторы, для художественного и разговорного стилей могло отличаться. Для обеспечения одинакового размера выборок и возможности их последующего объединения или сравнения, было необходимо привести оба набора векторизованных предложений к единой длине. С этой целью определялась минимальная длина среди двух полученных массивов векторов (массива векторов художественного стиля и массива векторов разговорного стиля). Затем оба массива обрезались до этой минимальной длины:

```
length = min(len(lit_vectors), len(tg_vectors))
lit_vectors_np = np.array(lit_vectors[:length])
tg_vectors_np = np.array(tg_vectors[:length])
```

После этого они были сохранены в формате npy, а их вес составил около 1 ГБ. Подробнее с кодом, который использовался для векторизации данных, можно ознакомиться в приложении В.

8.5 Визуализация полученных эмбеддингов

После того как были получены одномерные векторные представления для каждого предложения из художественного и разговорного стилей, возникла задача визуально оценить, насколько эти векторы отличаются. Прямая визуализация многомерных векторов затруднительна, поэтому был применен метод снижения размерности UMAP.

Цели визуализации:

1. Оценка разделимости стилей. Визуально определить, образуют ли векторные представления предложений разных стилей отдельные кластеры или группы в пространстве пониженной размерности.
2. Выявление структуры данных. Понять, существуют ли какие-либо закономерности или подструктуры внутри каждого стиля или между ними.
3. Качественная оценка эмбеддингов. Получить интуитивное представление о том, насколько успешно предыдущие этапы векторизации (включая сжатие эмбеддингов слов Navec с помощью PCA, обработку пунктуации и формирование векторов предложений) сохранили информацию, релевантную для стилистического анализа.

Процесс визуализации с использованием UMAP:

1. Подготовка данных для UMAP.
    a. Были загружены массивы `lit_vectors_np` и `conv_vectors_np`, содержащие векторные представления предложений.
    b. Для ускорения процесса обучения UMAP и получения быстрой оценки, из каждого набора данных была сформирована случайная выборка размером 2000 векторов. Это позволило сократить вычислительные затраты без существенной потери информации об общей структуре данных.

c. Выборки были объединены в один массив `all_vectors_sample_np`, который использовался для обучения UMAP. Это важно, так как UMAP строит единое многообразие для всех предоставленных данных, что позволяет корректно сопоставить положение векторов разных стилей в результирующем пространстве.

2. Снижение размерности для визуальной оценки.

   Для наглядной визуализации многомерных векторных представлений предложений и оценки их разделимости по стилям, было решено снизить их размерность до двух измерений (2D). Двухмерное пространство является наиболее интуитивно понятным для графического представления и позволяет визуально оценить наличие или отсутствие кластеризации данных, степень их перекрытия и общую структуру расположения векторов разных стилей.

   a. Для этой цели модель UMAP была настроена на проекцию исходных высокоразмерных данных в двухмерное пространство, путем установки параметра `n_components = 2`.
   b. Использовались общепринятые значения гиперпараметров: `n_neighbors=15` (число ближайших соседей, учитываемых при построении графа многообразия) и `min_dist=0.1` (минимальное желаемое расстояние между точками в низкоразмерном представлении). Выбор этих параметров направлен на сбалансированное сохранение как локальной, так и глобальной структуры исходных данных в проекции.
   c. После обучения на объединенной случайной выборке из 2000 векторов каждого стиля, модель UMAP трансформировала эти векторы предложений в их двухмерные координаты, которые затем использовались для построения диаграммы рассеяния.

3. Визуализация 2D-представлений.

a. Полученные двумерные координаты были разделены на два набора, соответствующие художественному и разговорному стилям.
b. С помощью библиотек matplotlib и seaborn была построена диаграмма рассеяния. Каждая точка на графике представляет одно предложение, а её цвет указывает на стилистическую принадлежность (голубой для художественного, лососевый для разговорного).
c. На графике (приложение Г) наблюдается небольшое разделение между облаками точек, представляющих разные стили. Хотя существует значительная зона перекрытия, что вполне естественно для такой сложной лингвистической задачи, как различение стилей на уровне отдельных предложений, общая тенденция к формированию отдельных кластеров заметна. Это указывает на то, что используемые векторные представления содержат информацию, позволяющую UMAP выявить и отобразить некоторые стилистические различия.

4. Снижение размерности до 1D и визуализация плотностей.
   a. Для альтернативного взгляда на разделимость данных была также выполнена проекция векторов в одномерное пространство с помощью UMAP (`n_components=1`).
   b. Результаты были визуализированы с помощью графиков ядерной оценки плотности (KDE plots). Для каждого стиля был построен свой график плотности, отражающий распределение его предложений вдоль единственной UMAP-компоненты.
   c. Графики плотности (приложение Д) показывают, что пики распределений для художественного и разговорного стилей несколько смещены относительно друг друга. Распределение для художественного стиля имеет выраженный пик в области отрицательных значений компоненты UMAP, в то время как

разговорный стиль демонстрирует более широкое распределение с пиками в положительной области. Это дополнительно подтверждает, что даже одна компонента, полученная с помощью UMAP, способна частично уловить и разделить стилистические особенности предложений.

Проведенная визуализация векторных представлений предложений, сформированных на основе эмбеддингов Navec с последующим сжатием и агрегацией, была нацелена на качественную оценку их способности выявлять стилистические различия между художественным и разговорным текстами.

Важно отметить, что анализируемые корпуса текстов не являлись параллельными, то есть отличались не только по стилю, но и по семантическому содержанию. В таких условиях задача выявления чисто стилистических маркеров усложняется, поскольку семантические различия также могут вносить вклад в распределение векторов.

Несмотря на эту сложность, анализ двухмерных проекций UMAP продемонстрировал, что векторные представления предложений разных стилей обнаруживают тенденцию к формированию отдельных, хотя и пересекающихся, областей в пространстве пониженной размерности. Сам факт наличия такой, пусть и неполной, сепарации в условиях семантически разнородных данных подчеркивает, что используемые эмбеддинги и методы их обработки способны улавливать не только тематические, но и собственно стилистические характеристики текста.

Аналогичные выводы следуют из анализа одномерных проекций: смещение пиков распределений для художественного и разговорного стилей на графиках ядерной оценки плотности дополнительно подтверждает, что даже одна компонента UMAP частично отражает стилистические особенности.

Таким образом, результаты визуализации свидетельствуют о том, что разработанный подход к векторизации предложений позволяет получить

репрезентации, содержащие стилистически релевантную информацию, которая проявляется даже на фоне семантической вариативности непарных текстов. Это подтверждает качественную состоятельность сформированных эмбеддингов.

С кодом, который использовался для визуализации, можно ознакомиться в приложении Е.

9 Обучение модели CycleGAN на эмбеддингах Navec

После векторизации предложений, где каждый текст был представлен одномерным вектором фиксированной длины на основе сжатых эмбеддингов Navec, следующим этапом исследования стала разработка и обучение модели для стилистического переноса между художественным и разговорным доменами. Для решения этой задачи была выбрана архитектура CycleGAN (Cycle-Consistent Generative Adversarial Network). Данная архитектура привлекательна тем, что позволяет обучать модель на непарных данных, то есть без необходимости иметь прямые стилистические аналоги для каждого предложения в обучающих корпусах, а также тем, что используется для похожих задач в компьютерном зрении.

Основу CycleGAN, как описано в оригинальной работе [2], составляют два генератора и два дискриминатора. Генератор `G_LitToConv` предназначен для преобразования векторов предложений из художественного стиля в векторы, имитирующие разговорный стиль, в то время как генератор `G_ConvToLit` выполняет обратную задачу. Соответствующие им дискриминаторы, `D_Conv` и `D_Lit`, обучаются отличать реальные (оригинальные) векторные представления предложений своего домена от тех, что были сгенерированы (трансформированы) генераторами.

Для эффективного обучения CycleGAN используются несколько типов функций потерь. Адверсиальная потеря заставляет генераторы создавать выходы, которые дискриминаторы не могут отличить от реальных данных, а дискриминаторы, в свою очередь, учатся как можно лучше это различие улавливать. Для адверсиальной потери использовалась бинарная кросс-энтропия. Ключевым компонентом, обеспечивающим сохранение семантического содержания при изменении стиля, является потеря цикловой согласованности. Она основана на идее, что если предложение перевести из стиля А в стиль Б, а затем обратно из стиля Б в стиль А, то результат должен быть максимально близок к исходному предложению. Это помогает предотвратить ситуацию, когда генератор просто отображает все входные

данные в одно и то же предложение целевого стиля, игнорируя содержание. Дополнительно может применяться потеря идентичности, которая стимулирует генератор не изменять входные данные, если они уже принадлежат его целевому домену (например, `G_LitToConv` не должен сильно изменять предложение, которое уже написано в художественном стиле). Это способствует стабилизации обучения и сохранению характеристик целевого домена.

В качестве базовых архитектурных элементов для генераторов и дискриминаторов были использованы рекуррентные нейронные сети типа LSTM (Long Short-Term Memory) [21]. Архитектура LSTM, подробно описанная, например, в работе Хохрайтера и Шмидхубера, специально разработана для эффективной обработки последовательных данных и запоминания долгосрочных зависимостей, что делает ее подходящей для работы с векторными представлениями предложений, которые по своей сути являются последовательностями векторов токенов. Исходные предложения были представлены векторами размерности 400, полученными конкатенацией 40 десятимерных сжатых эмбеддингов Navec для каждого токена.

Было проведено четыре последовательных эксперимента (итерации) с различными конфигурациями модели и гиперпараметров. Каждый такой эксперимент с уникальным набором настроек далее именуется “версией модели”.

9.1 Первая версия модели и гиперпараметров

В первой итерации генераторы состояли из одного LSTM-слоя со скрытым состоянием размерности 512, дискриминаторы также использовали один LSTM-слой, но со скрытым состоянием размерности 256, за которым следовал полносвязный слой (128 нейронов, LeakyReLU) и выходной слой с сигмоидой. Для потерь цикла и идентичности применялась функция L1-расстояния. Скорости обучения были установлены на 0.0002 для генераторов и 0.0001 для дискриминаторов, с весами для потерь цикла и идентичности, равными 5.

Анализ графиков потерь (Приложение Ж) выявил, что дискриминаторы успешно обучались, их потери снижались, хотя и показывали некоторую волатильность на валидационной выборке. Однако потери генераторов, как и потери цикла и идентичности, демонстрировали тенденцию к росту и гораздо большую нестабильность, особенно на валидационной выборке. Это указывало на возможный дисбаланс в обучении, где ”сильные” дискриминаторы подавляли генераторы, а также на трудности модели в сохранении семантики и идентичности при использовании L1-потери.

9.2 Вторая версия модели и гиперпараметров

Для решения проблем первой версии, на второй итерации были внесены корректировки. Скорость обучения дискриминаторов была уменьшена до 0.00005, а генераторов незначительно увеличена до 0.00025, с целью ослабить дискриминаторы. Вес потерь цикла (`lambda_cycle`) был повышен до 7, а потерь идентичности (`lambda_identity`) снижен до 2. Архитектура генератора также претерпела изменения: размерность скрытого состояния LSTM была уменьшена до 256, но количество LSTM-слоев увеличено до двух в поисках более глубокой, но менее широкой архитектуры.

Графики потерь для этой версии (Приложение З) не показали кардинального улучшения. Потери дискриминаторов вели себя схожим образом. Потери генераторов оставались высокими и волатильными, а потери цикла согласованности, несмотря на увеличенный вес, продолжали расти. Это свидетельствовало о том, что проблема сохранения семантической целостности оставалась актуальной.

9.3 Третья версия модели и гиперпараметров

Ключевым изменением в третьей версии стало внедрение комбинированных функций потерь для цикловой согласованности и идентичности. Теперь они представляли собой взвешенную сумму L1-расстояния и косинусного расстояния (с равными весами 0.5 для каждой компоненты). Косинусное расстояние было добавлено для лучшего учета

семантической близости векторов, оценивая сходство их направлений. Вес потерь цикла (`lambda_cycle`) был значительно увеличен до 10, а потерь идентичности (`lambda_identity`) – до 3.5.

Однако, как показали графики (Приложение И), эта конфигурация привела к серьезной дестабилизации обучения. Примерно после 57-ой эпохи наблюдался резкий и значительный рост потерь генераторов, а также потерь цикла и идентичности, которые совсем сильно подскочили, что указывало на коллапс обучения. Вероятно, комбинация новых функций потерь и высоких весов сделала оптимизационную задачу для генераторов неразрешимой в текущих условиях.

9.4 Четвертая версия модели и гиперпараметров

Учитывая предыдущие результаты, в четвертой версии были предприняты значительные изменения для стабилизации процесса. Веса потерь цикла и идентичности были снижены до 4, при сохранении комбинированной природы этих потерь. Архитектура генератора была вновь модифицирована: размерность скрытого LSTM-слоя теперь составляла 400, а количество слоев было увеличено до трех. Наиболее существенные изменения коснулись дискриминаторов: их архитектура была значительно упрощена (скрытый LSTM-слой размерностью 160, полносвязный слой размерностью 40), а также была усилена их регуляризация за счет увеличения `discriminator_dropout_rate` до 0.4 и `discriminator_weight_decay` до 1e-3.

Графики потерь для этой итерации обучения (Приложение К) продемонстрировали иную картину. Потери генераторов, цикла и идентичности быстро снизились до очень низких и стабильных значений – наилучших среди всех версий. Однако потери дискриминаторов, после нескольких начальных эпох, выросли и стабилизировались на уровне около 0.69, что свидетельствует о том, что дискриминаторы практически не обучились и их предсказания были близки к случайным. Это, вероятно, результат их чрезмерного упрощения и сильной регуляризации.

Качественная оценка результатов четвертой версии, проведенная путем ручного тестирования преобразования предложений, выявила неудовлетворительное качество переноса стиля. Несмотря на низкие численные значения потерь, связанных с генерацией и сохранением контента, сгенерированные предложения часто содержали большое количество неизвестных токенов (<unk>), знаков препинания. Отсутствовал какой-либо перенос стиля. Например, при попытке преобразовать текст ”У лукоморья дуб зелёный, златая цепь на дубе том, и днём, и ночью кот учёный всё ходит по цепи кругом” из художественного стиля в разговорный, получилось следующее предложение: ”«««««««««««««««««««««««««««« <unk> лукоморья дуб <unk> <unk> цепь на дубе том <unk> <unk> и ночью кот <unk> <unk> ходит по цепи кругом”. При попытке преобразовать этот текст обратно в художественный стиль, текст почти не изменился: ”«««««««««««««««««««««««««««« <unk> лукоморья дуб <unk> <unk> цепь на дубе том <unk> <unk> и ночью кот <unk> <unk> ходит по пленки–”. Это подтверждает, что ”слабые” дискриминаторы не смогли обеспечить генераторам адекватный обучающий сигнал для изучения сложных стилистических трансформаций. К тому же циклическое преобразование текста невозможно. Несмотря на долгие попытки улучшить алгоритм декодирования эмбеддингов, полученных на выходе генераторов, ничего хорошего получить не удалось, поскольку проблема заключалась не в нём.

9.5 Общие выводы по обучению CycleGAN на эмбеддингах Navec и переход к следующим этапам

Итеративный процесс настройки модели CycleGAN на основе сжатых эмбеддингов Navec продемонстрировал значительные трудности в достижении качественного переноса стиля при сохранении семантического содержания и обеспечении стабильности обучения. Ни одна из рассмотренных конфигураций не привела к полностью удовлетворительному результату. Основными выявленными ограничениями, связанными с

использованием статических эмбеддингов Navec в данной архитектуре, являются:

1. Отсутствие контекстуальности. Статические эмбеддинги Navec присваивают каждому слову один и тот же вектор вне зависимости от его окружения, что затрудняет улавливание тонких стилистических нюансов, часто определяемых контекстом.
2. Потеря информации при сжатии. Предварительное сжатие 300-мерных векторов Navec до 10 измерений с помощью PCA, необходимое для оптимизации, могло привести к потере значимой информации, усложняя задачу для модели.
3. Проблемы с реконструкцией текста. Ручные тесты показали, что восстановление осмысленного и стилистически корректного текста из сгенерированных 400-мерных векторов (агрегаций 10-мерных токенов) является сложной задачей. Частое появление ”<unk>” токенов, знаков препинания, семантическая несвязность и отсутствие циклической согласованности преобразований предложений свидетельствуют об ограниченности как самих сжатых эмбеддингов слов, так и, возможно, выразительной способности LSTM-генераторов для данной задачи.
4. Сложность обучения GAN. Достижение стабильного равновесия в adversarial обучении является известной проблемой, особенно при работе с текстовыми данными, где дискретность языка трансформируется в непрерывные векторные пространства.

Учитывая выявленные недостатки статических эмбеддингов Navec и трудности с качественной генерацией и переносом стиля, дальнейшие исследования были направлены на использование более мощных контекстуальных моделей представления текста, таких как BERT. Поскольку ожидалось, что способность BERT улавливать контекстные зависимости и формировать более богатые семантические и стилистические репрезентации позволит преодолеть ограничения, с которыми столкнулась модель,

построенная на основе векторных представлений Navec, и добиться более качественного переноса стиля.

С кодом обучения всех этих версий моделей можно ознакомиться в Приложении Л.

10 Обучение модели декодирования эмбеддингов BERT

Предыдущая глава была посвящена исследованию применения модели CycleGAN для стилистического переноса текста с использованием статических векторных представлений слов Navec. Как было показано в разделе 5.5, этот подход столкнулся с существенными ограничениями, включая отсутствие контекстуальности в эмбеддингах Navec, значительную потерю информации при сжатии размерности с 300 до 10 с помощью метода главных компонент, и неспособность генераторов реконструировать осмысленный текст, что проявлялось в появлении большого количества неизвестных токенов `<unk>` и нарушении грамматической структуры текста.

Эти проблемы указали на необходимость перехода к более мощным представлениям текста. В качестве альтернативы были выбраны контекстуальные эмбеддинги, получаемые с помощью архитектуры BERT, которая генерирует глубокие векторные представления, учитывающие контекст употребления слов. Для получения представления целого предложения был выбран эмбеддинг специального токена `[CLS]`, добавляемого в начало каждой последовательности при обработке BERT. Этот токен, по замыслу, агрегирует информацию о семантике и структуре предложения после прохождения через слои трансформера-кодировщика. В качестве базовой модели использовалась `DeepPavlov/rubert-base-cased`, адаптированная для русского языка.

Однако переход на BERT-эмбеддинги сам по себе не решает задачу генерации текста. Если бы мы продолжили использовать подход, подобный CycleGAN (где генераторы работают в пространстве эмбеддингов, как это было предложено, например, для адаптации CycleGAN к тексту в работе [4]), то после преобразования `[CLS]` эмбеддинга (например, из художественного стиля в разговорный) потребовалось бы преобразовать результирующий вектор обратно в текст. Таким образом, перед экспериментами со стилистическим переносом в пространстве BERT-эмбеддингов необходимо было решить подзадачу: разработка и обучение модели-декодера, способной

по [CLS] эмбеддингу восстанавливать исходную или семантически близкую текстовую последовательность. Успешное решение этой задачи подтвердило бы, что выбранные эмбеддинги содержат достаточно информации для реконструкции текста, открыв путь к дальнейшим исследованиям. Данный раздел подробно описывает процесс разработки, обучения и анализа результатов такой модели-декодера.

10.1 Подготовка данных для обучения модели декодирования

Для обучения модели, основной задачей которой является преобразование [CLS] эмбеддинга BERT обратно в текст, необходим параллельный набор данных, состоящий из пар: “эмбеддинг предложения, полученный от BERT” (вход) – “исходное предложение в виде последовательности токенов” (цель). Ключевым этапом подготовки входных данных для декодера стала векторизация текстовых предложений с использованием предобученной модели BERT. С полным кодом векторизации можно ознакомиться в приложении М. В качестве основы использовалась модель DeepPavlov/rubert-base-cased и соответствующий ей токенизатор BertTokenizer. Процесс векторизации для каждого предложения из исходного датасета data.csv (содержащего колонки lit_text для художественного стиля и tg_text для разговорного) включал токенизацию с помощью tokenizer(text, max_length=self.max_seq_len, truncation=True, padding='max_length', return_tensors='pt'), где максимальная длина последовательности max_seq_len была установлена в 96 токенов. Этот параметр был выбран на основе анализа распределения длин предложений в корпусе (Рисунок 2) для охвата большинства предложений при минимизации вычислительных затрат. Если последовательность была короче, она дополнялась паддинг-токенами (pad_token_id = 0), а если длиннее – обрезалась. Токенизатор автоматически добавлял специальные токены [CLS] в начало и [SEP] в конец. Подготовленные батчи токенизированных последовательностей подавались на вход модели BertModel.from_pretrained(”DeepPavlov/rubert-base-cased”). Из выходных

данных модели (outputs.last_hidden_state) извлекался вектор, соответствующий первому токену [CLS] (embed = outputs.last_hidden_state[:, 0, :]), который использовался как репрезентация всего предложения. Эти 768-мерные векторы были сохранены в файлы lit_embeddings.npy (художественный стиль) и conv_embeddings.npy (разговорный стиль), общим числом около 2.76 миллионов (по 1.38 миллиона на стиль). Для обучения декодера эти файлы объединялись в единый массив embeddings типа float32. Параллельно, целевые текстовые данные (исходные предложения) также токенизировались тем же AutoTokenizer.from_pretrained(”DeepPavlov/rubert-base-cased”). С полным кодом токенизации, как и построением, обучением декодера, можно ознакомиться в приложении Н. Процесс, реализованный в функции preprocess_texts, включал обрезку текста до 95 токенов, добавление [SEP] и дополнение паддинг-токенами до 96 токенов. Результат сохранялся в tokenized_texts.npy как массив np.int64. Для работы с данными в PyTorch был разработан класс EmbeddingToTokenDataset, преобразующий массивы в тензоры torch.float32 и torch.long. Датасет делился на обучающую (80%) и валидационную (20%) выборки с помощью random_split. Загрузчики данных DataLoader создавались с размером батча BATCH_SIZE = 4032 (что приводило примерно к 685 шагам обновления за эпоху), shuffle=True для обучения, и pin_memory=True. NUM_WORKERS был установлен в 0.

10.2 Архитектура модели декодера

Модель TransformerDecoder основана на архитектуре трансформера. Входной [CLS] эмбеддинг (размерность 768) сначала проходит слой нормализации nn.LayerNorm и линейный слой nn.Linear(768, 768), после чего его размерность изменяется на (batch_size, 1, 768) для использования в качестве memory (контекста) для декодера. Для целевой последовательности токенов используются стандартные слои эмбеддингов слов nn.Embedding(VOCAB_SIZE, EMBED_DIM) (где VOCAB_SIZE = 119547, EMBED_DIM = 768) и позиционных эмбеддингов nn.Embedding(MAX_SEQ_LEN, EMBED_DIM) (где

MAX_SEQ_LEN = 96). Сумма этих двух эмбеддингов формирует входное представление для декодера. Основой модели является стек из 3 слоев nn.TransformerDecoderLayer. Каждый такой слой включает маскированное многоголовочное внимание к себе (self-attention), многоголовочное внимание к memory (cross-attention), и полносвязную нейронную сеть (feed-forward network). Гиперпараметры слоя: NUM_HEADS = 8 (количество голов внимания), FF_DIM = 1024 (размерность скрытого слоя в FFN, что меньше стандартного значения 3072 для EMBED_DIM = 768, выбрано для уменьшения числа параметров), функция активации gelu и DROPOUT = 0.1. Весь декодер и его слои настроены на batch_first=True. Для предсказания следующего токена из большого словаря используется слой nn.AdaptiveLogSoftmaxWithLoss.

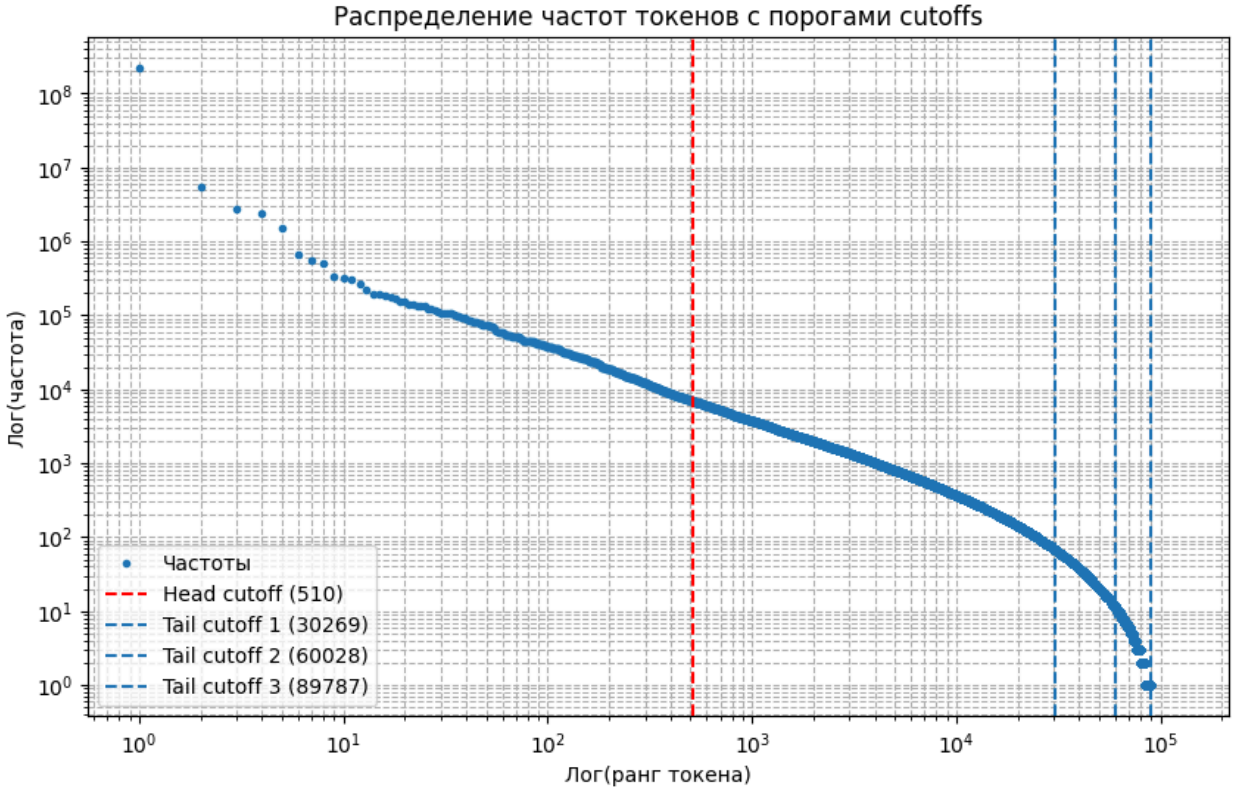


Рисунок 3 — Распределение частот токенов с порогами cutoffs

Параметр cutoffs = [510, 30269, 60028, 89787] определяет границы кластеров токенов, подобранные на основе анализа распределения их частот (Рисунок 3), где первые 510 токенов (покрывающие ~94% кумулятивной частоты) составляют “голову”, а остальные распределены по “хвостам” с фактором уменьшения размерности div_value = 4. С кодом построения

графика частот токенов можно ознакомиться в приложении О. Для предотвращения “заглядывания вперед” в авторегрессионном режиме используется кэшируемая треугольная маска (_get_tgt_mask), создаваемая с помощью torch.triu и masked_fill для установки значения -inf на запрещенных позициях. Метод генерации текста (generate) использует лучевой поиск (beam search) с параметрами: beam_width = 20, temperature = 0.7, top_k = 50, top_p = 0.7, штраф за длину alpha = 0.8 и минимальная длина генерации min_length = 10. Дополнительно, только на этапе генерации, применяется смещение логитов на основе частот токенов (freq_bias), вычисляемое как beta * torch.log(freqs + 1.0) (где beta = 2.0 при генерации, а частоты freqs загружаются из token_frequencies.npy). Отсутствие этого смещения во время обучения создает рассогласование между условиями тренировки и инференса. Метод прямого прохода (forward) реализует механизм Teacher Forcing, где коэффициент teacher_forcing_ratio (динамически уменьшаемый во время обучения) определяет вероятность использования истинного предыдущего токена или токена, предсказанного моделью.

10.3 Процесс обучения модели декодера

Обучение модели TransformerDecoder проводилось на объединенном датасете (художественный и разговорный стили) в течение 50 эпох. В качестве оптимизатора был выбран AdamW с начальной скоростью обучения 10^{-4}. Для управления скоростью обучения применялся линейный планировщик с "прогревом" (get_linear_schedule_with_warmup) на протяжении первых 1000 шагов. Для ускорения вычислений и экономии памяти GPU использовалась автоматическая смешанная точность (torch.cuda.amp.GradScaler и autocast). Функция потерь, как упоминалось, вычислялась непосредственно слоем model.output_layer (типа AdaptiveLogSoftmaxWithLoss). Важно отметить, что стандартная реализация

этого слоя в PyTorch не имеет параметра `ignore_index`, что означает, что паддинг-токены (`pad_token_id = 0` для используемого токенизатора), если они попадали в целевые последовательности `tgt_shifted`, могли учитываться при вычислении потерь и градиентов, потенциально искажая процесс обучения. В будущих работах этот аспект требует более внимательного рассмотрения, например, путем явного маскирования потерь для позиций с паддинг-токенами. На протяжении обучения коэффициент Teacher Forcing (`teacher_forcing_ratio`) линейно уменьшался с 1.0 (на начальных эпохах, когда модель полностью полагается на истинные предыдущие токены) до значения примерно 0.3875 к 50-й эпохе. Это означает, что к концу обучения модель примерно в 61% случаев на каждом шаге декодирования генерировала следующий токен на основе собственных предыдущих предсказаний. После каждой эпохи обучения проводилась валидация на отложенной выборке, где `teacher_forcing_ratio` всегда устанавливался в 0.0, чтобы оценить способность модели к автономной генерации. Лучшая модель по минимальному валидационному лоссу сохранялась. Из-за ограничений по времени и ресурсам обучение было остановлено на 50-й эпохе, не дожидаясь срабатывания критерия ранней остановки (`PATIENCE = 5`, `MIN_DELTA = 0.0005`).

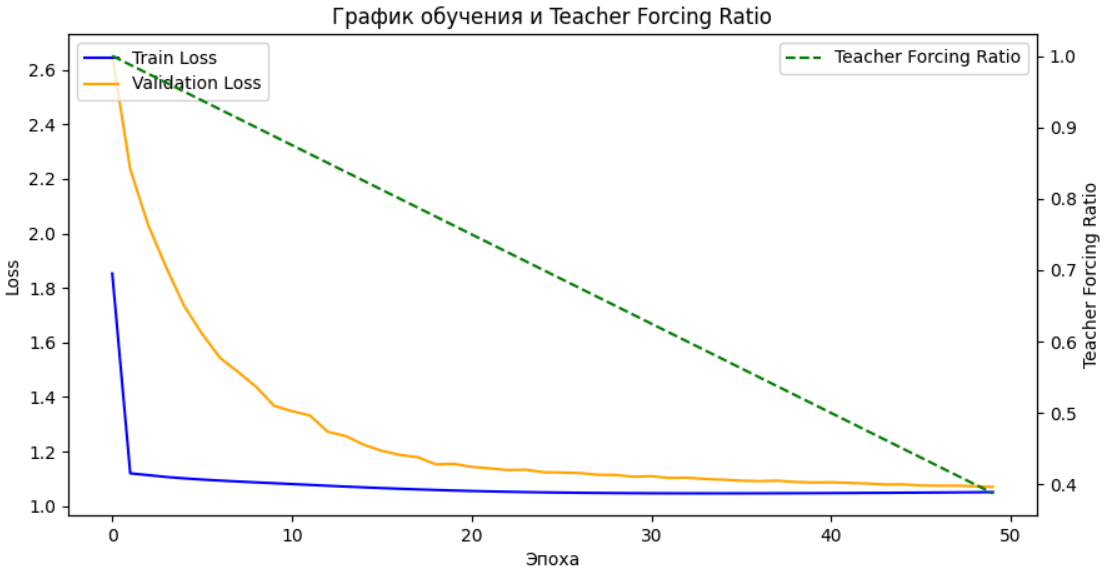


Рисунок 4 — График обучения и Teacher Forcing Ratio

График обучения (Рисунок 4) демонстрирует динамику тренировочного и валидационного лоссов, а также коэффициента Teacher Forcing. Тренировочный лосс (синяя линия) начался со значения приблизительно 1.85 на первой эпохе и плавно снижался, достигнув значения около 1.04-1.06 к 50-й эпохе. Валидационный лосс (оранжевая линия) стартовал с ~2.62, также демонстрируя устойчивое снижение и достигнув ~1.06-1.07 к концу 50-й эпохи. Небольшие флуктуации валидационного лосса и его близость к тренировочному указывают на отсутствие явного переобучения на данном этапе.

Таблица 1 — Количественные потери на ключевых эпохах.

Эпоха	Train loss	Val loss
1	~1.85	~2.62
10	~1.1	~1.37
30	~1.08	~1.17
50	~1.06	~1.07

Коэффициент Teacher Forcing (зеленая пунктирная линия) линейно уменьшался, как и было запланировано.

10.4 Результаты декодирования и критический анализ проблем

Несмотря на обнадёживающую динамику функции потерь, которая свидетельствовала о формальном успехе процесса оптимизации, качественная оценка способности обученной модели `TransformerDecoder` восстанавливать текст из `[CLS]` эмбеддингов BERT выявила её полную неэффективность для решения поставленной задачи. Для проверки было взято предложение “`какие сладкие булочки!`”. Его `[CLS]` эмбеддинг,

полученный от DeepPavlov/rubert-base-cased, был подан на вход метода model.generate(...) с параметрами лучевого поиска: beam_width = 20, temperature = 0.7, top_k = 50, top_p = 0.7, alpha = 0.8, min_length = 10, и beta = 2.0 для freq_bias. Сгенерированный текст “Правоохранительные Правоохранительныелашение [unused66] …”, не имел никакого семантического или лексического сходства с исходным предложением. Вместо этого он представлял собой набор часто встречающихся в датасете (например, “Правоохранительные”, что может указывать на его высокую частоту в одной из частей исходного корпуса) или специальных/неиспользуемых токенов ([unusedXX]), не образующих осмысленной фразы.

Анализ причин столь неудовлетворительного результата выявил несколько ключевых факторов:

1. Информационная недостаточность [CLS] эмбеддинга. Один 768-мерный вектор, даже полученный от такой мощной модели, как BERT, вероятно, не способен сохранить всю полноту информации, необходимой для точной реконструкции сложной структуры, лексики и грамматики исходного предложения. Этот вектор оптимизирован для задач классификации, а не для генерации.
2. Сложность задачи отображения “один-ко-многим”. Преобразование одного вектора в последовательность токенов требует от модели не только знания лексики, но и глубокого моделирования языковых закономерностей и долгосрочных зависимостей, что является фундаментально сложной задачей.
3. Ограничения процесса обучения.
    a. Неполное снижение Teacher Forcing. Остановка обучения на 50-й эпохе, когда teacher_forcing_ratio составлял ~0.39, не позволила модели в достаточной мере научиться генерировать текст полностью автономно, полагаясь исключительно на свои предыдущие предсказания.

b. Проблема с `pad_token_id` в `AdaptiveLogSoftmaxWithLoss`. Отсутствие игнорирования паддинг-токенов при вычислении потерь могло привести к тому, что модель обучалась предсказывать эти нерелевантные токены, что искажало градиенты и мешало обучению полезным закономерностям.

4. Архитектурные ограничения и гиперпараметры.
   a. Емкость модели-декодера. Возможно, трех слоев декодера с размерностью скрытого слоя FFN 1024 было недостаточно для моделирования столь сложной зависимости.
5. Рассогласование применения freq_bias. Использование смещения частот с `beta = 2.0` только на этапе генерации, в то время как во время обучения оно не применялось, создало значительное несоответствие условий. Это могло привести к тому, что модель на этапе инференса непропорционально сильно отдавала предпочтение высокочастотным токенам, таким как “Правоохранительные”, или специальным токенам, частота которых могла быть искусственно завышена или неправильно интерпретирована смещением.
6. Разрыв между оптимизируемой метрикой и качеством генерации. Успешное снижение функции потерь (связанной с perplexity) не всегда гарантирует генерацию осмысленного, связного и грамматически правильного текста, что и подтвердилось в данном эксперименте.

10.5 Выводы

Эксперименты, проведенные в рамках данного раздела, убедительно продемонстрировали, что задача построения эффективного декодера, способного восстанавливать осмысленный текст из статического `[CLS]` эмбеддинга предложения, полученного от модели BERT, в рамках выбранной архитектуры и процесса обучения не была успешно решена. Несмотря на формально хорошие показатели снижения функции потерь на обучающей и валидационной выборках, качество генерируемого текста оказалось крайне низким. Это делает данный подход (использование `[CLS]` эмбеддинга BERT

как промежуточного представления для последующей генерации) непригодным для задачи стилистического переноса текста.

Ключевые выводы по результатам данного раздела:

- Агрегированное представление предложения в виде одного `[CLS]` вектора, вероятно, несет недостаточно информации для его точной и полной реконструкции.
- Обучение модели для отображения “один-ко-многим” в данном контексте требует более совершенных подходов, тщательного подбора архитектуры, процесса обучения, и, возможно, значительно больших вычислительных ресурсов.
- Выявленные методологические проблемы, такие как обработка паддинг-токенов в функции потерь, рассогласование условий обучения и генерации (из-за `freq_bias`), и незавершенное снижение Teacher Forcing, также внесли свой вклад в неудовлетворительный результат.

Учитывая эти выводы, а также негативный опыт с моделью CycleGAN на эмбеддингах Navec, где даже при попытке оперировать в пространстве эмбеддингов возникали существенные проблемы с качеством генерации, было принято решение о кардинальном изменении стратегии. Вместо того чтобы продолжать разрабатывать сложные многокомпонентные системы с отдельными кодировщиками и декодерами, работающими с промежуточными векторными представлениями, целесообразно перейти к использованию предобученных sequence-to-sequence моделей. Эти модели, такие как BART или T5, изначально спроектированы для задач преобразования одной текстовой последовательности в другую (например, машинный перевод, реферирование, ответы на вопросы). Они обучаются end-to-end на больших объемах текстовых данных и способны эффективно улавливать сложные зависимости между входной и выходной текстовыми последовательностями, сохраняя при этом необходимую информацию для генерации когерентного и релевантного текста.

Для последующих экспериментов по стилистическому переносу текста была выбрана модель sn4kebyt3/ru-bart-large [22], представляющая собой версию mBART (multilingual BART), адаптированную и дообученную для русского языка. Модели семейства BART, основанные на мощной архитектуре кодер-декодер Трансформера, в целом продемонстрировали высокую эффективность в широком спектре задач генерации текста, включая работу с русским языком [8].

11 Разработка и обучение классификатора стилей текста

Исследования, проведённые в рамках данной работы, выявили значительные трудности в решении задачи стилистического переноса текста. Так, использование модели CycleGAN с применением статических векторных представлений слов Navec столкнулось с рядом ограничений. Среди них были отсутствие контекстуальной информации в эмбеддингах, значительная потеря данных при сжатии размерности с 300 до 10 с помощью метода главных компонент и неспособность генераторов создавать осмысленный текст, что проявлялось в большом количестве неизвестных токенов `<unk>` и нарушении грамматической структуры. Попытка применения контекстуальных эмбеддингов BERT, предпринятая позже, также не привела к желаемому результату. Вектор `[CLS]`, агрегирующий информацию о предложении, оказался недостаточно информативным для восстановления сложных текстовых последовательностей с помощью трансформерного декодера, что стало очевидным после анализа результатов декодирования. Эти ограничения указали на необходимость разработки нового подхода, где важную роль должна была сыграть стилистическая функция потерь, направляющая генеративную модель к созданию текста в заданном стиле — разговорном или художественном.

Для реализации такой функции потерь потребовалась предварительно обученная модель, способная с высокой точностью определять стиль текста. Эта модель должна была стать основой для оценки стиля сгенерированного текста, обеспечивая обратный сигнал для оптимизации генератора. В рамках данной главы была поставлена задача разработки, обучения и оценки бинарного классификатора стилей текста. Создание такого классификатора стало важным шагом, поскольку его результаты должны были использоваться в дальнейшем для интеграции в модели стилистического переноса, включая sequence-to-sequence архитектуры, что открыло бы новые перспективы для повышения качества генерации текста.

11.1 Подготовка данных для обучения классификатора стилей

Работа с классификатором началась с подготовки данных из датасета `data.csv`, который уже использовался ранее для задач векторизации и декодирования. Датасет содержал около 2.76 миллионов предложений, равномерно распределенных между двумя колонками: `tg_text`, включавшей примеры разговорного стиля (метка 0), и `lit_text`, представлявшей художественный стиль (метка 1). По 1.38 миллиона примеров на каждый стиль обеспечивали сбалансированность классов, что исключало перекос в обучении модели. Сначала данные загружались с использованием библиотеки `pandas`, после чего тексты из обеих колонок объединялись в один список, а соответствующие метки (0 для разговорного стиля и 1 для художественного) формировались в отдельный список для дальнейшей работы.

Датасет был разделен на три выборки: обучающую, валидационную и тестовую. Для этого применялась функция `train_test_split` из библиотеки `scikit-learn`. На первом этапе 80% данных (2.208 миллиона примеров) отводились в обучающую выборку, а оставшиеся 20% делились на валидационную и тестовую выборки. Параметр `VAL_TEST_SPLIT = 0.3` определял, что 70% из этих 20% (или 14% от общего объема, то есть 0.3864 миллиона примеров) становились валидационной выборкой, а оставшиеся 30% (6% от общего объема, то есть 0.1656 миллиона примеров) — тестовой. Параметр `RANDOM_STATE = 42` обеспечивал воспроизводимость разделения, что было важно для повторяемости экспериментов. Таким образом, итоговое распределение выборок составило 2.208 миллиона примеров для обучения, 0.3864 миллиона для валидации и 0.1656 миллиона для тестирования, что позволило создать репрезентативные наборы данных для всех этапов работы.

Следующим шагом стала токенизация текстов, необходимая для приведения их к формату, пригодному для работы с трансформерной моделью. Для этого использовался токенизатор `DistilBertTokenizer` из библиотеки `transformers`, соответствующий модели “`DeepPavlov/distilrubert-base-cased-conversational`”. Эта модель

представляла собой облегченную версию RuBERT, что позволило снизить вычислительные затраты, сохранив при этом качество контекстуальных представлений текста. Токенизация выполнялась с помощью специально разработанной функции `tokenize`. Максимальная длина последовательности устанавливалась на уровне 96 токенов, что было определено на основе анализа распределения длин предложений в датасете. Если текст был длиннее, он обрезался, а если короче — дополнялся паддинг-токенами с идентификатором `pad_token_id = 0`. Результат токенизации возвращался в виде PyTorch тензоров, что упрощало последующую работу с моделью. Токенизированные данные сохранялись для повторного использования, что сократило время на подготовку данных в дальнейших экспериментах.

На основе токенизированных текстов был создан класс `StyleDataset`, который объединял обработанные данные и их метки. Этот класс использовался для подготовки данных к обучению в PyTorch. Затем настраивались загрузчики данных с помощью `DataLoader`. Размер батча составлял 2448 примеров, что обеспечивало около 902 шагов обновления за одну эпоху (2.208 миллиона примеров, разделенные на 2448). Для обучающей выборки применялась опция `shuffle = True`, чтобы случайное перемешивание данных способствовало лучшей обобщающей способности модели. Кроме того, параметр `pin_memory = True` использовался для ускорения передачи данных на GPU, что оказалось особенно полезным при работе с большими объемами данных и позволило сократить время обучения.

11.2 Архитектура модели классификатора стилей

Модель классификатора, названная `StyleClassifier`, была построена на основе предобученной `DistilBertModel`, к которой был добавлен специализированный классификационный слой. Основная часть модели, представленная `DistilBertModel`, отвечала за генерацию 768-мерных контекстуальных эмбеддингов для каждого токена в последовательности. Особое внимание уделялось токену `[CLS]`, расположенному в начале текста, который агрегировал информацию о всей последовательности после

прохождения через слои трансформера-кодировщика. Этот вектор затем передавался на вход линейного слоя `nn.Linear(768, 2)`, который преобразовывал его в логиты для двух классов: метка 0 соответствовала разговорному стилю, а метка 1 — художественному. Такая архитектура была выбрана из-за ее простоты, что позволило достичь высокой скорости обучения и инференса, сохранив при этом способность модели точно классифицировать стиль текста.

11.3 Процесс обучения классификатора

Обучение модели проводилось в течение трех эпох с использованием оптимизатора `AdamW`, начальная скорость обучения которого составляла 2e-5. Этот выбор был обусловлен стандартными практиками тонкой настройки трансформерных моделей, где небольшая скорость обучения позволяет избежать резких изменений весов предобученной модели. В качестве функции потерь применялась `nn.CrossEntropyLoss`, которая включала в себя `LogSoftmax` и `NLLLoss`, что обеспечивало корректное вычисление потерь для задачи бинарной классификации. Для ускорения вычислений на GPU использовалась смешанная точность с помощью `torch.cuda.amp.GradScaler`, что позволило сократить время обучения и уменьшить потребление памяти.

Процесс обучения состоял из двух основных фаз: тренировочной и валидационной. На тренировочной фазе модель оптимизировалась на обучающей выборке: для каждого батча вычислялись потери, выполнялся обратный проход и обновлялись веса с учетом градиентов. На валидационной фазе, проводимой после каждой эпохи, оценивались потери, точность и взвешенная F1-метрика на валидационной выборке.

Результаты обучения показали стабильную динамику (Рисунок 5): на первой эпохе тренировочный лосс составил 0.1175, а валидационный — 0.0983, при этом точность и F1-метрика достигли значения 0.9604. На второй эпохе значения потерь снизились до 0.0871 для тренировочной выборки и 0.0918 для валидационной, а точность и F1-метрика выросли до 0.9632. К

третьей эпохе тренировочный лосс уменьшился до 0.0725, валидационный — до 0.0900, а метрики точности и F1 достигли 0.9644. Отсутствие значительных расхождений между тренировочными и валидационными потерями указывало на стабильность обучения и отсутствие переобучения. После завершения обучения модель была сохранена для дальнейшего использования.

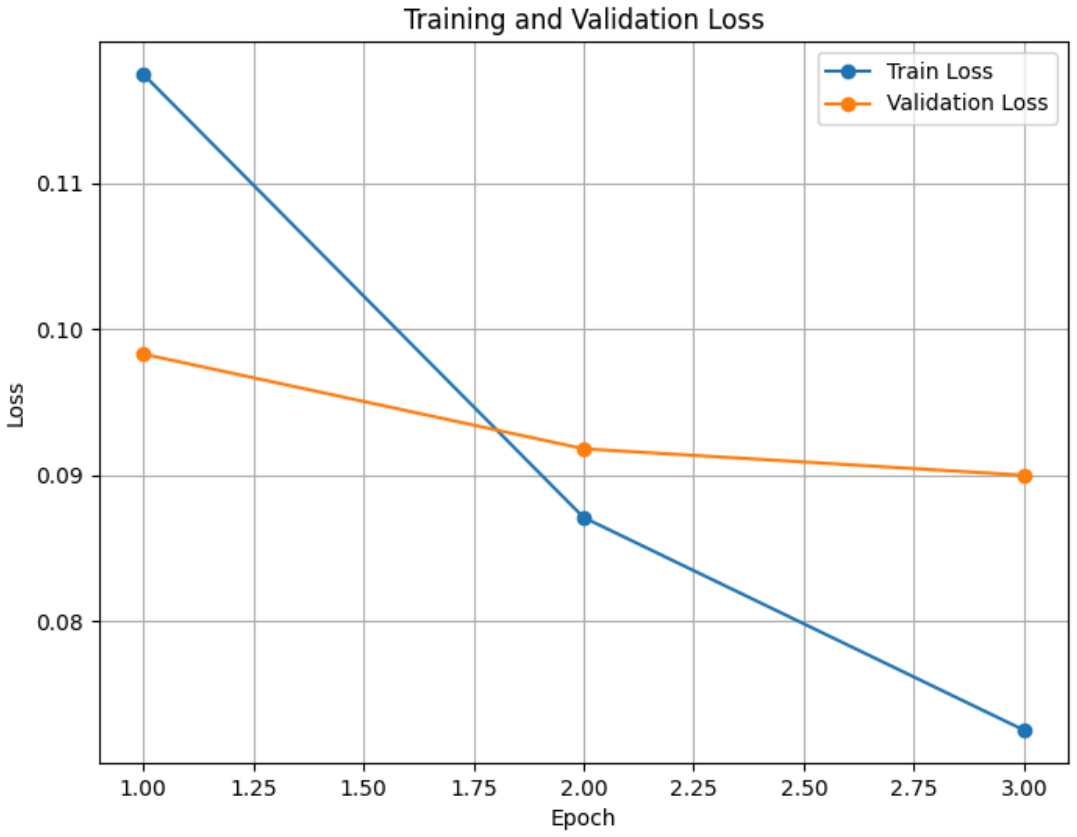


Рисунок 5 - Процесс обучения классификатора стилей

11.4 Оценка качества на тестовой выборке

После завершения обучения была проведена оценка модели на тестовой выборке, которая включала 0.1656 миллиона примеров и не использовалась в процессе обучения. Результаты показали, что модель достигла точности 0.9647 и взвешенной F1-метрики 0.9647. Эти высокие показатели подтвердили способность модели обобщать данные и успешно определять

стиль текста, что сделало ее подходящей для применения в качестве оценщика стиля в дальнейших экспериментах.

11.5 Функционал предсказания и стилистическая потеря

Для практического применения классификатора были разработаны две ключевые функции, которые расширили его возможности и сделали пригодным для интеграции в генеративные системы, а также позволили провести оценку качества работы модели. Первая функция, `predict_style`, была предназначена для определения стиля текста. Она могла возвращать либо метку класса (0 для разговорного стиля или 1 для художественного), либо вероятности принадлежности текста к каждому из стилей при использовании параметра `return_probs=True`. Эта функция стала основным инструментом для проверки способности модели классифицировать тексты различной сложности, что было особенно важно для анализа ее эффективности.

Вторая функция, `style_loss`, реализовывала вычисление стилистической функции потерь с использованием `CrossEntropyLoss` между предсказаниями классификатора и целевой меткой стиля. Она позволяла количественно оценить, насколько текст соответствовал заданному стилю, что делало ее полезной для оптимизации генеративных моделей. Например, низкое значение потерь для текста с целевой меткой, совпадающей с его стилем, указывало на высокую степень соответствия, тогда как высокое значение сигнализировало о необходимости корректировки генератора. Эти функции обеспечили гибкость применения классификатора, что стало важным шагом для его использования в задачах стилистического переноса текста.

11.6 Качественная оценка и анализ

Для качественной оценки работы классификатора и анализа его способности различать стили текста были отобраны как простые, так и более сложные примеры, чтобы проверить модель в различных условиях. Простые

примеры включали тексты с явными стилистическими признаками, тогда как сложные тексты содержали более тонкие различия, что позволило глубже понять поведение модели.

Среди простых примеров был рассмотрен текст разговорного стиля: “а бывает ты типа в толпе но как будто вообще один и никто тебя не видит и не услышит даже если крикнешь” (метка 0). При подаче этого текста в функцию `predict_style` с параметром `return_probs = True` модель выдала вероятности `[0.99999595, 0.00000408]`, что отражало чрезвычайно высокую уверенность в классификации текста как разговорного, с минимальной вероятностью ошибки в пользу художественного стиля. Такой результат подтверждал способность модели точно распознавать неформальную лексику (“типа”, “как будто вообще”), разговорные конструкции и фрагментарную структуру, характерные для разговорного стиля. Второй простой пример представлял собой текст художественного стиля: “Среди тысяч огней он чувствовал себя тенью, забытой даже собственным отражением” (метка 1). Для этого текста функция вернула вероятности `[0.00505348, 0.99494654]`, что указывало на уверенность модели в принадлежности текста к художественному стилю при небольшой вероятности ошибки в пользу разговорного стиля. Этот результат демонстрировал способность модели улавливать поэтические образы (“чувствовал себя тенью”, “забытой даже собственным отражением”) и сложные синтаксические конструкции, типичные для художественной литературы.

Далее были проанализированы более сложные примеры, где различия между стилями были менее очевидны. Первый сложный пример представлял собой текст разговорного стиля: "Было же время, всё было как будто чище, легче. Дышалось. Не знаю, как объяснить — но тогда просто жил и не думал, зачем. И это было нормально. А теперь всё как будто через фильтр, чужой" (метка 0). Модель выдала вероятности `[0.99698526, 0.00301473]`, что указывало на высокую уверенность в классификации текста как разговорного, с небольшой вероятностью ошибки. Этот текст содержал

элементы, которые могли затруднить классификацию, такие как философские размышления и образные выражения (“всё как будто через фильтр, чужой”), которые могли быть интерпретированы как художественные. Однако использование разговорных маркеров (“Не знаю, как объяснить”, “просто жил”), коротких предложений (“Дышалось”) и неформального тона позволило модели корректно определить стиль. Второй сложный пример был текстом художественного стиля: “Он вспоминал то время не как череду событий, а как состояние: утро, в котором не нужно ничего решать. Пустота, от которой не страшно. Тогда он просто существовал — не объясняя себе зачем. Теперь всё иначе, и в этом иначе не было покоя” (метка 1). Для этого текста модель вернула вероятности `[0.19649537, 0.8035046]`, что отражало уверенность в принадлежности текста к художественному стилю, но с заметной вероятностью ошибки в пользу разговорного стиля. Этот результат оказался менее уверенным по сравнению с простыми примерами. Тем не менее, модель уловила художественные черты текста, такие как метафоричность (“утро, в котором не нужно ничего решать”, “пустота, от которой не страшно”) и более сложная синтаксическая структура, что позволило ей сделать правильный выбор, хотя и с меньшей уверенностью.

Анализ результатов показал, что классификатор демонстрировал высокую точность на текстах с явными стилистическими признаками, где вероятности ошибок были минимальными. Однако при работе с более сложными текстами, где стилистические различия были менее выражены, модель проявляла меньшую уверенность, особенно в случае художественного текста, вероятность ошибки для которого составила около 0.1965. Это указывало на то, что для текстов с тонкими стилистическими различиями модель могла путать стили.

11.7 Выводы и значение для дальнейшей работы

В ходе работы был разработан и обучен классификатор стилей на основе `DistilRuBERT`, который достиг точности 96.47% на тестовой выборке. Модель успешно различала разговорный стиль (метка 0) и художественный

стиль (метка 1), что подтвердилось в ходе качественной оценки на примерах различной сложности, таких как “а бывает ты типа в толпе но как будто вообще один…” и “Он вспоминал то время не как череду событий…”. Анализ показал высокую точность классификации для текстов с явными стилистическими признаками, но выявил сложности при работе с текстами, где различия были менее выражены. Полученные результаты позволили использовать классификатор для формирования стилистической функции потерь, что стало важным шагом в совершенствовании подходов к стилистическому переносу текста. В дальнейшем планировалось интегрировать модель в sequence-to-sequence архитектуры, такие как mBART, адаптированные для русского языка, что могло способствовать созданию более эффективных систем генерации текста с учетом стилистических особенностей.

С полным кодом, связанным с обучением классификатора стилей, можно ознакомиться в приложении П.

12 Обучение модели CycleGAN для стилистического переноса текста с использованием mBART

12.1 Концепция модели и компоненты системы

Для реализации стилистического переноса текста между разговорным и художественным стилями была разработана архитектура, вдохновленная CycleGAN, но адаптированная для работы с текстовыми данными. В качестве базовой модели генератора была выбрана предобученная модель “sn4kebyt3/ru-bart-large” — версия mBART, оптимизированная для русского языка. Детали реализации представлены в Приложении Р.

Система включала три основных компонента. Генератор (G_model) был построен на основе MBartForConditionalGeneration. Его токенизатор MBart50TokenizerFast был дополнен двумя специальными токенами: TAG_TO_LIT = “[TO_LIT]” и TAG_TO_CONV = “[TO_CONV]”, которые добавлялись в начало входного текста для указания целевого стиля. Для учета новых токенов слой эмбеддингов модели был расширен (self.bart_model.resize_token_embeddings(len(gen_tokenizer))).
Корректная работа с русским языком обеспечивалась установкой параметров decoder_start_token_id и forced_bos_token_id равными идентификатору токена русского языка (RUSSIAN_TOKEN_ID = 24228). Генератор поддерживал как генерацию текста, так и вычисление потерь при наличии целевых меток, что использовалось для расчета потерь идентичности и цикловой консистентности.

Дискриминаторы (D_C_model для разговорного стиля и D_L_model для художественного) были реализованы как сверточные нейронные сети (CNN, класс Discriminator). Они включали слой эмбеддингов (nn.Embedding, размерность 128), два блока из сверточного слоя (nn.Conv1d), активации ReLU и максимального пулинга (nn.MaxPool1d), а также финальный полносвязный слой (nn.Linear), выдающий логит, который оценивал, является ли текст “реальным” или “сгенерированным”.

Классификатор стилей (style_classifier_main_model) был разработан и обучен ранее (в предыдущей главе). Он базировался на модели DistilBertModel (DeepPavlov/distilrubert-base-cased-conversational) с добавленным линейным слоем для бинарной классификации (0 — разговорный стиль, 1 — художественный стиль). В рамках текущего эксперимента веса классификатора были заморожены (param.requires_grad_(False)), и он использовался исключительно для формирования стилистической функции потерь (loss_G_style_total), оценивая, насколько сгенерированный текст соответствует целевому стилю.

12.2 Подготовка данных и конфигурация обучения

Для обучения использовался датасет data.csv, содержащий 1382549 пар текстов для каждого стиля (разговорного и художественного). Данные были разделены на обучающую (1244294 текста), валидационную (69127 текстов) и тестовую выборки (оставшаяся часть, не использовалась в данном эксперименте). Максимальная длина последовательности для токенизатора mBART (gen_tokenizer) и генератора была установлена как MAX_LENGTH = 128. Для токенизации текстов, подаваемых на вход классификатору стилей (style_tokenizer), использовалась та же длина. Данные организовывались с помощью класса StyleDataset, который добавлял управляющие теги (TAG_TO_LIT или TAG_TO_CONV) и токенизировал тексты. Обучающие и валидационные выборки загружались через DataLoader с параметрами BATCH_SIZE = 96 и NUM_WORKERS = 0.

Использовался оптимизатор AdamW. Для генератора (G_model) скорость обучения составляла LEARNING_RATE_GEN = 2e-5, для дискриминаторов (D_C_model, D_L_model) — LEARNING_RATE_DISC = 4e-5. Применялся линейный планировщик скорости обучения с прогревом (get_linear_schedule_with_warmup). Число шагов за эпоху было установлено как STEPS_PER_EPOCH = 125. Для повышения эффективности использовалась автоматическая смешанная точность (torch.cuda.amp.GradScaler).

Функции потерь включали: адверсиальную (`nn.BCEWithLogitsLoss`), стилистическую (на основе замороженного классификатора, `nn.CrossEntropyLoss`), а также потери идентичности и цикловой консистентности (вычисляемые через `loss` от `G_model`). Весовые коэффициенты для этих потерь (`LAMBDA_CYCLE`, `LAMBDA_IDENTITY`, `LAMBDA_STYLE`, `LAMBDA_ADV`) изменялись между итерациями обучения, что описано ниже.

12.3 Анализ результатов обучения первой итерации

Первая итерация обучения проводилась в течение 8 эпох с использованием следующих весовых коэффициентов для функций потерь:

- `LAMBDA_CYCLE = 7.0`
- `LAMBDA_IDENTITY = 10.0`
- `LAMBDA_STYLE = 7.0`
- `LAMBDA_ADV = 1.0`

Рассмотрим графики обучения модели на первой итерации (Рисунок 6). Тренировочные потери (верхний график): общая потеря генератора (“G Total Train (Full)”, синяя линия) начиналась с высокого значения (~360) и резко снижалась до ~130–150 ко второй эпохе, после чего продолжала медленно уменьшаться, достигнув ~120 к восьмой эпохе. Основной вклад в эту потерю вносила стилистическая компонента (“G Style Train”, зеленая пунктирная линия), которая также снижалась, но оставалась доминирующей, составляя большую часть общей потери (~100–110 к концу). Потери идентичности (“G Identity Train”, фиолетовая пунктирная линия) и цикловой консистентности (“G Cycle Train”, красная пунктирная линия) быстро стабилизировались на низком уровне (10–20), что объясняется высоким значением `LAMBDA_IDENTITY = 10.0`, из-за которого модель стремилась минимизировать эти потери, отдавая приоритет сохранению контента. Адверсиальная потеря (“G Adv Train”, оранжевая пунктирная линия) оставалась минимальной (~0.5–1.5), что указывает на слабое влияние этой компоненты. Потери дискриминаторов (“D

Total Train”, коричневая линия) колебались в диапазоне 0–2, что типично для адверсиального обучения, где дискриминаторы и генератор балансируют друг друга.

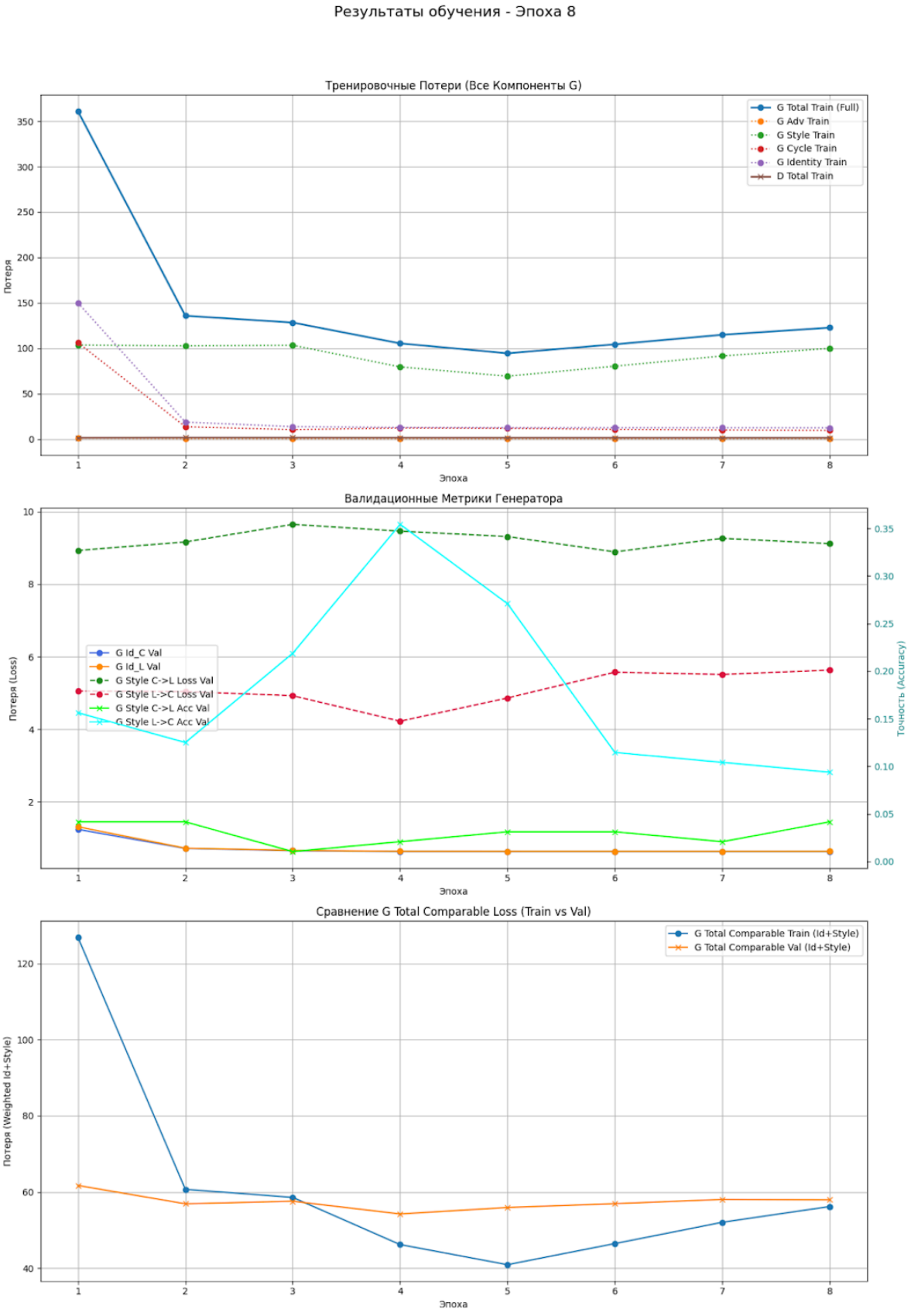


Рисунок 6 - Графики обучения модели на первой итерации

Валидационные метрики (средний график): потери идентичности (“G Id_C Val”, синяя линия, и “G Id_L Val”, оранжевая линия) снижались до ~0.6–0.7 к концу обучения, что указывало на хорошее сохранение контента при преобразовании текстов в их же стиль. Однако стилистические потери (“G Style C ≻ L Loss Val”, зеленая пунктирная линия, и “G Style L ≻ C Loss Val”, красная пунктирная линия) оставались высокими: около 9.0 для C ≻ L и 5.0 для L ≻ C, без значительного снижения. Это говорило о том, что сгенерированные тексты редко соответствовали целевому стилю. Точность стилистической классификации (“G Style C ≻ L Acc Val”, светло-зеленая линия с крестиками, и “G Style L ≻ C Acc Val”, голубая линия с крестиками) была крайне низкой: для C ≻ L она колебалась в диапазоне 2–10%, а для L ≻ C — 10–13%, что подтверждало неспособность модели эффективно менять стиль текста.

Сравнение общей сопоставимой потери (нижний график): тренировочная потеря (“G Total Comparable Train”, синяя линия), учитывающая только идентичность и стиль, снижалась с ~160 до ~60–70, тогда как валидационная (“G Total Comparable Val”, оранжевая линия) после снижения до ~80 на 3–4 эпохе начала расти, достигая ~90 к восьмой эпохе. Это расхождение указывало на переобучение: модель хорошо оптимизировала потери на обучающей выборке, но не могла обобщить на валидационной.

12.4 Корректировка гиперпараметров и анализ результатов второй итерации

Учитывая низкую стилистическую точность в первой итерации, были скорректированы весовые коэффициенты, чтобы усилить влияние стилистической и адверсиальной потерь. Вторая итерация проводилась в течение 9 эпох со следующими значениями гиперпараметров:

- `LAMBDA_CYCLE = 4.0` (уменьшено с 7.0)
- `LAMBDA_IDENTITY` = 4.0 (уменьшено с 10.0)

- LAMBDA_STYLE = 12.0 (увеличено с 7.0)
- LAMBDA_ADV = 1.5 (увеличено с 1.0)

Рассмотрим графики обучения модели на второй итерации (Рисунок 7). Тренировочные потери (верхний график): общая потеря генератора (“G Total Train (Full)”, синяя линия) начиналась с ~270 и снижалась до ~160–170 к девятой эпохе. Стилистическая потеря (Ц”G Style Train”, зеленая пунктирная линия) оставалась доминирующей, снижаясь с ~180 до ~120–130, что ожидаемо из-за увеличения LAMBDA_STYLE до 12.0, усилившего влияние этой компоненты. Потери идентичности (“G Identity Train”, фиолетовая пунктирная линия) и цикловой консистентности (“G Cycle Train”, красная пунктирная линия) были ниже, чем в первой итерации (0–5 и 40–50 соответственно), что логично из-за снижения их весов (LAMBDA_IDENTITY и LAMBDA_CYCLE уменьшены до 4.0). Адверсиальная потеря (“G Adv Train”, оранжевая пунктирная линия) оставалась низкой (~0.5–1.0), несмотря на увеличение LAMBDA_ADV до 1.5, что указывает на недостаточное влияние дискриминаторов на обучение генератора. Потери дискриминаторов (“D Total Train”, коричневая линия) стабильно колебались в диапазоне 0.5–1.5, что свидетельствует о сбалансированном противостоянии с генератором, но без значительного прогресса в улучшении качества генерации. Валидационные метрики (средний график): потери идентичности (“G Id_C Val”, синяя линия, и “G Id_L Val”, оранжевая линия) оставались низкими (0.6–0.9), что показывало, что снижение LAMBDA_IDENTITY не ухудшило сохранение контента. Стилистические потери (“G Style C ≻ L Loss Val”, зеленая пунктирная линия, и “G Style L ≻ C Loss Val”, красная пунктирная линия) демонстрировали высокую вариативность: C ≻ L колебалась в диапазоне 8.8–9.7, а L ≻ C — 4.2–5.5, без явного снижения. Это указывало на продолжающиеся сложности с достижением целевого стиля. Однако стилистическая точность показала улучшение: для C ≻ L (“G Style C ≻ L Acc Val”, светло-зеленая линия с крестиками) точность выросла с ~1.8% на

первой эпохе до пика ~31% на четвертой, но затем снизилась до 5–7% к девятой эпохе; для L ≻ C (“G Style L ≻ C Acc Val”, голубая линия с крестиками) точность увеличилась с ~12% до пика ~32% на четвертой эпохе, после чего упала до ~12–13%. Эти пиковые значения были выше, чем в первой итерации, но последующее снижение точности указывало на нестабильность обучения и возможное переобучение на стилистических признаках, которые модель не смогла устойчиво обобщить.

Качественный анализ. Логи валидации (эпоха 6) показали, что модель склонна копировать входные тексты:

- C ≻ L 1 IN: “За некоторыми исключениями …” ≻ OUT: “За некоторыми исключениями …”
- C ≻ L 2 IN: “Чарльз Буковски — Музыка горячей воды …” ≻ OUT: “Чарльз Буковски — Музыка горячей воды :....”
- L ≻ C 1 IN: “Я вам не соперница....” ≻ OUT: “Я вам не соперница....”
- L ≻ C 2 IN: “Но сказанного не воротишь....” ≻ OUT: “Но сказанного не воротишь....”

Снижение `LAMBDA_IDENTITY` до 4.0 и увеличение `LAMBDA_STYLE` до 12.0 не смогли преодолеть склонность базовой модели к копированию (без дообучения базовая модель просто копировала входной текст). Это указывает на то, что баланс между компонентами потерь остался неоптимальным: модель минимизировала потери через копирование, вместо того чтобы изменять стиль текста.

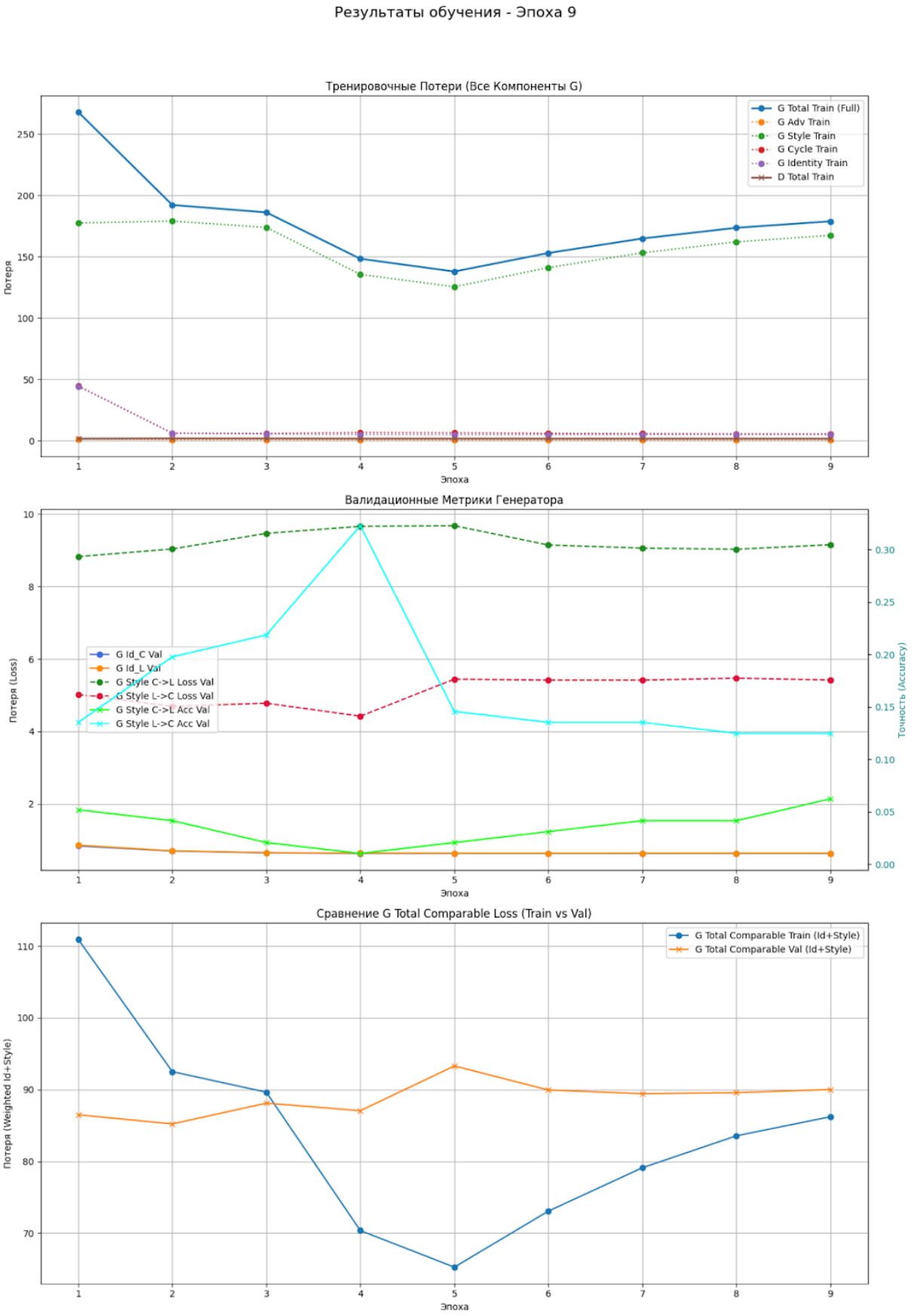


Рисунок 6 - Графики обучения модели на второй итерации

12.5 Общие выводы по обучению модели CycleGAN на mBART

Эксперименты с CycleGAN-подобной моделью на основе mBART показали, что задача стилистического переноса текста на непараллельных данных крайне сложна. В первой итерации (8 эпох) высокий вес идентичности (`LAMBDA_IDENTITY = 10.0`) привел к низкой стилистической точности (2–13%) и высоким стилистических потерям (9.0 для C ≻ L, 5.0 для L ≻ C). Графики показали переобучение: валидационная сопоставимая потеря начала расти после 4–5 эпохи. Во второй итерации (9 эпох) корректировка гиперпараметров дала временное улучшение (пиковая стилистическая точность ~31–32% на четвертой эпохе), но к концу обучения точность упала до 5–13%, а графики подтвердили нестабильность и переобучение (расхождение между тренировочной и валидационной потерями увеличилось). Модель копировала входные тексты, несмотря на изменения гиперпараметров.

Ключевая проблема заключалась в сложности балансировки потерь: высокий вес стилистической потери не смог преодолеть склонность модели к копированию, а снижение веса идентичности оказалось недостаточным. Низкое влияние адверсиальной потери (~0.5–1.0) указывало на слабую роль дискриминаторов. Учитывая ограниченный прогресс, дальнейшие эксперименты с данной архитектурой были прекращены. Возможные улучшения могли бы включать более тонкую настройку весов, усложнение дискриминаторов или использование параллельных данных, но это потребовало бы значительных ресурсов.

ЗАКЛЮЧЕНИЕ

В рамках данной курсовой работы было проведено комплексное исследование проблемы стилистического переноса текста между разговорным и художественным стилями русского языка с использованием современных методов машинного обучения, включая генеративно-состязательные сети и трансформерные архитектуры. Основное внимание было уделено моделям, способным работать с непараллельными данными, таким как CycleGAN, а также модели mBART с модифицированной функцией потерь.

Эксперименты показали, что использование статических эмбеддингов (Navec) в рамках CycleGAN оказалось недостаточно эффективным: модели не справлялись с генерацией связного и стилистически преобразованного текста. Попытка использовать контекстуальные представления BERT через декодирование `[CLS]` эмбеддинга также продемонстрировала ограничения — вектор не содержал достаточной информации для точной реконструкции текста, будучи оптимизированным в первую очередь для задач классификации.

Одним из наиболее успешных результатов исследования стала разработка и обучение бинарного классификатора стилей на базе DistilRuBERT. Данная модель достигла высокой точности (96.47% на тестовой выборке) в задаче распознавания стилистической принадлежности текста, подтвердив свою эффективность и потенциал для дальнейшего применения в качестве инструмента оценки или компонента функции потерь в более сложных архитектурах переноса стиля.

Наиболее перспективным из исследованных генеративных подходов представлялось использование модели mBART, адаптированной под схему CycleGAN. Однако, несмотря на корректировку гиперпараметров и временное улучшение качества генерации на валидационных данных в ходе одной из итераций, модель в целом оказалась нестабильной и продемонстрировала выраженную склонность к копированию входного

текста, не достигая желаемого стилистического преобразования. Основная сложность заключалась в настройке многокомпонентной функции потерь и достижении необходимого баланса между сохранением содержания и изменением стиля.

В целом, проведенное исследование подтверждает высокую сложность задачи стилистического переноса текста, особенно при работе с непараллельными данными. Тем не менее, полученные результаты, наработки (включая успешно обученный классификатор стилей) и выявленные ограничения различных подходов создают прочную основу для дальнейших исследований. Перспективными направлениями являются более тонкая настройка и разработка новых функций потерь, внедрение эффективных методов регуляризации для борьбы с копированием, исследование альтернативных архитектур генераторов и дискриминаторов, а также возможное использование частично параллельных данных или техник аугментации для улучшения качества обучения.

Таким образом, несмотря на то, что создание универсальной высококачественной модели стилистического переноса текста требует дальнейших изысканий, данная работа вносит ценный вклад в понимание проблем и вызовов, связанных со стилистической трансформацией текста, и формирует задел для последующих разработок в области обработки естественного языка.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Predictions for journalism in 2023: AI and tech [Электронный ресурс] // Journalism.co.uk. – URL: https://www.journalism.co.uk/news/predictions-for-journalism-in-2023-ai-and-tech/s2/a991916/ (дата обращения: 05.12.2024). – Загл. с экрана. – Яз. англ.
2. Zhu, J.-Y. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks [Электронный ресурс] / J.-Y. Zhu, T. Park, P. Isola, A. A. Efros // Proceedings of the IEEE International Conference on Computer Vision (ICCV). – 2017. – P. 2223–2232. – URL: https://doi.org/10.1109/ICCV.2017.244 (дата обращения: 24.09.2024). – Загл. с экрана. – Яз. англ.
3. Luo, F. Reinforcement learning based text style transfer without parallel training corpus [Электронный ресурс] / F. Luo, P. Li, W. Zhang, J. Wang, M. Zhang // Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. – 2019. – Vol. 1. – P. 3737–3747. – URL: https://aclanthology.org/N19-1320.pdf (дата обращения: 07.10.2024). – Загл. с экрана. – Яз. англ.
4. Lorandi, M. Adapting the CycleGAN Architecture for Text Style Transfer [Электронный ресурс] / M. Lorandi, M. A. Mohamed, K. McGuinness. – Dublin City University, 2023. – URL: https://dcu-nlg.github.io/publications/adapting-the-cyclegan-architecture-for-text-style-transfer/ (дата обращения: 16.10.2024). – Загл. с экрана. – Яз. англ.
5. Han, J. Text Style Transfer with Contrastive Transfer Pattern Mining [Электронный ресурс] / J. Han et al. // Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics. – 2023. – P. 6302–6312. – URL: https://aclanthology.org/2023.acl-long.439/ (дата обращения: 03.11.2024). – Загл. с экрана. – Яз. англ.

6. Sutskever, I. Sequence to Sequence Learning with Neural Networks [Электронный ресурс] / I. Sutskever, O. Vinyals, Q. V. Le // Advances in Neural Information Processing Systems. – 2014. – URL: https://arxiv.org/abs/1409.3215 (дата обращения: 01.02.2025). – Загл. с экрана. – Яз. англ.
7. Vaswani, A. Attention is All You Need [Электронный ресурс] / A. Vaswani et al. // Advances in Neural Information Processing Systems. – 2017. – URL: https://arxiv.org/abs/1706.03762 (дата обращения: 01.02.2025). – Загл. с экрана. – Яз. англ.
8. Liu, Y. Multilingual Denoising Pre-training for Neural Machine Translation [Электронный ресурс] / Y. Liu et al. // ACL. – 2020. – URL: https://arxiv.org/abs/2001.08210 (дата обращения: 06.02.2025). – Загл. с экрана. – Яз. англ.
9. Raffel, C. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer [Электронный ресурс] / C. Raffel et al. // Journal of Machine Learning Research. – 2020. – URL: https://arxiv.org/abs/1910.10683 (дата обращения: 08.02.2025). – Загл. с экрана. – Яз. англ.
10. Zhang, J. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization [Электронный ресурс] / J. Zhang et al. // ICML. – 2020. – URL: https://arxiv.org/abs/1912.08777 (дата обращения: 11.02.2025). – Загл. с экрана. – Яз. англ.
11. Sudhakar, A. “Transforming” Delete, Retrieve, Generate Approach for Controlled Text Style Transfer [Электронный ресурс] / A. Sudhakar, S. Upadhyay, M. Maheswaran // EMNLP. – 2019. – URL: https://arxiv.org/abs/1908.09368 (дата обращения: 15.02.2025). – Загл. с экрана. – Яз. англ.
12. Mikolov, T. Efficient Estimation of Word Representations in Vector Space [Электронный ресурс] / T. Mikolov et al. – 2013. – URL:

https://arxiv.org/abs/1301.3781 (дата обращения: 20.02.2025). – Загл. с экрана. – Яз. англ.

13. Pennington, J. GloVe: Global Vectors for Word Representation [Электронный ресурс] / J. Pennington, R. Socher, C. Manning // EMNLP. – 2014. – URL: https://aclanthology.org/D14-1162/ (дата обращения: 15.02.2025). – Загл. с экрана. – Яз. англ.
14. Bojanowski, P. Enriching Word Vectors with Subword Information [Электронный ресурс] / P. Bojanowski et al. // TACL. – 2017. – URL: https://arxiv.org/abs/1607.04606 (дата обращения: 15.02.2025). – Загл. с экрана. – Яз. англ.
15. Devlin, J. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding [Электронный ресурс] / J. Devlin et al. // NAACL. – 2019. – URL: https://arxiv.org/abs/1810.04805 (дата обращения: 17.03.2025). – Загл. с экрана. – Яз. англ.
16. McInnes, L. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction [Электронный ресурс] / L. McInnes, J. Healy, J. Melville. – 2018. – URL: https://arxiv.org/abs/1802.03426 (дата обращения: 24.03.2025). – Загл. с экрана. – Яз. англ.
17. Dolfik. Russian Telegram Chats History [Электронный ресурс] // Kaggle. – 2018. – URL: https://www.kaggle.com/datasets/dolfik/russian-telegram-chats-history (дата обращения: 04.11.2024). – Загл. с экрана. – Яз. англ.
18. Creative Commons. CC0 1.0 Universal: Public Domain Dedication [Электронный ресурс]. – URL: https://creativecommons.org/public-domain/cc0/ (дата обращения: 08.12.2024). – Загл. с экрана. – Яз. англ.
19. ЛитЛайф. Классическая проза [Электронный ресурс] // Litlife.club. – URL: https://litlife.club/genres/56-klassicheskaya-proza (дата обращения: 05.11.2024). – Загл. с экрана. – Яз. рус.

20. Национальный корпус русского языка. Словарь векторных представлений слов [Электронный ресурс]. – URL: https://natasha.github.io/navec/ (дата обращения: 12.11.2024). – Загл. с экрана. – Яз. рус.
21. Hochreiter, S. Long short-term memory [Электронный ресурс] / S. Hochreiter, J. Schmidhuber // Neural Computation. – 1997. – Vol. 9, № 8. – P. 1735–1780. – URL: https://www.bioinf.jku.at/publications/older/2604.pdf (дата обращения: 20.10.2024). – Загл. с экрана. – Яз. англ.
22. sn4kebyt3. ru-bart-large [Электронный ресурс] // Hugging Face. URL: https://huggingface.co/sn4kebyt3/ru-bart-large (дата обращения: 23.02.2025). Загл. с экрана. Яз. англ.

Приложение А

Сбор данных

```
!pip install selenium
!pip install webdriver_manager


from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
import time
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import requests
from bs4 import BeautifulSoup
import os
import zipfile
import re


def get_pages_with_books(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')
    books_links = soup.find_all('a', class_='text-decoration-none')
    books_urls = [link['href'] for link in books_links if ('href' in
link.attrs and re.match(r"^https://litlife\.club/books/\d+$",
link['href']))]
    return books_urls


def download_fb2_file(page_url, download_dir):
    options = webdriver.ChromeOptions()
    prefs = {"download.default_directory": download_dir}
    options.add_experimental_option("prefs", prefs)
    service = Service(ChromeDriverManager().install())
    driver = webdriver.Chrome(service=service, options=options)
    try:
        driver.get(page_url)
        wait = WebDriverWait(driver, 10)
        download_button =
wait.until(EC.element_to_be_clickable((By.XPATH, "//a[contains(@href,
'.fb2.zip')]")))
```

```
        download_button.click()
        time.sleep(15)
        print(f”Файл должен быть загружен в: {download_dir}”)
    except Exception as e:
        print(f”Произошла ошибка: {e}”)
    finally:
        driver.quit()


download_dir = ”data”
count_of_pages = 20
for x in range(count_of_pages):
    main_page_url =
f”https://litlife.club/genres/56-klassicheskaya-proza?page={x}”
    books_url = get_pages_with_books(main_page_url)
    for book in books_url:
        download_fb2_file(book, download_dir)


all_files = os.listdir(”data”)
for file_zip in all_files:
    with zipfile.ZipFile(f”data/{file_zip}”, 'r') as zip_ref:
        for file in zip_ref.namelist():
            if file.endswith(”.fb2”):
                try:
                    fb2_file = zip_ref.open(file)
                    fb2_content = fb2_file.read()
                    text_content = ''
                    soup = BeautifulSoup(fb2_content, 'lxml-xml')
                    for para in soup.find_all(['p', 'section']):
                        text_content += para.get_text(separator='\n',
strip=True) + '\n'

                    with open(f”books_txt/{file}.txt”, 'w',
encoding='utf-8') as txt_file:
                        txt_file.write(text_content)
                except FileNotFoundError:
                    ...
```

Приложение Б

Небольшая предобработка данных и формирование датасета

```
import json
import numpy
import os
import shutil
import zipfile
from google.colab import drive
import pandas as pd
import re
import nltk
from nltk.tokenize import sent_tokenize


nltk.download('punkt')


from google.colab import drive
drive.mount('/content/drive')


books_path =
”/content/drive/MyDrive/CycleGAN_for_TST_problem/books_txt.zip”
with zipfile.ZipFile(books_path, ”r”) as zip_ref:
    for filename in zip_ref.namelist():
        zip_ref.extract(filename, ”/content/books”)


tg_messages_path =
”/content/drive/MyDrive/CycleGAN_for_TST_problem/telegram_comments.zip
”
with zipfile.ZipFile(tg_messages_path, ”r”) as zip_ref:
    for filename in zip_ref.namelist():
        if ”telegram/telegram/” not in filename:
            zip_ref.extract(filename, ”/content/tg”)


low_chars_border = 8
high_chars_border = 200


lit_data = set()
```

```
all_files = os.listdir(”/content/books/books_txt/”)
for file in all_files:
    file_txt = open(f”/content/books/books_txt/{file}”)
    data_txt = file_txt.read()
    sentences = sent_tokenize(data_txt)
    for sentence in sentences:
        sentence = sentence.strip()
        sentence = re.sub(”[^a-яA-Я–––—,.!?:«» ]”, ””, sentence)
        while len(sentence) > 0 and sentence[0] in ”–––—,.!?:«» ”:
            sentence = sentence[1:]
        if low_chars_border < len(sentence) < high_chars_border:
            lit_data.add(sentence)


lit = pd.Series(list(lit_data))


comnts = set()
allFiles = os.listdir(”/content/tg/telegram/”)
for filename in allFiles:
    if len(comnts) > len(lit) - 1:
                break
    with open(f”/content/tg/telegram/{filename}”, 'r') as f:
            groupedComments = json.load(f)
            for comment in groupedComments:
                row = comment[”text”]
                sentences = sent_tokenize(row)
                for sentence in sentences:
                    sentence = sentence.strip()
                    sentence = re.sub(”[^a-яA-Я–––—,.!?:«» ]”, ””,
sentence)
                    while len(sentence) > 0 and sentence[0] in
”–––—,.!?:«» ”:
                        sentence = sentence[1:]
                    if low_chars_border < len(sentence) <
high_chars_border:
                        comnts.add(sentence)
                        if len(comnts) > len(lit) - 1:
                            break
                if len(comnts) > len(lit) - 1:
                            break
```

```
com = pd.Series(list(comnts))


data = pd.DataFrame({”lit_text”: lit, ”tg_text”: com})


data.to_csv(”/content/drive/MyDrive/CycleGAN_for_TST_problem/data.csv”
, index=False)
```

Приложение В

Векторизация данных

```
!pip install navec


!wget
https://storage.yandexcloud.net/natasha-navec/packs/navec_hudlit_v1_12
B_500K_300d_100q.tar


import pandas as pd
import numpy as np
import torch
from navec import Navec
import nltk
nltk.download('punkt_tab')
from nltk.tokenize import word_tokenize
from sklearn.decomposition import PCA
from collections import Counter
import seaborn as sb
import matplotlib.pyplot as plt


from google.colab import drive
drive.mount('/content/drive')


df =
pd.read_csv(”/content/drive/MyDrive/CycleGAN_for_TST_problem/data.csv”
)


path = 'navec_hudlit_v1_12B_500K_300d_100q.tar'
navec = Navec.load(path)


embeddings = list()
for word in navec.vocab.words:
        embeddings.append(navec[word])
embeddings = np.array(embeddings)
```

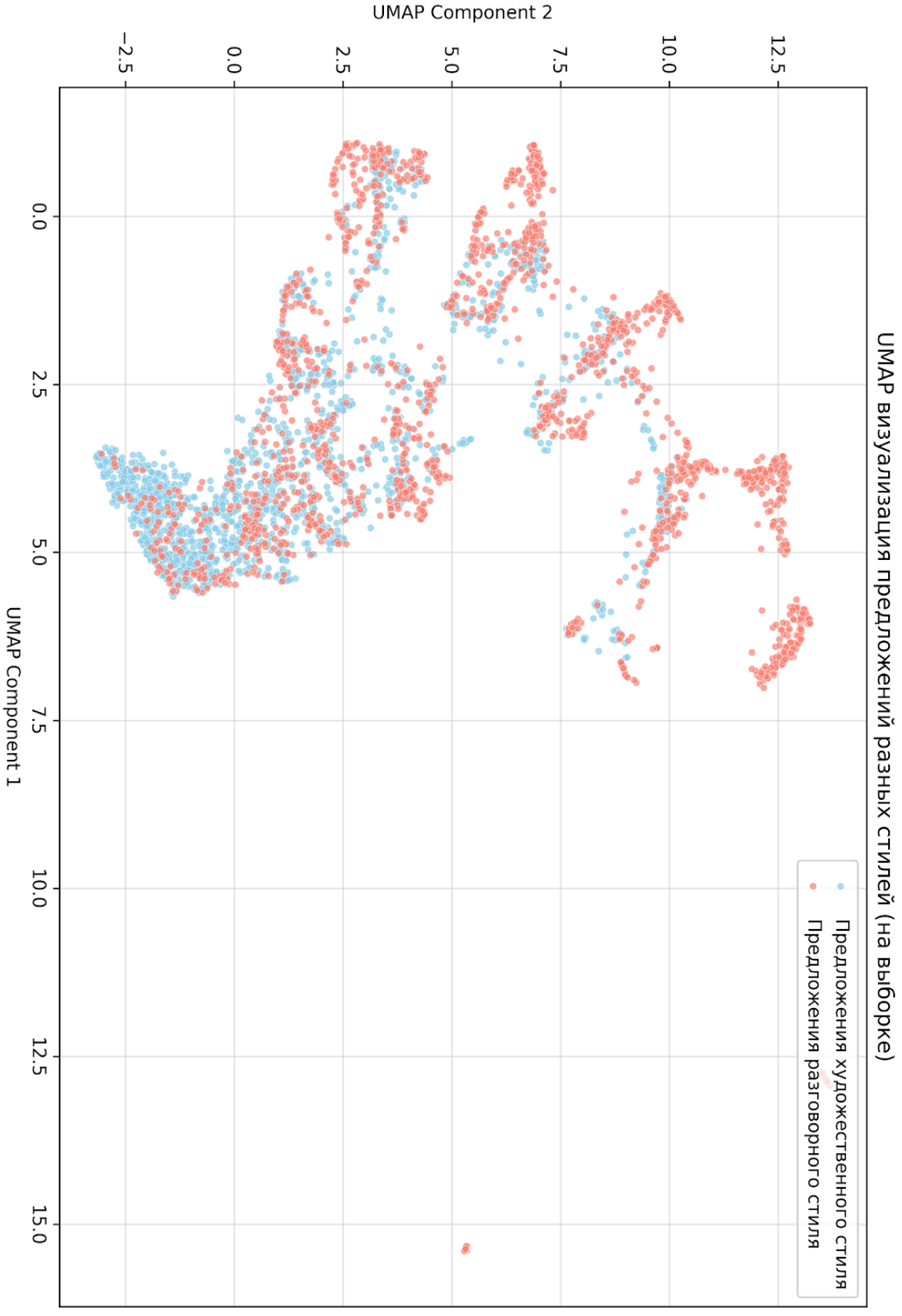
```
target_dim = 10
pca = PCA(n_components=target_dim)
reduced_embeddings = pca.fit_transform(embeddings)


reduced_navec = {word: np.array(reduced_embeddings[i],
dtype=np.float16) for i, word in enumerate(navec.vocab.words)}


punkt_vectors = {
    ”.”: np.array([0.0] * (target_dim - 1) + [1.0], dtype=np.float16),
    ”!”: np.array([0.0] * (target_dim - 1) + [0.9], dtype=np.float16),
    ”?”: np.array([0.0] * (target_dim - 1) + [0.8], dtype=np.float16),
    ”,”: np.array([0.0] * (target_dim - 1) + [0.5], dtype=np.float16),
    ”:”: np.array([0.0] * (target_dim - 1) + [0.6], dtype=np.float16),
    ”–”: np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    ”-”: np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    ”—”: np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    ”«”: np.array([0.0] * (target_dim - 1) + [0.2], dtype=np.float16),
    ”»”: np.array([0.0] * (target_dim - 1) + [0.3], dtype=np.float16),
}


vector_lengths = Counter(len(word_tokenize(line)) for line in
df.lit_text)


lengths = sorted(vector_lengths.keys())
counts = [vector_lengths[length] for length in lengths]
plt.figure(figsize=(14, 7))
data = pd.DataFrame({'Length': lengths, 'Count': counts})
sb.barplot(data=data, x='Length', y='Count', color='skyblue',
edgecolor='black')
plt.xlabel('Количество токенов в предложении', fontsize=14)
plt.ylabel('Количество предложений', fontsize=14)
plt.title('Распределение количества токенов в предложениях',
fontsize=14)
plt.grid(axis='y', linestyle='--', alpha=0.5)
plt.tight_layout()


max_vector_length = 40
```

```
lit_vectors = list()
for i, tokens in enumerate(lit_tokens):
    padding_for = max_vector_length - len(tokens)
    vector = [np.zeros(target_dim)] * padding_for
    for word in tokens:
        try:
            vector.append(reduced_navec[word])
        except KeyError:
            try:
                vector.append(punkt_vectors[word])
            except KeyError:
                vector.append(reduced_navec[”<unk>”])
    vector = np.array(vector,
dtype=np.float16).reshape(max_vector_length * target_dim)
    lit_vectors.append(vector)


tg_tokens = []
for sentence in df.tg_text:
    tokens = word_tokenize(sentence)
    if len(tokens) > max_vector_length:
        continue
    else:
        tg_tokens.append(tokens)


tg_vectors = list()
for i, tokens in enumerate(tg_tokens):
    padding_for = max_vector_length - len(tokens)
    vector = [np.zeros(target_dim)] * padding_for
    for word in tokens:
        try:
            vector.append(reduced_navec[word])
        except KeyError:
            try:
                vector.append(punkt_vectors[word])
            except KeyError:
                vector.append(reduced_navec[”<unk>”])
    vector = np.array(vector,
dtype=np.float16).reshape(max_vector_length * target_dim)
    tg_vectors.append(vector)
```

```
length = min(len(lit_vectors), len(tg_vectors))


lit_vectors_np = np.array(lit_vectors[:length])
tg_vectors_np = np.array(tg_vectors[:length])


np.save(”/content/drive/MyDrive/CycleGAN_for_TST_problem/vectorized_li
t.npy”, lit_vectors_np)
np.save(”/content/drive/MyDrive/CycleGAN_for_TST_problem/vectorized_tg
.npy”, tg_vectors_np)
```

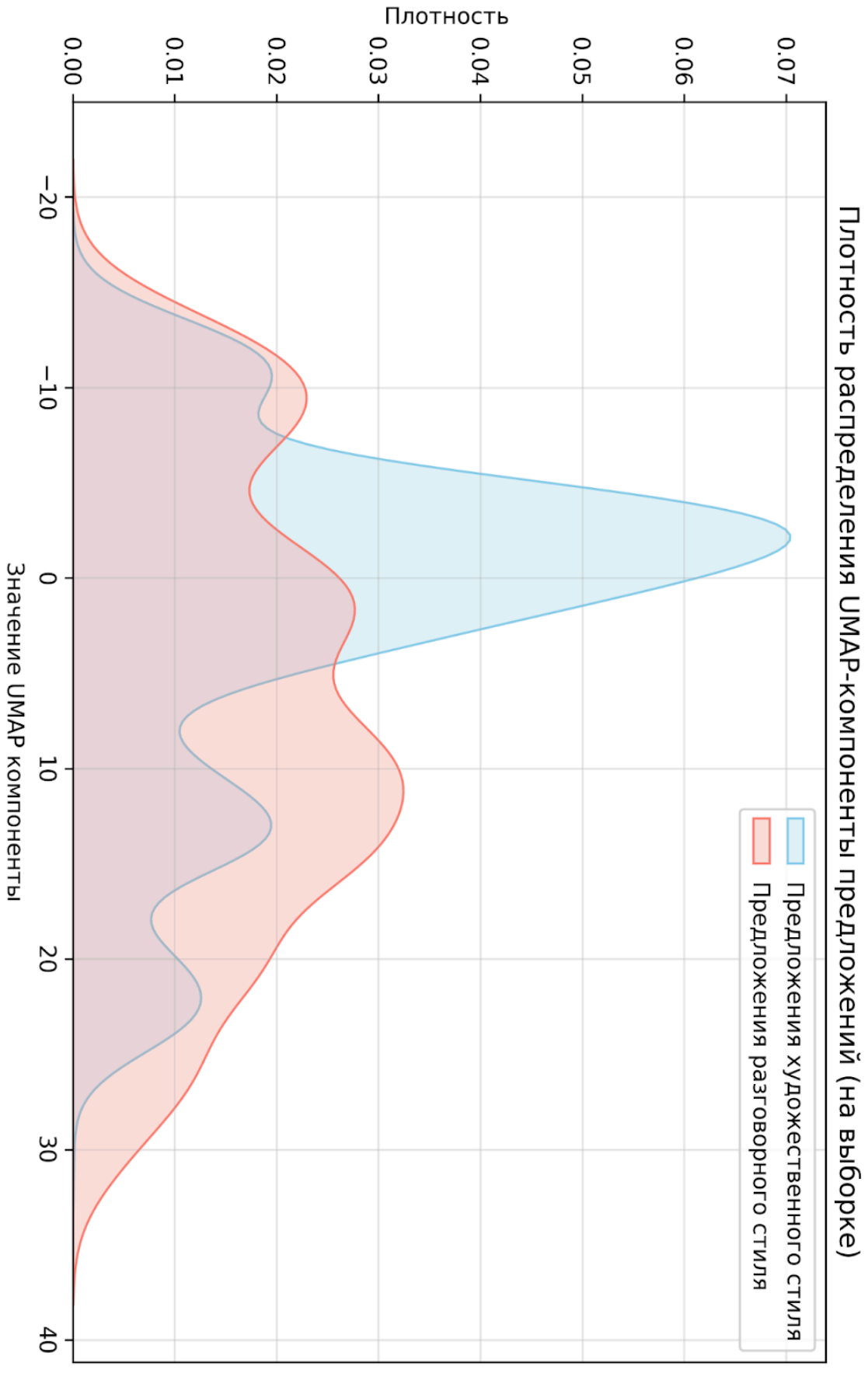
Приложение Г

UMAP визуализация предложений разных стилей



Приложение Д

Плотность распределения UMAP-компоненты предложений



Приложение Ж

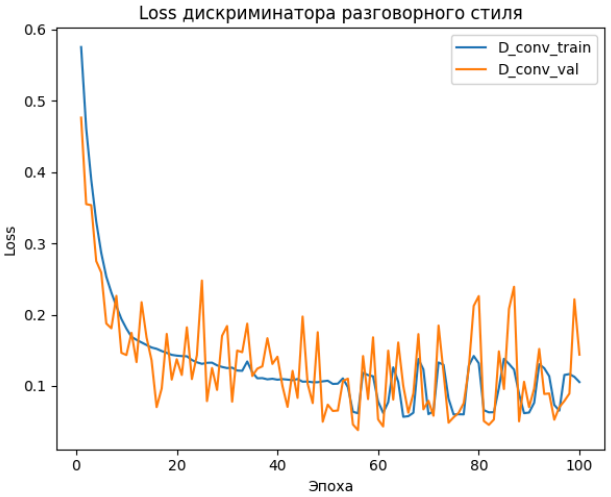
Визуализация одномерных векторных представлений предложений

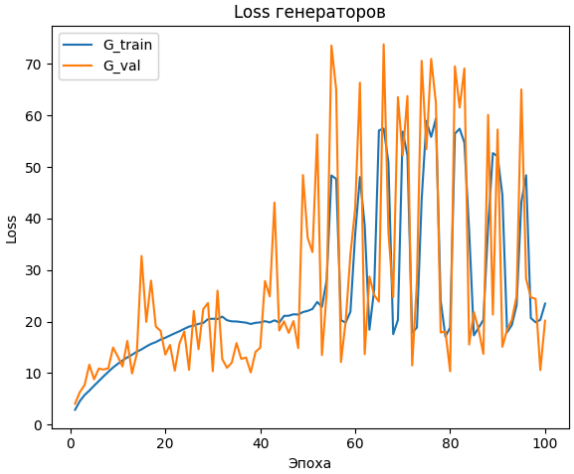
```
import numpy as np
import seaborn as sns
import umap


lit_vectors_np = np.load('vectorized_lit.npy')
conv_vectors_np = np.load('vectorized_tg.npy')


sample_size = 2000
num_rows = lit_vectors_np.shape[0]
indices = np.random.choice(num_rows, size=sample_size, replace=False)
lit_vectors_sample = lit_vectors_np[indices]
conv_vectors_sample = conv_vectors_np[indices]
all_vectors_sample_np = np.concatenate((lit_vectors_sample,
conv_vectors_sample), axis=0)


reducer_2d = umap.UMAP(n_components=2,
                       n_neighbors=15,
                       min_dist=0.1,
                       verbose=True)


embedding_2d_all_sample =
reducer_2d.fit_transform(all_vectors_sample_np)


umap_2d_lit_sample = embedding_2d_all_sample[:len(lit_vectors_sample)]
umap_2d_conv_sample =
embedding_2d_all_sample[len(lit_vectors_sample):]


plt.figure(figsize=(12, 8))
sns.scatterplot(x=umap_2d_lit_sample[:, 0], y=umap_2d_lit_sample[:,
1], color='skyblue', label='Предложения художественного стиля',
alpha=0.7, s=15)
sns.scatterplot(x=umap_2d_conv_sample[:, 0], y=umap_2d_conv_sample[:,
1], color='salmon', label='Предложения разговорного стиля', alpha=0.7,
s=15)
```

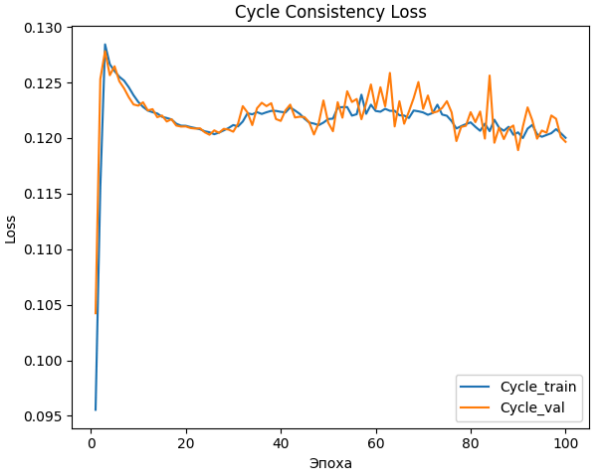
```
plt.xlabel('UMAP Component 1')
plt.ylabel('UMAP Component 2')
plt.title('UMAP визуализация предложений разных стилей (на выборке)')
plt.legend(loc='upper right')
plt.grid(True, alpha=0.3)
plt.savefig('umap_2d.png', dpi=300, bbox_inches='tight')
plt.show()



reducer_1d = umap.UMAP(n_components=1,
                       n_neighbors=15,
                       min_dist=0.1,
                       verbose=True)


embedding_1d_all_sample =
reducer_1d.fit_transform(all_vectors_sample_np)


umap_1d_lit_sample = embedding_1d_all_sample[:len(lit_vectors_sample)]
umap_1d_conv_sample =
embedding_1d_all_sample[len(lit_vectors_sample):]


plt.figure(figsize=(10, 6))
sns.kdeplot(umap_1d_lit_sample.flatten(), color='skyblue',
label='Предложения художественного стиля', fill=True)
sns.kdeplot(umap_1d_conv_sample.flatten(), color='salmon',
label='Предложения разговорного стиля', fill=True)
plt.xlabel('Значение UMAP компоненты')
plt.ylabel('Плотность')
plt.title('Плотность распределения UMAP-компоненты предложений (на
выборке)')
plt.legend(loc='upper right')
plt.grid(True, alpha=0.3)
plt.savefig('umap_1d.png', dpi=300, bbox_inches='tight')
plt.show()
```

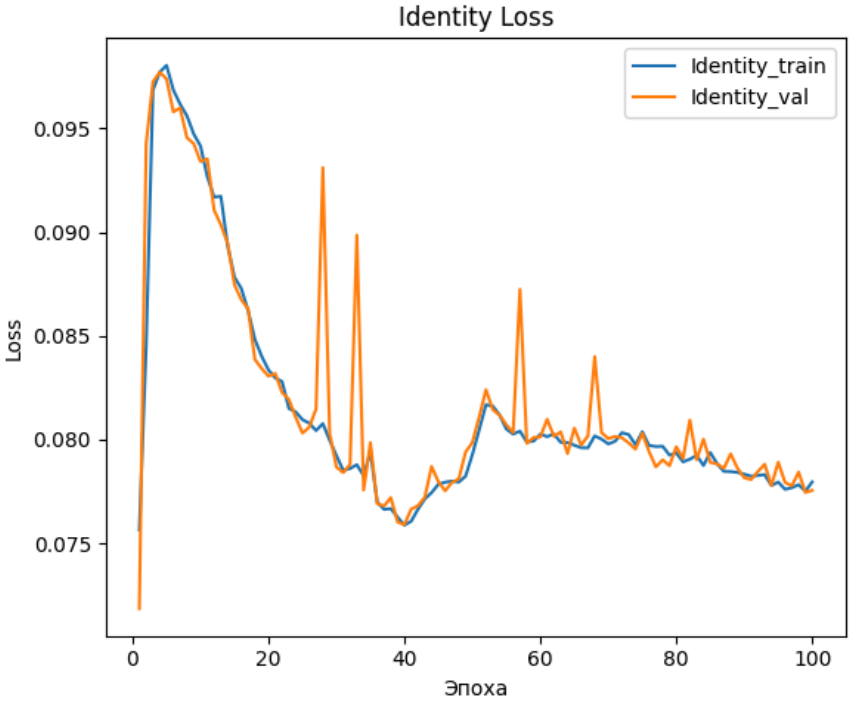
Приложение И

Графики потерь модели CycleGAN V1, обученной на эмбеддингах Navec



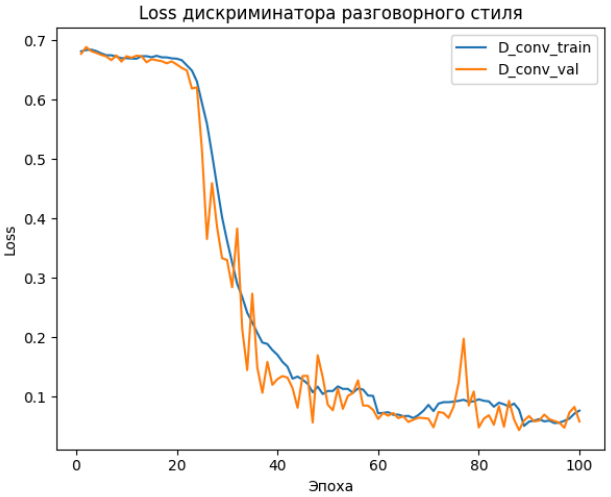


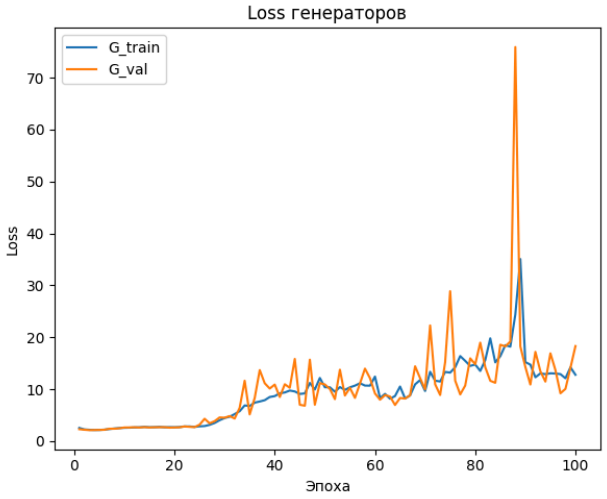


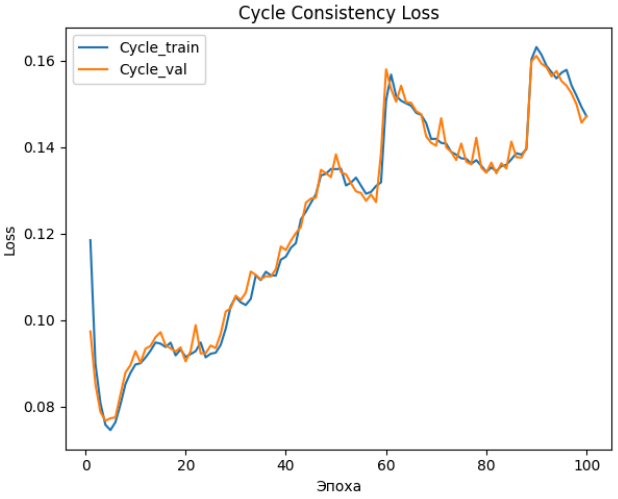


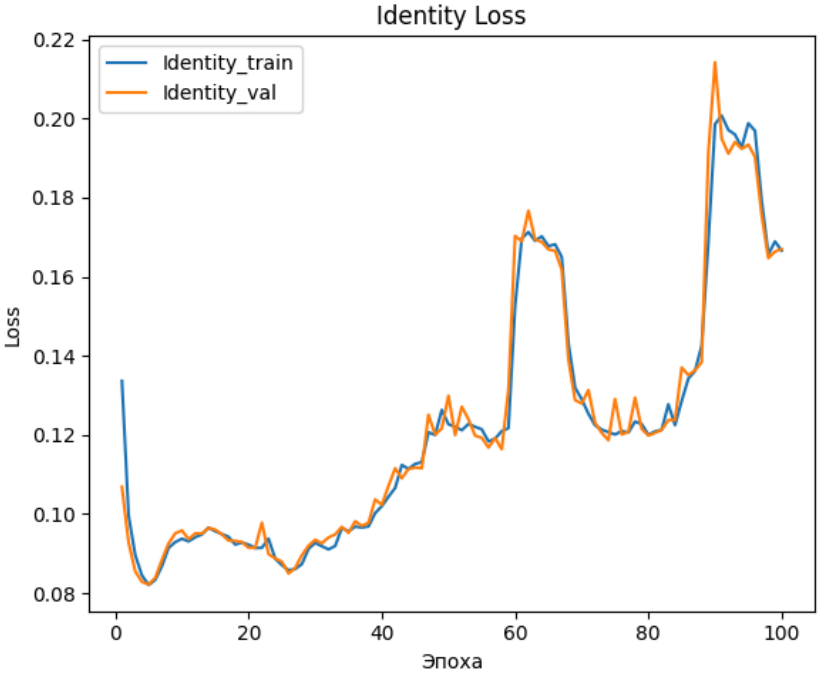
Приложение К

Графики потерь модели CycleGAN V2, обученной на эмбеддингах Navec



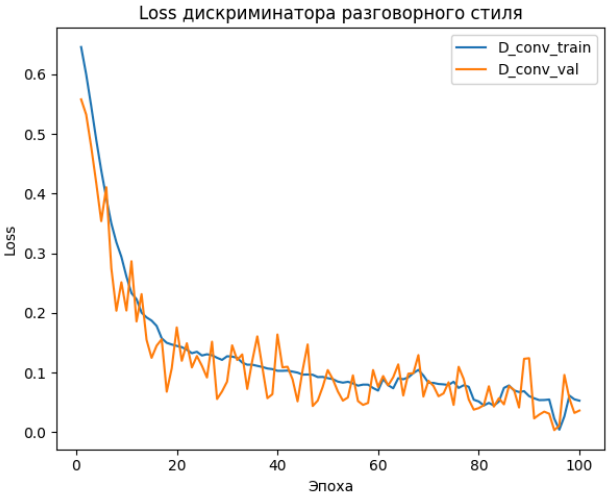


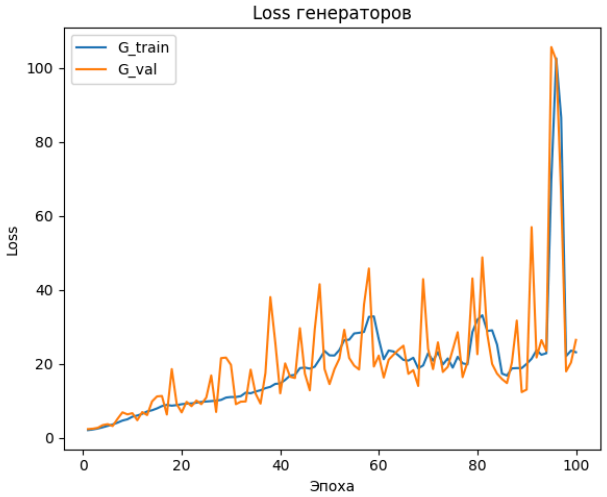


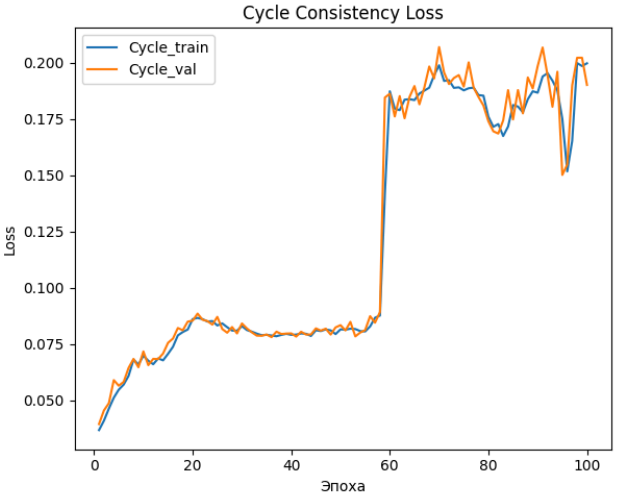


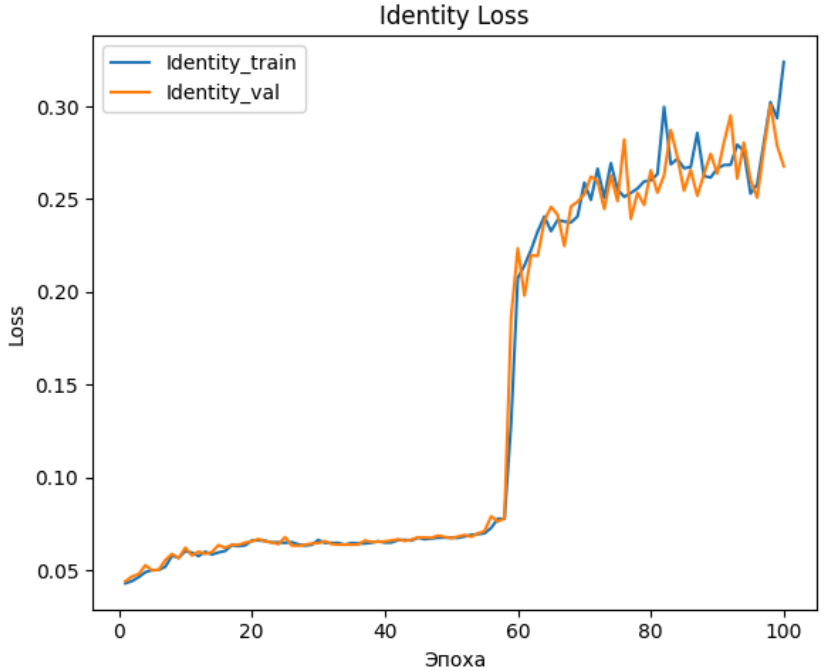
Приложение Л

Графики потерь модели CycleGAN V3, обученной на эмбеддингах Navec



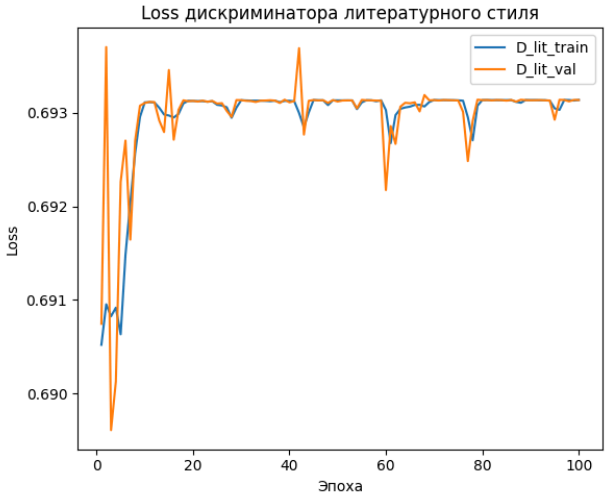


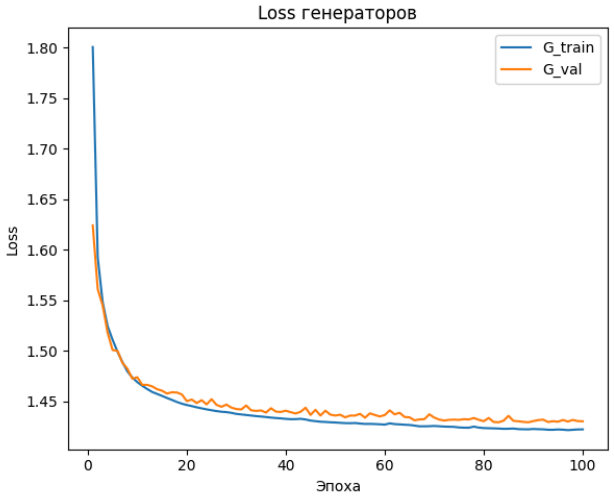


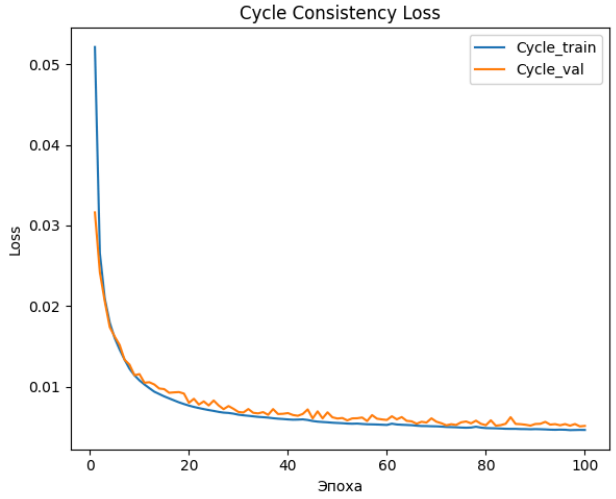


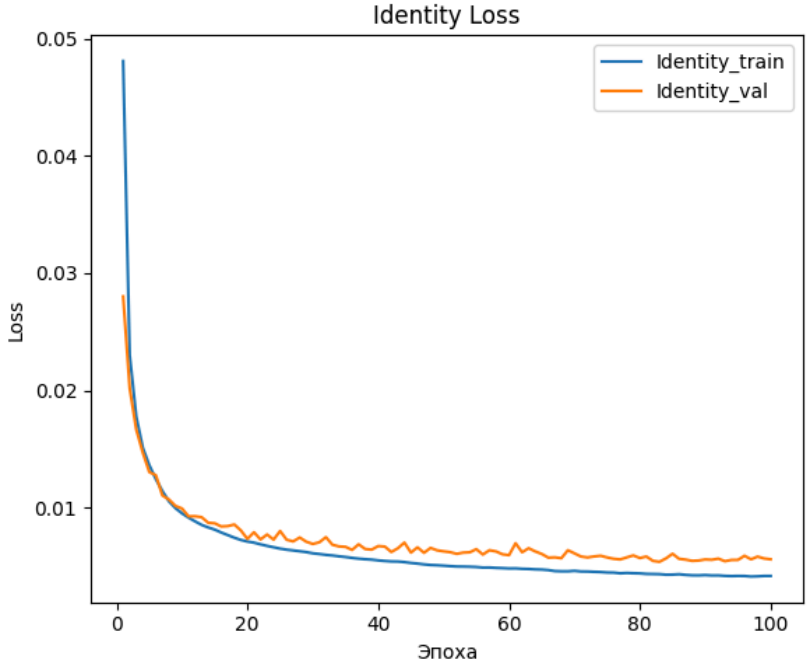
Приложение М

Графики потерь модели CycleGAN V4, обученной на эмбеддингах Navec









Приложение Н

Построение и обучение моделей CycleGAN на эмбеддингах Navec

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from sklearn.model_selection import train_test_split
from torch.utils.data import Dataset, DataLoader
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib
from sklearn.decomposition import PCA
from sklearn.decomposition import IncrementalPCA
import torch.nn.functional as F
import os
import re
import nltk
from nltk.tokenize import sent_tokenize
nltk.download('punkt')
nltk.download('punkt_tab')
from nltk.tokenize import word_tokenize
from navec import Navec
import scipy.spatial.distance


lit_vectors_np = np.load(”vectorized_lit.npy”)
conv_vectors_np = np.load(”vectorized_tg.npy”)

# -------------!!! Первая версия !!!-------------
# --- Гиперпараметры ---
vector_dimension = 400
batch_size = 64
learning_rate_generators = 0.0002
learning_rate_discriminators = 0.0001
num_epochs = 100
lambda_cycle = 5
lambda_identity = 5
beta1_adam = 0.5
beta2_adam = 0.999
```

```
val_split_ratio = 0.2
random_seed = 42
lstm_hidden_dim_generator = 512
lstm_num_layers_generator = 1
lstm_hidden_dim_discriminator = 256
lstm_num_layers_discriminator = 1
discriminator_layer_size_hidden = 128
leaky_relu_negative_slope = 0.2
num_dataloader_workers = 0
print_batch_interval = 500
plot_loss_interval_epochs = 5
output_dir = model_output_dir = ”version_1”
discriminator_dropout_rate = 0.2
discriminator_weight_decay = 1e-5
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


class Generator(nn.Module):
    def __init__(self, input_dim, output_dim,
lstm_hidden_dim=lstm_hidden_dim_generator,
num_layers=lstm_num_layers_generator):
        super(Generator, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim,
hidden_size=lstm_hidden_dim, num_layers=num_layers, batch_first=True)
        self.linear_out = nn.Linear(lstm_hidden_dim, output_dim)

    def forward(self, x):
        x = x.unsqueeze(1)
        lstm_out, _ = self.lstm(x)
        output = self.linear_out(lstm_out[:, -1, :])
        return output


class Discriminator(nn.Module):
    def __init__(self, input_dim,
lstm_hidden_dim=lstm_hidden_dim_discriminator,
num_layers=lstm_num_layers_discriminator,
negative_slope=leaky_relu_negative_slope,
layer_size_hidden=discriminator_layer_size_hidden):
        super(Discriminator, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim,
hidden_size=lstm_hidden_dim, num_layers=num_layers, batch_first=True)
        self.linear_hidden = nn.Linear(lstm_hidden_dim,
layer_size_hidden)
```

```
        self.leaky_relu = nn.LeakyReLU(negative_slope)
        self.linear_out = nn.Linear(layer_size_hidden, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = x.unsqueeze(1)
        lstm_out, _ = self.lstm(x)
        hidden_out = self.leaky_relu(self.linear_hidden(lstm_out[:,
-1, :]))
        output = self.linear_out(hidden_out)
        return self.sigmoid(output)


generator_LitToConv = Generator(vector_dimension,
vector_dimension).to(device)
generator_ConvToLit = Generator(vector_dimension,
vector_dimension).to(device)
discriminator_Literary = Discriminator(vector_dimension).to(device)
discriminator_Conversational =
Discriminator(vector_dimension).to(device)

optimizer_G = optim.Adam(list(generator_LitToConv.parameters()) +
list(generator_ConvToLit.parameters()),
                         lr=learning_rate_generators,
betas=(beta1_adam, beta2_adam))
optimizer_D_lit = optim.Adam(discriminator_Literary.parameters(),
lr=learning_rate_discriminators, betas=(beta1_adam, beta2_adam),
weight_decay=discriminator_weight_decay)
optimizer_D_conv =
optim.Adam(discriminator_Conversational.parameters(),
lr=learning_rate_discriminators, betas=(beta1_adam, beta2_adam),
weight_decay=discriminator_weight_decay)


# -------------!!! Вторая версия !!!-------------
# --- Гиперпараметры ---
vector_dimension = 400
batch_size = 64
learning_rate_generators = 0.00025
learning_rate_discriminators = 0.00005
num_epochs = 100
lambda_cycle = 7
lambda_identity = 2
beta1_adam = 0.5
```

```
beta2_adam = 0.999
val_split_ratio = 0.2
random_seed = 42
lstm_hidden_dim_generator = 256
lstm_num_layers_generator = 2
lstm_hidden_dim_discriminator = 256
lstm_num_layers_discriminator = 1
discriminator_layer_size_hidden = 128
leaky_relu_negative_slope = 0.2
num_dataloader_workers = 0
print_batch_interval = 500
plot_loss_interval_epochs = 5
output_dir = model_output_dir = ”version_2”
discriminator_dropout_rate = 0.2
discriminator_weight_decay = 1e-5
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


class Generator(nn.Module):
    def __init__(self, input_dim, output_dim,
lstm_hidden_dim=lstm_hidden_dim_generator,
num_layers=lstm_num_layers_generator): # Используем гиперпараметры
        super(Generator, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim,
hidden_size=lstm_hidden_dim, num_layers=num_layers, batch_first=True)
        self.linear_out = nn.Linear(lstm_hidden_dim, output_dim)

    def forward(self, x):
        x = x.unsqueeze(1)
        lstm_out, _ = self.lstm(x)
        output = self.linear_out(lstm_out[:, -1, :])
        return output


class Discriminator(nn.Module):
    def __init__(self, input_dim, lstm_hidden_dim, num_layers,
negative_slope, layer_size_hidden, dropout_rate):
        super(Discriminator, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim,
hidden_size=lstm_hidden_dim, num_layers=num_layers, batch_first=True)
        self.linear_hidden = nn.Linear(lstm_hidden_dim,
layer_size_hidden)
        self.leaky_relu = nn.LeakyReLU(negative_slope)
```

```
        self.dropout = nn.Dropout(dropout_rate)
        self.linear_out = nn.Linear(layer_size_hidden, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = x.unsqueeze(1)
        lstm_out, _ = self.lstm(x)
        hidden_out = self.leaky_relu(self.linear_hidden(lstm_out[:,
-1, :]))
        hidden_out_dropout = self.dropout(hidden_out)
        output = self.linear_out(hidden_out_dropout)
        return self.sigmoid(output)


generator_LitToConv = Generator(vector_dimension, vector_dimension,
lstm_hidden_dim_generator, lstm_num_layers_generator).to(device)
generator_ConvToLit = Generator(vector_dimension, vector_dimension,
lstm_hidden_dim_generator, lstm_num_layers_generator).to(device)
discriminator_Literary = Discriminator(vector_dimension,
lstm_hidden_dim_discriminator, lstm_num_layers_discriminator,
leaky_relu_negative_slope, discriminator_layer_size_hidden,
discriminator_dropout_rate).to(device)
discriminator_Conversational = Discriminator(vector_dimension,
lstm_hidden_dim_discriminator, lstm_num_layers_discriminator,
leaky_relu_negative_slope, discriminator_layer_size_hidden,
discriminator_dropout_rate).to(device)

optimizer_G = optim.Adam(list(generator_LitToConv.parameters()) +
list(generator_ConvToLit.parameters()),
                         lr=learning_rate_generators,
betas=(beta1_adam, beta2_adam))
optimizer_D_lit = optim.Adam(discriminator_Literary.parameters(),
lr=learning_rate_discriminators, betas=(beta1_adam, beta2_adam),
weight_decay=discriminator_weight_decay)
optimizer_D_conv =
optim.Adam(discriminator_Conversational.parameters(),
lr=learning_rate_discriminators, betas=(beta1_adam, beta2_adam),
weight_decay=discriminator_weight_decay)


# -------------!!! Третья версия !!!-------------
# --- Гиперпараметры ---
vector_dimension = 400
batch_size = 64
learning_rate_generators = 0.00025
```

```
learning_rate_discriminators = 0.00005
num_epochs = 100
lambda_cycle = 10
lambda_identity = 3.5
beta1_adam = 0.5
beta2_adam = 0.999
val_split_ratio = 0.2
random_seed = 42
lstm_hidden_dim_generator = 256
lstm_num_layers_generator = 2
lstm_hidden_dim_discriminator = 256
lstm_num_layers_discriminator = 1
discriminator_layer_size_hidden = 128
leaky_relu_negative_slope = 0.2
num_dataloader_workers = 0
print_batch_interval = 500
plot_loss_interval_epochs = 5
lambda_cycle_l1 = 0.5
lambda_cycle_cosine = 0.5
lambda_identity_l1 = 0.5
lambda_identity_cosine = 0.5
output_dir = model_output_dir = ”version_3”
discriminator_dropout_rate = 0.2
discriminator_weight_decay = 1e-5
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


class Generator(nn.Module):
    def __init__(self, input_dim, output_dim,
lstm_hidden_dim=lstm_hidden_dim_generator,
num_layers=lstm_num_layers_generator):
        super(Generator, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim,
hidden_size=lstm_hidden_dim, num_layers=num_layers, batch_first=True)
        self.linear_out = nn.Linear(lstm_hidden_dim, output_dim)

    def forward(self, x):
        x = x.unsqueeze(1)
        lstm_out, _ = self.lstm(x)
        output = self.linear_out(lstm_out[:, -1, :])
        return output
```

```
class Discriminator(nn.Module):
    def __init__(self, input_dim, lstm_hidden_dim, num_layers,
negative_slope, layer_size_hidden, dropout_rate):
        super(Discriminator, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim,
hidden_size=lstm_hidden_dim, num_layers=num_layers, batch_first=True)
        self.linear_hidden = nn.Linear(lstm_hidden_dim,
layer_size_hidden)
        self.leaky_relu = nn.LeakyReLU(negative_slope)
        self.dropout = nn.Dropout(dropout_rate)
        self.linear_out = nn.Linear(layer_size_hidden, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = x.unsqueeze(1)
        lstm_out, _ = self.lstm(x)
        hidden_out = self.leaky_relu(self.linear_hidden(lstm_out[:,
-1, :]))
        hidden_out_dropout = self.dropout(hidden_out)
        output = self.linear_out(hidden_out_dropout)
        return self.sigmoid(output)


generator_LitToConv = Generator(vector_dimension, vector_dimension,
lstm_hidden_dim_generator, lstm_num_layers_generator).to(device)
generator_ConvToLit = Generator(vector_dimension, vector_dimension,
lstm_hidden_dim_generator, lstm_num_layers_generator).to(device)
discriminator_Literary = Discriminator(vector_dimension,
lstm_hidden_dim_discriminator, lstm_num_layers_discriminator,
leaky_relu_negative_slope, discriminator_layer_size_hidden,
discriminator_dropout_rate).to(device)
discriminator_Conversational = Discriminator(vector_dimension,
lstm_hidden_dim_discriminator, lstm_num_layers_discriminator,
leaky_relu_negative_slope, discriminator_layer_size_hidden,
discriminator_dropout_rate).to(device)


optimizer_G = optim.Adam(list(generator_LitToConv.parameters()) +
list(generator_ConvToLit.parameters()),
                         lr=learning_rate_generators,
betas=(beta1_adam, beta2_adam))
optimizer_D_lit = optim.Adam(discriminator_Literary.parameters(),
lr=learning_rate_discriminators, betas=(beta1_adam, beta2_adam),
weight_decay=discriminator_weight_decay)
optimizer_D_conv =
optim.Adam(discriminator_Conversational.parameters(),
```

```
lr=learning_rate_discriminators, betas=(beta1_adam, beta2_adam),
weight_decay=discriminator_weight_decay)
# -------------!!! Четвёртая версия !!!-------------
# --- Гиперпараметры ---
vector_dimension = 400
batch_size = 64
learning_rate_generators = 0.0002
learning_rate_discriminators = 0.00005
num_epochs = 100
lambda_cycle = 4
lambda_identity = 4
beta1_adam = 0.5
beta2_adam = 0.999
val_split_ratio = 0.2
random_seed = 42
lstm_hidden_dim_generator = 400
lstm_num_layers_generator = 3
lstm_hidden_dim_discriminator = 160
lstm_num_layers_discriminator = 1
discriminator_layer_size_hidden = 40
leaky_relu_negative_slope = 0.2
num_dataloader_workers = 0
print_batch_interval = 500
plot_loss_interval_epochs = 5
lambda_cycle_l1 = 0.5
lambda_cycle_cosine = 0.5
lambda_identity_l1 = 0.5
lambda_identity_cosine = 0.5
output_dir = model_output_dir = ”version_4”
discriminator_dropout_rate = 0.4
discriminator_weight_decay = 1e-3
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


class Generator(nn.Module):
    def __init__(self, input_dim, output_dim,
lstm_hidden_dim=lstm_hidden_dim_generator,
num_layers=lstm_num_layers_generator):
        super(Generator, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim,
hidden_size=lstm_hidden_dim, num_layers=num_layers, batch_first=True)
        self.linear_out = nn.Linear(lstm_hidden_dim, output_dim)
```

```
    def forward(self, x):
        x = x.unsqueeze(1)
        lstm_out, _ = self.lstm(x)
        output = self.linear_out(lstm_out[:, -1, :])
        return output


class Discriminator(nn.Module):
    def __init__(self, input_dim, lstm_hidden_dim, num_layers,
negative_slope, layer_size_hidden, dropout_rate):
        super(Discriminator, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim,
hidden_size=lstm_hidden_dim, num_layers=num_layers, batch_first=True)
        self.linear_hidden = nn.Linear(lstm_hidden_dim,
layer_size_hidden)
        self.leaky_relu = nn.LeakyReLU(negative_slope)
        self.dropout = nn.Dropout(dropout_rate)
        self.linear_out = nn.Linear(layer_size_hidden, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = x.unsqueeze(1)
        lstm_out, _ = self.lstm(x)
        hidden_out = self.leaky_relu(self.linear_hidden(lstm_out[:,
-1, :]))
        hidden_out_dropout = self.dropout(hidden_out)
        output = self.linear_out(hidden_out_dropout)
        return self.sigmoid(output)


generator_LitToConv = Generator(vector_dimension, vector_dimension,
lstm_hidden_dim_generator, lstm_num_layers_generator).to(device)
generator_ConvToLit = Generator(vector_dimension, vector_dimension,
lstm_hidden_dim_generator, lstm_num_layers_generator).to(device)
discriminator_Literary = Discriminator(vector_dimension,
lstm_hidden_dim_discriminator, lstm_num_layers_discriminator,
leaky_relu_negative_slope, discriminator_layer_size_hidden,
discriminator_dropout_rate).to(device)
discriminator_Conversational = Discriminator(vector_dimension,
lstm_hidden_dim_discriminator, lstm_num_layers_discriminator,
leaky_relu_negative_slope, discriminator_layer_size_hidden,
discriminator_dropout_rate).to(device)

optimizer_G = optim.Adam(list(generator_LitToConv.parameters()) +
list(generator_ConvToLit.parameters()),
```

```
                          lr=learning_rate_generators,
betas=(beta1_adam, beta2_adam))
optimizer_D_lit = optim.Adam(discriminator_Literary.parameters(),
lr=learning_rate_discriminators, betas=(beta1_adam, beta2_adam),
weight_decay=discriminator_weight_decay)
optimizer_D_conv =
optim.Adam(discriminator_Conversational.parameters(),
lr=learning_rate_discriminators, betas=(beta1_adam, beta2_adam),
weight_decay=discriminator_weight_decay)


# -------------!!!Ф-ции сохранения моделей и графиков!!!-------------


def plot_losses(history, epoch, output_dir=output_dir):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    epochs_range = range(1, epoch + 1)

    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    sns.lineplot(x=epochs_range, y=history['D_lit_train'],
label='D_lit_train')
    sns.lineplot(x=epochs_range, y=history['D_lit_val'],
label='D_lit_val')
    plt.title('Loss дискриминатора литературного стиля')
    plt.xlabel('Эпоха')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    sns.lineplot(x=epochs_range, y=history['D_conv_train'],
label='D_conv_train')
    sns.lineplot(x=epochs_range, y=history['D_conv_val'],
label='D_conv_val')
    plt.title('Loss дискриминатора разговорного стиля')
    plt.xlabel('Эпоха')
    plt.ylabel('Loss')
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir,
f'discriminators_losses.png'))
    plt.close()
```

```
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    sns.lineplot(x=epochs_range, y=history['G_train'],
label='G_train')
    sns.lineplot(x=epochs_range, y=history['G_val'], label='G_val')
    plt.title('Loss генераторов')
    plt.xlabel('Эпоха')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    sns.lineplot(x=epochs_range, y=history['cycle_train'],
label='Cycle_train')
    sns.lineplot(x=epochs_range, y=history['cycle_val'],
label='Cycle_val')
    plt.title('Cycle Consistency Loss')
    plt.xlabel('Эпоха')
    plt.ylabel('Loss')
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir,
f'generators_and_cycle_losses.png'))
    plt.close()

    plt.figure(figsize=(6, 5))
    sns.lineplot(x=epochs_range, y=history['identity_train'],
label='Identity_train')
    sns.lineplot(x=epochs_range, y=history['identity_val'],
label='Identity_val')
    plt.title('Identity Loss')
    plt.xlabel('Эпоха')
    plt.ylabel('Loss')
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, f'identity_losses.png'))
    plt.close()


def save_models(epoch, generator_lit_to_conv, generator_conv_to_lit,
discriminator_literary, discriminator_conversational, optimizer_g,
optimizer_d_lit, optimizer_d_conv, output_dir=model_output_dir):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
```

```
    torch.save(generator_lit_to_conv.state_dict(),
os.path.join(output_dir, f'generator_lit_to_conv.pth'))
    torch.save(generator_conv_to_lit.state_dict(),
os.path.join(output_dir, f'generator_conv_to_lit.pth'))
    torch.save(discriminator_literary.state_dict(),
os.path.join(output_dir, f'discriminator_literary.pth'))
    torch.save(discriminator_conversational.state_dict(),
os.path.join(output_dir, f'discriminator_conversational.pth'))
    torch.save(optimizer_g.state_dict(), os.path.join(output_dir,
f'optimizer_g.pth'))
    torch.save(optimizer_d_lit.state_dict(), os.path.join(output_dir,
f'optimizer_d_lit.pth'))
    torch.save(optimizer_d_conv.state_dict(), os.path.join(output_dir,
f'optimizer_d_conv.pth'))
    print(f”Модели перезаписаны в {output_dir} на {epoch} эпохе”)


# -------------!!!Общая подготовка к обучению!!!-------------


class SentenceDataset(torch.utils.data.Dataset):
    def __init__(self, literary_vectors, conversational_vectors):
        self.literary_data = torch.tensor(literary_vectors,
dtype=torch.float32)
        self.conversational_data =
torch.tensor(conversational_vectors, dtype=torch.float32)
        self.dataset_len = len(self.literary_data)

    def __len__(self):
        return self.dataset_len
    def __getitem__(self, idx):
        return self.literary_data[idx], self.conversational_data[idx]


lit_train, lit_val, conv_train, conv_val = train_test_split(
    lit_vectors_np, conv_vectors_np, test_size=val_split_ratio,
random_state=random_seed
)


train_dataset = SentenceDataset(lit_train, conv_train)
val_dataset = SentenceDataset(lit_val, conv_val)
```

```
train_dataloader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, shuffle=True,
num_workers=num_dataloader_workers)
val_dataloader = torch.utils.data.DataLoader(val_dataset,
batch_size=batch_size, shuffle=False,
num_workers=num_dataloader_workers)


history = {'D_lit_train': [], 'D_conv_train': [], 'G_train': [],
'cycle_train': [], 'identity_train': [],
           'D_lit_val': [], 'D_conv_val': [], 'G_val': [],
'cycle_val': [], 'identity_val': []}


# -----------!!!Подготовка и обучение для 1 и 2 версий!!!-----------


criterion_GAN = nn.BCELoss()
criterion_cycle = nn.L1Loss()
criterion_identity = nn.L1Loss()


for epoch in range(num_epochs):
    generator_LitToConv.train()
    generator_ConvToLit.train()
    discriminator_Literary.train()
    discriminator_Conversational.train()

    train_loss_D_lit_accum = 0.0
    train_loss_D_conv_accum = 0.0
    train_loss_G_accum = 0.0
    train_loss_cycle_accum = 0.0
    train_loss_identity_accum = 0.0
    num_train_batches = 0

    for i, (real_literary, real_conversational) in
enumerate(train_dataloader):
        real_literary = real_literary.to(device)
        real_conversational = real_conversational.to(device)

        valid = torch.ones(real_literary.size(0), 1).to(device)
        fake = torch.zeros(real_literary.size(0), 1).to(device)

        optimizer_D_lit.zero_grad()
```

```
        optimizer_D_conv.zero_grad()

        loss_D_lit_real =
criterion_GAN(discriminator_Literary(real_literary), valid)
        fake_literary = generator_ConvToLit(real_conversational)
        loss_D_lit_fake =
criterion_GAN(discriminator_Literary(fake_literary.detach()), fake)
        loss_D_lit = (loss_D_lit_real + loss_D_lit_fake) / 2
        loss_D_lit.backward()
        optimizer_D_lit.step()

        loss_D_conv_real =
criterion_GAN(discriminator_Conversational(real_conversational),
valid)
        fake_conversational = generator_LitToConv(real_literary)
        loss_D_conv_fake =
criterion_GAN(discriminator_Conversational(fake_conversational.detach(
)), fake)
        loss_D_conv = (loss_D_conv_real + loss_D_conv_fake) / 2
        loss_D_conv.backward()
        optimizer_D_conv.step()

        optimizer_G.zero_grad()

        fake_conversational = generator_LitToConv(real_literary)
        loss_GAN_LitToConv =
criterion_GAN(discriminator_Conversational(fake_conversational),
valid)

        fake_literary = generator_ConvToLit(real_conversational)
        loss_GAN_ConvToLit =
criterion_GAN(discriminator_Literary(fake_literary), valid)

        recovered_literary = generator_ConvToLit(fake_conversational)
        loss_cycle_lit = criterion_cycle(recovered_literary,
real_literary)

        recovered_conversational = generator_LitToConv(fake_literary)
        loss_cycle_conv = criterion_cycle(recovered_conversational,
real_conversational)

        loss_cycle = (loss_cycle_lit + loss_cycle_conv) / 2

        identity_conversational =
generator_LitToConv(real_conversational)
```

```
        loss_identity_conv =
criterion_identity(identity_conversational, real_conversational)

        identity_literary = generator_ConvToLit(real_literary)
        loss_identity_lit = criterion_identity(identity_literary,
real_literary)

        loss_identity = (loss_identity_conv + loss_identity_lit) / 2

        loss_G =  (loss_GAN_LitToConv + loss_GAN_ConvToLit) + \
                  lambda_cycle * loss_cycle + \
                  lambda_identity * loss_identity
        loss_G.backward()
        optimizer_G.step()

        train_loss_D_lit_accum += loss_D_lit.item()
        train_loss_D_conv_accum += loss_D_conv.item()
        train_loss_G_accum += loss_G.item()
        train_loss_cycle_accum += loss_cycle.item()
        train_loss_identity_accum += loss_identity.item()
        num_train_batches += 1

        if i % print_batch_interval == 0:
            print(f”Эпоха [{epoch + 1}/{num_epochs}] Батч
[{i}/{len(train_dataloader)}] ”
                  f”D_lit_loss: {loss_D_lit.item():.4f} D_conv_loss:
{loss_D_conv.item():.4f} ”
                  f”G_loss: {loss_G.item():.4f}”)

    generator_LitToConv.eval()
    generator_ConvToLit.eval()
    discriminator_Literary.eval()
    discriminator_Conversational.eval()

    val_loss_D_lit_accum = 0.0
    val_loss_D_conv_accum = 0.0
    val_loss_G_accum = 0.0
    val_loss_cycle_accum = 0.0
    val_loss_identity_accum = 0.0
    num_val_batches = 0

    with torch.no_grad():
```

```
        for i_val, (val_literary, val_conversational) in
enumerate(val_dataloader):
            val_literary = val_literary.to(device)
            val_conversational = val_conversational.to(device)
            val_valid = torch.ones(val_literary.size(0), 1).to(device)
            val_fake = torch.zeros(val_literary.size(0), 1).to(device)

            val_loss_D_lit_real =
criterion_GAN(discriminator_Literary(val_literary), val_valid)
            val_fake_literary =
generator_ConvToLit(val_conversational)
            val_loss_D_lit_fake =
criterion_GAN(discriminator_Literary(val_fake_literary), val_fake)
            val_loss_D_lit_val = (val_loss_D_lit_real +
val_loss_D_lit_fake) / 2

            val_loss_D_conv_real =
criterion_GAN(discriminator_Conversational(val_conversational),
val_valid)
            val_fake_conversational =
generator_LitToConv(val_literary)
            val_loss_D_conv_fake =
criterion_GAN(discriminator_Conversational(val_fake_conversational),
val_fake)
            val_loss_D_conv_val = (val_loss_D_conv_real +
val_loss_D_conv_fake) / 2

            val_loss_GAN_LitToConv =
criterion_GAN(discriminator_Conversational(val_fake_conversational),
val_valid)
            val_loss_GAN_ConvToLit =
criterion_GAN(discriminator_Literary(val_fake_literary), val_valid)

            val_recovered_literary =
generator_ConvToLit(val_fake_conversational)
            val_loss_cycle_lit =
criterion_cycle(val_recovered_literary, val_literary)
            val_recovered_conversational =
generator_LitToConv(val_fake_literary)
            val_loss_cycle_conv =
criterion_cycle(val_recovered_conversational, val_conversational)
            val_loss_cycle_val = (val_loss_cycle_lit +
val_loss_cycle_conv) / 2

            val_identity_conversational =
generator_LitToConv(val_conversational)
```

```
            val_loss_identity_conv =
criterion_identity(val_identity_conversational, val_conversational)
            val_identity_literary = generator_ConvToLit(val_literary)
            val_loss_identity_lit =
criterion_identity(val_identity_literary, val_literary)
            val_loss_identity_val = (val_loss_identity_conv +
val_loss_identity_lit) / 2

            val_loss_G_val = (val_loss_GAN_LitToConv +
val_loss_GAN_ConvToLit) + \
                             lambda_cycle * val_loss_cycle_val + \
                             lambda_identity * val_loss_identity_val

            val_loss_D_lit_accum += val_loss_D_lit_val.item()
            val_loss_D_conv_accum += val_loss_D_conv_val.item()
            val_loss_G_accum += val_loss_G_val.item()
            val_loss_cycle_accum += val_loss_cycle_val.item()
            val_loss_identity_accum += val_loss_identity_val.item()
            num_val_batches += 1

    avg_val_loss_D_lit = val_loss_D_lit_accum / num_val_batches
    avg_val_loss_D_conv = val_loss_D_conv_accum / num_val_batches
    avg_val_loss_G = val_loss_G_accum / num_val_batches
    avg_val_loss_cycle = val_loss_cycle_accum / num_val_batches
    avg_val_loss_identity = val_loss_identity_accum / num_val_batches

    generator_LitToConv.train()
    generator_ConvToLit.train()
    discriminator_Literary.train()
    discriminator_Conversational.train()

    history['D_lit_train'].append(train_loss_D_lit_accum /
num_train_batches)
    history['D_conv_train'].append(train_loss_D_conv_accum /
num_train_batches)
    history['G_train'].append(train_loss_G_accum / num_train_batches)
    history['cycle_train'].append(train_loss_cycle_accum /
num_train_batches)
    history['identity_train'].append(train_loss_identity_accum /
num_train_batches)

    history['D_lit_val'].append(avg_val_loss_D_lit)
    history['D_conv_val'].append(avg_val_loss_D_conv)
    history['G_val'].append(avg_val_loss_G)
```

```
    history['cycle_val'].append(avg_val_loss_cycle)
    history['identity_val'].append(avg_val_loss_identity)

    print(f”Эпоха [{epoch + 1}/{num_epochs}] Обучение--- ”
          f”D_lit_loss: {history['D_lit_train'][-1]:.4f} D_conv_loss:
{history['D_conv_train'][-1]:.4f} ”
          f”G_loss: {history['G_train'][-1]:.4f} cycle_loss:
{history['cycle_train'][-1]:.4f} identity_loss:
{history['identity_train'][-1]:.4f}”)
    print(f”Эпоха [{epoch + 1}/{num_epochs}] Валидация --- ”
          f”Val_D_lit_loss: {history['D_lit_val'][-1]:.4f}
Val_D_conv_loss: {history['D_conv_val'][-1]:.4f} ”
          f”Val_G_loss: {history['G_val'][-1]:.4f} Val_cycle_loss:
{history['cycle_val'][-1]:.4f} Val_identity_loss:
{history['identity_val'][-1]:.4f}”)

    save_models(epoch + 1, generator_LitToConv, generator_ConvToLit,
discriminator_Literary, discriminator_Conversational, optimizer_G,
optimizer_D_lit, optimizer_D_conv)
    plot_losses(history, epoch + 1)

print(”Обучение завершено!”)

plot_losses(history, num_epochs)


# ----------!!!Подготовка и обучение для 3 и 4 версий!!!----------


def cosine_distance_loss(output, target):
    cosine_similarity = F.cosine_similarity(output, target)
    cosine_distance = (1 - cosine_similarity) / 2
    loss = torch.mean(cosine_distance)
    return loss


criterion_GAN = nn.BCELoss()
criterion_cycle_l1 = nn.L1Loss()
criterion_cycle_cosine = cosine_distance_loss
criterion_identity_l1 = nn.L1Loss()
criterion_identity_cosine = cosine_distance_loss

for epoch in range(num_epochs):
    generator_LitToConv.train()
```

```
    generator_ConvToLit.train()
    discriminator_Literary.train()
    discriminator_Conversational.train()

    train_loss_D_lit_accum = 0.0
    train_loss_D_conv_accum = 0.0
    train_loss_G_accum = 0.0
    train_loss_cycle_accum = 0.0
    train_loss_identity_accum = 0.0
    num_train_batches = 0

    for i, (real_literary, real_conversational) in
enumerate(train_dataloader):
        real_literary = real_literary.to(device)
        real_conversational = real_conversational.to(device)

        valid = torch.ones(real_literary.size(0), 1).to(device)
        fake = torch.zeros(real_literary.size(0), 1).to(device)

        optimizer_D_lit.zero_grad()
        optimizer_D_conv.zero_grad()

        loss_D_lit_real =
criterion_GAN(discriminator_Literary(real_literary), valid)
        fake_literary = generator_ConvToLit(real_conversational)
        loss_D_lit_fake =
criterion_GAN(discriminator_Literary(fake_literary.detach()), fake)
        loss_D_lit = (loss_D_lit_real + loss_D_lit_fake) / 2
        loss_D_lit.backward()
        optimizer_D_lit.step()

        loss_D_conv_real =
criterion_GAN(discriminator_Conversational(real_conversational),
valid)
        fake_conversational = generator_LitToConv(real_literary)
        loss_D_conv_fake =
criterion_GAN(discriminator_Conversational(fake_conversational.detach(
)), fake)
        loss_D_conv = (loss_D_conv_real + loss_D_conv_fake) / 2
        loss_D_conv.backward()
        optimizer_D_conv.step()

        optimizer_G.zero_grad()
```

```
        fake_conversational = generator_LitToConv(real_literary)
        loss_GAN_LitToConv =
criterion_GAN(discriminator_Conversational(fake_conversational),
valid)

        fake_literary = generator_ConvToLit(real_conversational)
        loss_GAN_ConvToLit =
criterion_GAN(discriminator_Literary(fake_literary), valid)

        recovered_literary = generator_ConvToLit(fake_conversational)
        loss_cycle_lit_l1 = criterion_cycle_l1(recovered_literary,
real_literary)
        loss_cycle_lit_cosine =
criterion_cycle_cosine(recovered_literary, real_literary)
        loss_cycle_lit = lambda_cycle_l1 * loss_cycle_lit_l1 +
lambda_cycle_cosine * loss_cycle_lit_cosine

        recovered_conversational = generator_LitToConv(fake_literary)
        loss_cycle_conv_l1 =
criterion_cycle_l1(recovered_conversational, real_conversational)
        loss_cycle_conv_cosine =
criterion_cycle_cosine(recovered_conversational, real_conversational)
        loss_cycle_conv = lambda_cycle_l1 * loss_cycle_conv_l1 +
lambda_cycle_cosine * loss_cycle_conv_cosine

        loss_cycle = (loss_cycle_lit + loss_cycle_conv) / 2

        identity_conversational =
generator_LitToConv(real_conversational)
        loss_identity_conv_l1 =
criterion_identity_l1(identity_conversational, real_conversational)
        loss_identity_conv_cosine =
criterion_identity_cosine(identity_conversational,
real_conversational)
        loss_identity_conv = lambda_identity_l1 *
loss_identity_conv_l1 + lambda_identity_cosine *
loss_identity_conv_cosine

        identity_literary = generator_ConvToLit(real_literary)
        loss_identity_lit_l1 =
criterion_identity_l1(identity_literary, real_literary)
        loss_identity_lit_cosine =
criterion_identity_cosine(identity_literary, real_literary)
        loss_identity_lit = lambda_identity_l1 * loss_identity_lit_l1
+ lambda_identity_cosine * loss_identity_lit_cosine
```

```
        loss_identity = (loss_identity_conv + loss_identity_lit) / 2

        loss_G =  (loss_GAN_LitToConv + loss_GAN_ConvToLit) + \
                  lambda_cycle * loss_cycle + \
                  lambda_identity * loss_identity
        loss_G.backward()
        optimizer_G.step()

        train_loss_D_lit_accum += loss_D_lit.item()
        train_loss_D_conv_accum += loss_D_conv.item()
        train_loss_G_accum += loss_G.item()
        train_loss_cycle_accum += loss_cycle.item()
        train_loss_identity_accum += loss_identity.item()
        num_train_batches += 1

        if i % print_batch_interval == 0:
            print(f"Эпоха [{epoch + 1}/{num_epochs}] Батч
[{i}/{len(train_dataloader)}] "
                  f"D_lit_loss: {loss_D_lit.item():.4f} D_conv_loss:
{loss_D_conv.item():.4f} "
                  f"G_loss: {loss_G.item():.4f}")

    generator_LitToConv.eval()
    generator_ConvToLit.eval()
    discriminator_Literary.eval()
    discriminator_Conversational.eval()

    val_loss_D_lit_accum = 0.0
    val_loss_D_conv_accum = 0.0
    val_loss_G_accum = 0.0
    val_loss_cycle_accum = 0.0
    val_loss_identity_accum = 0.0
    num_val_batches = 0

    with torch.no_grad():
        for i_val, (val_literary, val_conversational) in
enumerate(val_dataloader):
            val_literary = val_literary.to(device)
            val_conversational = val_conversational.to(device)
            val_valid = torch.ones(val_literary.size(0), 1).to(device)
            val_fake = torch.zeros(val_literary.size(0), 1).to(device)
```

```
            val_loss_D_lit_real =
criterion_GAN(discriminator_Literary(val_literary), val_valid)
            val_fake_literary =
generator_ConvToLit(val_conversational)
            val_loss_D_lit_fake =
criterion_GAN(discriminator_Literary(val_fake_literary), val_fake)
            val_loss_D_lit_val = (val_loss_D_lit_real +
val_loss_D_lit_fake) / 2

            val_loss_D_conv_real =
criterion_GAN(discriminator_Conversational(val_conversational),
val_valid)
            val_fake_conversational =
generator_LitToConv(val_literary)
            val_loss_D_conv_fake =
criterion_GAN(discriminator_Conversational(val_fake_conversational),
val_fake)
            val_loss_D_conv_val = (val_loss_D_conv_real +
val_loss_D_conv_fake) / 2

            val_loss_GAN_LitToConv =
criterion_GAN(discriminator_Conversational(val_fake_conversational),
val_valid)
            val_loss_GAN_ConvToLit =
criterion_GAN(discriminator_Literary(val_fake_literary), val_valid)

            val_fake_conversational =
generator_LitToConv(val_literary)
            val_fake_literary =
generator_ConvToLit(val_conversational)

            val_recovered_literary =
generator_ConvToLit(val_fake_conversational)
            val_recovered_conversational =
generator_LitToConv(val_fake_literary)

            val_loss_cycle_lit_l1 =
criterion_cycle_l1(val_recovered_literary, val_literary)
            val_loss_cycle_lit_cosine =
criterion_cycle_cosine(val_recovered_literary, val_literary)
            val_loss_cycle_lit_val = lambda_cycle_l1 *
val_loss_cycle_lit_l1 + lambda_cycle_cosine *
val_loss_cycle_lit_cosine

            val_loss_cycle_conv_l1 =
criterion_cycle_l1(val_recovered_conversational, val_conversational)
```

```
            val_loss_cycle_conv_cosine =
criterion_cycle_cosine(val_recovered_conversational,
val_conversational)
            val_loss_cycle_conv_val = lambda_cycle_l1 *
val_loss_cycle_conv_l1 + lambda_cycle_cosine *
val_loss_cycle_conv_cosine

            val_loss_cycle_val = (val_loss_cycle_lit_val +
val_loss_cycle_conv_val) / 2

            val_identity_conversational =
generator_LitToConv(val_conversational)
            val_identity_literary = generator_ConvToLit(val_literary)

            val_loss_identity_conv_l1 =
criterion_identity_l1(val_identity_conversational, val_conversational)
            val_loss_identity_conv_cosine =
criterion_identity_cosine(val_identity_conversational,
val_conversational)
            val_loss_identity_conv_val = lambda_identity_l1 *
val_loss_identity_conv_l1 + lambda_identity_cosine *
val_loss_identity_conv_cosine

            val_loss_identity_lit_l1 =
criterion_identity_l1(val_identity_literary, val_literary)
            val_loss_identity_lit_cosine =
criterion_identity_cosine(val_identity_literary, val_literary)
            val_loss_identity_lit_val = lambda_identity_l1 *
val_loss_identity_lit_l1 + lambda_identity_cosine *
val_loss_identity_lit_cosine

            val_loss_identity_val = (val_loss_identity_conv_val +
val_loss_identity_lit_val) / 2

            val_loss_G_val = (val_loss_GAN_LitToConv +
val_loss_GAN_ConvToLit) + \
                             lambda_cycle * val_loss_cycle_val + \
                             lambda_identity * val_loss_identity_val

            val_loss_D_lit_accum += val_loss_D_lit_val.item()
            val_loss_D_conv_accum += val_loss_D_conv_val.item()
            val_loss_G_accum += val_loss_G_val.item()
            val_loss_cycle_accum += val_loss_cycle_val.item()
            val_loss_identity_accum += val_loss_identity_val.item()
            num_val_batches += 1
```

```
    avg_val_loss_D_lit = val_loss_D_lit_accum / num_val_batches
    avg_val_loss_D_conv = val_loss_D_conv_accum / num_val_batches
    avg_val_loss_G = val_loss_G_accum / num_val_batches
    avg_val_loss_cycle = val_loss_cycle_accum / num_val_batches
    avg_val_loss_identity = val_loss_identity_accum / num_val_batches

    generator_LitToConv.train()
    generator_ConvToLit.train()
    discriminator_Literary.train()
    discriminator_Conversational.train()

    history['D_lit_train'].append(train_loss_D_lit_accum /
num_train_batches)
    history['D_conv_train'].append(train_loss_D_conv_accum /
num_train_batches)
    history['G_train'].append(train_loss_G_accum / num_train_batches)
    history['cycle_train'].append(train_loss_cycle_accum /
num_train_batches)
    history['identity_train'].append(train_loss_identity_accum /
num_train_batches)

    history['D_lit_val'].append(avg_val_loss_D_lit)
    history['D_conv_val'].append(avg_val_loss_D_conv)
    history['G_val'].append(avg_val_loss_G)
    history['cycle_val'].append(avg_val_loss_cycle)
    history['identity_val'].append(avg_val_loss_identity)

    print(f"Эпоха [{epoch + 1}/{num_epochs}] Обучение--- "
          f"D_lit_loss: {history['D_lit_train'][-1]:.4f} D_conv_loss:
{history['D_conv_train'][-1]:.4f} "
          f"G_loss: {history['G_train'][-1]:.4f} cycle_loss:
{history['cycle_train'][-1]:.4f} identity_loss:
{history['identity_train'][-1]:.4f}")
    print(f"Эпоха [{epoch + 1}/{num_epochs}] Валидация --- "
          f"Val_D_lit_loss: {history['D_lit_val'][-1]:.4f}
Val_D_conv_loss: {history['D_conv_val'][-1]:.4f} "
          f"Val_G_loss: {history['G_val'][-1]:.4f} Val_cycle_loss:
{history['cycle_val'][-1]:.4f} Val_identity_loss:
{history['identity_val'][-1]:.4f}")

    if (epoch + 1) % plot_loss_interval_epochs == 0:
        plot_losses(history, epoch + 1)
```

```
    save_models(epoch + 1, generator_LitToConv, generator_ConvToLit,
discriminator_Literary, discriminator_Conversational, optimizer_G,
optimizer_D_lit, optimizer_D_conv)


print(”Обучение завершено!”)


# -----------!!!Ручные тесты!!!-----------


path = 'navec_hudlit_v1_12B_500K_300d_100q.tar'
navec = Navec.load(path)

embeddings = list()
for word in navec.vocab.words:
        embeddings.append(navec[word])
embeddings = np.array(embeddings)

target_dim = 10
max_vector_length = 40
pca = PCA(n_components=target_dim)
reduced_embeddings = pca.fit_transform(embeddings)

reduced_navec = {word: np.array(reduced_embeddings[i],
dtype=np.float16) for i, word in enumerate(navec.vocab.words)}

punkt_vectors = {
    ”.”: np.array([0.0] * (target_dim - 1) + [1.0], dtype=np.float16),
    ”!”: np.array([0.0] * (target_dim - 1) + [0.9], dtype=np.float16),
    ”?”: np.array([0.0] * (target_dim - 1) + [0.8], dtype=np.float16),
    ”,”: np.array([0.0] * (target_dim - 1) + [0.5], dtype=np.float16),
    ”:”: np.array([0.0] * (target_dim - 1) + [0.6], dtype=np.float16),
    ”–”: np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    ”-”: np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    ”—”: np.array([0.0] * (target_dim - 1) + [0.4], dtype=np.float16),
    ”«”: np.array([0.0] * (target_dim - 1) + [0.2], dtype=np.float16),
    ”»”: np.array([0.0] * (target_dim - 1) + [0.3], dtype=np.float16),
}

combined_vectors = {**reduced_navec, **punkt_vectors}
combined_items = list(combined_vectors.keys())
combined_embeddings = np.array(list(combined_vectors.values()))
```

```
def text_to_vec(text):
    sentence = text.strip()
    sentence = sentence.strip()
    sentence = re.sub(”[^а-яА-Я---—,.!?:«» ]”, ””, sentence)
    start_removed = 0
    while len(sentence) > 0 and sentence[0] in ”---—,.!?:«» ”:
        start_removed += 1
    sentence = sentence[start_removed:]
    tokens = word_tokenize(sentence)
    tokens = tokens[:max_vector_length]
    padding_for = max_vector_length - len(tokens)
    vector = [np.zeros(target_dim)] * padding_for
    for word in tokens:
        try:
            vector.append(reduced_navec[word])
        except KeyError:
            try:
                vector.append(punkt_vectors[word])
            except KeyError:
                vector.append(reduced_navec[”<unk>”])
    vector = np.array(vector,
dtype=np.float16).reshape(max_vector_length * target_dim)
    return torch.tensor(vector,
dtype=torch.float32).unsqueeze(0).to(device)


def vec_to_text(vector):
    vector_2d = vector.reshape(max_vector_length, target_dim)

    tokens = []
    combined_vectors = {**reduced_navec, **punkt_vectors}
    combined_items = list(combined_vectors.keys())
    combined_embeddings = np.array(list(combined_vectors.values()))

    for vec in vector_2d:
        if isinstance(vec, torch.Tensor):
            vec_np = vec.cpu().numpy()
        else:
            vec_np = vec

        if np.all(vec_np == 0):
```

```
            continue

        if isinstance(vec, torch.Tensor):
            distances = scipy.spatial.distance.cdist(vec.reshape(1,
-1).cpu().numpy(), combined_embeddings, metric='euclidean')[0]
        else:
            distances = scipy.spatial.distance.cdist(vec.reshape(1,
-1), combined_embeddings, metric='euclidean')[0]
        closest_token_index = np.argmin(distances)
        closest_token = combined_items[closest_token_index]
        tokens.append(closest_token)

    reconstructed_text_parts = []
    for token in tokens:
        if token in punkt_vectors:
            reconstructed_text_parts.append(token)
        else:
            reconstructed_text_parts.append(" " + token)

    reconstructed_text = "".join(reconstructed_text_parts).strip()
    return reconstructed_text


generator_LitToConv = Generator(vector_dimension,
vector_dimension).to(device)
generator_ConvToLit = Generator(vector_dimension,
vector_dimension).to(device)
discriminator_Literary = Discriminator(vector_dimension,
lstm_hidden_dim_discriminator, lstm_num_layers_discriminator,
leaky_relu_negative_slope, discriminator_layer_size_hidden,
discriminator_dropout_rate).to(device)
discriminator_Conversational = Discriminator(vector_dimension,
lstm_hidden_dim_discriminator, lstm_num_layers_discriminator,
leaky_relu_negative_slope, discriminator_layer_size_hidden,
discriminator_dropout_rate).to(device)


generator_LitToConv.load_state_dict(torch.load(os.path.join(output_dir
, 'generator_lit_to_conv.pth'), map_location=device))
generator_ConvToLit.load_state_dict(torch.load(os.path.join(output_dir
, 'generator_conv_to_lit.pth'), map_location=device))
generator_LitToConv.eval()
generator_ConvToLit.eval()
```

```
lit_text = ”У лукоморья дуб зелёный Златая цепь на дубе том И днём и ночью кот учёный Всё ходит по цепи кругом”
lit_vec = text_to_vec(lit_text)


with torch.no_grad():
    fake_conv = generator_LitToConv(lit_vec)
    fake_lit = generator_ConvToLit(conv_vec)
    fake_conv_to_lit = generator_ConvToLit(fake_conv)
    fake_lit_to_conv = generator_LitToConv(fake_lit)


print(lit_text)
print(vec_to_text(fake_conv))
print(vec_to_text(fake_conv_to_lit))
```

Приложение П

Векторизация предложений с использованием ruBERT

```
import torch
from transformers import BertTokenizer, BertModel
from torch.utils.data import Dataset, DataLoader
import pandas as pd
import numpy as np
from tqdm import tqdm
import time


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 2048
max_seq_len = 96
num_workers = 16


tokenizer =
BertTokenizer.from_pretrained("DeepPavlov/rubert-base-cased",
force_download=True)
model = BertModel.from_pretrained("DeepPavlov/rubert-base-cased",
force_download=True).to(device)


class TextDataset(Dataset):
    def __init__(self, texts, tokenizer, max_seq_len):
        self.texts = texts
        self.tokenizer = tokenizer
        self.max_seq_len = max_seq_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        encoding = self.tokenizer(text, max_length=self.max_seq_len,
truncation=True, padding='max_length', return_tensors='pt')
        return {k: v.squeeze(0) for k, v in encoding.items()}


data = pd.read_csv("data.csv")
lit_texts = data["lit_text"].tolist()
```

```
tg_texts = data["tg_text"].tolist()


lit_dataset = TextDataset(lit_texts, tokenizer, max_seq_len)
tg_dataset = TextDataset(tg_texts, tokenizer, max_seq_len)
lit_dataloader = DataLoader(lit_dataset, batch_size=batch_size,
shuffle=False, num_workers=num_workers, pin_memory=True)
tg_dataloader = DataLoader(tg_dataset, batch_size=batch_size,
shuffle=False, num_workers=num_workers, pin_memory=True)


def vectorize_texts(dataloader, model, output_file):
    embeddings = []
    with torch.no_grad():
        for batch in tqdm(dataloader, desc=f"Векторизация
{output_file}"):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)

            with torch.cuda.amp.autocast():
                outputs = model(input_ids,
attention_mask=attention_mask)
                embed = outputs.last_hidden_state[:, 0, :]

            embeddings.append(embed.cpu().numpy())
            torch.cuda.empty_cache()

        embeddings = np.concatenate(embeddings, axis=0)
        np.save(output_file, embeddings)
        return embeddings


start_time = time.time()
lit_embeddings = vectorize_texts(lit_dataloader, model,
"lit_embeddings.npy")
tg_embeddings = vectorize_texts(tg_dataloader, model,
"tg_embeddings.npy")
print(f"Векторизация завершена за {time.time() - start_time:.2f}
секунд")
```

Приложение Р

Токенизация предложений, построение и обучение декодера

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset, random_split
from torch.cuda.amp import GradScaler, autocast
from tqdm import tqdm
import numpy as np
import os
import matplotlib.pyplot as plt
from transformers import AutoTokenizer,
get_linear_schedule_with_warmup, BertModel
import time
import torch.profiler
import torch.nn.functional as F


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Используем устройство: {device}")

TOKENIZER_MODEL = "DeepPavlov/rubert-base-cased"
tokenizer = AutoTokenizer.from_pretrained(TOKENIZER_MODEL,
force_download=True)

EMBED_DIM = 768
VOCAB_SIZE = tokenizer.vocab_size
MAX_SEQ_LEN = 96
BATCH_SIZE = 4032
NUM_EPOCHS = 100
LEARNING_RATE = 1e-4
WARMUP_STEPS = 1000
PATIENCE = 5
MIN_DELTA = 0.0005
NUM_LAYERS = 3
NUM_HEADS = 8
FF_DIM = 1024
DROPOUT = 0.1
NUM_WORKERS = 0
VALIDATION_SPLIT = 0.2
BEAM_WIDTH = 5
TEMPERATURE = 1.0
```

```
ADAPTIVE_CUTOFFS = [510, 30269, 60028, 89787]
ADAPTIVE_DIV_VALUE = 4

train_losses = []
val_losses = []
teacher_forcing_ratios = []


def clear_gpu_memory():
    torch.cuda.empty_cache()

def preprocess_texts(texts=None, tokenizer=tokenizer,
max_seq_len=MAX_SEQ_LEN, output_file="tokenized_texts.npy"):
    if os.path.exists(output_file):
        print(f"Загружаем готовые токены из {output_file}")
        return np.load(output_file, allow_pickle=True)
    if texts is None:
        raise ValueError("Файл tokenized_texts.npy не найден, и тексты
не предоставлены!")
    print("Токенизация текстов...")
    tokenized = []
    for text in tqdm(texts, desc="Токенизация"):
        tokens = tokenizer.encode(text, max_length=max_seq_len - 1,
truncation=True, padding='max_length')
        tokens = tokens + [tokenizer.sep_token_id]
        if len(tokens) < max_seq_len:
            tokens += [tokenizer.pad_token_id] * (max_seq_len -
len(tokens))
        tokenized.append(tokens)
    tokenized = np.array(tokenized, dtype=np.int64)
    np.save(output_file, tokenized)
    print(f"Токены сохранены в {output_file}")
    return tokenized

def get_teacher_forcing_ratio(epoch):
    if epoch < 80:
        return 1.0 - (epoch / 80)
    else:
        return 0.0


class TransformerDecoder(nn.Module):
    def __init__(self, embed_dim=768, vocab_size=119547, num_layers=3,
```

```
                 num_heads=8, ff_dim=1024, dropout=0.1,
max_seq_len=96, freq_file="token_frequencies.npy", beta=0.5):
        super().__init__()
        self.embed_dim = embed_dim
        self.vocab_size = vocab_size
        self.max_seq_len = max_seq_len

        self.input_norm = nn.LayerNorm(embed_dim)
        self.input_proj = nn.Linear(embed_dim, embed_dim)

        self.token_embedding = nn.Embedding(vocab_size, embed_dim)
        self.pos_embedding = nn.Embedding(max_seq_len, embed_dim)

        self.register_buffer("position_ids",
torch.arange(max_seq_len).unsqueeze(0), persistent=False)

        decoder_layer = nn.TransformerDecoderLayer(
            d_model=embed_dim, nhead=num_heads,
dim_feedforward=ff_dim,
            dropout=dropout, batch_first=True, activation='gelu'
        )
        self.decoder = nn.TransformerDecoder(decoder_layer,
num_layers=num_layers)
        self.dropout = nn.Dropout(dropout)

        self.output_layer = nn.AdaptiveLogSoftmaxWithLoss(
            in_features=embed_dim,
            n_classes=vocab_size,
            cutoffs=[510, 30269, 60028, 89787],
            div_value=4,
            head_bias=True
        )

        self.mask_cache = {}

        if os.path.exists(freq_file):
            freqs = np.load(freq_file)
            freq_tensor = torch.from_numpy(freqs).float()
            bias = beta * torch.log(freq_tensor + 1.0)
            self.register_buffer('freq_bias', bias)
        else:
            print(f"Файл {freq_file} не найден, freq_bias не будет
использоваться.")
```

```
    def _get_tgt_mask(self, seq_len, device):
        if seq_len not in self.mask_cache:
            mask = torch.triu(torch.ones(seq_len, seq_len,
device=device), diagonal=1)
            mask = mask.masked_fill(mask == 1, float('-inf'))
            self.mask_cache[seq_len] = mask
        return self.mask_cache[seq_len]

    def forward(self, src_embed, tgt, teacher_forcing_ratio=1.0):
        batch_size, seq_len = tgt.size()
        device = tgt.device

        src_embed = self.input_norm(src_embed)
        memory = self.input_proj(src_embed).unsqueeze(1)

        positions = self.position_ids[:, :seq_len].expand(batch_size,
seq_len)
        tgt_mask = self._get_tgt_mask(seq_len, device)

        if teacher_forcing_ratio >= 1.0:
            input_ids = tgt
        else:
            with torch.no_grad():
                decoder_input_ids = torch.zeros_like(tgt)
                decoder_input_ids[:, 0] = tokenizer.cls_token_id

                embedded = self.token_embedding(decoder_input_ids) +
self.pos_embedding(positions)
                logits = self.decoder(self.dropout(embedded), memory,
tgt_mask=tgt_mask)

                predicted_ids =
self.output_layer.predict(logits.reshape(-1,
self.embed_dim)).reshape(batch_size, seq_len)

            use_teacher = torch.rand(batch_size, seq_len - 1,
device=device) < teacher_forcing_ratio
            input_ids = tgt.clone()
            input_ids[:, 1:] = torch.where(use_teacher, tgt[:, 1:],
predicted_ids[:, 1:])

        decoder_input = self.token_embedding(input_ids) +
self.pos_embedding(positions)
```

```
        decoder_input = self.dropout(decoder_input)

        output = self.decoder(decoder_input, memory,
tgt_mask=tgt_mask)
        output = self.dropout(output)

        return output

    def generate(self, src_embed, max_len=96, start_token_id=None,
beam_width=5, temperature=1.0, alpha=0.7, top_k=50, top_p=0.9,
min_length=10):
        batch_size = src_embed.size(0)
        device = src_embed.device
        start_token_id = start_token_id or tokenizer.cls_token_id
        sep_token_id = tokenizer.sep_token_id

        src_embed = self.input_norm(src_embed).unsqueeze(1)
        memory = self.input_proj(src_embed)

        beams = [(torch.full((1, 1), start_token_id, device=device),
0.0)]
        completed_beams = []

        for step in range(max_len - 1):
            all_candidates = []
            for seq, score in beams:
                if seq[0, -1] == sep_token_id and step >= min_length:
                    completed_beams.append((seq, score))
                    continue

                seq_len = seq.size(1)
                positions = self.position_ids[:, :seq_len].expand(1,
seq_len)
                tgt_embed = self.token_embedding(seq) +
self.pos_embedding(positions)
                mask = self._get_tgt_mask(seq_len, device)

                output = self.decoder(tgt_embed, memory,
tgt_mask=mask)
                hidden = output[:, -1, :]

                log_probs = self.output_layer.log_prob(hidden)
                log_probs = log_probs / temperature
                probs = torch.exp(log_probs)
```

```
                if hasattr(self, 'freq_bias'):
                    log_probs = log_probs - self.freq_bias
                    probs = torch.exp(log_probs)

                sorted_probs, sorted_indices = torch.sort(probs,
descending=True, dim=-1)
                cumsum_probs = torch.cumsum(sorted_probs, dim=-1)
                selected_mask = cumsum_probs <= top_p
                selected_mask[:, 0] = True
                num_selected = selected_mask.sum(dim=-1).max()
                top_p_probs = sorted_probs[:, :num_selected]
                top_p_indices = sorted_indices[:, :num_selected]

                top_k = min(top_k, num_selected.item())
                top_k_probs, top_k_indices = torch.topk(top_p_probs,
top_k, dim=-1)
                top_k_indices = top_p_indices.gather(-1,
top_k_indices)

                top_beam_probs, top_beam_indices =
torch.topk(top_k_probs, beam_width, dim=-1)
                top_beam_indices = top_k_indices.gather(-1,
top_beam_indices)

                for i in range(beam_width):
                    next_token = top_beam_indices[:, i].unsqueeze(0)
                    log_prob = torch.log(top_beam_probs[:, i] + 1e-10)
                    new_seq = torch.cat([seq, next_token], dim=1)
                    new_score = score + log_prob.item()
                    all_candidates.append((new_seq, new_score))

            beams = sorted(all_candidates, key=lambda x: x[1] / ((5 +
len(x[0][0])) ** alpha / (6 ** alpha)), reverse=True)[:beam_width]
            if not beams:
                break

        beams.extend(completed_beams)
        if not beams:
            return torch.full((batch_size, 1), start_token_id,
device=device)

        best_seq = max(beams, key=lambda x: x[1])[0]
        return best_seq.expand(batch_size, -1)
```

```
tokenized_texts = preprocess_texts()


counts = {}

for text in tokenized_texts:
    for token_id in text:
        if token_id in counts:
            counts[token_id] += 1
        else:
            counts[token_id] = 1

frequencies = np.zeros(VOCAB_SIZE, dtype=np.int32)

for token_id, freq in counts.items():
    frequencies[token_id] = freq

np.save("token_frequencies.npy", frequencies)


freqs = np.load("token_frequencies.npy")

lit_embeddings = np.load("lit_embeddings.npy")
tg_embeddings = np.load("conv_embeddings.npy")
embeddings = np.concatenate([lit_embeddings, tg_embeddings], axis=0)
tokenized_texts = preprocess_texts()


TOTAL_STEPS = (len(embeddings) // BATCH_SIZE) * NUM_EPOCHS


class EmbeddingToTokenDataset(Dataset):
    def __init__(self, embeddings, tokenized_texts):
        self.embeddings = torch.tensor(embeddings,
dtype=torch.float32)
        self.tokenized_texts = torch.tensor(tokenized_texts,
dtype=torch.long)

    def __len__(self):
        return len(self.embeddings)
```

```
    def __getitem__(self, idx):
        return self.embeddings[idx], self.tokenized_texts[idx]


dataset = EmbeddingToTokenDataset(embeddings, tokenized_texts)
dataset_size = len(dataset)
val_size = int(VALIDATION_SPLIT * dataset_size)
train_size = dataset_size - val_size
train_dataset, val_dataset = random_split(dataset, [train_size,
val_size])


train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
shuffle=True, pin_memory=True, num_workers=NUM_WORKERS)
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE,
shuffle=False, pin_memory=True, num_workers=NUM_WORKERS)


model = TransformerDecoder().to(device)
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=WARMUP_STEPS, num_training_steps=TOTAL_STEPS)
criterion = nn.CrossEntropyLoss(ignore_index=tokenizer.pad_token_id)
scaler = GradScaler()


best_val_loss = float('inf')
patience_counter = 0


train_times = []
val_times = []

for epoch in range(NUM_EPOCHS):
    epoch_start_time = time.time()

    teacher_forcing_ratio = get_teacher_forcing_ratio(epoch)
    teacher_forcing_ratios.append(teacher_forcing_ratio)
    print(f"Эпоха {epoch+1}, Teacher Forcing Ratio:
{teacher_forcing_ratio:.3f}")

    model.train()
    train_loss_total = 0
```

```
    train_start_time = time.time()
    for i, (src_embed, tgt) in enumerate(tqdm(train_dataloader,
desc=f"Тренировка, Эпоха {epoch+1}/{NUM_EPOCHS}")):
        src_embed = src_embed.to(device, non_blocking=True)
        tgt = tgt.to(device, non_blocking=True)

        optimizer.zero_grad(set_to_none=True)

        with autocast(dtype=torch.float16):
            output = model(src_embed, tgt[:, :-1],
teacher_forcing_ratio)
            tgt_shifted = tgt[:, 1:].reshape(-1)
            loss_output = model.output_layer(output.reshape(-1,
EMBED_DIM), tgt_shifted)
            loss = loss_output.loss

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scheduler.step()
        scaler.update()

        train_loss_total += loss.item()

        if i == 0:
            print(f"Размер src_embed: {src_embed.shape}")
            print(f"Размер tgt: {tgt.shape}")
            print(f"Память GPU после первого батча:
{torch.cuda.memory_allocated()/1024**3:.2f} ГБ")

    train_end_time = time.time()
    train_times.append(train_end_time - train_start_time)
    avg_train_loss = train_loss_total / len(train_dataloader)
    train_losses.append(avg_train_loss)
    print(f"Эпоха {epoch+1}/{NUM_EPOCHS}, Тренировочный лосс:
{avg_train_loss:.4f}, Время тренировки: {train_times[-1]:.2f} секунд")

    model.eval()
    val_loss_total = 0
    val_start_time = time.time()
    with torch.no_grad():
        for src_embed, tgt in tqdm(val_dataloader, desc=f"Валидация,
Эпоха {epoch+1}/{NUM_EPOCHS}"):
            src_embed = src_embed.to(device, non_blocking=True)
```

```
            tgt = tgt.to(device, non_blocking=True)
            with autocast(dtype=torch.float16):
                output = model(src_embed, tgt[:, :-1],
teacher_forcing_ratio=0.0)
                tgt_shifted = tgt[:, 1:].reshape(-1)
                loss_output = model.output_layer(output.reshape(-1,
EMBED_DIM), tgt_shifted)
                loss = loss_output.loss

            val_loss_total += loss.item()

    val_end_time = time.time()
    val_times.append(val_end_time - val_start_time)
    avg_val_loss = val_loss_total / len(val_dataloader)
    val_losses.append(avg_val_loss)
    print(f"Эпоха {epoch+1}/{NUM_EPOCHS}, Валидационный лосс:
{avg_val_loss:.4f}, Время валидации: {val_times[-1]:.2f} секунд")

    fig, ax1 = plt.subplots(figsize=(10, 5))
    ax1.plot(train_losses, label='Train Loss', color='blue')
    ax1.plot(val_losses, label='Validation Loss', color='orange')
    ax1.set_xlabel('Эпоха')
    ax1.set_ylabel('Loss')
    ax1.legend(loc='upper left')

    ax2 = ax1.twinx()
    ax2.plot(teacher_forcing_ratios, label='Teacher Forcing Ratio',
color='green', linestyle='--')
    ax2.set_ylabel('Teacher Forcing Ratio')
    ax2.legend(loc='upper right')

    plt.title('График обучения и Teacher Forcing Ratio')
    plt.savefig("loss.png")
    plt.close()

    if avg_val_loss < best_val_loss - MIN_DELTA:
        best_val_loss = avg_val_loss
        patience_counter = 0
        torch.save(model, "decoder_model.pth")
    else:
        patience_counter += 1
        if patience_counter >= PATIENCE:
            print("Ранний останов")
```

```
            break

    epoch_end_time = time.time()
    print(f"Общее время эпохи {epoch+1}: {epoch_end_time - epoch_start_time:.2f} секунд")
    clear_gpu_memory()

print("Финальная модель сохранена")
clear_gpu_memory()

bert_model = BertModel.from_pretrained("DeepPavlov/rubert-base-cased", force_download=True).to(device)
bert_model.eval()



device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = TransformerDecoder(beta=2.0).to(device)
model.load_state_dict(torch.load("decoder_model.pth", map_location=device).state_dict(), strict=False)
model.eval()

freqs = np.load("token_frequencies.npy")

text = "какие сладкие булочки!"
tokens = tokenizer.encode(text, max_length=MAX_SEQ_LEN - 1, truncation=True, padding='max_length')
tokens = tokens + [tokenizer.sep_token_id]
input_ids = torch.tensor([tokens]).to(device)
attention_mask = (input_ids != tokenizer.pad_token_id).long().to(device)

with torch.no_grad():
    outputs = bert_model(input_ids=input_ids, attention_mask=attention_mask)
    cls_embedding = outputs.last_hidden_state[:, 0, :]
    generated = model.generate(cls_embedding, max_len=MAX_SEQ_LEN, beam_width=20, temperature=0.7, top_k=50, top_p=0.7, alpha=0.8, min_length=10)
    res = tokenizer.decode(generated[0].tolist(), skip_special_tokens=True)
    print("Generated token IDs:", generated[0].tolist())
    print("Corresponding tokens:", tokenizer.convert_ids_to_tokens(generated[0].tolist()))
    print("Decoded output:", res)
```

Приложение С

Построение графика частот токенов

```
import numpy as np
import matplotlib.pyplot as plt


token_freqs = np.load('token_frequencies.npy')


vocab_size = token_freqs.shape[0]
print(f'Размер словаря: {vocab_size}')
print('Некоторые статистики по частотам:')
print(f'Максимальная частота: {token_freqs.max()}')
print(f'Минимальная частота: {token_freqs.min()}')


sorted_freqs = np.sort(token_freqs)[::-1]
ranks = np.arange(1, vocab_size + 1)


total_freq = sorted_freqs.sum()
print(f'Общее количество вхождений токенов: {total_freq}')


cumulative = np.cumsum(sorted_freqs) / total_freq


head_threshold = 0.94
head_idx = np.searchsorted(cumulative, head_threshold)
print(f'Хэдовая группа: первые {head_idx} токенов покрывают
{head_threshold*100:.0f}% частот')


num_clusters = 3
tail_size = vocab_size - head_idx
tail_cutoffs = [int(head_idx + tail_size * (i+1) / (num_clusters+1))
for i in range(num_clusters)]
print("Предлагаемые разделения (cutoffs) для adaptive softmax:")
print("Head: 0 -", head_idx)
for i, cutoff in enumerate(tail_cutoffs):
    print(f"Tail cluster {i+1}: {head_idx if i==0 else
tail_cutoffs[i-1]} - {cutoff}")
```

```
print(f"Последний кластер: {tail_cutoffs[-1]} - {vocab_size}")


plt.figure(figsize=(10, 6))
plt.loglog(ranks, sorted_freqs, marker='.', linestyle='none',
label='Частоты')
plt.axvline(head_idx, color='red', linestyle='--', label=f'Head cutoff
({head_idx})')
for i, cutoff in enumerate(tail_cutoffs):
    plt.axvline(cutoff, linestyle='--', label=f'Tail cutoff {i+1}
({cutoff})')
plt.xlabel('Лог(ранг токена)')
plt.ylabel('Лог(частота)')
plt.title('Распределение частот токенов с порогами cutoffs')
plt.grid(True, which="both", ls="--")
plt.legend()
plt.show()
```

Приложение Т

Построение и обучение классификатора стилей

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
from torch.optim import AdamW
from transformers import DistilBertTokenizer, DistilBertModel
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
import pandas as pd
import numpy as np
from tqdm import tqdm
import os
import matplotlib.pyplot as plt
import torch.nn.functional as F


MODEL_NAME = "DeepPavlov/distilrubert-base-cased-conversational"
MAX_LENGTH = 96
NUM_LABELS = 2
LEARNING_RATE = 2e-5
NUM_EPOCHS = 3
BATCH_SIZE = 2448
TEST_SIZE = 0.2
VAL_TEST_SPLIT = 0.3
RANDOM_STATE = 42
SAVE_DIR = "style_classifier"
MODEL_PATH = os.path.join(SAVE_DIR, "model.pth")
LOSS_PLOT_PATH = os.path.join(SAVE_DIR, "training_loss.png")

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


data = pd.read_csv("data.csv")
texts = list(data['tg_text']) + list(data['lit_text'])
labels = [0] * len(data['tg_text']) + [1] * len(data['lit_text'])


train_texts, temp_texts, train_labels, temp_labels = train_test_split(
    texts, labels, test_size=TEST_SIZE, random_state=RANDOM_STATE
)
```

```
val_texts, test_texts, val_labels, test_labels = train_test_split(
    temp_texts, temp_labels, test_size=VAL_TEST_SPLIT,
random_state=RANDOM_STATE
)


tokenizer = DistilBertTokenizer.from_pretrained(MODEL_NAME)


def tokenize(texts, max_length=MAX_LENGTH):
    return tokenizer(texts, padding=True, truncation=True,
max_length=max_length, return_tensors='pt')


train_encodings = tokenize(train_texts)
val_encodings = tokenize(val_texts)
test_encodings = tokenize(test_texts)

os.makedirs(SAVE_DIR, exist_ok=True)

torch.save(train_encodings, os.path.join(SAVE_DIR,
"train_encodings.pt"))
torch.save(val_encodings, os.path.join(SAVE_DIR, "val_encodings.pt"))
torch.save(test_encodings, os.path.join(SAVE_DIR,
"test_encodings.pt"))


train_encodings = torch.load(os.path.join(SAVE_DIR,
"train_encodings.pt"))
val_encodings = torch.load(os.path.join(SAVE_DIR, "val_encodings.pt"))
test_encodings = torch.load(os.path.join(SAVE_DIR,
"test_encodings.pt"))


class StyleDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
```

```
        item = {key: val[idx] for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx],
dtype=torch.long)
        return item


train_dataset = StyleDataset(train_encodings, train_labels)
val_dataset = StyleDataset(val_encodings, val_labels)
test_dataset = StyleDataset(test_encodings, test_labels)


class StyleClassifier(nn.Module):
    def __init__(self, num_labels=NUM_LABELS):
        super().__init__()
        self.bert = DistilBertModel.from_pretrained(MODEL_NAME)
        self.classifier = nn.Linear(self.bert.config.hidden_size,
num_labels)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids,
attention_mask=attention_mask)
        cls_output = outputs.last_hidden_state[:, 0, :]
        logits = self.classifier(cls_output)
        return logits


model = StyleClassifier().to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = AdamW(model.parameters(), lr=LEARNING_RATE)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE)
scaler = torch.cuda.amp.GradScaler()


train_losses = []
val_losses = []

for epoch in range(NUM_EPOCHS):
    model.train()
    total_train_loss = 0
    for batch in tqdm(train_loader):
```

```
            optimizer.zero_grad()
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            with torch.amp.autocast(device.type):
                logits = model(input_ids, attention_mask)
                loss = loss_fn(logits, labels)

            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
            total_train_loss += loss.item()

        avg_train_loss = total_train_loss / len(train_loader)
        train_losses.append(avg_train_loss)
        print(f"Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}")

        model.eval()
        total_val_loss = 0
        val_preds = []
        val_true = []
        with torch.no_grad():
            for batch in val_loader:
                input_ids = batch['input_ids'].to(device)
                attention_mask = batch['attention_mask'].to(device)
                labels = batch['labels'].to(device)

                with torch.amp.autocast(device.type):
                    logits = model(input_ids, attention_mask)
                    loss = loss_fn(logits, labels)
                    preds = torch.argmax(logits, dim=1)

                total_val_loss += loss.item()
                val_preds.extend(preds.cpu().numpy())
                val_true.extend(labels.cpu().numpy())

        avg_val_loss = total_val_loss / len(val_loader)
        val_losses.append(avg_val_loss)
        val_accuracy = accuracy_score(val_true, val_preds)
        val_f1 = f1_score(val_true, val_preds, average='weighted')
        print(f"Epoch {epoch+1}, Validation Loss: {avg_val_loss:.4f},
Accuracy: {val_accuracy:.4f}, F1 Score: {val_f1:.4f}")
```

```
model.eval()
test_preds = []
test_true = []
with torch.no_grad():
    for batch in test_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        with torch.amp.autocast(device.type):
            logits = model(input_ids, attention_mask)
            preds = torch.argmax(logits, dim=1)

        test_preds.extend(preds.cpu().numpy())
        test_true.extend(labels.cpu().numpy())

test_accuracy = accuracy_score(test_true, test_preds)
test_f1 = f1_score(test_true, test_preds, average='weighted')
print(f"Test Accuracy: {test_accuracy:.4f}, F1 Score: {test_f1:.4f}")


plt.figure(figsize=(8, 6))
plt.plot(range(1, NUM_EPOCHS + 1), train_losses, label='Train Loss',
marker='o')
plt.plot(range(1, NUM_EPOCHS + 1), val_losses, label='Validation
Loss', marker='o')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.savefig(LOSS_PLOT_PATH)
plt.close()


os.makedirs(SAVE_DIR, exist_ok=True)
torch.save(model.state_dict(), MODEL_PATH)
tokenizer.save_pretrained(SAVE_DIR)

model = StyleClassifier()
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
```

```
model.to(device)
model.eval()

tokenizer = DistilBertTokenizer.from_pretrained(SAVE_DIR)


def predict_style(texts, return_probs=False):
    encodings = tokenizer(texts, padding=True, truncation=True,
max_length=MAX_LENGTH, return_tensors="pt")
    input_ids = encodings['input_ids'].to(device)
    attention_mask = encodings['attention_mask'].to(device)

    with torch.no_grad():
        logits = model(input_ids, attention_mask)
        probs = torch.softmax(logits, dim=1)

    if return_probs:
        return probs.cpu().numpy()
    else:
        preds = torch.argmax(probs, dim=1)
        return preds.cpu().numpy()


conv_text = "Было же время, всё было как будто чище, легче. Дышалось.
Не знаю, как объяснить — но тогда просто жил и не думал, зачем. И это
было нормально. а теперь всё как будто через фильтр, чужой."
lit_text = "Он вспоминал то время не как череду событий, а как
состояние: утро, в котором не нужно ничего решать. Пустота, от которой
не страшно. Тогда он просто существовал — не объясняя себе зачем.
Теперь всё иначе, и в этом иначе не было покоя."
predict_style(conv_text, return_probs=True), predict_style(lit_text,
return_probs=True)


def style_loss(predicted_texts, target_style_label):
    encodings = tokenizer(predicted_texts, padding=True,
truncation=True, max_length=96, return_tensors='pt').to(device)

    with torch.no_grad():
        logits = model(encodings['input_ids'],
encodings['attention_mask'])

    target_labels = torch.full((logits.size(0),), target_style_label,
dtype=torch.long, device=device)
```

```
loss = F.cross_entropy(logits, target_labels)
return loss
```

Приложение У

Обучение модели CycleGAN с использованием mBART

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
from torch.optim import AdamW
from transformers import (
    MBartForConditionalGeneration, MBart50TokenizerFast,
    DistilBertModel, DistilBertTokenizer,
    get_linear_schedule_with_warmup, GenerationConfig
)
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
from tqdm import tqdm
import os
import matplotlib.pyplot as plt
import random
from torch.cuda.amp import autocast, GradScaler
import itertools


torch.manual_seed(42)
torch.cuda.manual_seed_all(42)
np.random.seed(42)
random.seed(42)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

TAG_TO_LIT = "[TO_LIT]"
TAG_TO_CONV = "[TO_CONV]"

GEN_MODEL_NAME = "sn4kebyt3/ru-bart-large"
CLASSIFIER_MODEL_NAME =
"DeepPavlov/distilrubert-base-cased-conversational"
MAX_LENGTH = 128
NUM_EPOCHS = 12
```

```
BATCH_SIZE = 96
STEPS_PER_EPOCH = 125

LEARNING_RATE_GEN = 2e-5
LEARNING_RATE_DISC = 4e-5
TEST_SIZE = 0.1
VAL_TEST_SPLIT = 0.5
RANDOM_STATE = 42
SAVE_DIR = "tag_cyclegan_bart_final_v2"
MODEL_PATH_G_BEST = os.path.join(SAVE_DIR, "G_bart_best.pth")
MODEL_PATH_D_C_BEST = os.path.join(SAVE_DIR, "D_C_best.pth")
MODEL_PATH_D_L_BEST = os.path.join(SAVE_DIR, "D_L_best.pth")
PLOT_PATH = os.path.join(SAVE_DIR, "training_plots_final.png")

LAMBDA_CYCLE = 4.0
LAMBDA_IDENTITY = 4.0
LAMBDA_STYLE = 12.0
LAMBDA_ADV = 1.5

base_gen_config_train = GenerationConfig(
    max_length=MAX_LENGTH, min_length=5, num_beams=3,
    early_stopping=True, temperature=1.0,
    repetition_penalty=1.0, no_repeat_ngram_size=0
)
base_gen_config_val_log = GenerationConfig(
    max_length=MAX_LENGTH, min_length=10, num_beams=3,
    early_stopping=True, temperature=1.0,
    repetition_penalty=1.2, no_repeat_ngram_size=4
)
NUM_VAL_STYLE_EXAMPLES = BATCH_SIZE


try:
    data = pd.read_csv("data.csv")
    texts_C_full = list(data['tg_text'].astype(str))
    texts_L_full = list(data['lit_text'].astype(str))
    print(f"Загружено {len(texts_C_full)} разговорных и
{len(texts_L_full)} литературных текстов.")
except Exception as e:
    print(f"Ошибка загрузки данных: {e}"); raise

train_texts_C, temp_texts_C = train_test_split(texts_C_full,
test_size=TEST_SIZE, random_state=RANDOM_STATE)
```

```
train_texts_L, temp_texts_L = train_test_split(texts_L_full,
test_size=TEST_SIZE, random_state=RANDOM_STATE)
val_texts_C, _ = train_test_split(temp_texts_C,
test_size=VAL_TEST_SPLIT, random_state=RANDOM_STATE)
val_texts_L, _ = train_test_split(temp_texts_L,
test_size=VAL_TEST_SPLIT, random_state=RANDOM_STATE)
print(f"Размеры датасетов: Train C: {len(train_texts_C)}, Val C:
{len(val_texts_C)}")
print(f"                   Train L: {len(train_texts_L)}, Val L:
{len(val_texts_L)}")



try:
    gen_tokenizer =
MBart50TokenizerFast.from_pretrained(GEN_MODEL_NAME)
    style_tokenizer =
DistilBertTokenizer.from_pretrained(CLASSIFIER_MODEL_NAME)


    special_tokens_to_add = {'additional_special_tokens': [TAG_TO_LIT,
TAG_TO_CONV]}
    num_added_toks =
gen_tokenizer.add_special_tokens(special_tokens_to_add)
    print(f"Добавлено {num_added_toks} спец. токенов в gen_tokenizer:
{TAG_TO_LIT}, {TAG_TO_CONV} (Новый размер словаря:
{len(gen_tokenizer)})")


    RUSSIAN_TOKEN_ID = gen_tokenizer.lang_code_to_id.get("ru_RU",
gen_tokenizer.eos_token_id)
    if RUSSIAN_TOKEN_ID == gen_tokenizer.eos_token_id and "ru_RU" not
in gen_tokenizer.lang_code_to_id:
        print(f"ПРЕДУПРЕЖДЕНИЕ: Токен языка 'ru_RU' не найден.
Используется eos_token_id ({RUSSIAN_TOKEN_ID})")
    print(f"ID токена русского языка для mBART генератора:
{RUSSIAN_TOKEN_ID}")
except Exception as e:
    print(f"Ошибка инициализации токенизаторов: {e}"); raise



def update_generation_config(base_config, model_config_obj,
tokenizer_pad_token_id):
    updated_config = GenerationConfig.from_dict(base_config.to_dict())
    updated_config.decoder_start_token_id =
model_config_obj.decoder_start_token_id
    updated_config.eos_token_id = model_config_obj.eos_token_id
    updated_config.pad_token_id = tokenizer_pad_token_id
```

```
    if hasattr(model_config_obj, 'forced_bos_token_id') and
model_config_obj.forced_bos_token_id is not None:
        updated_config.forced_bos_token_id =
model_config_obj.forced_bos_token_id
    return updated_config


class StyleDataset(Dataset):
    def __init__(self, texts_list, generator_tokenizer,
max_sequence_length,
                 generator_input_tag=None,
create_identity_labels=False):
        self.texts = [str(text) for text in texts_list]
        self.generator_tokenizer = generator_tokenizer
        self.max_sequence_length = max_sequence_length
        self.generator_input_tag = generator_input_tag
        self.create_identity_labels = create_identity_labels

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        original_text_string = self.texts[idx]
        input_text_for_g = f"{self.generator_input_tag}
{original_text_string}" if self.generator_input_tag else
original_text_string
        tokenized_g_input = self.generator_tokenizer(input_text_for_g,
padding="max_length", truncation=True,
max_length=self.max_sequence_length, return_tensors='pt')
        item = {key: val.squeeze(0) for key, val in
tokenized_g_input.items()}
        item['original_text_str'] = original_text_string
        tokenized_original =
self.generator_tokenizer(original_text_string, padding="max_length",
truncation=True, max_length=self.max_sequence_length,
return_tensors='pt')
        item['original_ids'] =
tokenized_original['input_ids'].squeeze(0)
        item['original_mask'] =
tokenized_original['attention_mask'].squeeze(0)
        if self.create_identity_labels: item['labels'] =
item['original_ids'].clone()
        return item
```

```
train_dataset_C_main = StyleDataset(train_texts_C, gen_tokenizer,
MAX_LENGTH)
train_dataset_L_main = StyleDataset(train_texts_L, gen_tokenizer,
MAX_LENGTH)
train_dataloader_C = DataLoader(train_dataset_C_main,
batch_size=BATCH_SIZE, shuffle=True, drop_last=True, num_workers=0,
pin_memory=True if device.type == 'cuda' else False)
train_dataloader_L = DataLoader(train_dataset_L_main,
batch_size=BATCH_SIZE, shuffle=True, drop_last=True, num_workers=0,
pin_memory=True if device.type == 'cuda' else False)


val_dataset_L_identity_main = StyleDataset(val_texts_L, gen_tokenizer,
MAX_LENGTH, generator_input_tag=TAG_TO_LIT,
create_identity_labels=True)
val_dataset_C_identity_main = StyleDataset(val_texts_C, gen_tokenizer,
MAX_LENGTH, generator_input_tag=TAG_TO_CONV,
create_identity_labels=True)
val_dataloader_L_identity = DataLoader(val_dataset_L_identity_main,
batch_size=BATCH_SIZE, num_workers=0, pin_memory=True if device.type
== 'cuda' else False)
val_dataloader_C_identity = DataLoader(val_dataset_C_identity_main,
batch_size=BATCH_SIZE, num_workers=0, pin_memory=True if device.type
== 'cuda' else False)


val_dataset_C_for_style = StyleDataset(val_texts_C, gen_tokenizer,
MAX_LENGTH)
val_dataset_L_for_style = StyleDataset(val_texts_L, gen_tokenizer,
MAX_LENGTH)
val_dataloader_C_for_style = DataLoader(val_dataset_C_for_style,
batch_size=BATCH_SIZE, num_workers=0, pin_memory=True if device.type
== 'cuda' else False)
val_dataloader_L_for_style = DataLoader(val_dataset_L_for_style,
batch_size=BATCH_SIZE, num_workers=0, pin_memory=True if device.type
== 'cuda' else False)



class Generator(nn.Module):
    def __init__(self, model_name_str, tokenizer_vocabulary_size,
russian_language_token_id):
        super().__init__()
        self.bart_model =
MBartForConditionalGeneration.from_pretrained(model_name_str)

self.bart_model.resize_token_embeddings(tokenizer_vocabulary_size)
        self.generation_config_internal = self.bart_model.config
        self.generation_config_internal.forced_bos_token_id =
russian_language_token_id
```

```
        self.generation_config_internal.decoder_start_token_id =
russian_language_token_id
        self.pad_token_id = gen_tokenizer.pad_token_id
        print(f"Генератор инициализирован:
forced_bos={self.generation_config_internal.forced_bos_token_id},
dec_start={self.generation_config_internal.decoder_start_token_id}")

    def forward(self, input_ids, attention_mask, labels=None):
        return self.bart_model(input_ids=input_ids,
attention_mask=attention_mask, labels=labels)

    def generate_texts(self, input_ids, attention_mask,
external_generation_config):
        return self.bart_model.generate(input_ids=input_ids,
attention_mask=attention_mask,
generation_config=external_generation_config)

    def get_attention_mask_for_generated(self, generated_ids_tensor):
        return (generated_ids_tensor != self.pad_token_id).long()


class Discriminator(nn.Module):
    def __init__(self, vocabulary_size, max_sequence_length):
        super().__init__()
        self.embedding_layer = nn.Embedding(vocabulary_size, 128)
        self.cnn_layers = nn.Sequential(
            nn.Conv1d(128, 256, kernel_size=3, padding=1), nn.ReLU(),
nn.MaxPool1d(kernel_size=2),
            nn.Conv1d(256, 512, kernel_size=3, padding=1), nn.ReLU(),
nn.MaxPool1d(kernel_size=2))
        self.fc_layer = nn.Linear(512, 1)

    def forward(self, input_ids_tensor, attention_mask_tensor=None):
        embedded_x = self.embedding_layer(input_ids_tensor);
        if attention_mask_tensor is not None: embedded_x = embedded_x
* attention_mask_tensor.unsqueeze(-1)
        permuted_x = embedded_x.permute(0,2,1); convolved_x =
self.cnn_layers(permuted_x)
        pooled_x = F.adaptive_avg_pool1d(convolved_x,1).squeeze(-1);
logits = self.fc_layer(pooled_x)
        return logits


class StyleClassifier(nn.Module):
    def __init__(self, classifier_model_name_str):
```

```
        super().__init__()
        self.bert =
DistilBertModel.from_pretrained(classifier_model_name_str)
        self.classifier = nn.Linear(self.bert.config.hidden_size, 2)

    def forward(self, input_ids_tensor, attention_mask_tensor):
        bert_output = self.bert(input_ids=input_ids_tensor,
attention_mask=attention_mask_tensor)
        cls_token_embedding = bert_output.last_hidden_state[:,0,:]
        logits = self.classifier(cls_token_embedding)
        return logits


G_model = Generator(GEN_MODEL_NAME, len(gen_tokenizer),
RUSSIAN_TOKEN_ID).to(device)
D_C_model = Discriminator(len(gen_tokenizer), MAX_LENGTH).to(device)
D_L_model = Discriminator(len(gen_tokenizer), MAX_LENGTH).to(device)

try:
    style_classifier_main_model =
StyleClassifier(CLASSIFIER_MODEL_NAME).to(device)
    style_classifier_model_path = "style_classifier/model.pth";
    if not os.path.exists(style_classifier_model_path): raise
FileNotFoundError(f"Нет классификатора:
{style_classifier_model_path}")

style_classifier_main_model.load_state_dict(torch.load(style_classifie
r_model_path, map_location=device)); print("Классификатор стиля
загружен.")
except Exception as e: print(f"Ошибка StyleClassifier: {e}"); raise
style_classifier_main_model.eval(); [param.requires_grad_(False) for
param in style_classifier_main_model.parameters()]

gen_config_train = update_generation_config(base_gen_config_train,
G_model.generation_config_internal, gen_tokenizer.pad_token_id)
gen_config_val_log = update_generation_config(base_gen_config_val_log,
G_model.generation_config_internal, gen_tokenizer.pad_token_id)


optimizer_G = AdamW(G_model.parameters(), lr=LEARNING_RATE_GEN,
eps=1e-8)
optimizer_D_C = AdamW(D_C_model.parameters(), lr=LEARNING_RATE_DISC,
eps=1e-8)
optimizer_D_L = AdamW(D_L_model.parameters(), lr=LEARNING_RATE_DISC,
eps=1e-8)
```

```
num_total_training_steps = NUM_EPOCHS * STEPS_PER_EPOCH;
num_warmup_steps_sched = int(0.05 * num_total_training_steps)

scheduler_G = get_linear_schedule_with_warmup(optimizer_G,
num_warmup_steps=num_warmup_steps_sched,
num_training_steps=num_total_training_steps)
scheduler_D_C = get_linear_schedule_with_warmup(optimizer_D_C,
num_warmup_steps=num_warmup_steps_sched,
num_training_steps=num_total_training_steps)
scheduler_D_L = get_linear_schedule_with_warmup(optimizer_D_L,
num_warmup_steps=num_warmup_steps_sched,
num_training_steps=num_total_training_steps)



grad_scaler = GradScaler()
adversarial_loss_fn = nn.BCEWithLogitsLoss()
cross_entropy_loss_fn_for_G =
nn.CrossEntropyLoss(ignore_index=gen_tokenizer.pad_token_id)
style_classification_loss_fn = nn.CrossEntropyLoss()



def tokenize_for_style_classifier_utility(texts_batch,
style_cls_tokenizer, max_len_val, current_dev):
    return style_cls_tokenizer(texts_batch, padding="max_length",
truncation=True, max_length=max_len_val,
return_tensors='pt').to(current_dev)



@torch.no_grad()
def run_validation_final(
    generator_to_validate,
    val_dl_C_for_id, val_dl_L_for_id,
    val_dl_C_for_style_acc, val_dl_L_for_style_acc,
    style_classifier_for_val,
    current_validation_gen_config,
    num_examples_for_style_eval, current_eval_device
):
    generator_to_validate.eval()
    val_epoch_metrics = {}

    current_id_loss_C, num_id_batches_C = 0.0, 0
    for batch_data in tqdm(val_dl_C_for_id, desc="Validating Identity
C->C", leave=False, ncols=100):
        input_ids = batch_data['input_ids'].to(current_eval_device)
```

```
        attention_mask =
batch_data['attention_mask'].to(current_eval_device)
        labels = batch_data['labels'].to(current_eval_device)
        with autocast():
            outputs = generator_to_validate(input_ids, attention_mask,
labels=labels)
            current_id_loss_C += outputs.loss.item()
        num_id_batches_C += 1
    val_epoch_metrics['id_loss_C_val'] = current_id_loss_C /
num_id_batches_C if num_id_batches_C > 0 else float('inf')

    current_id_loss_L, num_id_batches_L = 0.0, 0
    for batch_data in tqdm(val_dl_L_for_id, desc="Validating Identity
L->L", leave=False, ncols=100):
        input_ids = batch_data['input_ids'].to(current_eval_device)
        attention_mask =
batch_data['attention_mask'].to(current_eval_device)
        labels = batch_data['labels'].to(current_eval_device)
        with autocast():
            outputs = generator_to_validate(input_ids, attention_mask,
labels=labels)
            current_id_loss_L += outputs.loss.item()
        num_id_batches_L += 1
    val_epoch_metrics['id_loss_L_val'] = current_id_loss_L /
num_id_batches_L if num_id_batches_L > 0 else float('inf')

    generated_L_examples_val, total_s_loss_C2L, correct_s_preds_C2L,
count_s_C2L = [], 0.0, 0, 0
    num_batches_to_process_C = min(len(val_dl_C_for_style_acc),
(num_examples_for_style_eval + BATCH_SIZE - 1) // BATCH_SIZE)

    for batch_data in tqdm(itertools.islice(val_dl_C_for_style_acc,
num_batches_to_process_C), desc="Validating Style C->L", leave=False,
ncols=100, total=num_batches_to_process_C):
        real_C_text_strings = batch_data['original_text_str']
        g_input_val_texts = [f"{TAG_TO_LIT} {text}" for text in
real_C_text_strings]
        tokenized_g_val_input = gen_tokenizer(g_input_val_texts,
padding="max_length", truncation=True, max_length=MAX_LENGTH,
return_tensors='pt').to(current_eval_device)
        with autocast():

            fake_L_ids_generated =
generator_to_validate.generate_texts(tokenized_g_val_input.input_ids,
```

```
tokenized_g_val_input.attention_mask,
external_generation_config=current_validation_gen_config)
        fake_L_text_strings =
gen_tokenizer.batch_decode(fake_L_ids_generated,
skip_special_tokens=True)
        tokenized_for_style_input =
tokenize_for_style_classifier_utility(fake_L_text_strings,
style_tokenizer, MAX_LENGTH, current_eval_device)
        with autocast():
            style_logits_output =
style_classifier_for_val(tokenized_for_style_input.input_ids,
tokenized_for_style_input.attention_mask)
        target_style_labels = torch.ones(style_logits_output.size(0),
dtype=torch.long, device=current_eval_device)
        total_s_loss_C2L +=
style_classification_loss_fn(style_logits_output,
target_style_labels).item() * style_logits_output.size(0)
        predicted_style_labels = torch.argmax(style_logits_output,
dim=1)
        correct_s_preds_C2L += (predicted_style_labels ==
target_style_labels).sum().item()
        count_s_C2L += style_logits_output.size(0)
        if not generated_L_examples_val:
generated_L_examples_val.extend(list(zip(real_C_text_strings[:3],
fake_L_text_strings[:3])))
    val_epoch_metrics['style_loss_C2L_val'] = total_s_loss_C2L /
count_s_C2L if count_s_C2L > 0 else float('inf')
    val_epoch_metrics['style_acc_C2L_val'] = correct_s_preds_C2L /
count_s_C2L if count_s_C2L > 0 else 0.0
    val_epoch_metrics['example_C2L_gen_val'] =
generated_L_examples_val


    generated_C_examples_val, total_s_loss_L2C, correct_s_preds_L2C,
count_s_L2C = [], 0.0, 0, 0
    num_batches_to_process_L = min(len(val_dl_L_for_style_acc),
(num_examples_for_style_eval + BATCH_SIZE - 1) // BATCH_SIZE)
    for batch_data in tqdm(itertools.islice(val_dl_L_for_style_acc,
num_batches_to_process_L), desc="Validating Style L->C", leave=False,
ncols=100, total=num_batches_to_process_L):
        real_L_text_strings = batch_data['original_text_str']
        g_input_val_texts = [f"{TAG_TO_CONV} {text}" for text in
real_L_text_strings]
        tokenized_g_val_input = gen_tokenizer(g_input_val_texts,
padding="max_length", truncation=True, max_length=MAX_LENGTH,
return_tensors='pt').to(current_eval_device)
        with autocast():
```

```
            fake_C_ids_generated =
generator_to_validate.generate_texts(tokenized_g_val_input.input_ids,
tokenized_g_val_input.attention_mask,
external_generation_config=current_validation_gen_config)
        fake_C_text_strings =
gen_tokenizer.batch_decode(fake_C_ids_generated,
skip_special_tokens=True)
        tokenized_for_style_input =
tokenize_for_style_classifier_utility(fake_C_text_strings,
style_tokenizer, MAX_LENGTH, current_eval_device)
        with autocast():
            style_logits_output =
style_classifier_for_val(tokenized_for_style_input.input_ids,
tokenized_for_style_input.attention_mask)
        target_style_labels = torch.zeros(style_logits_output.size(0),
dtype=torch.long, device=current_eval_device)
        total_s_loss_L2C +=
style_classification_loss_fn(style_logits_output,
target_style_labels).item() * style_logits_output.size(0)
        predicted_style_labels = torch.argmax(style_logits_output,
dim=1)
        correct_s_preds_L2C += (predicted_style_labels ==
target_style_labels).sum().item()
        count_s_L2C += style_logits_output.size(0)
        if not generated_C_examples_val:
generated_C_examples_val.extend(list(zip(real_L_text_strings[:3],
fake_C_text_strings[:3])))
    val_epoch_metrics['style_loss_L2C_val'] = total_s_loss_L2C /
count_s_L2C if count_s_L2C > 0 else float('inf')
    val_epoch_metrics['style_acc_L2C_val'] = correct_s_preds_L2C /
count_s_L2C if count_s_L2C > 0 else 0.0
    val_epoch_metrics['example_L2C_gen_val'] =
generated_C_examples_val

    val_epoch_metrics['G_total_val_comparable'] = \
        (val_epoch_metrics['id_loss_C_val'] +
val_epoch_metrics['id_loss_L_val']) * 0.5 * LAMBDA_IDENTITY + \
        (val_epoch_metrics['style_loss_C2L_val'] +
val_epoch_metrics['style_loss_L2C_val']) * 0.5 * LAMBDA_STYLE

    generator_to_validate.train()
    return val_epoch_metrics


training_history = {
    'epoch': [],
```

```
    'G_total_train_full': [], 'G_adv_train': [], 'G_cycle_train': [],
    'G_identity_train': [], 'G_style_train': [], 'D_total_train': [],
    'G_total_train_comparable': [],
    'G_id_loss_C_val': [], 'G_id_loss_L_val': [],
    'G_style_loss_C2L_val': [], 'G_style_acc_C2L_val': [],
    'G_style_loss_L2C_val': [], 'G_style_acc_L2C_val': [],
    'G_total_val_comparable': []
}
best_validation_metric = float('inf')
os.makedirs(SAVE_DIR, exist_ok=True)

data_iterator_C = itertools.cycle(train_dataloader_C)
data_iterator_L = itertools.cycle(train_dataloader_L)

for epoch_num in range(NUM_EPOCHS):
    G_model.train(); D_C_model.train(); D_L_model.train()

    current_epoch_train_losses_sum = {
        'G_total_train_full': 0.0, 'G_adv_train': 0.0,
'G_cycle_train': 0.0,
        'G_identity_train': 0.0, 'G_style_train': 0.0,
'D_total_train': 0.0,
        'G_total_train_comparable': 0.0  # Для сопоставимого графика
    }

    progress_bar = tqdm(range(STEPS_PER_EPOCH), desc=f"Эпоха
{epoch_num + 1}/{NUM_EPOCHS}", ncols=120)

    for step_num in progress_bar:
        batch_C_data = next(data_iterator_C)
        batch_L_data = next(data_iterator_L)

        real_C_original_ids = batch_C_data['original_ids'].to(device)
        real_C_original_mask =
batch_C_data['original_mask'].to(device)
        real_C_original_texts_list = batch_C_data['original_text_str']

        real_L_original_ids = batch_L_data['original_ids'].to(device)
        real_L_original_mask =
batch_L_data['original_mask'].to(device)
        real_L_original_texts_list = batch_L_data['original_text_str']

        optimizer_D_C.zero_grad()
```

```
        optimizer_D_L.zero_grad()

        g_input_C_to_L_texts_list = [f"{TAG_TO_LIT} {text}" for text
in real_C_original_texts_list]
        tokenized_g_input_C_to_L =
gen_tokenizer(g_input_C_to_L_texts_list, padding="max_length",
truncation=True, max_length=MAX_LENGTH,
return_tensors='pt').to(device)

        g_input_L_to_C_texts_list = [f"{TAG_TO_CONV} {text}" for text
in real_L_original_texts_list]
        tokenized_g_input_L_to_C =
gen_tokenizer(g_input_L_to_C_texts_list, padding="max_length",
truncation=True, max_length=MAX_LENGTH,
return_tensors='pt').to(device)

        with torch.no_grad():
            with autocast():
                fake_L_generated_ids =
G_model.generate_texts(tokenized_g_input_C_to_L.input_ids,
tokenized_g_input_C_to_L.attention_mask,
external_generation_config=gen_config_train)
                fake_C_generated_ids =
G_model.generate_texts(tokenized_g_input_L_to_C.input_ids,
tokenized_g_input_L_to_C.attention_mask,
external_generation_config=gen_config_train)

        fake_L_generated_mask =
G_model.get_attention_mask_for_generated(fake_L_generated_ids)
        fake_C_generated_mask =
G_model.get_attention_mask_for_generated(fake_C_generated_ids)

        with autocast():
            d_l_pred_on_real = D_L_model(real_L_original_ids,
real_L_original_mask)
            d_l_pred_on_fake =
D_L_model(fake_L_generated_ids.detach(), fake_L_generated_mask)
            loss_D_L_total_step =
(adversarial_loss_fn(d_l_pred_on_real,
torch.ones_like(d_l_pred_on_real)) + \

adversarial_loss_fn(d_l_pred_on_fake,
torch.zeros_like(d_l_pred_on_fake))) * 0.5

            d_c_pred_on_real = D_C_model(real_C_original_ids,
real_C_original_mask)
```

```
            d_c_pred_on_fake =
D_C_model(fake_C_generated_ids.detach(), fake_C_generated_mask)
            loss_D_C_total_step =
(adversarial_loss_fn(d_c_pred_on_real,
torch.ones_like(d_c_pred_on_real)) + \

adversarial_loss_fn(d_c_pred_on_fake,
torch.zeros_like(d_c_pred_on_fake))) * 0.5


            loss_D_combined_step = loss_D_L_total_step +
loss_D_C_total_step


        grad_scaler.scale(loss_D_combined_step).backward() # backward
для D


        optimizer_G.zero_grad()
        with autocast():
            fake_L_ids_for_G =
G_model.generate_texts(tokenized_g_input_C_to_L.input_ids,
tokenized_g_input_C_to_L.attention_mask,
external_generation_config=gen_config_train)
            fake_C_ids_for_G =
G_model.generate_texts(tokenized_g_input_L_to_C.input_ids,
tokenized_g_input_L_to_C.attention_mask,
external_generation_config=gen_config_train)
            fake_L_mask_for_G =
G_model.get_attention_mask_for_generated(fake_L_ids_for_G)
            fake_C_mask_for_G =
G_model.get_attention_mask_for_generated(fake_C_ids_for_G)


            loss_G_adv_L_component =
adversarial_loss_fn(D_L_model(fake_L_ids_for_G, fake_L_mask_for_G),
torch.ones_like(D_L_model(fake_L_ids_for_G, fake_L_mask_for_G)))
            loss_G_adv_C_component =
adversarial_loss_fn(D_C_model(fake_C_ids_for_G, fake_C_mask_for_G),
torch.ones_like(D_C_model(fake_C_ids_for_G, fake_C_mask_for_G)))
            loss_G_adversarial_total = (loss_G_adv_L_component +
loss_G_adv_C_component) * LAMBDA_ADV


            fake_L_texts_for_style_clf =
gen_tokenizer.batch_decode(fake_L_ids_for_G, skip_special_tokens=True)
            fake_C_texts_for_style_clf =
gen_tokenizer.batch_decode(fake_C_ids_for_G, skip_special_tokens=True)
            tokenized_L_for_style =
tokenize_for_style_classifier_utility(fake_L_texts_for_style_clf,
style_tokenizer, MAX_LENGTH, device)
```

```
            tokenized_C_for_style =
tokenize_for_style_classifier_utility(fake_C_texts_for_style_clf,
style_tokenizer, MAX_LENGTH, device)
            style_L_predictions =
style_classifier_main_model(tokenized_L_for_style.input_ids,
tokenized_L_for_style.attention_mask)
            style_C_predictions =
style_classifier_main_model(tokenized_C_for_style.input_ids,
tokenized_C_for_style.attention_mask)
            loss_G_style_L_comp =
style_classification_loss_fn(style_L_predictions,
torch.ones(style_L_predictions.size(0), dtype=torch.long,
device=device))
            loss_G_style_C_comp =
style_classification_loss_fn(style_C_predictions,
torch.zeros(style_C_predictions.size(0), dtype=torch.long,
device=device))
            loss_G_style_total = (loss_G_style_L_comp +
loss_G_style_C_comp) * LAMBDA_STYLE


            g_input_reconstruct_C_texts_list = [f"{TAG_TO_CONV}
{text}" for text in fake_L_texts_for_style_clf]
            tokenized_g_input_reconstruct_C =
gen_tokenizer(g_input_reconstruct_C_texts_list, padding="max_length",
truncation=True, max_length=MAX_LENGTH,
return_tensors='pt').to(device)
            reconstructed_C_outputs =
G_model(tokenized_g_input_reconstruct_C.input_ids,
tokenized_g_input_reconstruct_C.attention_mask,
labels=real_C_original_ids)
            loss_G_cycle_C_component = reconstructed_C_outputs.loss
            g_input_reconstruct_L_texts_list = [f"{TAG_TO_LIT} {text}"
for text in fake_C_texts_for_style_clf]
            tokenized_g_input_reconstruct_L =
gen_tokenizer(g_input_reconstruct_L_texts_list, padding="max_length",
truncation=True, max_length=MAX_LENGTH,
return_tensors='pt').to(device)
            reconstructed_L_outputs =
G_model(tokenized_g_input_reconstruct_L.input_ids,
tokenized_g_input_reconstruct_L.attention_mask,
labels=real_L_original_ids)
            loss_G_cycle_L_component = reconstructed_L_outputs.loss
            loss_G_cycle_total = (loss_G_cycle_C_component +
loss_G_cycle_L_component) * LAMBDA_CYCLE


            g_input_identity_L_texts_list = [f"{TAG_TO_LIT} {text}"
for text in real_L_original_texts_list]
```

```
            tokenized_g_input_identity_L =
gen_tokenizer(g_input_identity_L_texts_list, padding="max_length",
truncation=True, max_length=MAX_LENGTH,
return_tensors='pt').to(device)
            identity_L_outputs =
G_model(tokenized_g_input_identity_L.input_ids,
tokenized_g_input_identity_L.attention_mask,
labels=real_L_original_ids)
            loss_G_identity_L_component = identity_L_outputs.loss
            g_input_identity_C_texts_list = [f"{TAG_TO_CONV} {text}"
for text in real_C_original_texts_list]
            tokenized_g_input_identity_C =
gen_tokenizer(g_input_identity_C_texts_list, padding="max_length",
truncation=True, max_length=MAX_LENGTH,
return_tensors='pt').to(device)
            identity_C_outputs =
G_model(tokenized_g_input_identity_C.input_ids,
tokenized_g_input_identity_C.attention_mask,
labels=real_C_original_ids)
            loss_G_identity_C_component = identity_C_outputs.loss
            loss_G_identity_total = (loss_G_identity_L_component +
loss_G_identity_C_component) * LAMBDA_IDENTITY


            loss_G_full_step = loss_G_adversarial_total +
loss_G_style_total + loss_G_cycle_total + loss_G_identity_total


        grad_scaler.scale(loss_G_full_step).backward() # backward для
G


        grad_scaler.step(optimizer_D_L)
        grad_scaler.step(optimizer_D_C)
        torch.nn.utils.clip_grad_norm_(G_model.parameters(), 1.0)
        grad_scaler.step(optimizer_G)


        grad_scaler.update()


        scheduler_G.step()
        scheduler_D_C.step()
        scheduler_D_L.step()


        current_epoch_train_losses_sum['D_total_train'] +=
loss_D_combined_step.item()
        current_epoch_train_losses_sum['G_total_train_full'] +=
loss_G_full_step.item()
        current_epoch_train_losses_sum['G_adv_train'] +=
loss_G_adversarial_total.item()
```

```
        current_epoch_train_losses_sum['G_style_train'] +=
loss_G_style_total.item()
        current_epoch_train_losses_sum['G_cycle_train'] +=
loss_G_cycle_total.item()
        current_epoch_train_losses_sum['G_identity_train'] +=
loss_G_identity_total.item()

        g_total_comparable_train_step_unweighted_id =
(loss_G_identity_L_component.item() +
loss_G_identity_C_component.item()) * 0.5
        g_total_comparable_train_step_unweighted_style =
(loss_G_style_L_comp.item() + loss_G_style_C_comp.item()) * 0.5
        g_total_comparable_train_step =
(g_total_comparable_train_step_unweighted_id * LAMBDA_IDENTITY) + \

(g_total_comparable_train_step_unweighted_style * LAMBDA_STYLE)
        current_epoch_train_losses_sum['G_total_train_comparable'] +=
g_total_comparable_train_step

        progress_bar.set_postfix({
            "G_Full": f"{loss_G_full_step.item():.2f}",
            "D_Total": f"{loss_D_combined_step.item():.2f}",
            "LR_G": f"{scheduler_G.get_last_lr()[0]:.2e}"
        })

    training_history['epoch'].append(epoch_num + 1)
    for key_hist in current_epoch_train_losses_sum:

training_history[key_hist].append(current_epoch_train_losses_sum[key_h
ist] / STEPS_PER_EPOCH)

    validation_epoch_results = run_validation_final(
        G_model, val_dataloader_C_identity, val_dataloader_L_identity,
        val_dataloader_C_for_style, val_dataloader_L_for_style,
        style_classifier_main_model,
        gen_config_val_log,
        NUM_VAL_STYLE_EXAMPLES, device
    )


training_history['G_id_loss_C_val'].append(validation_epoch_results['i
d_loss_C_val'])

training_history['G_id_loss_L_val'].append(validation_epoch_results['i
d_loss_L_val'])
```

```
training_history['G_style_loss_C2L_val'].append(validation_epoch_resul
ts['style_loss_C2L_val'])

training_history['G_style_acc_C2L_val'].append(validation_epoch_result
s['style_acc_C2L_val'])

training_history['G_style_loss_L2C_val'].append(validation_epoch_resul
ts['style_loss_L2C_val'])

training_history['G_style_acc_L2C_val'].append(validation_epoch_result
s['style_acc_L2C_val'])

training_history['G_total_val_comparable'].append(validation_epoch_res
ults['G_total_val_comparable'])


    print(f"\n--- Результаты Эпохи {epoch_num + 1}/{NUM_EPOCHS} ---")
    print(f"  Потери Тренировки:
G_Full={training_history['G_total_train_full'][-1]:.3f} "
          f"(Adv={training_history['G_adv_train'][-1]:.3f},
Style={training_history['G_style_train'][-1]:.3f}, "
          f"Cyc={training_history['G_cycle_train'][-1]:.3f},
Id={training_history['G_identity_train'][-1]:.3f}), "
          f"D_Total={training_history['D_total_train'][-1]:.3f}")
    print(f"
G_Comparable_Train={training_history['G_total_train_comparable'][-1]:.
3f}")
    print(f"  Метрики Валидации:
G_Id(C|L)={validation_epoch_results['id_loss_C_val']:.3f}|{validation_
epoch_results['id_loss_L_val']:.3f}, "
          f"G_Style(C2L
L|A)={validation_epoch_results['style_loss_C2L_val']:.3f}|{validation_
epoch_results['style_acc_C2L_val']:.2%}, "
          f"G_Style(L2C
L|A)={validation_epoch_results['style_loss_L2C_val']:.3f}|{validation_
epoch_results['style_acc_L2C_val']:.2%}")
    print(f"
G_Total_Comparable_Val={validation_epoch_results['G_total_val_comparab
le']:.3f}")


    print("  Примеры генерации (Валидация):")
    for i, (inp_text, gen_text) in
enumerate(validation_epoch_results['example_C2L_gen_val']):
        print(f"    C->L {i+1} IN:  \"{inp_text[:70].replace(chr(10),'
')}...\"")
        print(f"            OUT: \"{gen_text[:70].replace(chr(10),'
')}...\"")
```

```
    for i, (inp_text, gen_text) in
enumerate(validation_epoch_results['example_L2C_gen_val']):
        print(f"    L->C {i+1} IN:  \"{inp_text[:70].replace(chr(10),'
')}...\"")
        print(f"            OUT: \"{gen_text[:70].replace(chr(10),'
')}...\"")

    current_validation_metric_for_saving =
validation_epoch_results['G_total_val_comparable']
    if current_validation_metric_for_saving < best_validation_metric:
        best_validation_metric = current_validation_metric_for_saving
        torch.save(G_model.state_dict(), MODEL_PATH_G_BEST)
        torch.save(D_C_model.state_dict(), MODEL_PATH_D_C_BEST)
        torch.save(D_L_model.state_dict(), MODEL_PATH_D_L_BEST)
        print(f"  *** Новая лучшая модель сохранена! Val
G_Comparable_Loss: {best_validation_metric:.3f} ***")

    fig, axs = plt.subplots(3, 1, figsize=(14, 21))
    fig.suptitle(f"Результаты обучения - Эпоха {epoch_num + 1}",
fontsize=16)

    axs[0].plot(training_history['epoch'],
training_history['G_total_train_full'], '-o', label='G Total Train
(Full)', linewidth=2)
    axs[0].plot(training_history['epoch'],
training_history['G_adv_train'], ':o', label='G Adv Train')
    axs[0].plot(training_history['epoch'],
training_history['G_style_train'], ':o', label='G Style Train')
    axs[0].plot(training_history['epoch'],
training_history['G_cycle_train'], ':o', label='G Cycle Train')
    axs[0].plot(training_history['epoch'],
training_history['G_identity_train'], ':o', label='G Identity Train')
    axs[0].plot(training_history['epoch'],
training_history['D_total_train'], '-x', label='D Total Train',
linewidth=2)
    axs[0].set_title('Тренировочные Потери (Все Компоненты G)');
axs[0].set_xlabel('Эпоха'); axs[0].set_ylabel('Потеря')
    axs[0].legend(loc='upper right'); axs[0].grid(True)

    ax1_val_loss = axs[1]; ax1_val_acc = axs[1].twinx()
    p1, = ax1_val_loss.plot(training_history['epoch'],
training_history['G_id_loss_C_val'], '-o', label='G Id_C Val',
color='royalblue')
    p2, = ax1_val_loss.plot(training_history['epoch'],
training_history['G_id_loss_L_val'], '-o', label='G Id_L Val',
color='darkorange')
```

```
    p3, = ax1_val_loss.plot(training_history['epoch'],
training_history['G_style_loss_C2L_val'], '--o', label='G Style C->L
Loss Val', color='forestgreen')
    p4, = ax1_val_loss.plot(training_history['epoch'],
training_history['G_style_loss_L2C_val'], '--o', label='G Style L->C
Loss Val', color='crimson')
    p5, = ax1_val_acc.plot(training_history['epoch'],
training_history['G_style_acc_C2L_val'], '-x', label='G Style C->L Acc
Val', color='lime')
    p6, = ax1_val_acc.plot(training_history['epoch'],
training_history['G_style_acc_L2C_val'], '-x', label='G Style L->C Acc
Val', color='cyan')
    ax1_val_loss.set_title('Валидационные Метрики Генератора');
ax1_val_loss.set_xlabel('Эпоха'); ax1_val_loss.set_ylabel('Потеря
(Loss)')
    ax1_val_acc.set_ylabel('Точность (Accuracy)', color='teal');
ax1_val_acc.tick_params(axis='y', labelcolor='teal')
    handles1, labels1 = ax1_val_loss.get_legend_handles_labels();
handles2, labels2 = ax1_val_acc.get_legend_handles_labels()
    ax1_val_loss.legend(handles=handles1 + handles2, labels=labels1 +
labels2, loc='center left', bbox_to_anchor=(0.05, 0.5))
    ax1_val_loss.grid(True)


    axs[2].plot(training_history['epoch'],
training_history['G_total_train_comparable'], '-o', label='G Total
Comparable Train (Id+Style)')
    axs[2].plot(training_history['epoch'],
training_history['G_total_val_comparable'], '-x', label='G Total
Comparable Val (Id+Style)')
    axs[2].set_title('Сравнение G Total Comparable Loss (Train vs
Val)'); axs[2].set_xlabel('Эпоха'); axs[2].set_ylabel('Потеря
(Weighted Id+Style)')
    axs[2].legend(loc='upper right'); axs[2].grid(True)


    plt.tight_layout(rect=[0, 0.03, 1, 0.95]); plt.savefig(PLOT_PATH);
plt.close(fig)
    print(f"Графики сохранены по пути: {PLOT_PATH}")


print("--- Обучение завершено ---")
```