



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования

«МИРЭА – Российский технологический университет»

**РТУ МИРЭА**

Институт информационных технологий

Кафедра вычислительной техники

## КУРСОВАЯ РАБОТА

По дисциплине

«Проектирование и обучение нейронных сетей»

(наименование дисциплины)

Тема курсовой работы

Проектирование и разработка нейронных сетей для различных

(наименование темы)

систем. Сравнительный анализ предобученной и дообученной модели

Студент группы

ИКБО-04-22

(учебная группа)

Кликушин В.И.

(Фамилия Имя Отчество)

(подпись студента)

Руководитель курсовой работы

Ст. преподаватель ВТ Семенов Р.Э.

(Должность, звание, ученая степень)

(подпись руководителя)

Консультант

Зав. каф. ВТ, к.т.н. Платонова О.В.

(Должность, звание, ученая степень)

(подпись консультанта)

Работа представлена к защите « 14 » 05 2025 г.

Допущен к защите « 14 » 05 2025 г.

Москва 2025 г.



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования

«МИРЭА – Российский технологический университет»

**РТУ МИРЭА**

Институт информационных технологий

Кафедра вычислительной техники

Утверждаю

Заведующий кафедрой

Платонова О.В.

ФИО

«15» февраля 2025г.

### ЗАДАНИЕ

На выполнение курсовой работы

по дисциплине «Проектирование и обучение нейронных сетей»

Студент

Кликушин В.И.

Группа

ИКБО-04-22

Тема

Проектирование и разработка нейронных сетей для различных систем.

Сравнительный анализ предобученной и дообученной модели.

Исходные данные:

1. Описания основных методов разработки нейронных сетей.
2. Архитектуры современных нейронных сетей.

Перечень вопросов, подлежащих разработке, и обязательного графического материала:

1. Реализованный метод обучения нейронной сети на основе архитектуры «Трансформер».
2. Реализованный метод обучения нейронной сети на основе генеративной сети (GAN).
3. Реализованный метод обучения нейронной сети на основе графовой сети (GNN).
4. Сравнительный анализ предобученной и дообученной модели.

Срок представления к защите курсовой работы: до «30» мая 2025 г.

Задание на курсовую работу выдал

(Семенов Р.Э.)

Подпись

ФИО консультанта

«15» февраля 2025 г.

Задание на курсовую работу получил

(Кликушин В.И.)

Подпись

ФИО исполнителя

«15» февраля 2025 г.

Москва 2025г.

## ОТЗЫВ

на курсовую работу

по дисциплине «Проектирование и обучение нейронных сетей»

Студент Кликушин В.И. группа ИКБО-04-22  
(ФИО студента) (Группа)

### Характеристика курсовой работы

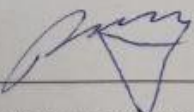
Критерий	Да	Нет	Не полностью
1. Соответствие содержания курсовой работы указанной теме	✓		
2. Соответствие курсовой работы заданию	✓		
3. Соответствие рекомендациям по оформлению текста, таблиц, рисунков и пр.	✓		
4. Полнота выполнения всех пунктов задания	✓		
5. Логичность и системность содержания курсовой работы	✓		
6. Отсутствие фактических грубых ошибок	✓		

Замечания:

нет

Рекомендуемая оценка:

отлично

  
(Подпись руководителя)

Ст. преподаватель ВТ Семенов Р.Э.

(ФИО руководителя)

# СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	6
1 ТРАНСФОРМЕР .....	8
1.1 Теоретический раздел .....	8
1.1.1 История появления архитектуры.....	8
1.1.2 Описание архитектуры .....	9
1.1.3 Применение трансформеров .....	13
1.2 Постановка задачи.....	14
1.3 Документация к данным.....	14
1.4 Обучение модели.....	15
1.5 Полученные результаты .....	18
1.6 Выводы по разделу.....	20
2 ГЕНЕРАТИВНО-СОСТЯЗАТЕЛЬНАЯ СЕТЬ.....	21
2.1 Теоретический раздел .....	21
2.1.1 История появления архитектуры.....	21
2.1.2 Описание архитектуры .....	22
2.2 Постановка задачи.....	27
2.3 Документация к данным.....	28
2.4 Обучение модели.....	34
2.5 Полученные результаты .....	36
2.6 Выводы по разделу.....	39
3 ГРАФОВАЯ СЕТЬ .....	40
3.1 Теоретический раздел .....	40
3.1.1 Описание графовых данных.....	40
3.1.2 Задачи на графах .....	42
3.1.2 История появления архитектуры.....	43
3.1.3 Описание архитектуры .....	44
3.2 Постановка задачи.....	47
3.3 Документация к данным.....	47

3.4 Обучение модели.....	51
3.5 Полученные результаты .....	54
3.6 Выводы по разделу.....	55
4 СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПРЕДОБУЧЕННОЙ И ДООБУЧЕННОЙ МОДЕЛИ.....	56
4.1 Теоретический раздел .....	56
4.2 Постановка задачи.....	57
4.3 Документация к данным.....	57
4.4 Предобученная модель .....	58
4.5 Дообучение модели.....	59
4.5 Выводы по разделу.....	60
ЗАКЛЮЧЕНИЕ .....	61
ПРИЛОЖЕНИЯ.....	62

## ВВЕДЕНИЕ

В последние десятилетия развитие методов глубокого обучения радикально преобразовало подходы к решению задач в области искусственного интеллекта. Начавшись с простых многослойных перцептронов и постепенного совершенствования алгоритмов обратного распространения ошибки, нейронные сети эволюционировали в сложные архитектуры, способные автоматически извлекать и обобщать высокоуровневые представления из огромных объёмов данных. Это привело к выдающимся достижениям в распознавании образов, обработке естественного языка, генерации реалистичных изображений и моделировании сложных структурных взаимосвязей.

Одной из ключевых вех в истории глубокого обучения стала архитектура трансформера, предложенная в 2017 году. Отказавшись от рекуррентных и сверточных механизмов в пользу глобального внимания, трансформеры позволили эффективно обрабатывать длинные последовательности текста и кода. Механизм многоголового внимания обеспечил одновременное взаимодействие каждого элемента входа со всеми другими, что значительно ускорило обучение и открыло путь к масштабированию моделей до миллиардов параметров. Результатом стали такие революционные системы, как GPT и BERT, способные решать задачи перевода, суммаризации, ответов на вопросы и генерации связного текста.

Другим направлением, кардинально изменившим представления о генеративных моделях, стали генеративно-состязательные сети (GAN). Введённые в 2014 году Янном Гудфеллоу и коллегами, эти сети состоят из двух противоборствующих компонентов: генератора, создающего синтетические примеры, и дискриминатора, выявляющего фальсификации. В процессе обучения генератор постепенно совершенствует свои образцы до тех пор, пока дискриминатор не сможет их отличить от реальных данных. GAN продемонстрировали великолепные результаты в создании фотореалистичных

изображений, текстур, а также нашли применение в задачах улучшения качества изображений и преобразования стилей.

Наконец, задачи обработки структурированных данных и сложных взаимосвязей между объектами привели к появлению графовых нейронных сетей (GNN). В отличие от классических сетей, рассчитанных на табличные или последовательные данные, GNN работают с произвольными графовыми структурами, позволяя учитывать как признаки объектов, так и их связи. Метод агрегирования информации от соседних узлов и обновления состояний каждой вершины открывает новые возможности в анализе социальных сетей, биомолекулярных структур, рекомендационных систем и многих других областях, где данные естественно представлены в виде графов.

Таким образом, архитектуры трансформеров, генеративно-состязательных сетей и графовых нейронных сетей образуют фундамент современного глубокого обучения, обеспечивая универсальные инструменты для решения широкого спектра задач.

# 1 ТРАНСФОРМЕР

## 1.1 Теоретический раздел

### 1.1.1 История появления архитектуры

Архитектура трансформера была представлена в июне 2017 года. В центре внимания первоначального исследования были задачи перевода. Затем было представлено несколько влиятельных моделей, в том числе:

- июнь 2018 года: GPT, первая предварительно обученная модель трансформера, использовалась для дообучения на различных задачах NLP и получила лучшие результаты;
- октябрь 2018 года: BERT, еще одна большая предварительно обученная модель, предназначенная для создания лучших резюме текстов;
- февраль 2019 года: GPT-2, улучшенная (и более крупная) версия GPT, которая не была сразу опубликована по этическим соображениям;
- октябрь 2019 года: DistilBERT, дистиллированная версия BERT, которая на 60% быстрее, на 40% легче и сохраняет 97% производительности BERT;
- октябрь 2019 года: BART и T5, две большие предварительно обученные модели, использующие ту же архитектуру, что и оригинальная модель трансформера;
- май 2020 года: GPT-3, еще более крупная версия GPT-2, способная хорошо справляться с различными задачами без необходимости в дообучении.

Этот список далеко не полный и призван лишь выделить несколько видов моделей трансформеров. В целом их можно разделить на три категории:

- GPT-подобные (также называемые авторегрессионными моделями



трансформеров);

- BERT-подобные (также называемые автокодирующими моделями трансформеров);
- BART/T5-подобные (также называемые моделями трансформации последовательности в последовательность).

### 1.1.2 Описание архитектуры

Все упомянутые выше модели трансформеров были обучены как языковые модели. Это означает, что они обучались на больших объемах необработанного текста в режиме саморегулируемого обучения. Саморегулируемое обучение — это парадигма машинного обучения, при которой модель обучается решению задачи, используя сами данные для генерации управляющих сигналов, а не полагаясь на метки, предоставленные извне. Это означает, что люди не нужны для разметки данных.

За исключением нескольких моделей, общая стратегия достижения лучшей производительности заключается в увеличении размеров моделей и объема данных, на которых они предварительно обучаются. На Рисунке 1.1.1 представлен график, отображающий рост числа параметров моделей со временем.

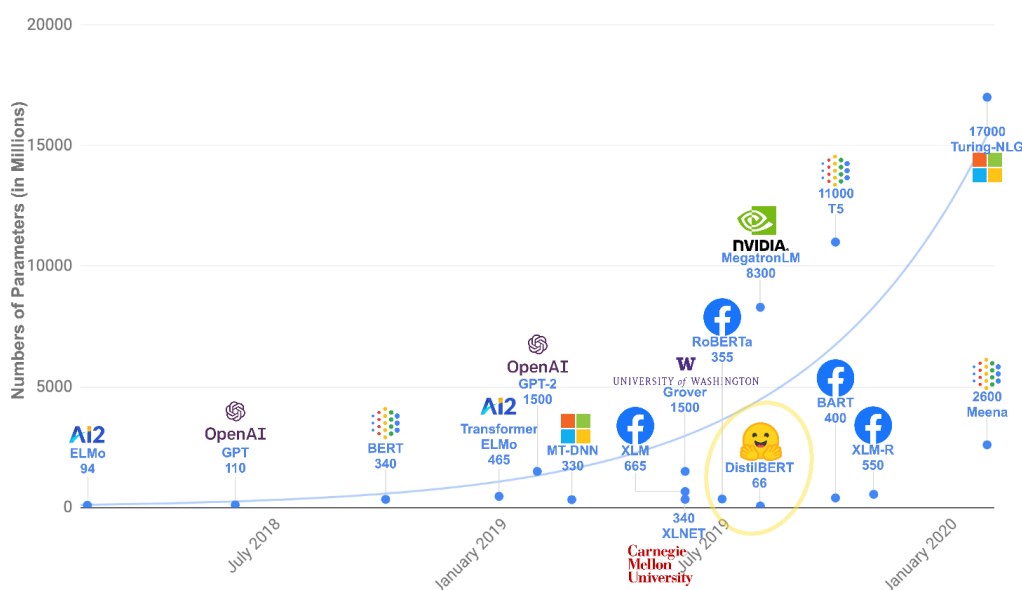
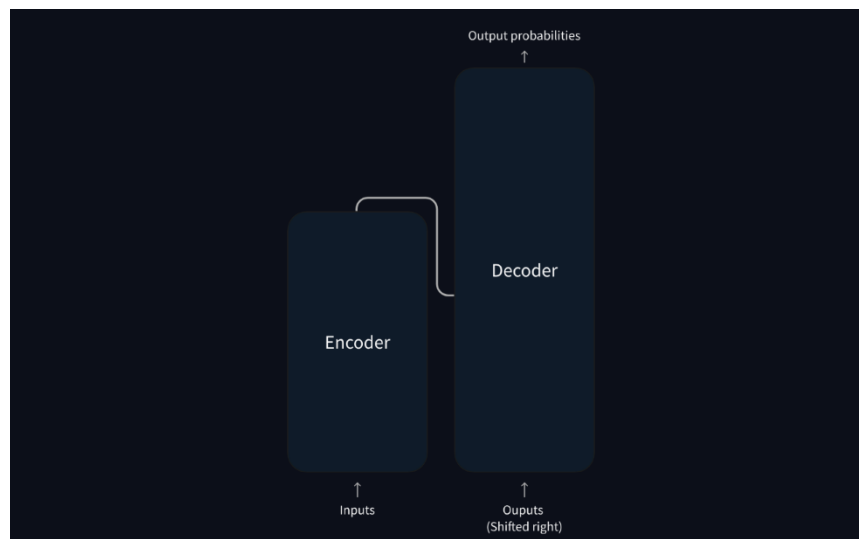


Рисунок 1.1.1 - Рост числа параметров моделей со временем

Идея архитектуры трансформера отображена на Рисунке 1.1.2.



**Рисунок 1.1.2 - Идея архитектуры трансформера**

Архитектура трансформера состоит из двух блоков:

- **кодировщик:** кодировщик получает входной сигнал и строит его представление (его признаки). Это означает, что модель оптимизирована для получения понимания от входных данных;
- **декодер:** декодер использует представление (признаки) кодера вместе с другими входными данными для создания целевой последовательности. Это означает, что модель оптимизирована для генерации выходных данных.

Каждая из этих частей может использоваться независимо, в зависимости от задачи:

- модели, включающие только кодировщик: хорошо подходят для задач, требующих понимания входных данных, таких как классификация предложений и распознавание именованных сущностей;
- модели, использующие только декодер: хорошо подходят для генеративных задач, таких как генерация текста;
- модели кодировщика-декодировщика или модели «последовательность-последовательность»: хорошо подходят для генеративных задач, требующих входных данных, таких как перевод

или обобщение.

Ключевой особенностью моделей трансформеров является то, что они строятся с помощью специальных слоев, называемых слоями внимания. Фактически, название статьи, представляющей архитектуру трансформера, было «Attention Is All You Need»!

Архитектура трансформера изначально была разработана для перевода. В процессе обучения кодер получает входные данные (предложения) на определенном языке, а декодер - те же предложения на нужном целевом языке. В кодировщике слои внимания могут использовать все слова в предложении (поскольку, перевод данного слова может зависеть от того, что находится как после, так и до него в предложении). Декодер, однако, работает последовательно и может обращать внимание только на те слова в предложении, которые он уже перевел (то есть только на те, которые предшествуют слову, генерируемому в данный момент). Оригинальная архитектура трансформера представлена на Рисунке 1.1.3.

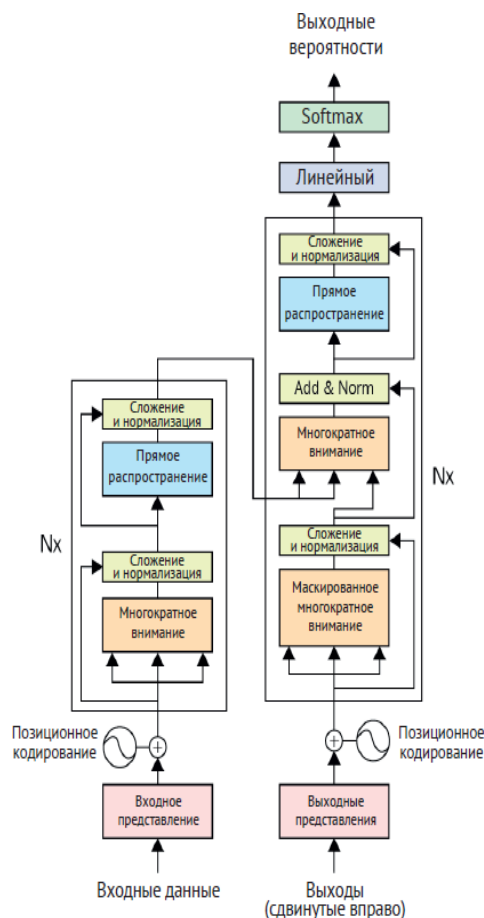


Рисунок 1.1.3 – Оригинальная архитектура трансформера

Модели кодировщиков используют только кодировщик модели трансформера. На каждом этапе уровни внимания могут обращаться ко всем словам в исходном предложении. Эти модели часто характеризуются как обладающие «двунаправленным» вниманием, и их часто называют моделями автокодирования. Предварительное обучение таких моделей обычно сводится к тому, чтобы каким-то образом испортить данное предложение (например, замаскировать в нем случайные слова) и поставить перед моделью задачу найти или восстановить исходное предложение. Модели-кодировщики лучше всего подходят для задач, требующих понимания полного предложения, таких как классификация предложений, распознавание именованных сущностей (и в более общем случае классификация слов) и экстрактивные ответы на вопросы.

Декодерные модели используют только декодер модели трансформера. На каждом этапе для данного слова слои внимания могут обращаться только к словам, расположенным перед ним в предложении. Такие модели часто называют авторегрессионными. Предварительное обучение моделей-декодеров обычно сводится к предсказанию следующего слова в предложении. Эти модели лучше всего подходят для задач, связанных с генерацией текста.

Модели кодировщика-декодировщика (также называемые моделями последовательности) используют обе части архитектуры трансформера. На каждом этапе слои внимания кодера получают доступ ко всем словам в исходном предложении, в то время как слои внимания декодера получают доступ только к словам, расположенным перед данным словом во входном сообщении. Предварительное обучение этих моделей может быть выполнено с использованием задач моделей кодировщика или декодировщика, но обычно для этого требуется нечто более сложное. Модели «последовательность-последовательность» лучше всего подходят для задач, связанных с генерацией новых предложений в зависимости от заданного исходного текста, таких как обобщение, перевод или генеративный ответ на вопрос.

Архитектура трансформера основана на механизме самовнимания, который позволяет модели анализировать взаимосвязи между всеми элементами

входной последовательности независимо от их расстояния друг от друга. Ниже приведены ключевые компоненты и математические основы архитектуры.

### 1.1.3 Применение трансформеров

Трансформеры стали основой для большинства современных NLP-решений. Среди ключевых задач:

- zero-shot classification – классификация неразмеченных данных;
- генерация текста;
- mask filling - задача предсказать правильное слово (точнее, лексему) в середине последовательности;
- named entity recognition — это задача в области обработки естественного языка (NLP), направленная на выделение и классификацию именованных сущностей в тексте, таких как имена людей, названия организаций, даты, местоположения, суммы денег и другие типы специфических объектов;
- ответы на вопросы (извлечение ответа из контекста);
- обобщение - задача сократить текст до более короткого, сохранив при этом все (или большинство) важных аспектов, упомянутых в тексте;
- машинный перевод.

Более того, трансформеры успешно адаптированы для работы с изображениями, преодолевая ограничения классических сверточных сетей:

- классификация изображений – модель Vision Transformer (ViT) разбивает изображение на патчи, обрабатывая их как последовательности, и достигает точности, сопоставимой с CNN.
- обнаружение объектов – архитектура DETR (Detection Transformer) использует трансформеры для прямого предсказания bounding box, исключая сложные постобработки.
- сегментация – модели типа Segmenter применяют механизм внимания для точного разделения объектов на пиксельном уровне.

- генерация изображений – трансформеры используются в моделях, таких как ImageGPT, для создания изображений на основе текстовых описаний.

## 1.2 Постановка задачи

Цель: реализовать обучение трансформера для решения задачи ответов на вопросы по заданному контексту.

Задачи: изучить архитектуру трансформера, выбрать данные для обучения и выполнить их предобработку, выбрать архитектуру для модели и обучить модель, интерпретировать полученные результаты.

## 1.3 Документация к данным

В качестве данных выбран датасет SberQuad (аналог SQuAD для русского языка), который загружается через Hugging Face Datasets. Структура датасета представлена на Рисунке 1.3.1.

```
Структура датасета:

DatasetDict({
  train: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 45328
  })
  validation: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 5036
  })
  test: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 23936
  })
})
Количество записей в train датасете: 45328
Количество записей в validation датасете: 5036
Количество записей в test датасете: 23936
Общее количество записей: 74300
```

Рисунок 1.3.1 – Структура датасета

Тренировочная выборка содержит 45328 записей, валидационная выборка 5,036 записей, тестовая выборка: 23,936 записей.

Пример записи представлен на Рисунке 1.3.2.

```
Пример точки данных из тренировочной выборки:
{
  "id": 62310,
  "title": "SherChallenge",
  "context": "В протерозойских отложениях органические остатки встречается намного чаще, чем в архейских. Они представлены известковыми выделениями синие-зелёных водорослей, хитином червей, остатками кишечнополостных. Кроме известковых водорослей, к числу древнейших",
  "question": "чем представлены органические остатки?",
  "answers": {
    "text": "известковыми выделениями синие-зелёных водорослей",
    "answer_start": 109
  }
}
id: 62310
title: SherChallenge
context: В протерозойских отложениях органические остатки встречается намного чаще, чем в архейских. Они представлены известковыми выделениями синие-зелёных водорослей, хитином червей, остатками кишечнополостных. Кроме известковых водорослей, к числу древнейших
question: чем представлены органические остатки?
answer: ['известковыми выделениями синие-зелёных водорослей']
answer_start: [109]
```

**Рисунок 1.3.2 – Пример записи датасета**

Описание полей датасета представлено в Таблице 1.3.1.

*Таблица 1.3.1 – Описание полей*

Поле	Тип	Описание
id	str	Уникальный идентификатор примера вопрос-ответ
title	str	Название документа или раздела, откуда взят контекст. Может использоваться для группировки или анализа по темам
context	str	Отрывок текста (контекст), в котором содержится ответ на вопрос
question	str	Вопрос, на который нужно ответить, используя контекст
answers	dict	Словарь, содержащий: text — список текстов ответов (обычно длиной 1), answer_start — список позиций начала ответа в context (соответствует text)

## 1.4 Обучение модели

Выбранная архитектура: DeepPavlov/rubert-base-cased - предобученная BERT-модель для русского языка. Особенности архитектуры:

- 12 слоев трансформера, 768 скрытых единиц, 12 голов внимания;
- обучена на корпусах Wikipedia, news и других русскоязычных текстов;
- поддерживает токенизацию с учетом морфологии русского языка;

- сохраняет регистр символов и изначально предназначена для задач обработки естественного языка на русском языке, включая классификацию, извлечение сущностей и задачу вопрос-ответ.

Выбранная модель оптимальна для QA-задач благодаря двунаправленному вниманию и способности работать с длинными контекстами.

Обучение проводится с использованием класса `Trainer` из библиотеки Hugging Face. Заданные гиперпараметры описаны в Таблице 1.4.1.

Таблица 1.4.1 – Выбранные гиперпараметры

Параметр	Значение	Описание
output_dir	./results	Директория для сохранения чекпойнтов и логов
learning_rate	2e-5	Скорость обучения. Типичное значение для BERT
per_device_train_batch_size	8	Размер батча на одно устройство
num_train_epochs	3	Количество эпох обучения
weight_decay	0.01	Коэффициент регуляризации (снижение переобучения)
eval_strategy	epoch	Оценка качества модели проводится после каждой эпохи
save_strategy	epoch	Сохранение модели также выполняется после каждой эпохи
logging_dir	./logs	Путь к директории для логов TensorBoard
fp16	True, если доступна CUDA	Использование 16-битных чисел с плавающей точкой (для ускорения и уменьшения памяти)

Процесс обучения запускается через метод `trainer.train()`, который:

1. Загружает и токенизирует датасет.
2. Производит обучение модели по токенизированным данным.
3. Выполняет оценку на валидационном наборе.
4. Сохраняет модель и токенизатор в директорию `./sberquad_qa_model`.

Обучение осуществляется следующим образом: сначала входные тексты — пары «вопрос-контекст» — токенизируются с помощью предварительно обученного токенизатора `DeepPavlov/rubert-base-cased`. В результате текст



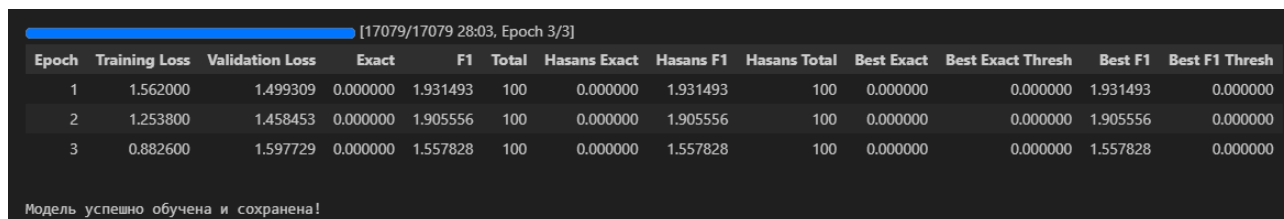
превращается в последовательности числовых идентификаторов токенов, а также создаются сопутствующие маски внимания и маркеры сегментов, отличающие вопрос от контекста. Токенизированные данные дополнительно аннотируются позициями начала и конца ответа в пределах контекста, что позволяет модели в дальнейшем учиться определять границы ответа в тексте.

Затем эти подготовленные входы подаются в сам трансформер. Архитектура модели представляет собой многоуровневый энкодер, где каждый слой состоит из механизма многоголового самовнимания и последующего позиционно-независимого полносвязного слоя. После прохождения всех слоёв трансформера на выходе формируются скрытые представления всех токенов последовательности, которые далее обрабатываются двумя отдельными линейными слоями. Эти слои вычисляют вероятности начала и конца ответа на основе каждого токена в контексте. Таким образом, модель получает два распределения: по позициям старта и по позициям окончания ответа.

Далее происходит вычисление функции потерь — суммарной кросс-энтропии между предсказанными и истинными позициями начала и конца ответа. Затем, благодаря механизму обратного распространения ошибки, градиенты ошибки передаются от выходных логитов через линейные и трансформерные слои назад, к весам модели. Оптимизатор использует эти градиенты для обновления параметров модели с целью минимизации потерь. Этот процесс повторяется для каждого батча в обучающем наборе и продолжается в течение заданного количества эпох.

На каждом этапе обучения производится периодическая оценка качества модели на отложенной валидационной выборке. Модель предсказывает ответы на вопросы, используя свои текущие параметры, а затем предсказанные ответы сравниваются с реальными с помощью метрик точного совпадения (exact match) и F1-меры. Эти метрики помогают отслеживать прогресс и предотвращают переобучение. После завершения всего процесса обучения модель вместе с её токенизатором сохраняется на диск для последующего использования в inference-режиме — при ответах на новые вопросы.

Процесс обучения представлен на Рисунке 1.4.1.



Epoch	Training Loss	Validation Loss	Exact	F1	Total	Hasans Exact	Hasans F1	Hasans Total	Best Exact	Best Exact Thresh	Best F1	Best F1 Thresh
1	1.562000	1.499309	0.000000	1.931493	100	0.000000	1.931493	100	0.000000	0.000000	1.931493	0.000000
2	1.253800	1.458453	0.000000	1.905556	100	0.000000	1.905556	100	0.000000	0.000000	1.905556	0.000000
3	0.882600	1.597729	0.000000	1.557828	100	0.000000	1.557828	100	0.000000	0.000000	1.557828	0.000000

Модель успешно обучена и сохранена!

**Рисунок 1.4.1 – Обучение модели**

Модель прошла все 3 эпохи обучения и была успешно сохранена.

Потери на обучении последовательно снижаются. Training Loss по эпохам принимает значения [1.562000; 1.253800; 0.882600], что говорит о том, что модель с каждой эпохой всё лучше подстраивается под обучающую выборку.

Потери на валидации не демонстрируют стабильного улучшения. Validation Loss принимает значения на эпохах обучения [1.499309; 1.458453; 1.597729]. Хотя на второй эпохе наблюдается небольшое снижение, на третьей валидационные потери возросли. Это может быть признаком начавшегося переобучения, когда модель запоминает тренировочные примеры, но теряет обобщающую способность.

## 1.5 Полученные результаты

Метрики качества остаются низкими в процессе обучения, что говорит о том, что данные могут быть неправильно размечены, проведено недостаточно эпох обучения, обучение проходит неправильно.

Для оценки качества модели используются две метрики:

- Exact Match (ЕМ, точное совпадение) проверяет, полностью ли предсказанный ответ в точности совпадает с правильным ответом, игнорируя пунктуацию, регистр и пробелы;
- F1 — гармоническое среднее между точностью (precision) и полнотой (recall) на уровне токенов.

На Рисунке 1.5.1 представлен ответ модели на вопрос, который не относится к датасету.

## Пример использования модели

```

qa_system = QAPipeline()

context = "Автокодировщиком называется нейронная сеть, обученная пытаться скопировать свой вход в выход"
question = "Что такое автокодировщик?"

result = qa_system.predict(context, question)
print(f"Результирующий словарь: {result}")
print("Результат предсказания:")
print(f"Вопрос: {question}")
print(f"Ответ: {result['answer']}")
print(f"Точность: {result['score']:.2f}")

```

15]

```

.. Device set to use cuda:0
Результирующий словарь: {'score': 0.8621006608009338, 'start': 28, 'end': 42, 'answer': 'нейронная сеть'}
Результат предсказания:
Вопрос: Что такое автокодировщик?
Ответ: нейронная сеть
Точность: 0.86

```

Рисунок 1.5.1 – Пример использования обученной модели

Модель вернула осмысленный, хоть и не полный ответ. Возвращаемый результат включает score - уверенность модели в правильности ответа, start, end – позиции начала и конца ответа в контексте, answer – строковый ответ.

Протестируем модель на тестовой выборке. Возьмем пять примеров и оценим ответы модели. Сведем информацию в Таблицу 1.5.1.

Таблица 1.5.1 – Предсказания модели на тестовой выборке

Вопрос	Контекст	Предсказание	Точность
У каких организмов отсутствуют настоящие дифференцированные клетки?	Многоклеточный организм — внесистематическая категория живых организмов, тело которых состоит из многих клеток, большая часть которых (кроме стволовых, например, клеток камбия у растений) дифференцированы, то есть различаются по строению и выполняемым функциям. Следует отличать многоклеточность и колониальность. У колониальных организмов отсутствуют настоящие дифференцированные клетки, а следовательно, и разделение тела на ткани. Граница между многоклеточностью и колониальностью нечёткая. Например, вольвокс часто относят к колониальным организмам, хотя в его колониях есть чёткое деление клеток на генеративные и соматические. Кроме дифференциации клеток, для многоклеточных характерен и более высокий уровень интеграции, чем для колониальных форм. Многоклеточные животные, возможно, появились на Земле 2,1 миллиарда лет назад, вскоре после кислородной революции	У колониальных организмов	0.35
Какие животные появились на Земле 2,1 миллиарда лет назад?	Многоклеточные животные, возможно, появились на Земле 2,1 миллиарда лет назад, вскоре после кислородной революции	Многоклеточные	0.63
Когда предположительно появились многоклеточные животные?	Многоклеточные животные, возможно, появились на Земле 2,1 миллиарда лет назад, вскоре после кислородной революции	2,1 миллиарда лет назад	0.17

## **1.6 Выводы по разделу**

В ходе выполнения работы построена и обучена модель трансформера для ответов на вопросы по контексту. Выяснено, что данная архитектура является хорошим инструментом для задач NLP, однако требует существенных ресурсов и времени для обучения.

## **2 ГЕНЕРАТИВНО-СОСТЯЗАТЕЛЬНАЯ СЕТЬ**

### **2.1 Теоретический раздел**

#### **2.1.1 История появления архитектуры**

Генеративно-состязательные сети (GAN, Generative Adversarial Networks) были впервые представлены в 2014 году группой исследователей под руководством Яна Гудфеллоу в статье «Generative Adversarial Nets». Эта работа стала революционной в области генеративного моделирования, предложив принципиально новый подход к созданию синтетических данных. До появления GAN доминирующими методами были вариационные автоэнкодеры и авторегрессивные модели, которые, однако, имели ограничения в качестве и разнообразии генерируемых данных.

Идея GAN возникла на стыке машинного обучения и теории игр. Авторы вдохновлялись концепцией минимаксной игры, где две модели соревнуются друг с другом: генератор создает данные, а дискриминатор оценивает их правдоподобие. Такой подход позволил избежать явного моделирования сложных распределений данных, что было ключевой проблемой предыдущих методов.

С момента публикации оригинальной статьи GAN быстро набрал популярность. Уже в первые годы появились модификации, такие как DCGAN (Deep Convolutional GAN), которые адаптировали сверточные слои для генерации изображений, и WGAN (Wasserstein GAN), решившие проблему нестабильности обучения. Эти разработки заложили основу для применения GAN в компьютерном зрении, обработке естественного языка и других областях.

## 2.1.2 Описание архитектуры

Генеративно-состязательная сеть (GAN) состоит из двух частей:

- генератор учится генерировать правдоподобные данные. Сгенерированные экземпляры становятся отрицательными обучающими примерами для дискриминатора;
- дискриминатор учится отличать поддельные данные генератора от реальных данных. Дискриминатор наказывает генератор за получение неправдоподобных результатов.

Когда начинается обучение, генератор выдает явно фальшивые данные, и дискриминатор быстро учится определять, что это фейковые данные. По мере обучения генератор приближается к выдаче выходных данных, которые могут обмануть дискриминатор. Наконец, если обучение генератора проходит хорошо, дискриминатор становится хуже отличать настоящее от поддельного. Он начинает классифицировать фейковые данные как настоящие, и их точность снижается. Принцип работы архитектуры представлен на Рисунке 2.1.1.

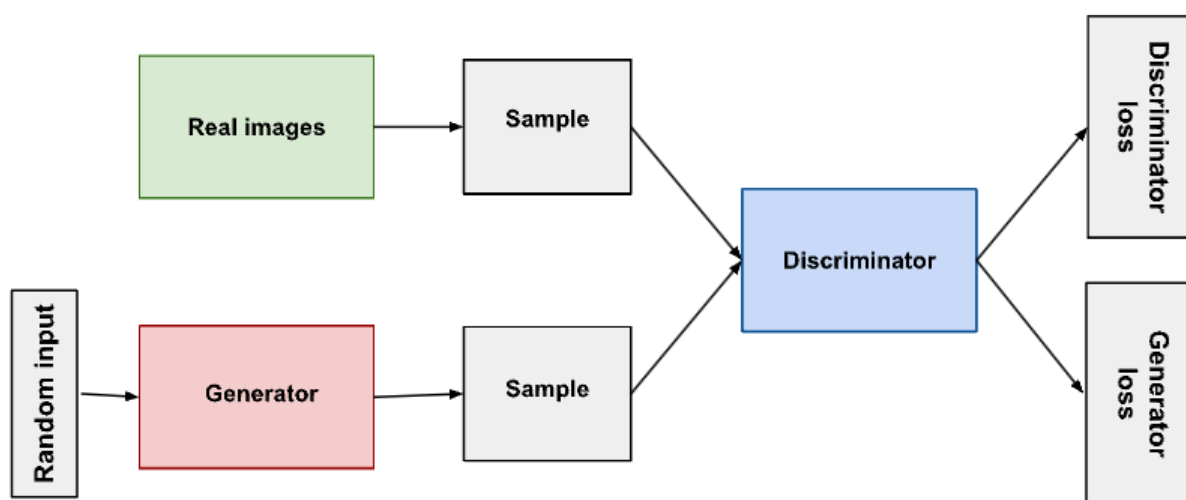


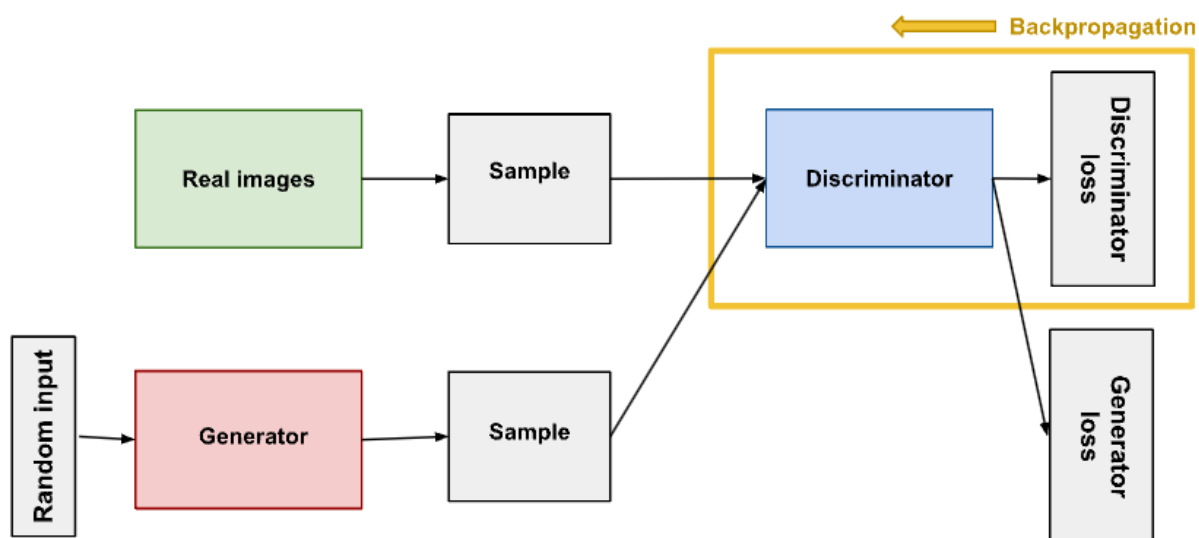
Рисунок 2.1.1 – Принцип работы GAN

Дискриминатор в GAN — это просто классификатор. Он пытается отличить реальные данные от данных, созданных генератором. Он может использовать любую сетевую архитектуру, соответствующую типу данных, которые он классифицирует.

Данные обучения дискриминатора поступают из двух источников:

- реальные экземпляры данных, например реальные фотографии людей. Дискриминатор использует эти случаи как положительные примеры во время обучения;
- поддельные экземпляры данных, созданные генератором. Дискриминатор использует эти случаи как отрицательные примеры во время обучения.

На Рисунке 2.1.1 два поля «Sample» представляют эти два источника данных, поступающих в дискриминатор. Во время обучения дискриминатора генератор не обучается. Его веса остаются постоянными, пока он создает примеры для обучения дискриминатора. Процесс обучения дискриминатора представлен на Рисунке 2.1.2.



**Рисунок 2.1.2 - Обратное распространение ошибки при обучении дискриминатора**

Дискриминатор подключается к двум функциям потерь. Во время обучения дискриминатора дискриминатор игнорирует потери генератора и просто использует потери дискриминатора. Обучение дискриминатора включает следующие шаги:

1. Дискриминатор классифицирует как реальные данные, так и поддельные данные от генератора.
2. Потеря дискриминатора наказывает дискриминатор за ошибочную классификацию реального экземпляра как поддельного или поддельного

экземпляра как настоящего.

3. Дискриминатор обновляет свои веса посредством обратного распространения ошибки дискриминатора через сеть дискриминатора.

Генераторная часть GAN учится создавать фальшивые данные, учитывая обратную связь от дискриминатора. Он учится заставлять дискриминатор классифицировать его выходные данные как реальные. Обучение генератора требует более тесной интеграции между генератором и дискриминатором, чем требует обучение дискриминатора. Часть GAN, которая обучает генератор, включает в себя:

- случайный ввод;
- сеть генератора, которая преобразует случайные входные данные в экземпляр данных;
- сеть дискриминатора, которая классифицирует сгенерированные данные;
- выход дискриминатора;
- потери генератора, которые наказывают генератор за то, что он не смог обмануть дискриминатор.

Процесс обучение генератора представлен на Рисунке 2.1.3.

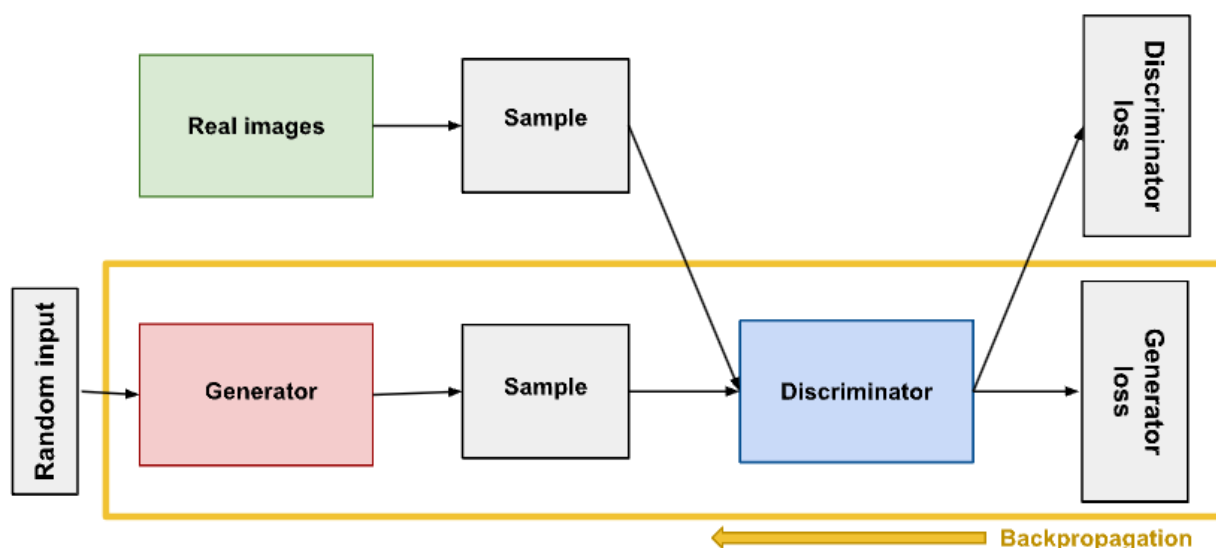


Рисунок 2.1.3 - Обратное распространение ошибки при обучении генератора

В своей самой простой форме GAN принимает на вход случайный шум. Затем генератор преобразует этот шум в значимый результат. Введя шум, GAN



может генерировать самые разнообразные данные, производя выборку из разных мест целевого распределения.

Эксперименты показывают, что распределение шума не имеет большого значения, поэтому мы можем выбрать что-то, из чего легко производить выборку, например равномерное распределение. Для удобства пространство, из которого производится выборка шума, обычно имеет меньшую размерность, чем размерность выходного пространства.

Чтобы обучить нейронную сеть, необходимо корректировать ее веса, чтобы уменьшить ошибку или потерю выходных данных. Однако в архитектуре GAN генератор не связан напрямую с потерями, на которые мы пытаемся повлиять. Генератор подает сигнал в сеть дискриминатора, и *дискриминатор* выдает результат, на который требуется повлиять. Потеря генератора наказывает генератор за создание образца, который сеть дискриминатора классифицирует как поддельный.

Этот дополнительный участок сети должен быть включен в обратное распространение ошибки. Обратное распространение корректирует каждый вес в правильном направлении, вычисляя влияние веса на выходные данные — как изменится результат, если изменить вес. Но влияние веса генератора зависит от влияния весов дискриминатора, в которые он подается. Таким образом, обратное распространение начинается на выходе и возвращается через дискриминатор в генератор.

В то же время дискриминатор не должен меняться во время обучения генератора. Попытка поразить движущуюся цель еще больше усложнит задачу генератора. Итак, обучение генератора состоит из следующих этапов:

1. Пример случайного шума.
2. Воспроизведение выходного сигнала генератора из выборочного случайного шума.
3. Получение классификации дискриминатора «Настоящая» или «Поддельная» для выхода генератора.
4. Расчет потери от классификации дискриминатора.

5. Обратное распространение ошибки через дискриминатор и генератор для получения градиентов.
6. Использование градиентов, чтобы изменить только веса генератора.

Обучение GAN протекает в чередующиеся периоды, так как генератор и дискриминатор имеют разные процессы обучения:

1. Дискриминатор обучается в течение одной или нескольких эпох.
2. Генератор тренируется в течение одной или нескольких эпох.
3. Шаги 1 и 2 повторяются, чтобы продолжить обучение сетей генератора и дискриминатора.

Генератор сохраняется постоянным на этапе обучения дискриминатора. Поскольку обучение дискриминатора пытается выяснить, как отличить настоящие данные от поддельных, оно должно научиться распознавать недостатки генератора. Это другая проблема для тщательно обученного генератора, чем для необученного генератора, выдающего случайный результат.

Дискриминатор сохраняется постоянным на этапе обучения генератора. В противном случае генератор будет пытаться поразить движущуюся цель и никогда не сойдется.

По мере того, как генератор улучшается по мере обучения, производительность дискриминатора ухудшается, поскольку дискриминатор не может легко отличить настоящее от поддельного. Если генератор работает идеально, то точность дискриминатора составляет 50%. По сути, дискриминатор подбрасывает монету, чтобы сделать прогноз.

Такое развитие событий создает проблему для конвергенции GAN в целом: обратная связь дискриминатора со временем становится менее значимой. Если GAN продолжит обучение после того момента, когда дискриминатор дает совершенно случайную обратную связь, то генератор начнет тренироваться на нежелательной обратной связи, и его собственное качество может ухудшиться.

GAN пытаются воспроизвести распределение вероятностей. Поэтому им следует использовать функции потерь, которые отражают расстояние между

распределением данных, сгенерированных GAN, и распределением реальных данных.

В статье, посвященной GAN, генератор пытается минимизировать следующую функцию, а дискриминатор пытается ее максимизировать (Формула 2.1.1).

$$E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))], \quad (2.1.1)$$

где  $D(x)$  – оценка дискриминатора вероятности того, что реальный экземпляр данных  $x$  является реальным;

$E_x$  – ожидаемое значение для всех экземпляров реальных данных;

$G(z)$  – выходной сигнал генератора при заданном шуме  $z$ ;

$D(G(z))$  – оценка дискриминатора вероятности того, что поддельный экземпляр является реальным;

$E_z$  – ожидаемое значение для всех случайных входных данных генератора (фактически, ожидаемое значение для всех сгенерированных поддельных экземпляров  $G(z)$ ).

Генератор не может напрямую влиять на член  $\log(D(x))$  в функции, поэтому для генератора минимизация потерь эквивалентна минимизации  $\log(1 - D(G(z)))$ .

## 2.2 Постановка задачи

Цель: реализовать обучение генеративно-сопоставительной сети для генерации новых точек данных.

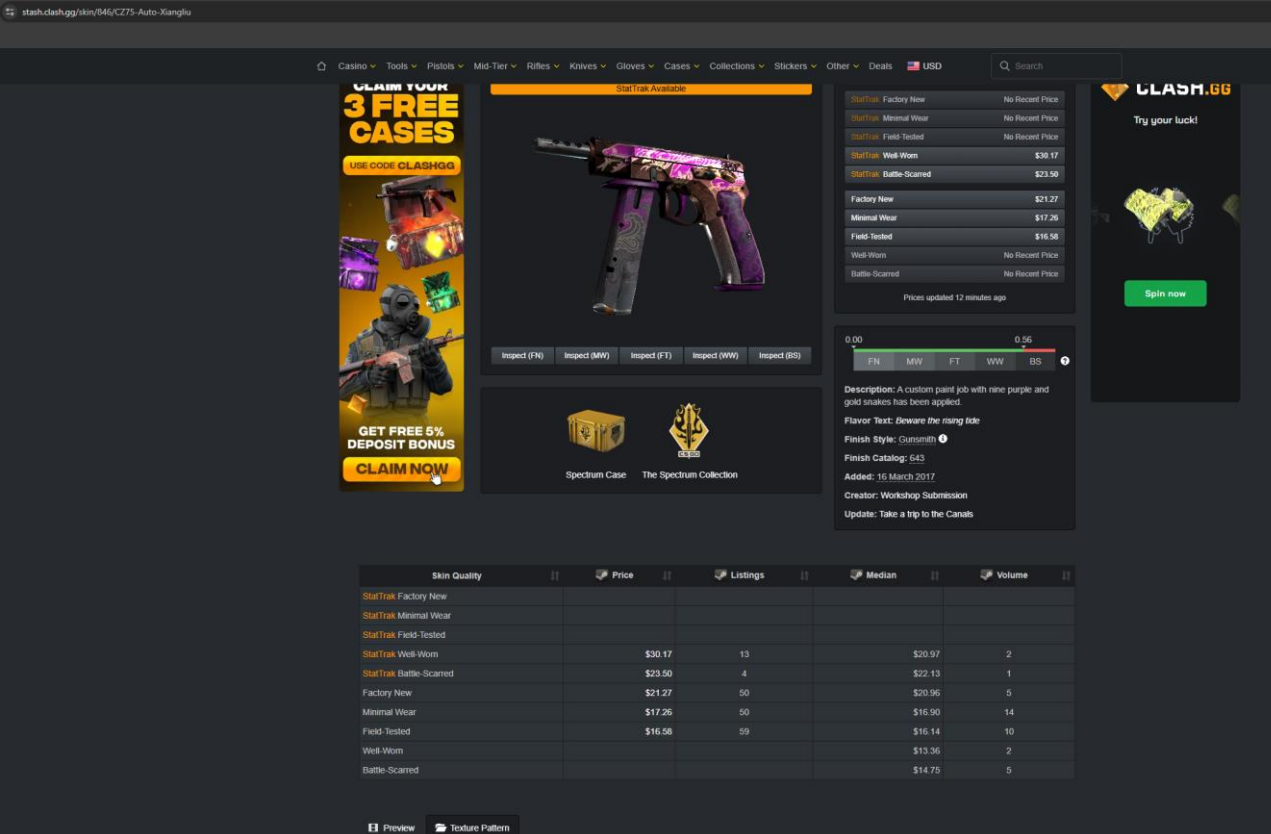
Задачи: изучить устройство генеративно-сопоставительных нейронных сетей (GAN), выбрать данные для обучения и выполнить их предобработку, обучить составленную GAN, интерпретировать полученные результаты.

## 2.3 Документация к данным

В качестве набора данных для обучения генеративно-сопоставительной сети возьмем текстуры скинов из игры Counter-Strike 2.

В открытом доступе найти текстуры не удалось, поэтому принято решение написать парсер, который соберет все существующие текстуры и сохранит локально в созданную директорию.

Ссылка на ресурс, который предоставляет подробную информацию о внутриигровых скинах Counter-Strike 2, включая текстуры, вынесена в список информационных источников. На Рисунке 2.3.1 отображена карточка отдельного взятого предмета (скина), которая содержит информацию о названии предмета, его стоимости в разных качествах, предпросмотр в игре и нужную текстуру (раздел Texture Pattern).



Skin Quality	Price	Listings	Median	Volume
StatTrak Factory New				
StatTrak Minimal Wear				
StatTrak Field-Tested				
StatTrak Well-Worn	\$30.17	13	\$20.97	2
StatTrak Battle-Scarred	\$23.50	4	\$22.13	1
Factory New	\$21.27	50	\$20.96	5
Minimal Wear	\$17.26	50	\$16.90	14
Field-Tested	\$16.58	59	\$16.14	10
Well-Worn			\$13.36	2
Battle-Scarred			\$14.75	5

Рисунок 2.3.1 – Карточка отдельного скина

Для написания парсера выбран фреймворк Selenium, который используется для автоматического тестирования веб-приложений и автоматизации действий в браузере. Код парсера представлен в Приложении Б.1.

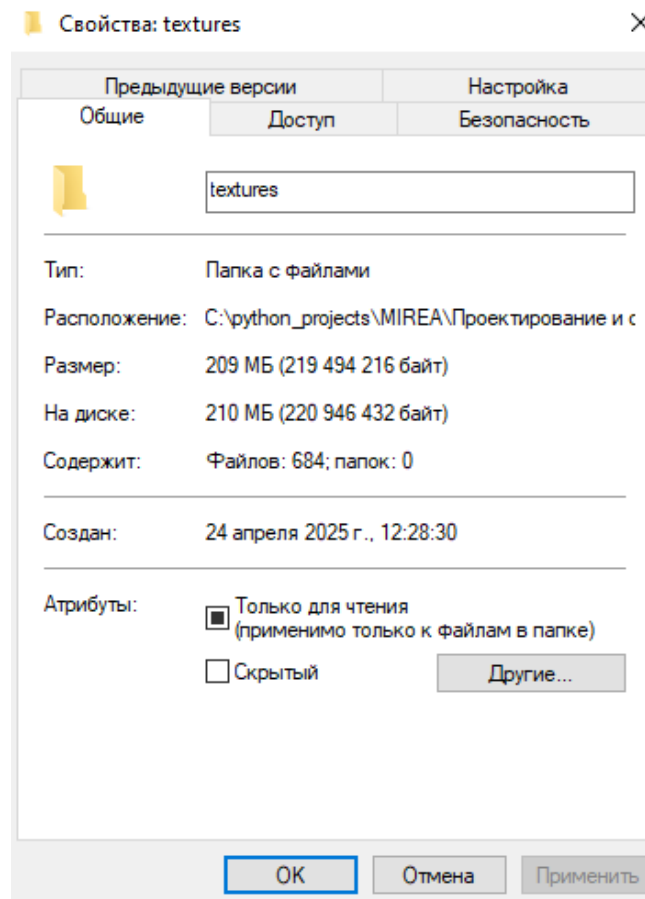
Отметим, что на сайте загружены текстуры не для всех скинов, поэтому необходимо выводить логи о результатах парсинга каждой карточки скина, чтобы отслеживать возможные ошибки при извлечении данных (Рисунок 2.3.2).

```
(.venv) PS C:\python_projects\Parsers\ParserCs2SkinsTextures> python textures.py
ERROR, msg=texture not found, page=https://stash.clash.gg/skin/1584/Zeus-x27-Olympus
ERROR, msg=texture not found, page=https://stash.clash.gg/skin/1656/Zeus-x27-Dragon-Snore
ERROR, msg=texture not found, page=https://stash.clash.gg/skin/1789/Zeus-x27-Charged-Up
ERROR, msg=texture not found, page=https://stash.clash.gg/skin/1683/Zeus-x27-Tosai
ERROR, msg=texture not found, page=https://stash.clash.gg/skin/1706/Zeus-x27-Electric-Blue
ERROR, msg=texture not found, page=https://stash.clash.gg/skin/1722/Zeus-x27-Swamp-DDPAT
OK, texture saved: https://stash.clash.gg/skin/262/CZ75-Auto-Victoria
OK, page=https://stash.clash.gg/skin/262/CZ75-Auto-Victoria
OK, texture saved: https://stash.clash.gg/skin/846/CZ75-Auto-Xiangliu
OK, page=https://stash.clash.gg/skin/846/CZ75-Auto-Xiangliu
OK, texture saved: https://stash.clash.gg/skin/572/CZ75-Auto-Yellow-Jacket
OK, page=https://stash.clash.gg/skin/572/CZ75-Auto-Yellow-Jacket
ERROR, msg=texture not found, page=https://stash.clash.gg/skin/260/CZ75-Auto-The-Fuschia-Is-Now
OK, texture saved: https://stash.clash.gg/skin/938/CZ75-Auto-Eco
OK, page=https://stash.clash.gg/skin/938/CZ75-Auto-Eco
OK, texture saved: https://stash.clash.gg/skin/906/CZ75-Auto-Tacticat
OK, page=https://stash.clash.gg/skin/906/CZ75-Auto-Tacticat
OK, texture saved: https://stash.clash.gg/skin/737/CZ75-Auto-Red-Astor
OK, page=https://stash.clash.gg/skin/737/CZ75-Auto-Red-Astor
█
```

Рисунок 2.3.2 – Логи при парсинге

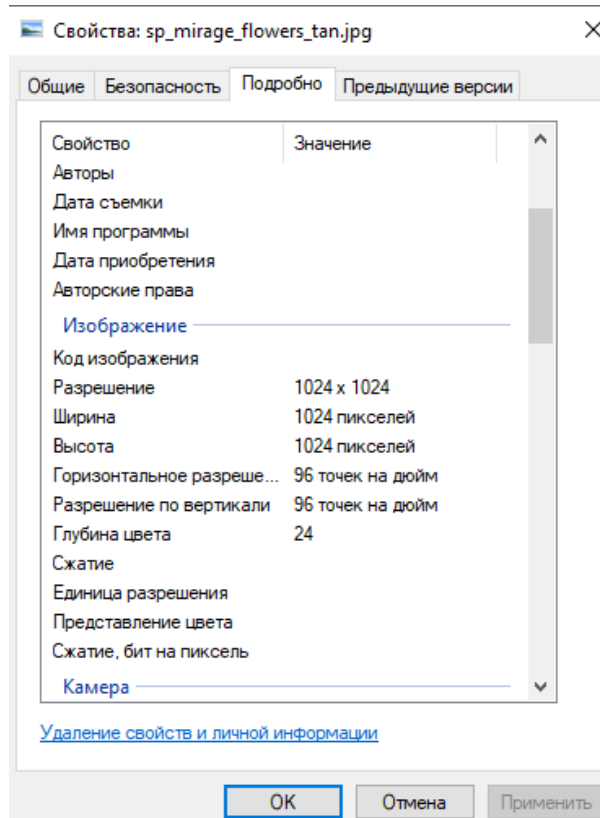
Парсинг всех карточек и загрузка изображений заняла примерно 15 минут. Итоговый датасет содержит 800 изображений текстур скинов. При внимательном просмотре загруженных текстур можно заметить, что текстуры повторяются. Это связано с тем, что несколько оружий могут быть реализованы с одним скинов.

Произведена предобработка данных: удалены повторяющиеся текстуры, удалены текстуры с разрешением, не равным 1024\*1024 пикселей (основная часть текстур загружена в таком разрешении). Предобработанный датасет содержит 684 текстуры (Рисунок 2.3.3).



**Рисунок 2.3.3 – Предобработанный датасет**

Свойства отдельно взятого изображения показаны на Рисунке 2.3.4.



**Рисунок 2.3.4 – Свойства отдельно взятого изображения**



Примеры текстур из датасета представлены на Рисунках 2.3.5–2.3.8.



Рисунок 2.3.5 – Текстура sp\_mirage\_flowers\_tan

Данная текстура симметрична относительно центра изображения и содержит повторяющиеся геометрические узоры, выполненные в ярких цветах.

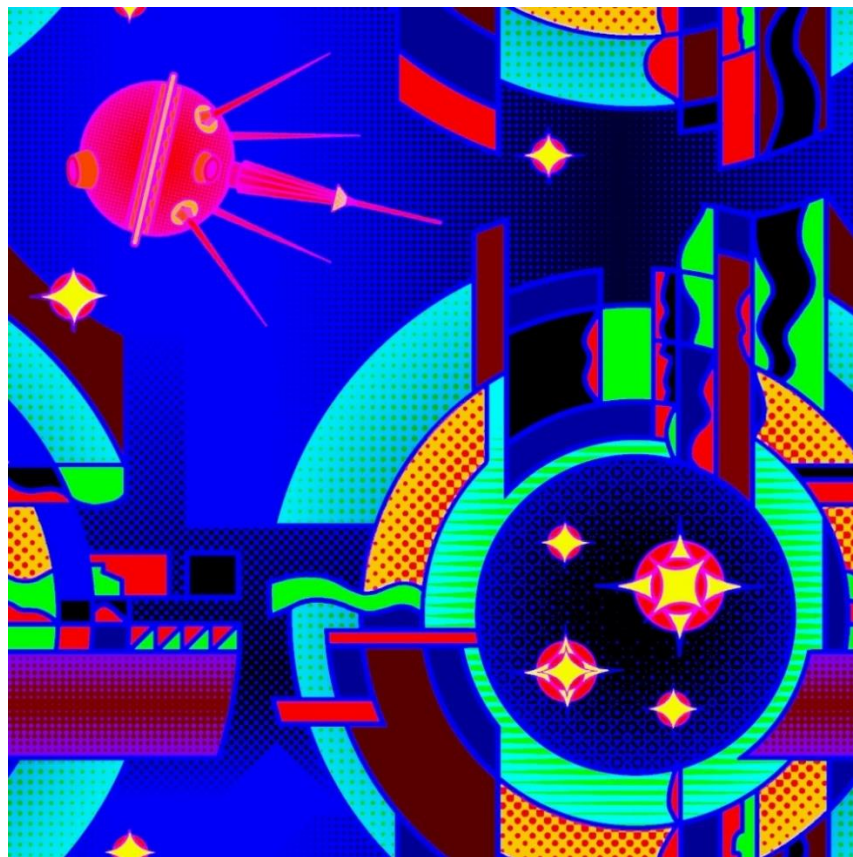


Рисунок 2.3.6 – Текстура aa\_spacespace\_orange



Это изображение выглядит как абстрактная текстура с яркими, неоновыми цветами, что создаёт эффект 3D-пространства. На изображении можно увидеть несколько геометрических фигур, таких как круги, прямоугольники и полосы, выполненные в контрастных цветах — синим, зелёным, красным, жёлтым и фиолетовым. В верхней части изображены звезды, а также нечто напоминающее спутник с лучами, что придаёт изображению космическую тематику. В целом, текстура выглядит ярко и насыщенно, с элементами поп-арта, создающими эффект визуальной динамики и движения.

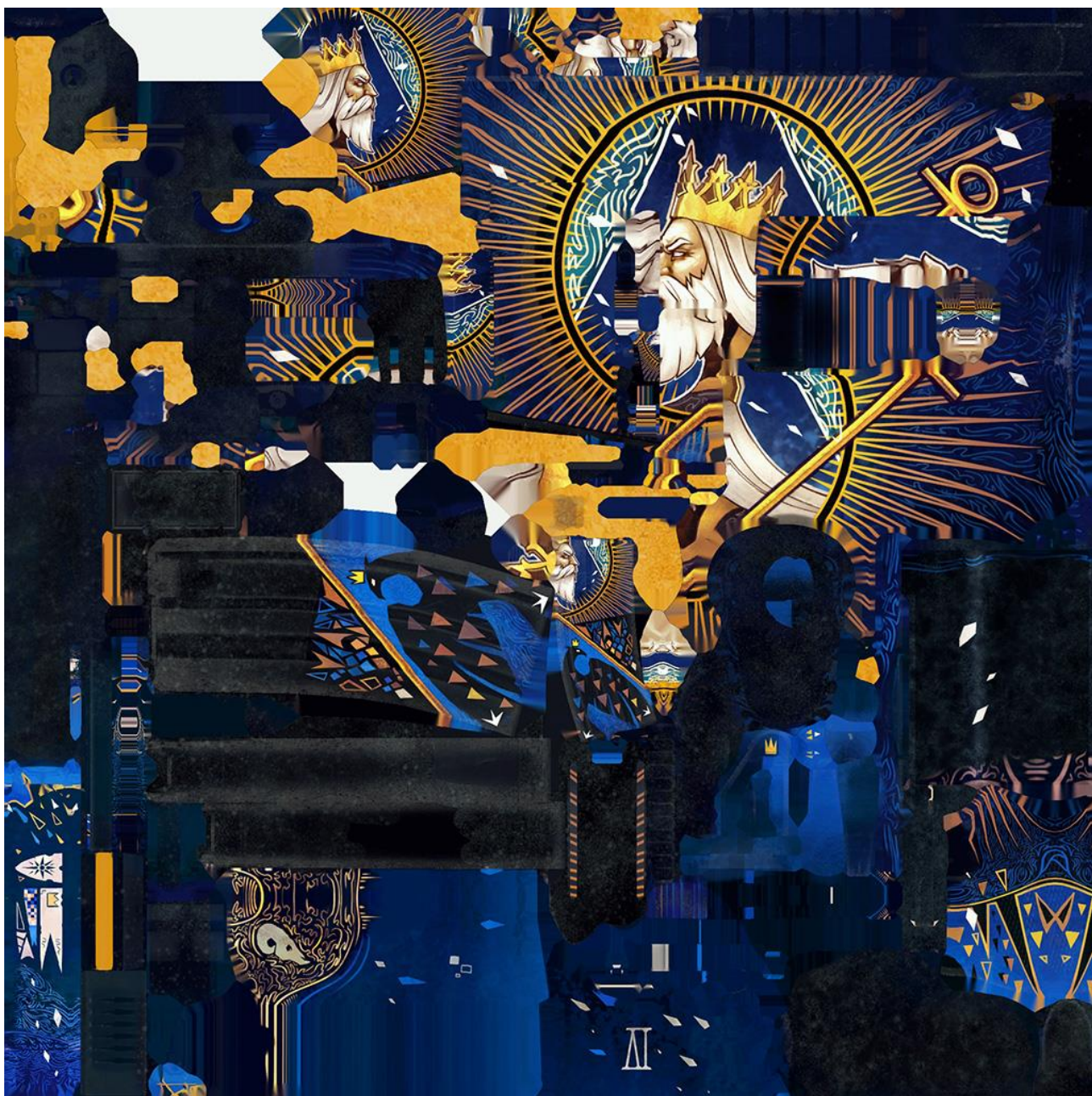


Рисунок 2.3.7 – Текстура gs\_m4a4\_emperor



Это изображение представляет собой текстуру с элементами старинного арта и фэнтезийной тематики. В центре изображён величественный король с длинной бородой и золотой короной, окружённый солнечными лучами. Элементы текста и орнамента (синие и золотые) сопровождают изображение. Текстура имеет довольно сильные абстракционные участки, которые создают эффект повреждённости или деформации изображения. Видны фрагменты в виде геометрических форм и темных пятен, что придаёт эффект разрушенной или старинной карты.

Цветовая палитра преимущественно включает темно-синие и золотистые тона, с яркими контрастами, что усиливает драматичность изображения.



Рисунок 2.3.8 – Текстура `su_money_glock`

На изображении показано множество стодолларовых купюр, разбросанных беспорядочно.

## 2.4 Обучение модели

Генератор принимает на вход вектор случайного шума из латентного пространства размерностью 128. На первом этапе вектор преобразуется в плоское представление с помощью полносвязного слоя Linear, расширяющего его в тензор размерности  $512 \times 4 \times 4$ . Далее генератор последовательно увеличивает пространственное разрешение изображения через серию блоков апсемплинга. Каждый блок состоит из билинейного апсемплинга изображения в два раза, сверточного слоя с ядром размером  $1 \times 1$  для изменения числа каналов, нормализации признаков с помощью InstanceNorm и функции активации ReLU. Затем применяется сверточный слой с ядром  $3 \times 3$ , нормализация и повторная активация. Последовательное применение семи таких блоков позволяет увеличить разрешение изображения с  $4 \times 4$  до  $512 \times 512$  пикселей. После последнего блока изображение дополнительно увеличивается до  $1024 \times 1024$  за счет апсемплинга, после чего применяется сверточный слой с выходными тремя каналами (RGB) и активацией Tanh, обеспечивающей значения пикселей в диапазоне  $[-1; 1]$ .

Дискриминатор получает на вход изображение размером  $1024 \times 1024$  и последовательно понижает его пространственное разрешение через сверточные блоки. Каждый блок включает сверточный слой с ядром  $4 \times 4$ , страйдом 2 и паддингом 1, функцию активации LeakyReLU с отрицательным наклоном 0.2, а также слой регуляризации Dropout для предотвращения переобучения. После прохождения четырех блоков разрешение уменьшается до  $64 \times 64$ , и применяется адаптивный усредняющий пуллинг, приводящий тензор к размеру  $1 \times 1$ . Затем данные передаются через полносвязный слой и активируются с помощью сигмоидной функции, дающей на выходе вероятность того, что входное изображение является реальным.

Перед началом обучения веса всех слоев генератора и дискриминатора были инициализированы согласно нормальному распределению с нулевым средним и стандартным отклонением 0.02, что позволяет ускорить и

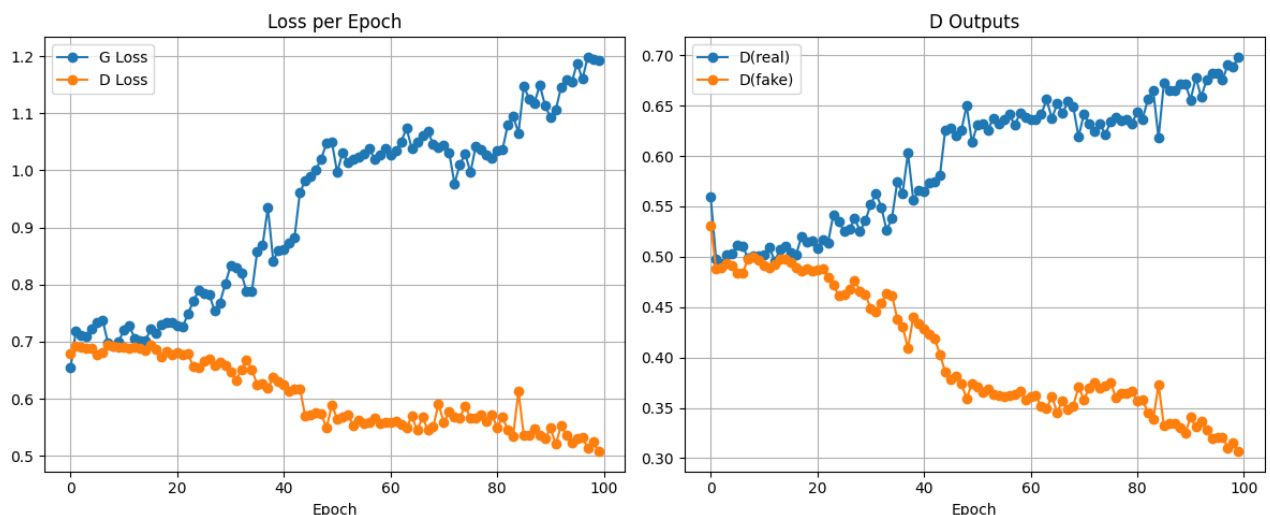
стабилизировать процесс сходимости модели. Для оптимизации использовались отдельные экземпляры алгоритма Adam для каждой из сетей. Скорость обучения генератора была установлена на уровне  $2 * 10^{-4}$ , а дискриминатора —  $1 * 10^{-4}$ . Использование разных скоростей обучения обусловлено необходимостью предотвратить доминирование дискриминатора над генератором в начальных этапах обучения.

В качестве функции потерь применялась бинарная кросс-энтропия (BCELoss), которая сравнивает выходы дискриминатора с целевыми метками (реальное или сгенерированное изображение).

Обучение модели велось в течение заранее 100 эпох, при этом на каждой итерации производился расчет и сохранение значений функции потерь генератора и дискриминатора, а также вероятностей, предсказанных дискриминатором для реальных и синтетических изображений.

Накопленные метрики позволяли анализировать динамику процесса обучения и своевременно выявлять признаки нестабильности, такие как коллапс модели или перетренированность дискриминатора.

Обучение в течение ста эпох заняло около одного часа. На Рисунке 2.4.1 представлены метрики, иллюстрирующие процесс обучения модели.



**Рисунок 2.4.1 – Метрики обучения**

Функция потерь генератора (G loss) отражает, насколько хорошо генератор обманывает дискриминатор. Чем ниже значение G loss, тем успешнее генератор

генерирует изображения, которые дискриминатор воспринимает как настоящие. В свою очередь, функция потерь дискриминатора (D loss) показывает, насколько точно дискриминатор различает реальные и синтетические изображения.

В ходе обучения было зафиксировано постепенное увеличение G loss с 0.5 до 1.2 и снижение D loss с 0.7 до 0.5 за первые 100 итераций. Рост G loss указывает на то, что с течением времени генератору становится сложнее обманывать дискриминатор, так как тот улучшает свои способности к распознаванию фальшивых изображений. Одновременно снижение D loss свидетельствует о том, что дискриминатор обучается правильно классифицировать изображения, достигая все большей уверенности в своих предсказаниях.

## 2.5 Полученные результаты

На первых итерациях генерировались изображения, представленные на Рисунках 2.5.1–2.5.2.



Рисунок 2.5.1 – Изображение, создаваемое генератором на первой эпохе обучения



На первых итерациях генератор выдал шум, который не имеет ничего схожего с текстурами в датасете.

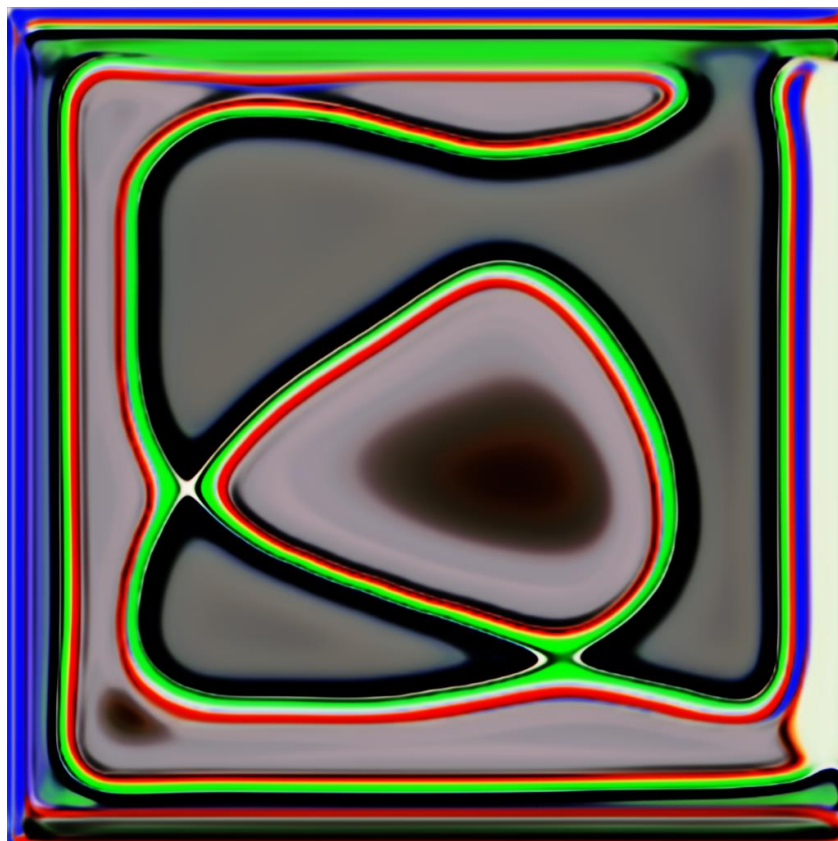


Рисунок 2.5.2 - Изображение, создаваемое генератором на третьей эпохе обучения

Изображение на десятой эпохе обучения представлено на Рисунке 2.5.3.



Рисунок 2.5.3 - Изображение, создаваемое генератором на десятой эпохе обучения

Изображение, созданное на пятидесятой эпохе обучения представлено на Рисунке 2.5.4.

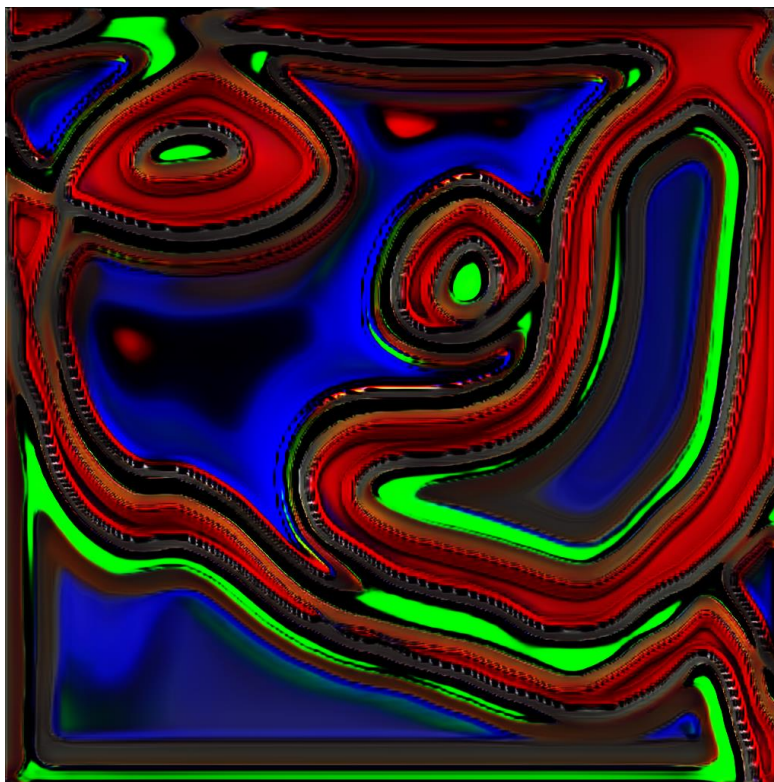


Рисунок 2.5.4 - Изображение, создаваемое генератором на пятидесятой эпохе обучения

Изображение, созданное на последних эпохах обучения представлено на Рисунке 2.5.5.

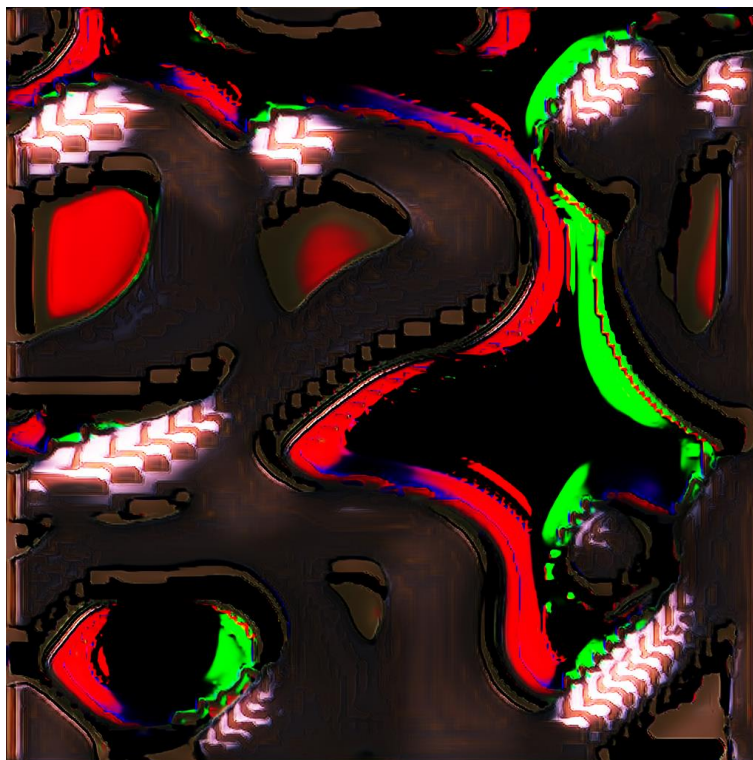


Рисунок 2.5.5 – Изображение, созданное на последних эпохах обучения

С ростом числа эпох детализация создаваемых текстур растет, однако, на картинках присутствуют черные области и полосы, а изображения все еще выглядят случайными и неструктурированными, отсутствуют узнаваемые узоры, закономерности или геометрические элементы.

Такой результат вероятнее всего связан с недостаточным количеством примеров в датасете и их большой вариативностью или недостаточным числом эпох, так как на ранних этапах обучения генеративные модели обычно создают шумоподобные структуры, и только при более длительном обучении начинают формировать осмысленные детали. Кроме того, стоит отметить, что дискриминатор обучается быстрее генератора, что видно по снижению D loss. Это приводит к ситуации, когда дискриминатор слишком уверенно отличает фейковые изображения, а генератору не хватает времени адаптироваться и научиться создавать что-то более осмысленное. В результате он «угадывает» случайные шаблоны, не приближаясь к формированию настоящих текстур.

## **2.6 Выводы по разделу**

В ходе выполнения работы обучена генеративно-сопоставительная сеть для генерации текстур скинов. Модель показала неплохие результаты в генерации, однако для создания более правдоподобных текстур необходимо увеличивать количество эпох, расширять датасет и выбирать более серьезную архитектуру.

## 3 ГРАФОВАЯ СЕТЬ

Наряду с обработкой табличных, текстовых, аудио данных и изображений, в глубинном обучении довольно часто приходится решать задачи на данных, имеющих графовую структуру. К таким данным относятся, к примеру, описания дорожных и компьютерных сетей, социальных графов и графов цитирований, молекулярных графов, а также графов знаний, описывающих взаимосвязи между сущностями, событиями и абстрактными категориями.

### 3.1 Теоретический раздел

#### 3.1.1 Описание графовых данных

Граф  $G = (V, E)$  принято представлять двумя множествами: множеством  $V$ , содержащим вершины и их признаковые описания, а также множеством  $E$ , содержащим связи между вершинами (то есть рёбра) и признаковые описания этих связей. На Рисунке 3.1.1 изображен пример графа.

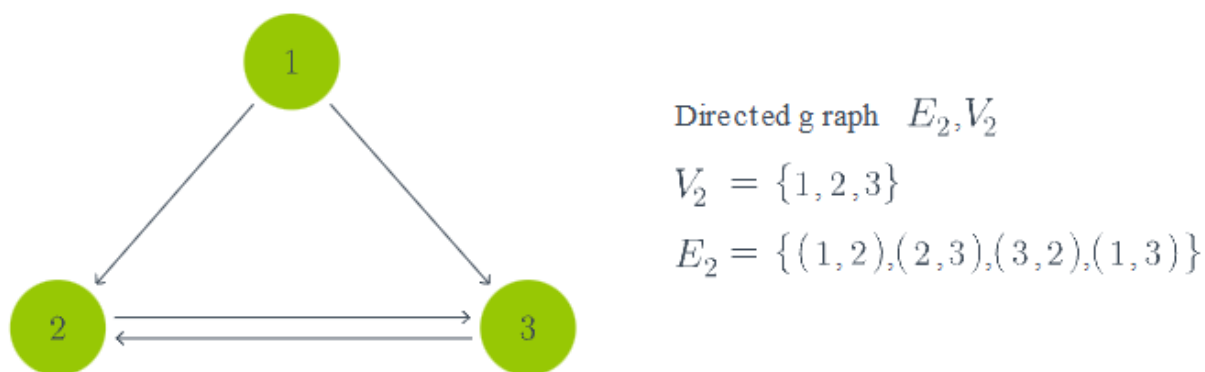


Рисунок 3.1.1 – Пример графа

Графовые данные довольно разнообразны. Они могут отличаться между собой в следующих моментах:

- по размеру, то есть количеству вершин и/или ребер;
- по наличию признаковых описаний вершин и рёбер. В зависимости от решаемой задачи, графы могут содержать информацию только в



вершинах, только в ребрах, либо же и там и там;

- графы могут быть гомо- и гетерогенными — в зависимости от того, имеют ли вершины и ребра графа одну природу либо же нет. Гомогенный и гетерогенный граф изображены на Рисунке 3.1.2.

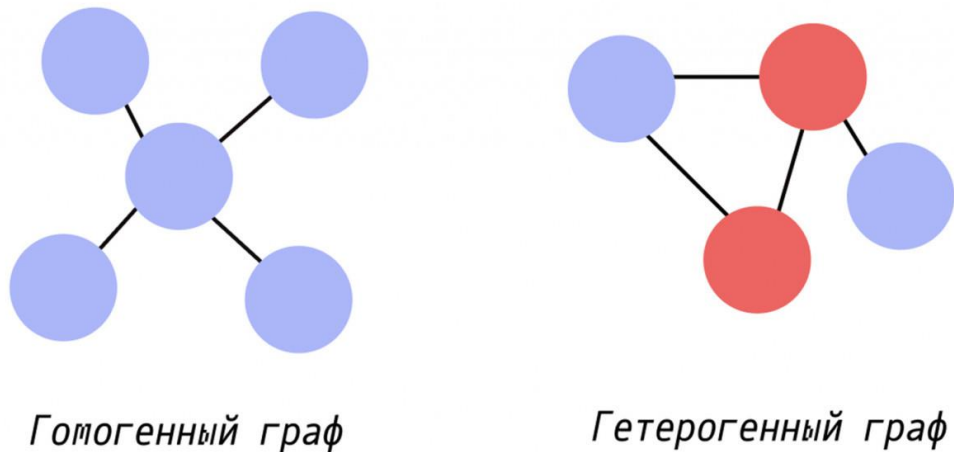


Рисунок 3.1.2 – Гомогенный и гетерогенный граф

Например, социальные графы содержат огромное количество вершин и ребер, часто измеряющееся в тысячах, содержат информацию в вершинах и очень редко в ребрах, а также являются гомогенными, так как все вершины имеют один тип. В то же время, молекулярные графы — это пример графов с, как правило, средним количеством вершин и ребер; вершины и связи в молекулярных графах имеют признаковое описание (типы атомов и ковалентных связей, а также информацию о зарядах и так далее), но при этом также являются гомогенными графами. К классу гетерогенных графов относятся, например, графы знаний, описывающие некоторую систему, различные сущности в ней и взаимодействия между этими сущностями. Вершины (сущности) и связи (ребра) такого графа могут иметь различную природу: скажем, вершинами могут быть сотрудники и подразделения компании, а рёбра могут отвечать отношениям «X работает в подразделении Y», «X и Z коллеги» и так далее.

### 3.1.2 Задачи на графах

Разнообразие графовых данных закономерно породило множество разнообразных задач, которые решаются на этих данных.

Среди них можно встретить классические постановки классификации, регрессии и кластеризации, но есть и специфичные задачи, не встречающиеся в других областях — например, задача восстановления пропущенных связей внутри графа или генерации графов с нужными свойствами. Однако даже классические задачи могут решаться на различных *уровнях*: классифицировать можно весь граф (graph-level), а можно отдельные его вершины (node-level) или связи (edge-level).

Так, в качестве примера graph-level задач можно привести классификацию и регрессию на молекулярных графах. Имея датасет с размеченными молекулами, можно предсказывать их принадлежность к лекарственной категории и различные химико-биологические свойства.

На node-level, как правило, классифицируют вершины одного огромного графа, например, социального. Имея частичную разметку, хочется восстановить метки неразмеченных вершин. Например, предсказать интересы нового пользователя по интересам его друзей.

Часто бывает такое, что граф приходит полностью неразмеченным и хочется без учителя разделить на компоненты. Например, имея граф цитирований, выделить в нем подгруппы соавторов или выделить области исследования. В таком случае принято говорить о node-level кластеризации графа.

Наконец, довольно интересна задача предсказания пропущенных связей в графе. В больших графах часто некоторые связи отсутствуют. Например, в социальном графе пользователь может добавить не всех знакомых в друзья. А в графе знаний могут быть проставлены только простые взаимосвязи, а высокоуровневые могут быть пропущены.

В конце, хотелось бы отметить очень важные особенности всех задач, связанных с графами. Алгоритмы решения этих задач должны обладать двумя свойствами:

- во-первых, графы в датасетах, как правило, могут отличаться по размерам: как по количеству вершин, так и по количеству связей. Алгоритмы решения задач на графах должны уметь принимать графы различных размеров;
- во-вторых, алгоритмы должны быть инварианты к перестановкам порядка вершин. То есть если взять тот же граф и перенумеровать его вершины, то алгоритмы должны выдавать те же предсказания с учетом этой перестановки.

### **3.1.2 История появления архитектуры**

Идея обработки данных, представленных в виде графов, возникла значительно раньше, чем само понятие «графовая нейронная сеть». Однако классические методы машинного обучения плохо справлялись с задачами на графах, поскольку они предполагали фиксированную размерность входных данных. Первые попытки адаптировать нейронные сети к графам относятся к началу 2000-х годов.

В 2005 году была представлена модель Graph Neural Networks (GNN) в работе Scarselli et al., которая вводила идею итеративного обновления признаков вершин на основе их соседей. Однако из-за ограничений вычислительных ресурсов и сложности обучения, первые GNN не получили широкого распространения.

Ситуация изменилась в 2017 году после выхода ряда ключевых работ:

- Graph Convolutional Networks (GCN) от Thomas Kipf и Max Welling предложили упрощенную схему обучения на графах, используя спектральное приближение сверточной операции для графов. Эта работа стала отправной точкой для бурного роста интереса к

графовым сетям;

- в этом же году появились GraphSAGE (Hamilton et al.) и GAT (Graph Attention Networks) (Velickovic et al.), предложившие методы агрегации информации от соседних вершин с помощью обучаемых функций и механизмов внимания.

С тех пор графовые нейронные сети быстро развивались. Появились такие направления, как:

- Graph Isomorphism Networks (GIN) – модели, приближающие теоретическую выразительность классических алгоритмов проверки изоморфизма графов.
- Spatial GNN и Spectral GNN – архитектуры, которые используют разные подходы к определению сверточной операции на графах.
- Dynamic GNN – модели, которые обрабатывают графы с изменяющейся структурой (например, временные графы в социальных сетях).
- Large-scale GNN – разработки для масштабирования GNN на очень большие графы, например, графы социальных сетей.

### 3.1.3 Описание архитектуры

Графовые нейронные сети по принципу работы и построения идейно очень похожи на сверточные нейронные сети. Более того, графовые нейронные сети являются обобщением сверточных нейронных сетей.

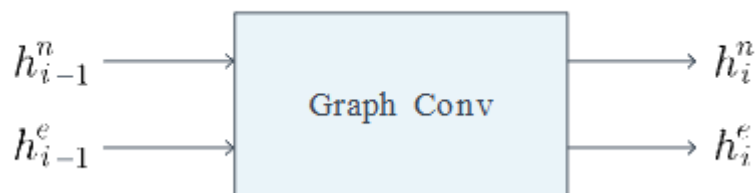
На вход графовой нейронной сети подается граф. В отличие от сверточных нейронных сетей, которые требуют, чтобы все картинки в батче были одинакового размера, графовые нейронные сети допускают разные размеры у объектов батча. Кроме того, в отличие от картинок, у которых информация довольно однородна (это, как правило, несколько цветовых каналов) и хранится в пикселях, у графов информация может также храниться в вершинах и/или ребрах. Причем в одних задачах информация может быть только в вершинах, в

других только в ребрах, а в-третьих, и там, и там. Сама информация может быть довольно разнородной: это могут быть и вещественные значения, и дискретные значения, в зависимости от природы графа и от типа решаемой задачи. Поэтому, довольно часто первым слоем в графовых нейронных сетях идут Embedding слои, которые переводят дискретные токены в вещественные векторы (Формула 3.1.1).

$$h_0^n = Emb(V), h_0^e = Emb(E) \quad (3.1.1)$$

Однако, сама суть работы у графовых и сверточных сетей совпадает. В графовой нейронной сети по очереди применяются слои, которые собирают информацию с соседей и обновляют информацию в вершине. То же самое делают и обычные свертки. Поэтому такие слои и называются графовыми свертками. Графовая свертка принимает на вход граф со скрытыми состояниями у вершин и ребер и выдает тот же граф, но уже с обновленными более информативными скрытыми состояниями.

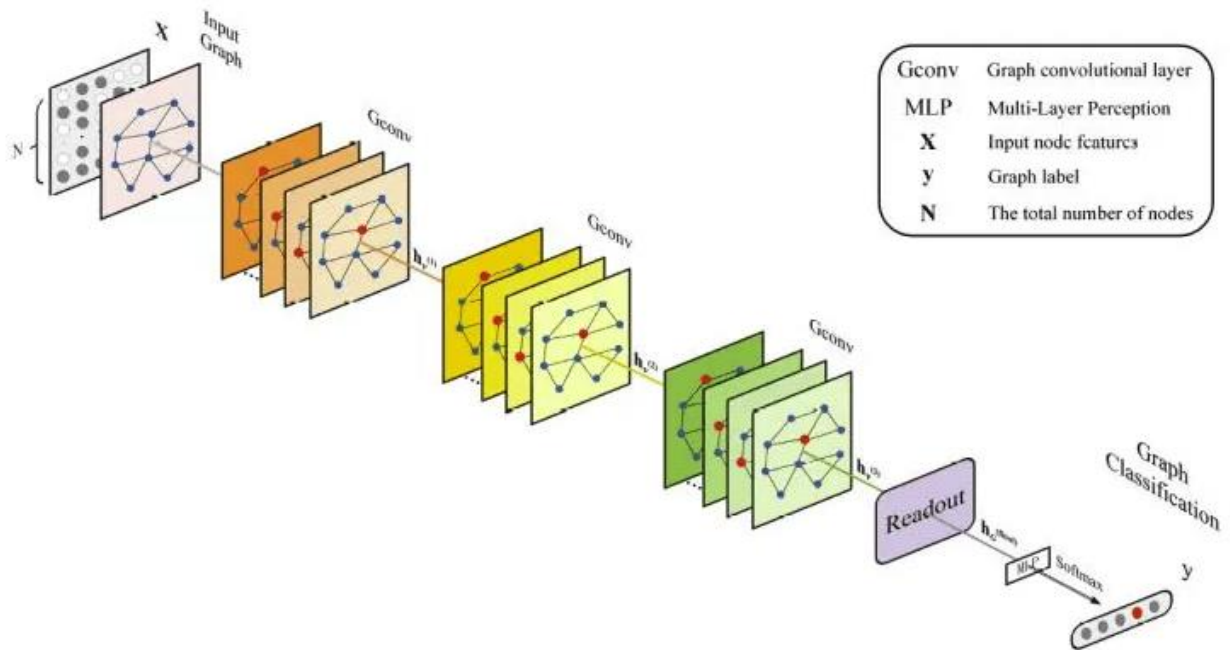
В отличие от сверточных нейронных сетей, при обработке графа pooling слои вставляют редко, в основном в graph-level задачах, при этом придумать разумную концепцию графового пулинга оказалось нелегко. В большинстве же архитектур пулинги не используются, и структура графа на входе и выходе графовой нейронной сети совпадает (Рисунок 3.1.3).



**Рисунок 3.1.3 – Обработка графа без изменения структуры**

Полученная после череды сверток информация с вершин и ребер в конце обрабатывается с помощью полносвязных сетей для получения ответа на задачу. Для node-level классификации и регрессии полносвязная сеть применяется к

скрытым состояниям вершин  $h_K^n$ , а для edge-level, соответственно, к скрытым состояниям ребер  $h_K^e$ . Для получения ответа на graph-level уровне информация с вершин и ребер сначала агрегируется с помощью readout операции. На месте readout операции могут располагаться любые инвариантные к перестановкам операции: подсчет максимума, среднего или даже обучаемый self-attention слой. Архитектура графовой сети изображена на Рисунке 3.1.4.



**Рисунок 3.1.4 – Архитектура графовой сети**

Как говорилось ранее, графовые нейронные сети являются обобщением сверточных. Если представить пиксели изображения вершинами графа, соединить соседние по свертке пиксели ребрами и предоставить относительную позицию пикселей в информации о ребре, то графовая свертка на таком графе будет работать так же, как и свертка над изображением.

К графовым нейронным сетям, как и к сверточным, применим термин receptive field. Это та область графа, которая будет влиять на скрытое состояние вершины после  $N$  сверток. Для графов receptive field после  $N$  графовых сверток — это все вершины и ребра графа, до которых можно дойти от фиксированной вершины не более чем за  $N$  переходов. Знание receptive field полезно при проектировании нейронной сети - имея представление о том, с какой

окрестности вершины надо собрать информацию для решения задачи, можно подбирать нужное количество графовых сверток.

Многие техники стабилизации обучения и повышения обобщаемости, такие как Dropout, BatchNorm и Residual Connections, применимы и к графовым нейронным сетям. Однако стоит помнить про их особенности. Эти операции могут независимо применяться (или не применяться) к вершинам и ребрам. Так, если вы применяете Dropout, то вы вправе поставить для вершин и для рёбер различные значения dropout rate. Аналогично и для Residual Connections - они могут применяться только для вершин, только для ребер или же и там и там.

### **3.2 Постановка задачи**

Цель: реализовать обучение графовой сети для решения задачи классификации вершин.

Задачи: изучить устройство графовых нейронных сетей (GNN), выбрать данные для обучения и выполнить их предобработку, обучить составленную GNN, интерпретировать полученные результаты.

### **3.3 Документация к данным**

В качестве набора данных выбран Cora - один из самых известных графовых датасетов из коллекции Planetoid. Состоит из научных статей по информатике, связанных цитированиями. Основная задача — классификация статей по тематическим категориям на основе их текстового содержания и структуры цитирований.

Каждая вершина (node) — это научная статья. Каждое ребро (edge) — это цитирование одной статьи другой. Статьи разделены на 7 тематических классов. Параметры графа, описывающего датасет, представлены на Рисунке 3.3.1.

Метрики графа:

Metric	Value
Nodes	2708
Edges	10556
Node features	1433
Classes	7
Train nodes	140
Validation nodes	500
Test nodes	1000
Avg degree	3.89808
Min degree	1
Max degree	168

**Рисунок 3.3.1 – Параметры графа**

Описание параметров графа:

- Nodes - количество научных статей (узлов) в графе;
- Edges - количество цитирований между статьями (направленные рёбра);
- Node features - размерность признакового описания статей (бинарные векторы слов);
- Classes - число тематических категорий;
- Train nodes - количество статей в обучающей выборке (5% от общего числа);
- Validation nodes - количество статей в валидационной выборке (для настройки модели);
- Test nodes - количество статей в тестовой выборке (для оценки качества модели);
- Avg degree - среднее количество цитирований на статью;
- Min degree - минимальное количество цитирований у статьи (есть статьи с 1 ссылкой);
- Max degree - максимальное количество цитирований у статьи (популярная статья).

Распределение статей по тематическим категориям (классам) отображено на Рисунке 3.3.2.



Метки классов:

Label ID	Label Name	Count
0	Case_Based	351
1	Genetic_Algorithms	217
2	Neural_Networks	418
3	Probabilistic_Methods	818
4	Reinforcement_Learning	426
5	Rule_Learning	298
6	Theory	180

Рисунок 3.3.2 – Распределение статей по тематическим категориям

Диаграмма баланса классов показана на Рисунке 3.3.3.

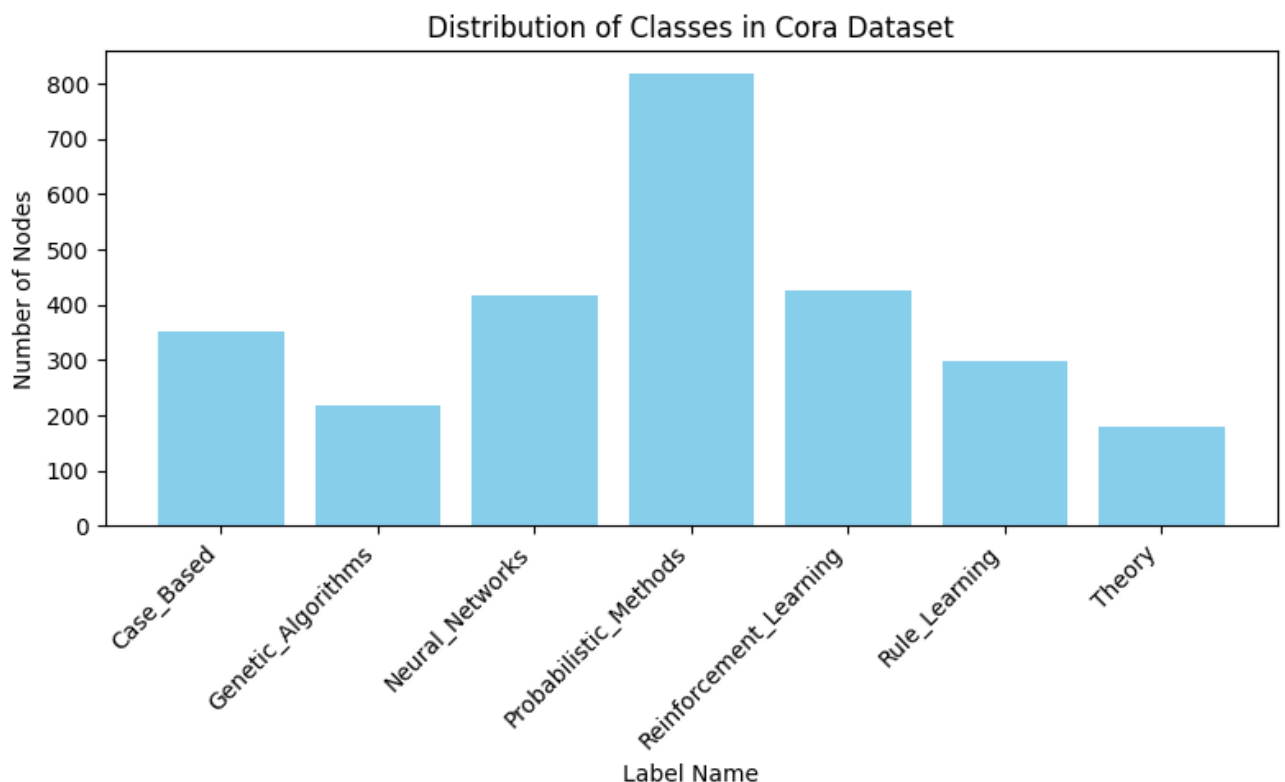


Рисунок 3.3.3 – Диаграмма баланса классов

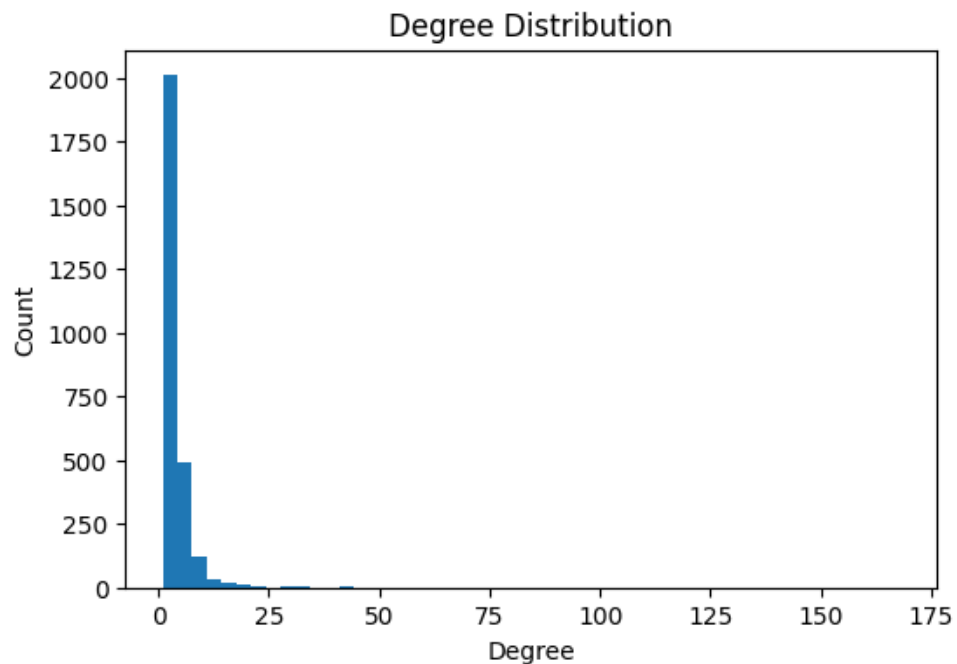
Пример первых десяти вершин графа представлен на Рисунке 3.3.4.

Первые 10 вершин графа:

Node	0	label=3	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	1	label=4	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	2	label=4	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	3	label=0	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	4	label=3	features=[0.0, 0.0, 0.0, 0.0555555559694767, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	5	label=2	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	6	label=0	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	7	label=3	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	8	label=3	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...
Node	9	label=2	features=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...

Рисунок 3.3.4 - Пример первых десяти вершин графа

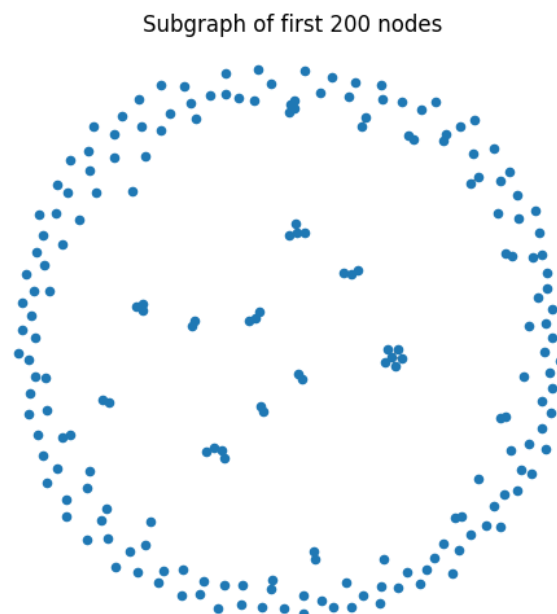
Гистограмма распределения степеней вершин представлена на Рисунке 3.3.5.



**Рисунок 3.3.5 – Гистограмма распределения степеней вершин**

Гистограмма показывает, как распределены узлы графа Cora по количеству связей (степеням). Каждый столбец соответствует определённому диапазону степеней, а его высота отражает количество узлов, имеющих такое число связей. По гистограмме видно, что большинство узлов имеют малое количество связей.

Визуализация подграфа из 200 вершин отражена на Рисунке 3.3.6.



**Рисунок 3.3.6 - Визуализация подграфа из 200 вершин**

Визуализация помогает понять, есть ли в графе «плотные» кластеры или он случайный, обнаружить статьи, которые не ссылаются ни на кого.

### 3.4 Обучение модели

Для решения задачи классификации вершин в графе был реализован процесс обучения графовой нейронной сети (GNN) с использованием модели GCN (Graph Convolutional Network). Архитектура модели представлена на Рисунке 3.4.1.

Архитектура и число параметров:

Layer (type:depth-idx)	Input Shape	Output Shape	Param #
GCN	[2708, 1433]	[2708, 7]	--
├─GCNConv: 1-1	[2708, 1433]	[2708, 16]	16
│   ├─Linear: 2-1	[2708, 1433]	[2708, 16]	22,928
│   └─SumAggregation: 2-2	[13264, 16]	[2708, 16]	--
├─GCNConv: 1-2	[2708, 16]	[2708, 7]	7
│   ├─Linear: 2-3	[2708, 16]	[2708, 7]	112
│   └─SumAggregation: 2-4	[13264, 7]	[2708, 7]	--
=====			
Total params: 23,063			
Trainable params: 23,063			
Non-trainable params: 0			
Total mult-adds (Units.MEGABYTES): 62.39			
=====			
Input size (MB): 15.69			
Forward/backward pass size (MB): 0.50			
Params size (MB): 0.09			
Estimated Total Size (MB): 16.28			
=====			

**Рисунок 3.4.1 – Архитектура модели**

Модель представляет собой двухслойную Graph Convolutional Network (GCN), предназначенную для решения задачи классификации вершин на графе. Каждый слой GCN агрегирует информацию от соседних вершин, что позволяет учитывать структуру графа в процессе обучения. Модель состоит из следующих компонентов:

1. Первый сверточный слой (GCNConv).
2. Функция активации ReLU.
3. Операция Dropout.
4. Второй сверточный слой (GCNConv).
5. Функция активации log\_softmax.

Слои архитектуры описаны в Таблице 3.4.1.

Таблица 3.4.1 – Слои архитектуры

Номер компонента	Компонент	Описание
1	GCNConv(input_dim, hidden_dim)	Графовая свёртка: преобразует входные признаки вершин в скрытые представления размерности hidden_dim. Учитывает структуру графа через агрегацию признаков соседей
2	ReLU	Нелинейная функция активации ReLU применяется к выходу первого слоя, чтобы повысить способность модели моделировать сложные зависимости
3	Dropout(p=0.5)	Для регуляризации используется Dropout с вероятностью 0.5: случайное обнуление части выходных признаков после первого слоя
4	GCNConv(hidden_dim, num_classes)	Второй графовый свёрточный слой: сжимает скрытые представления до числа классов (num_classes)
5	log_softmax(dim=1)	На выходе применяется нормализация логарифмированным softmax для получения лог-вероятностей по классам. Это удобно для дальнейшей работы с функцией потерь CrossEntropyLoss

Обучение осуществлялось с помощью специально разработанного менеджера обучения TrainingManager, который инкапсулирует весь цикл тренировки, раннюю остановку и сбор статистики по эпохам.

На каждом шаге обучения выполнялись следующие действия:

- прямое распространение (forward pass): модель получает в качестве входа признаки вершин и структуру графа (рёбра) и предсказывает метку класса для каждой вершины;
- вычисление функции потерь: в качестве функции потерь

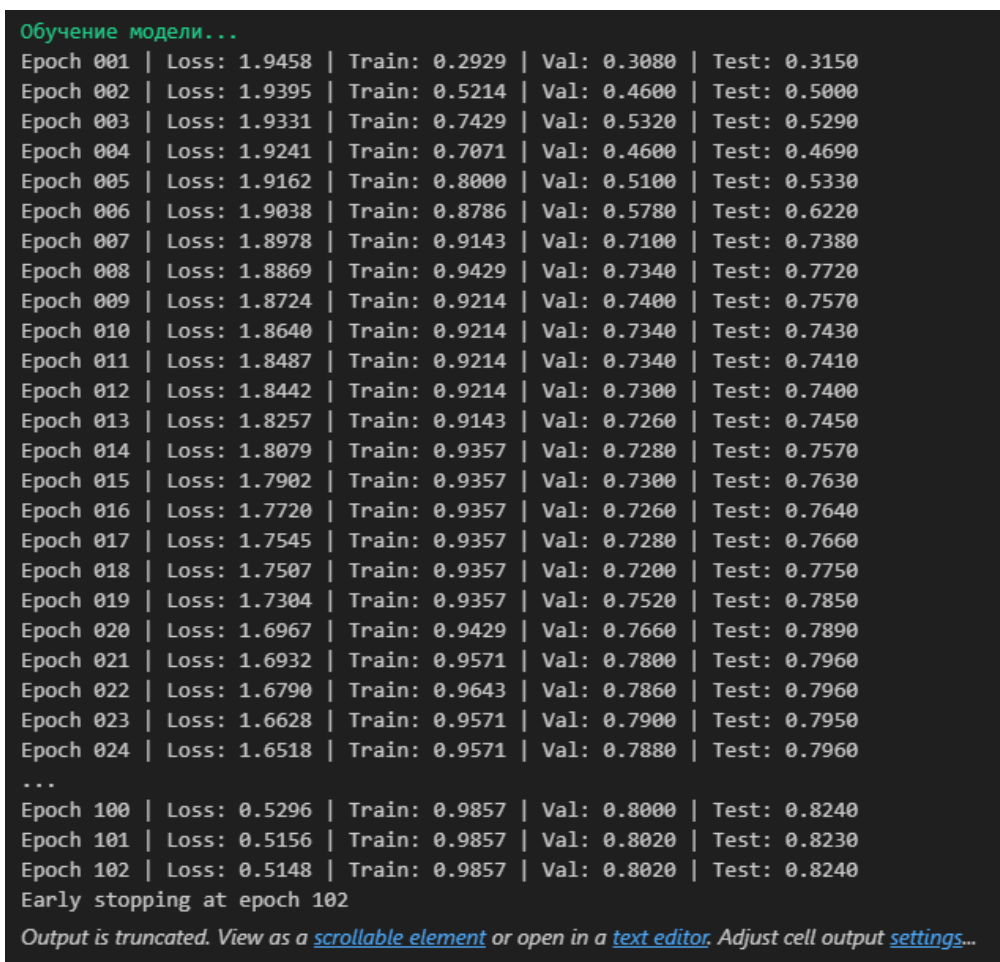
использовалась перекрёстная энтропия (CrossEntropyLoss), вычисляемая только на тренировочной части данных (train\_mask);

- обратное распространение ошибки (backward pass) и обновление весов: градиенты функции потерь рассчитывались и применялись к параметрам модели с использованием оптимизатора Adam;
- оценка точности: после каждой эпохи производилась оценка качества классификации (accuracy) на тренировочной, валидационной и тестовой выборках.

Для предотвращения переобучения применялась ранняя остановка обучения: если в течение заданного числа эпох не наблюдалось улучшения точности на валидационной выборке, обучение автоматически завершалось.

Такой подход позволяет избежать деградации качества модели и экономит вычислительные ресурсы.

Процесс обучения изображен на Рисунке 3.4.2.



```
Обучение модели...
Epoch 001 | Loss: 1.9458 | Train: 0.2929 | Val: 0.3080 | Test: 0.3150
Epoch 002 | Loss: 1.9395 | Train: 0.5214 | Val: 0.4600 | Test: 0.5000
Epoch 003 | Loss: 1.9331 | Train: 0.7429 | Val: 0.5320 | Test: 0.5290
Epoch 004 | Loss: 1.9241 | Train: 0.7071 | Val: 0.4600 | Test: 0.4690
Epoch 005 | Loss: 1.9162 | Train: 0.8000 | Val: 0.5100 | Test: 0.5330
Epoch 006 | Loss: 1.9038 | Train: 0.8786 | Val: 0.5780 | Test: 0.6220
Epoch 007 | Loss: 1.8978 | Train: 0.9143 | Val: 0.7100 | Test: 0.7380
Epoch 008 | Loss: 1.8869 | Train: 0.9429 | Val: 0.7340 | Test: 0.7720
Epoch 009 | Loss: 1.8724 | Train: 0.9214 | Val: 0.7400 | Test: 0.7570
Epoch 010 | Loss: 1.8640 | Train: 0.9214 | Val: 0.7340 | Test: 0.7430
Epoch 011 | Loss: 1.8487 | Train: 0.9214 | Val: 0.7340 | Test: 0.7410
Epoch 012 | Loss: 1.8442 | Train: 0.9214 | Val: 0.7300 | Test: 0.7400
Epoch 013 | Loss: 1.8257 | Train: 0.9143 | Val: 0.7260 | Test: 0.7450
Epoch 014 | Loss: 1.8079 | Train: 0.9357 | Val: 0.7280 | Test: 0.7570
Epoch 015 | Loss: 1.7902 | Train: 0.9357 | Val: 0.7300 | Test: 0.7630
Epoch 016 | Loss: 1.7720 | Train: 0.9357 | Val: 0.7260 | Test: 0.7640
Epoch 017 | Loss: 1.7545 | Train: 0.9357 | Val: 0.7280 | Test: 0.7660
Epoch 018 | Loss: 1.7507 | Train: 0.9357 | Val: 0.7200 | Test: 0.7750
Epoch 019 | Loss: 1.7304 | Train: 0.9357 | Val: 0.7520 | Test: 0.7850
Epoch 020 | Loss: 1.6967 | Train: 0.9429 | Val: 0.7660 | Test: 0.7890
Epoch 021 | Loss: 1.6932 | Train: 0.9571 | Val: 0.7800 | Test: 0.7960
Epoch 022 | Loss: 1.6790 | Train: 0.9643 | Val: 0.7860 | Test: 0.7960
Epoch 023 | Loss: 1.6628 | Train: 0.9571 | Val: 0.7900 | Test: 0.7950
Epoch 024 | Loss: 1.6518 | Train: 0.9571 | Val: 0.7880 | Test: 0.7960
...
Epoch 100 | Loss: 0.5296 | Train: 0.9857 | Val: 0.8000 | Test: 0.8240
Epoch 101 | Loss: 0.5156 | Train: 0.9857 | Val: 0.8020 | Test: 0.8230
Epoch 102 | Loss: 0.5148 | Train: 0.9857 | Val: 0.8020 | Test: 0.8240
Early stopping at epoch 102
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Рисунок 3.4.2 – Процесс обучения модели

### 3.5 Полученные результаты

Во время обучения автоматически сохранялись значения функции потерь (loss) по эпохам, точности (accuracy) на тренировочной, валидационной и тестовой выборках.

Построен график изменения функции потерь по эпохам (Рисунок 3.5.1) и график изменения точности на обучающей, валидационной и тестовой выборках (Рисунок 3.5.2).

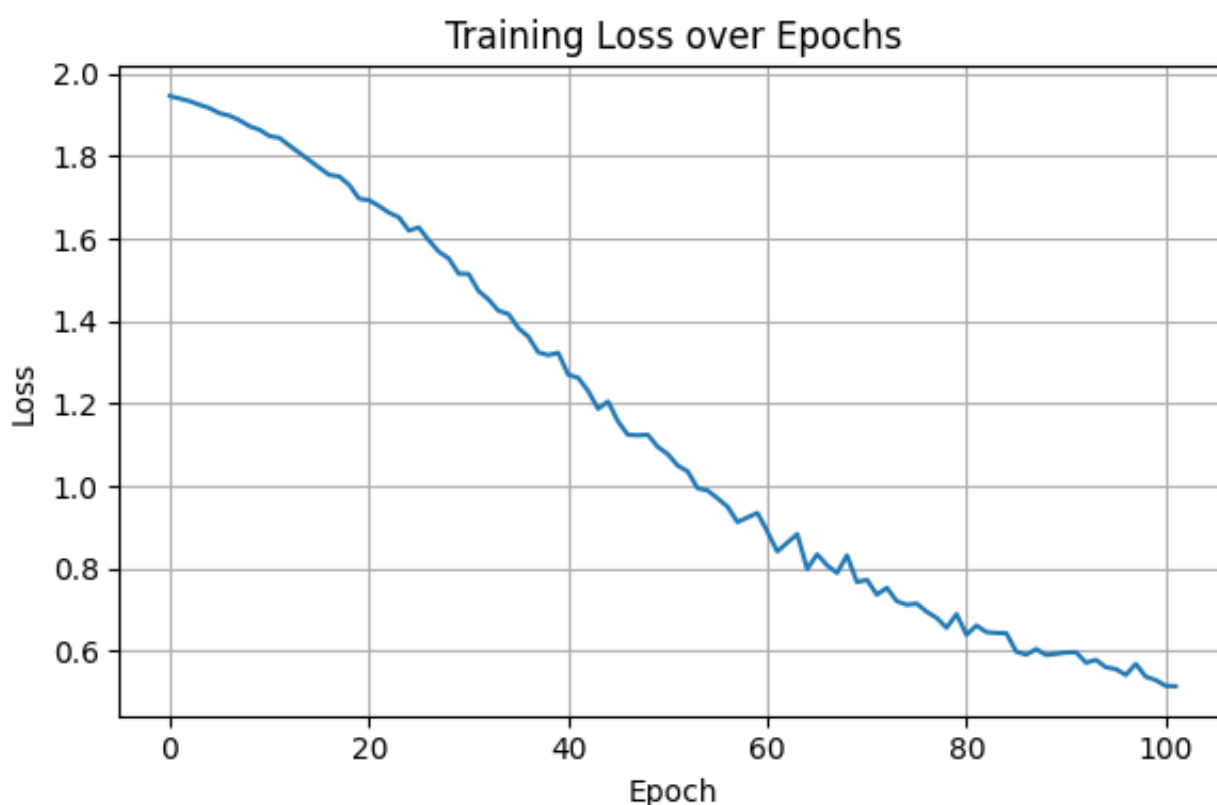
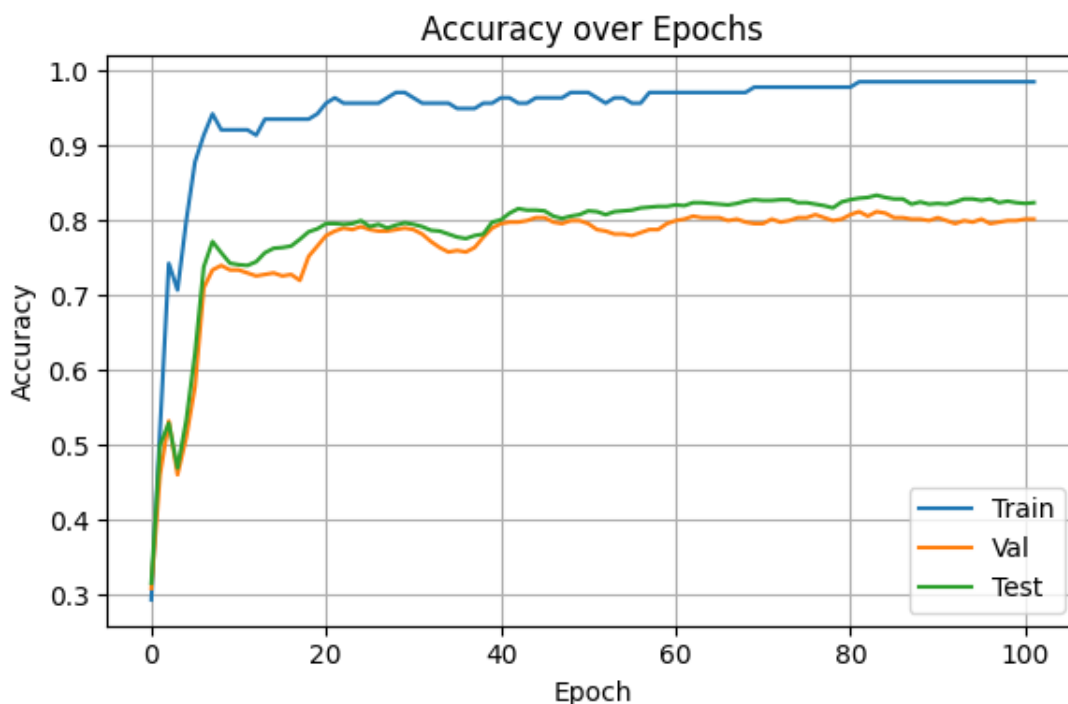


Рисунок 3.5.1 - График изменения функции потерь по эпохам

График изменения функции потерь показал устойчивое снижение loss на протяжении большинства эпох, что свидетельствует о корректной настройке обучения и сходимости модели.



**Рисунок 3.5.2 - График изменения точности на обучающей, валидационной и тестовой выборках**

Точность на обучающей выборке росла стабильно, а валидационная и тестовая точности также демонстрировали рост до момента стабилизации.

Итоговая точность на тестовой выборке равна 0.8080. Модель успешно обучилась и достигла высокой точности на тестовой выборке, что подтверждает её способность к обобщению.

### **3.6 Выводы по разделу**

Построенная графовая нейронная сеть на основе архитектуры GCN успешно справилась с задачей классификации вершин в графе Cora, обеспечив высокую точность на тестовой выборке. Результаты подтверждают, что использование GNN позволяет эффективно учитывать как признаки вершин, так и структуру их связей для решения задач классификации.

## **4 СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПРЕДОБУЧЕННОЙ И ДООБУЧЕННОЙ МОДЕЛИ**

### **4.1 Теоретический раздел**

Современные подходы в области машинного обучения демонстрируют значительный прогресс благодаря использованию предобученных моделей, таких как трансформеры. Эти модели, благодаря своей универсальности и способности обучаться на больших объемах данных, достигли высоких результатов в решении различных задач, включая обработку естественного языка, машинный перевод, классификацию текста и другие. Одним из ключевых преимуществ таких моделей является возможность их дообучения на специфических данных, что позволяет адаптировать их к конкретным требованиям и задачам.

Дообучение предобученной модели представляет собой процесс, в ходе которого модель, уже обладающая общими знаниями, получает дополнительное обучение на более специфичных данных для улучшения своей производительности на конкретной задаче. Это позволяет не только сократить время и ресурсы, необходимые для обучения, но и повысить точность модели, особенно когда доступ к большим объемам данных ограничен.

Сравнительный анализ предобученной и дообученной модели предоставляет ценную информацию о влиянии дополнительного обучения на результативность. Важно отметить, что, несмотря на впечатляющие достижения предобученных моделей, их эффективность может быть ограничена на специфических задачах, где требуется обработка специализированных данных. Дообучение позволяет устранить этот недостаток, подстраивая модель под специфическую доменную задачу, что делает её более точной и адаптированной.



## 4.2 Постановка задачи

Цель: реализовать дообучение предобученной модели трансформера и провести сравнительный анализ её эффективности по сравнению с оригинальной предобученной моделью.

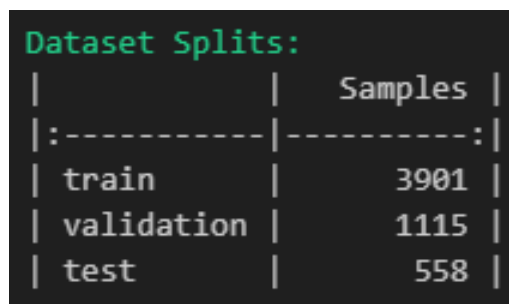
Задачи: изучить архитектуру трансформеров и особенности их применения в задачах обработки естественного языка, выбрать подходящий набор данных для дообучения модели, провести их предобработку и подготовку, реализовать процесс дообучения предобученной модели трансформера с использованием выбранных данных, оценить производительность дообученной модели и сравнить её с результатами работы предобученной модели, провести анализ полученных результатов, интерпретировать влияние дообучения на точность модели.

## 4.3 Документация к данным

В качестве датасета выбран набор данных «sms\_spam», который доступен через библиотеку datasets от Hugging Face. Этот набор данных содержит метки для классификации сообщений на два класса: спам и не спам. Для загрузки, анализа и подготовки датасета написан класс DatasetManager.

Датасет предварительно разделён на три части: обучающую, валидационную и тестовую выборки, что позволяет проводить эффективную оценку модели на различных этапах её обучения.

Распределение примеров между тренировочной, валидационной и тестовой выборками представлено на Рисунке 4.3.1.

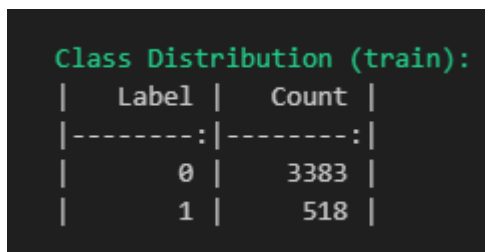


The image shows a terminal window with a dark background. The title 'Dataset Splits:' is in green. Below it is a table with two columns: the split name and the number of samples. The table is enclosed in a box with dashed lines for the header.

Dataset Splits:	
	Samples
train	3901
validation	1115
test	558

Рисунок 4.3.1 – Распределение примеров между выборками

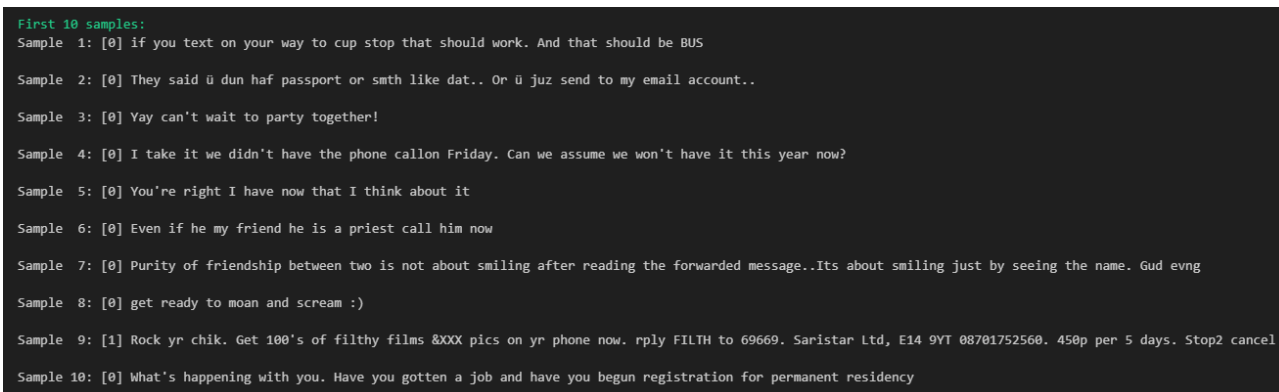
Распределение классов отображено на Рисунке 4.3.2.



```
Class Distribution (train):
| Label | Count |
|-----:|-----:|
|      0 |   3383 |
|      1 |    518 |
```

Рисунок 4.3.2 – Распределение классов в датасете

Пример десяти записей (писем) с метками представлен на Рисунке 4.3.3.



```
First 10 samples:
Sample 1: [0] if you text on your way to cup stop that should work. And that should be BUS
Sample 2: [0] They said ü dun haf passport or smth like dat.. Or ü juz send to my email account..
Sample 3: [0] Yay can't wait to party together!
Sample 4: [0] I take it we didn't have the phone callon Friday. Can we assume we won't have it this year now?
Sample 5: [0] You're right I have now that I think about it
Sample 6: [0] Even if he my friend he is a priest call him now
Sample 7: [0] Purity of friendship between two is not about smiling after reading the forwarded message..Its about smiling just by seeing the name. Gud evng
Sample 8: [0] get ready to moan and scream :)
Sample 9: [1] Rock yr chik. Get 100's of filthy films &XXX pics on yr phone now. rply FILTH to 69669. Saristar Ltd, E14 9YT 08701752560. 450p per 5 days. Stop2 cancel
Sample 10: [0] What's happening with you. Have you gotten a job and have you begun registration for permanent residency
```

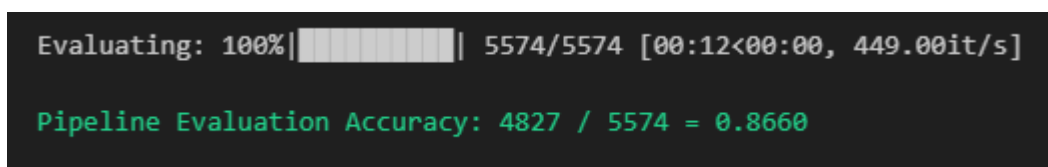
Рисунок 4.3.3 - Пример десяти записей (писем) с метками

## 4.4 Предобученная модель

В качестве модели выбрана DistilBERT — оптимизированная версия BERT, разработанная для снижения вычислительных затрат при сохранении высокой точности. DistilBERT достигает около 95% производительности оригинального BERT, но содержит на 40% меньше параметров и работает в 1.6 раза быстрее.

Использована функция-конвейер pipeline из библиотеки transformers, которая загружает и кеширует модель для дальнейшего использования. Код использования предобученной модели представлен в Приложении Г.1.

Процесс оценки точности модели на выбранном датасете проиллюстрирован на Рисунке 4.4.1.



```
Evaluating: 100%|██████████| 5574/5574 [00:12<00:00, 449.00it/s]

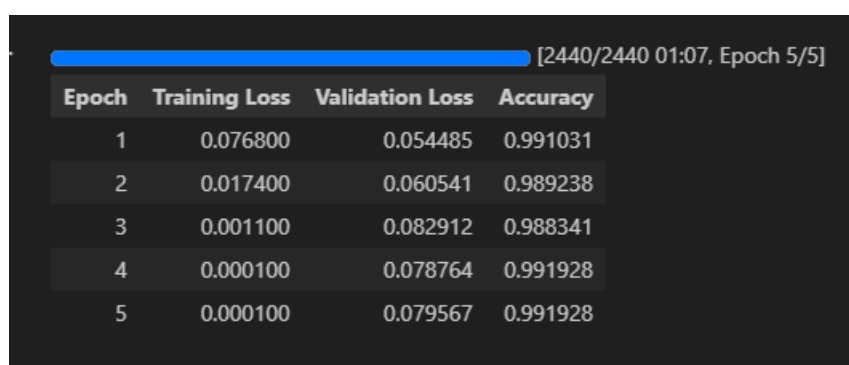
Pipeline Evaluation Accuracy: 4827 / 5574 = 0.8660
```

Рисунок 4.4.1 – Оценка точности предобученной модели

## 4.5 Дообучение модели

Для адаптации предобученной архитектуры DistilBERT к задаче детекции спама была выполнена процедура fine-tuning на конкретном SMS-корпусе. Исходный набор сообщений, содержащий только сплит «train», был последовательно разделён на три части — обучающую, валидационную и тестовую — в соотношении приблизительно 80 % / 10 % / 10 %. После этого каждая запись подвергалась токенизации с помощью WordPiece-токенизатора длиной до 128 токенов; из полученных input\_ids и attention\_mask формировались батчи с помощью DataCollatorWithPadding, что позволяло автоматически выравнивать последовательности по длине.

Процесс дообучения модели представлен на Рисунке 4.5.1.



Epoch	Training Loss	Validation Loss	Accuracy
1	0.076800	0.054485	0.991031
2	0.017400	0.060541	0.989238
3	0.001100	0.082912	0.988341
4	0.000100	0.078764	0.991928
5	0.000100	0.079567	0.991928

Рисунок 4.5.1 – Дообучение модели

График ошибки модели изображен на Рисунке 4.5.2.

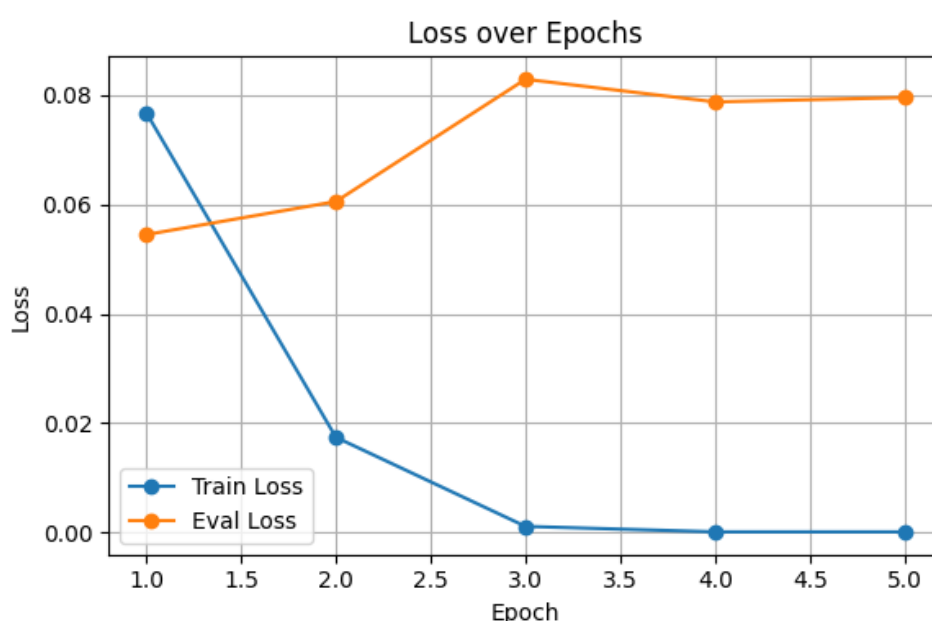


Рисунок 4.5.2 – График ошибки модели

По графику видно, что за три эпохи обучения ошибка модели стремится к нулю, так как датасет считается простым в силу малого числа записей.

График точности модели представлен на Рисунке 4.5.3.

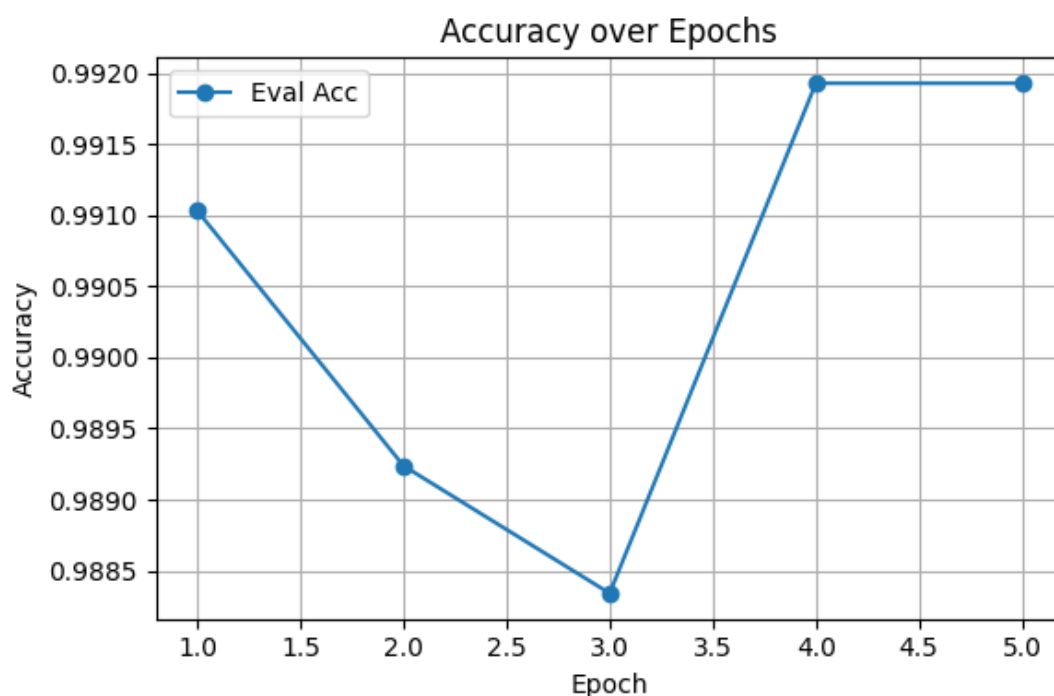


Рисунок 4.5.3 – График точности модели

Проверка модели на тестовой выборке и финальная точность представлены на Рисунке 4.5.4.

```
Testing on test split...  
Test Accuracy: 0.9946  
Training complete. Model and metrics saved to: ./spam\_model
```

Рисунок 4.5.4 – Точность модели на тестовой выборке

## 4.5 Выводы по разделу

В ходе дообучения наблюдался устойчивый рост качества модели: с каждой эпохой среднее значение функции потерь на тренировочных данных снижалось, а точность на валидационном наборе непрерывно возрастала и к окончанию обучения превысила 98%. При финальной проверке на тестовой выборке модель продемонстрировала высокие результаты, что свидетельствует об эффективности процедуры трансферного обучения в повышении точности классификации спама.

## ЗАКЛЮЧЕНИЕ

В рамках данной курсовой работы была проведена адаптация и дообучение предобученной трансформерной модели DistilBERT для задачи бинарной классификации SMS-сообщений на спам и не-спам. Теоретический анализ ключевых архитектур глубокого обучения — трансформеров, генеративно-сопоставительных и графовых сетей — показал универсальность и мощь методов трансферного обучения и внимания при работе с текстовыми данными.

Анализ динамики функции потерь и метрики точности подтвердил стабильную сходимость процесса обучения и отсутствие признаков переобучения. Полученные результаты демонстрируют, что даже относительно небольшой объём размеченных данных в сочетании с мощными предобученными архитектурами способен обеспечить надёжную и эффективную фильтрацию спама. Всё это подчёркивает перспективность подходов, основанных на глубоком обучении и трансферном обучении, для решения прикладных задач NLP с ограниченными вычислительными ресурсами.

## ПРИЛОЖЕНИЯ

Приложение А — Реализация модели трансформера.

Приложение Б.1 — Парсер веб-сайта с текстурами.

Приложение Б.2 — Реализация генеративно-состязательной сети.

Приложение В — Реализация графовой сети.

Приложение Г.1 — Использование предобученной модели.

Приложение Г.2 — Реализация трансферного обучения.

## Приложение А

### Реализации модели трансформера

#### *Листинг А – Код реализации модели трансформера*

```
import torch
import evaluate
from datasets import load_dataset
from transformers import AutoTokenizer
from transformers import AutoModelForQuestionAnswering
from transformers import TrainingArguments
from transformers import Trainer
from transformers import pipeline
from transformers import default_data_collator
from datasets import load_dataset

MODEL_NAME = "DeepPavlov/rubert-base-cased"

class QADatasetProcessor:
    """
    Класс для загрузки, токенизации и предобработки данных для задачи вопрос-
    ответ (QA).
    Использует SQuAD-подобный датасет SberQuad.
    """

    def __init__(self, model_name: str = MODEL_NAME):
        """
        Инициализирует токенизатор на основе заданной модели.

        Параметры:
            model_name (str): название модели, совместимой с Hugging Face.
        """
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)

    def load_data(self):
        """
        Загружает датасет SberQuad из Hugging Face Datasets.

        Возвращает:
            DatasetDict: словарь с разбивкой на train и validation.
        """
        self.dataset = load_dataset("kuznetsoffandrey/sberquad")
        return self.dataset

    def _preprocess_examples(self, examples):
        """
        Токенизирует и преобразует примеры в формат, совместимый с моделью
        вопрос-ответ.

        Параметры:
            examples (dict): батч примеров из датасета.

        Возвращает:
            dict: словарь с токенами и позициями начала и конца ответов.
        """
        questions = examples["question"]
        contexts = examples["context"]
        answers = examples["answers"]

        answer_starts = [ans["answer_start"][0] for ans in answers]
        answer_texts = [ans["text"][0] for ans in answers]
```

```
inputs = self.tokenizer(
    questions,
    contexts,
    max_length=384,
    truncation="only_second",
    stride=128,
    return_overflowing_tokens=True,
    return_offsets_mapping=True,
    padding="max_length",
)

offset_mapping = inputs.pop("offset_mapping")
overflow_to_sample_mapping = inputs.pop("overflow_to_sample_mapping")

start_positions = []
end_positions = []

for i, offsets in enumerate(offset_mapping):
    sample_idx = overflow_to_sample_mapping[i]
    start_char = answer_starts[sample_idx]
    end_char = start_char + len(answer_texts[sample_idx])

    sequence_ids = inputs.sequence_ids(i)
    context_start = sequence_ids.index(1)
    context_end = len(sequence_ids) - 1 - sequence_ids[::-1].index(1)

    if not (
        offsets[context_start][0]
        <= start_char
        < end_char
        <= offsets[context_end][1]
    ):
        start_positions.append(0)
        end_positions.append(0)
        continue

    token_start = context_start
    while token_start <= context_end and offsets[token_start][0] <=
start_char:
        token_start += 1
    start_positions.append(token_start - 1)

    token_end = context_end
    while token_end >= context_start and offsets[token_end][1] >=
end_char:
        token_end -= 1
    end_positions.append(token_end + 1)

inputs["start_positions"] = start_positions
inputs["end_positions"] = end_positions
inputs["overflow_to_sample_mapping"] = overflow_to_sample_mapping
return inputs

def get_tokenized_dataset(self, dataset):
    """
    Применяет токенизацию ко всему датасету.

    Параметры:
        dataset (DatasetDict): оригинальный датасет SberQuad.
```



### Продолжение Листинга А

```
        Возвращает:
            DatasetDict: токенизированный датасет.
        """
        return dataset.map(
            self._preprocess_examples,
            batched=True,
            remove_columns=dataset["train"].column_names,
            batch_size=100,
        )

class QAModelTrainer:
    """
    Класс для обучения модели на задаче вопрос-ответ с использованием Trainer
    API.
    """

    def __init__(
        self, model_name: str = MODEL_NAME, tokenizer=None, training_args: dict
        = None
    ):
        """
        Инициализация модели и аргументов тренировки.

        Параметры:
            model_name (str): имя модели.
            tokenizer: токенизатор, используемый в pipeline.
            training_args (dict): словарь с параметрами обучения.
        """
        self.model = AutoModelForQuestionAnswering.from_pretrained(model_name)
        self.tokenizer = tokenizer
        self.training_args = training_args or {
            "output_dir": "./results",
            "learning_rate": 2e-5,
            "per_device_train_batch_size": 8,
            "num_train_epochs": 3,
            "weight_decay": 0.01,
            "eval_strategy": "epoch",
            "save_strategy": "epoch",
            "logging_dir": "./logs",
            "fp16": torch.cuda.is_available(),
        }

    def setup_trainer(self, tokenized_dataset, original_dataset):
        """
        Создаёт Trainer с метриками, моделью и параметрами.

        Параметры:
            tokenized_dataset (DatasetDict): токенизированный датасет.
            original_dataset (DatasetDict): исходный SberQuad.

        Возвращает:
            Trainer: объект Trainer.
        """
        args = TrainingArguments(**self.training_args)
        squad_metric = evaluate.load("squad_v2")

    def compute_metrics(p):
        """Вычисляет метрики точности для задачи QA."""
        start_logits, end_logits = p.predictions
        start_pred = torch.argmax(torch.tensor(start_logits), dim=1).numpy()
        end_pred = torch.argmax(torch.tensor(end_logits), dim=1).numpy()
```

```

        formatted_predictions = []
        references = []

        for i in range(len(start_pred)):
            sample_idx = tokenized_dataset["validation"][i][
                "overflow_to_sample_mapping"
            ]
            original_sample = original_dataset["validation"][sample_idx]

            prediction_text = self.tokenizer.decode(
                tokenized_dataset["validation"][i]["input_ids"][
                    start_pred[i] : end_pred[i] + 1
                ],
                skip_special_tokens=True,
            )

            references.append(
                {
                    "id": str(original_sample["id"]),
                    "answers": {
                        "text": original_sample["answers"]["text"],
                        "answer_start":
original_sample["answers"]["answer_start"],
                    },
                }
            )

            formatted_predictions.append(
                {
                    "id": str(original_sample["id"]),
                    "prediction_text": prediction_text,
                    "no_answer_probability": 0.0,
                }
            )

        return squad_metric.compute(
            predictions=formatted_predictions, references=references
        )

    self.trainer = Trainer(
        model=self.model,
        args=args,
        train_dataset=tokenized_dataset["train"],
        eval_dataset=tokenized_dataset["validation"],
        data_collator=default_data_collator,
        compute_metrics=compute_metrics,
    )
    return self.trainer

def train(self):
    """Запуск обучения модели."""
    return self.trainer.train()

def save_model(self, path: str = "./sberquad_qa_model"):
    """
    Сохраняет модель и токенизатор в указанную директорию.

    Параметры:
        path (str): путь для сохранения.
    """

```

### Продолжение Листинга А

```
        self.model.save_pretrained(path)
        self.tokenizer.save_pretrained(path)

class QAPipeline:
    """
    Класс для выполнения предсказаний с помощью обученной модели в формате
    question-answering pipeline.
    """

    def __init__(self, model_path: str = "./sberquad_qa_model"):
        """
        Инициализирует модель и токенизатор из указанного пути.

        Параметры:
            model_path (str): путь до директории с моделью.
        """
        self.model = AutoModelForQuestionAnswering.from_pretrained(model_path)
        self.tokenizer = AutoTokenizer.from_pretrained(model_path)
        self.pipeline = pipeline(
            "question-answering",
            model = self.model,
            tokenizer = self.tokenizer,
            device = 0 if torch.cuda.is_available() else -1,
        )

    def predict(self, context: str, question: str):
        """
        Предсказывает ответ на вопрос по заданному контексту.

        Параметры:
            context (str): текстовый контекст.
            question (str): вопрос к контексту.

        Возвращает:
            dict: словарь с ответом и оценкой.
        """
        return self.pipeline(
            question=question,
            context=context,
            max_seq_len=384,
            doc_stride=128,
            handle_impossible_answer=True,
        )

if __name__ == "__main__":
    processor = QADatasetProcessor()
    dataset = processor.load_data()
    tokenized_dataset = processor.get_tokenized_dataset(dataset)

    trainer = QAModelTrainer(tokenizer=processor.tokenizer)
    trainer.setup_trainer(tokenized_dataset, dataset)
    trainer.train()
    trainer.save_model()

    qa_system = QAPipeline()
    context = (
        "Первые упоминания о строении человеческого тела встречаются в Древнем  
Египте."
    )
    question = "Где встречаются первые упоминания о строении человеческого  
тела?"
```

```

result = qa_system.predict(context, question)
print(f"Результирующий словарь: {result}")
print(f"Результат предсказания:")
print(f"Вопрос: {question}")
print(f"Ответ: {result['answer']}")
print(f"Точность: {result['score']:.2f}")

print("📊 Результаты на тестовой выборке:")
test_data = dataset["test"]

for i, sample in enumerate(test_data.select(range(10))):
    context = sample["context"]
    question = sample["question"]
    true_answer = sample["answers"]["text"][0]

    prediction = qa_system.predict(context, question)

    print(f"Пример {i + 1}")
    print(f"Вопрос: {question}")
    print(f"Контекст: {context}")
    print(f"Правильный ответ: {true_answer}")
    print(f"Предсказание: {prediction['answer']}")
    print(f"Точность: {prediction['score']:.2f}")

print("\n📊 Вычисляем метрики на всей тестовой выборке...")

squad_metric = evaluate.load("squad_v2")

test_data = dataset["test"]

predictions = []
references = []

for sample in test_data:
    context = sample["context"]
    question = sample["question"]
    true_answers = sample["answers"]["text"]

    result = qa_system.predict(context, question)

    predictions.append({
        "id": str(sample["id"]),
        "prediction_text": result["answer"],
        "no_answer_probability": 0.0
    })

    references.append({
        "id": str(sample["id"]),
        "answers": {
            "text": true_answers,
            "answer_start": sample["answers"]["answer_start"]
        }
    })

metrics = squad_metric.compute(predictions=predictions,
references=references)
print("✅ Метрики качества на тестовой выборке:")
print(f"🔴 Exact Match (EM): {metrics['exact']:.2f}")
print(f"🔴 F1 Score: {metrics['f1']:.2f}")

```

## Приложение Б.1

### Парсер веб-сайта с текстурами

*Листинг Б.1 – Код парсера веб-сайта с текстурами*

```
import time
import base64
import os
import shutil
import certifi
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.support.ui import WebDriverWait
from selenium.common.exceptions import NoSuchElementException, TimeoutException

WEAPONS = ['Zeus-x27', 'CZ75-Auto', 'Desert-Eagle', 'Dual-Berettas', 'Five-
Seven', 'Glock-18', 'P2000', 'P250', 'R8-Revolver', 'Tec-9', 'USP-S',
          'MAC-10', 'MP5-SD', 'MP7', 'MP9', 'PP-Bizon', 'P90', 'UMP-45',
          'MAG-7', 'Nova', 'Sawed-Off', 'XM1014',
          'M249', 'Negev',
          'AK-47', 'AUG', 'AWP', 'FAMAS', 'G3SG1', 'Galil-AR', 'M4A1-S',
'M4A4', 'SCAR-20', 'SG-553', 'SSG-08',
          'Karambit']

class ParserTextures:
    def __init__(self, base_url, texture_dir):
        self.chrome_options = webdriver.ChromeOptions()
        self.chrome_options.add_argument('--ignore-certificate-errors')
        self.chrome_options.add_argument('--ignore-ssl-errors')
        self.chrome_options.add_argument(f"--ssl-certificates-
path={certifi.where()}")
        # self.chrome_options.add_argument("--autoplay-policy=no-user-gesture-
required")
        self.chrome_options.add_argument('--disable-cache')
        # self.chrome_options.add_argument('--headless')
        # self.chrome_options.add_argument('--disable-gpu')
        # self.chrome_options.add_experimental_option("prefs",
{"profile.managed_default_content_settings.images": 2,
# "profile.man
aged_default_content_settings.stylesheet": 2,})
        self.chrome_options.add_experimental_option(
            "excludeSwitches", ['enable-automation', 'enable-logging'])
        self.base_url = base_url
        self.texture_dir = texture_dir
        self.browser = webdriver.Chrome(options=self.chrome_options)

    def __enter__(self):
        self.browser.maximize_window()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        try:
            if self.browser.service.process:
                self.browser.quit()
        except Exception as e:
            print(f"Error during shutdown: {e}")
        finally:
            if hasattr(self.browser, 'service'):
                self.browser.service.stop()
```

### Продолжение Листинга Б.1

```
def create_directory(self):
    if os.path.exists(self.texture_dir):
        shutil.rmtree(self.texture_dir)
    os.makedirs(self.texture_dir, exist_ok=True)

def get_all_weapons(self):
    self.create_directory()
    for weapon in WEAPONS:
        self.browser.get(f'{self.base_url}/weapon/{weapon}')
        skin_links = [el.get_attribute('href') for el in
self.browser.find_elements(By.CSS_SELECTOR, '.well.result-box.nomargin >
a:not(.nounderline)')]
        for skin_link in skin_links:
            self.parse_item_page(skin_link)
            time.sleep(0.5)

def parse_item_page(self, skin_link):
    self.browser.get(skin_link)
    try:
        WebDriverWait(self.browser, 3).until(
            EC.presence_of_element_located((By.CSS_SELECTOR, 'h1'))
        )
    except TimeoutException:
        print("\033[91m" + f"ERROR, msg=time out error, page={skin_link}" +
"\033[0m")
        return
    try:
        texture_button = self.browser.find_element(By.CSS_SELECTOR,
'a[href="#preview-texture"] > span.hidden-xs')
        except NoSuchElementException:
            print("\033[91m" + f"ERROR, msg=texture not found, page={skin_link}"
+ "\033[0m")
            return
        texture_button.click()
        texture_locator = (By.CSS_SELECTOR, 'div.active .skin-details-previews
a')
        WebDriverWait(self.browser,
2).until(EC.presence_of_all_elements_located(texture_locator))

        image_url =
self.browser.find_element(*texture_locator).get_attribute('href')

        js_script = f"""
        const done = arguments[0];
        fetch("{image_url}")
            .then(resp => resp.blob())
            .then(blob => {{
                const reader = new FileReader();
                reader.onloadend = () => done(reader.result);
                reader.readAsDataURL(blob);
            }})
            .catch(err => done(null));
        """
    try:
        base64_data = self.browser.execute_async_script(js_script)
    except Exception as e:
        print(f"\033[91m + JS execution error: {e}" + "\033[0m")
        return
    if base64_data is None:
        print(f"\033[91m" + "JS fetch failed or blocked: {image_url}" +
"\033[0m")
```

### Окончание Листинга Б.1

```
        return

    try:
        _, encoded = base64_data.split(",", 1)
        data = base64.b64decode(encoded)
        filename = os.path.basename(image_url.split('?')[0])
        filepath = os.path.join(self.texture_dir, filename)
        with open(filepath, 'wb') as f:
            f.write(data)
        print("\033[92m" + f"OK, texture saved: {skin_link}" + "\033[0m")
    except Exception as e:
        print(f"\033[91m" + "Error saving file: {e}, url={image_url}" +
              "\033[0m")

    print("\033[92m" + f"OK, page={skin_link}" + "\033[0m")

if __name__ == "__main__":
    start = time.perf_counter()
    with ParserTextures('https://stash.clash.gg', texture_dir='textures') as
    parser:
        parser.get_all_weapons()
    print(f'Время выполнения скрипта: {time.perf_counter() - start} секунд')
```



## Приложение Б.2

### Реализация генеративно-сопоставительной сети

*Листинг Б.2 – Код реализации генеративно-сопоставительной сети*

```
import os
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader
from PIL import Image
from typing import Optional, Callable

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
os.makedirs('gan_textures', exist_ok=True)

class TexturesDataset(Dataset):
    """Кастомный датасет для загрузки текстурных изображений из папки."""

    def __init__(self, root_dir: str, transform: Optional[Callable] = None) -> None:
        """
        Args:
            root_dir (str): Путь к директории с изображениями.
            transform (Callable, optional): Трансформации, применяемые к
            изображениям.
        """
        self.paths = sorted([
            os.path.join(root_dir, f)
            for f in os.listdir(root_dir)
            if f.lower().endswith(('.png', '.jpg', 'jpeg', 'bmp'))
        ])
        self.transform = transform

    def __len__(self) -> int:
        """Возвращает количество изображений в датасете."""
        return len(self.paths)

    def __getitem__(self, idx: int) -> torch.Tensor:
        """Загружает и возвращает одно изображение по индексу."""
        img = Image.open(self.paths[idx]).convert('RGB')
        if self.transform:
            img = self.transform(img)
        return img

transform = transforms.Compose([
    transforms.Resize((1024,1024)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,)*3, (0.5,)*3),
])

dataset = TexturesDataset('textures', transform)
dataloader = DataLoader(dataset, batch_size=8, shuffle=True,
                        num_workers=0, pin_memory=False)

def weights_init(m):
    if isinstance(m, (nn.Conv2d, nn.Linear)):
        nn.init.normal_(m.weight, 0.0, 0.02)
```

```

        if m.bias is not None:
            nn.init.zeros_(m.bias)
    elif isinstance(m, nn.BatchNorm2d):
        if m.weight is not None:
            nn.init.normal_(m.weight, 1.0, 0.02)
            nn.init.zeros_(m.bias)
    elif isinstance(m, nn.ConvTranspose2d):
        if m.weight is not None:
            nn.init.normal_(m.weight, 0.0, 0.02)
        if m.bias is not None:
            nn.init.zeros_(m.bias)

class Generator(nn.Module):
    """Генератор изображений для GAN."""

    def __init__(self, latent_dim: int = 100) -> None:
        """
        Args:
            latent_dim (int): Размерность латентного вектора.
        """
        super().__init__()
        self.latent_dim = latent_dim
        self.fc = nn.Linear(latent_dim, 512*4*4)

        def up(in_c: int, out_c: int) -> nn.Sequential:
            """Строит блок апсемплинга."""
            return nn.Sequential(
                nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=False),
                nn.Conv2d(in_c, out_c, 1),
                nn.InstanceNorm2d(out_c),
                nn.ReLU(True),
                nn.Conv2d(out_c, out_c, 3, padding=1),
                nn.InstanceNorm2d(out_c),
                nn.ReLU(True),
            )

        self.net = nn.Sequential(
            self.fc,
            nn.ReLU(True),
            nn.Unflatten(1, (512, 4, 4)),
            up(512, 512), # 4 → 8
            up(512, 256), # 8 → 16
            up(256, 256), # 16 → 32
            up(256, 128), # 32 → 64
            up(128, 64), # 64 → 128
            up(64, 32), # 128 → 256
            up(32, 16), # 256 → 512
            nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=False), # 512 → 1024
            nn.Conv2d(16, 3, 3, padding=1),
            nn.Tanh()
        )

        def forward(self, z: torch.Tensor) -> torch.Tensor:
            """Прямой проход генератора."""
            return self.net(z)

class Discriminator(nn.Module):
    """Дискриминатор для оценки реальности изображений."""

```

## Продолжение Листинга Б.2

```
def __init__(self) -> None:
    super().__init__()

    def block(in_c: int, out_c: int) -> nn.Sequential:
        """Строит сверточный блок."""
        return nn.Sequential(
            nn.Conv2d(in_c, out_c, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.25)
        )

    self.features = nn.Sequential(
        block(3, 64), # 1024 → 512
        block(64, 128), # 512 → 256
        block(128, 256), # 256 → 128
        block(256, 512), # 128 → 64
    )
    self.pool = nn.AdaptiveAvgPool2d(1)
    self.classifier = nn.Sequential(
        nn.Flatten(),
        nn.Linear(512, 1),
        nn.Sigmoid()
    )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Прямой проход дискриминатора."""
        x = self.features(x)
        x = self.pool(x)
        return self.classifier(x)

latent_dim = 128
epochs = 200
G = Generator(latent_dim).to(device)
D = Discriminator().to(device)
G.apply(weights_init)
D.apply(weights_init)

criterion = nn.BCELoss()
opt_G = optim.Adam(G.parameters(), lr=2e-4, betas=(0.5, 0.999))
opt_D = optim.Adam(D.parameters(), lr=1e-4, betas=(0.5, 0.999))

g_losses, d_losses = [], []
real_probs, fake_probs = [], []

for epoch in range(1, epochs + 1):
    ep_g, ep_d = 0.0, 0.0
    ep_r, ep_f = 0.0, 0.0
    nb = len(dataloader)

    for i, real in enumerate(dataloader, 1):
        real = real.to(device)
        b = real.size(0)

        valid = torch.full((b, 1), 0.9, device=device)
        fake_lbl = torch.full((b, 1), 0.1, device=device)

        opt_D.zero_grad()
        out_r = D(real)
        loss_r = criterion(out_r, valid)

        z = torch.randn(b, latent_dim, device=device)
```

```

fake = G(z).detach()
out_f = D(fake)
loss_f = criterion(out_f, fake_lbl)

d_loss = 0.5*(loss_r + loss_f)
d_loss.backward()
opt_D.step()

for _ in range(2):
    opt_G.zero_grad()
    z2 = torch.randn(b, latent_dim, device=device)
    gen = G(z2)
    out_gen = D(gen)
    g_loss = criterion(out_gen, valid)
    g_loss.backward()
    opt_G.step()

ep_d += d_loss.item()
ep_g += g_loss.item()
ep_r += out_r.mean().item()
ep_f += out_f.mean().item()

print(f"[{epoch:03d}/{epochs}]")
      f" [{i:03d}/{nb:03d}]"
      f" D_loss:{d_loss:.4f}"
      f" G_loss:{g_loss:.4f}"
      f" R:{out_r.mean().item():.2f}"
      f" F:{out_f.mean().item():.2f}")

d_losses.append(ep_d/nb)
g_losses.append(ep_g/nb)
real_probs.append(ep_r/nb)
fake_probs.append(ep_f/nb)

save_dir = os.path.join('gan_textures', f'epoch{epoch}')
os.makedirs(save_dir, exist_ok=True)
with torch.no_grad():
    samp_z = torch.randn(32, latent_dim, device=device)
    samples = G(samp_z).cpu()
    for idx, img in enumerate(samples, 1):
        torchvision.utils.save_image(img,
                                     os.path.join(save_dir, f"{idx:02d}.png"),
                                     normalize=True, value_range=(-1, 1))

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(g_losses, '-o', label='G Loss')
plt.plot(d_losses, '-o', label='D Loss')
plt.title('Loss per Epoch')
plt.xlabel('Epoch'); plt.legend(); plt.grid(True)
plt.subplot(1, 2, 2)
plt.plot(real_probs, '-o', label='D(real)')
plt.plot(fake_probs, '-o', label='D(fake)')
plt.title('D Outputs')
plt.xlabel('Epoch'); plt.legend(); plt.grid(True)
plt.tight_layout()
plt.savefig(os.path.join('gan_textures', 'metrics.png'))
plt.close()

print("Done. Все результаты – в папке gan_textures/")

```

## Приложение В

### Реализации графовой сети

#### *Листинг В – Код реализации графовой сети*

```
import torch
import torch.nn.functional as F
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
from typing import List, Dict
from torchinfo import summary
from torch_geometric.utils import to_networkx, degree
from torch_geometric.data import Data
from torch_geometric.datasets import Planetoid
from torch_geometric.nn import GCNConv
from torch_geometric.transforms import NormalizeFeatures

class DatasetManager:
    def __init__(self, dataset_name: str = "Cora", root_dir: str = "data") ->
None:
        """
        Загружает Planetoid-датасет и нормализует признаки.

        Параметры:
            dataset_name (str): Название датасета.
            root_dir (str): Путь к директории для хранения данных.
        """
        self.dataset = Planetoid(
            root=f"{root_dir}/{dataset_name}",
            name=dataset_name,
            transform=NormalizeFeatures(),
        )
        self.data: Data = self.dataset[0]

    def summary(self) -> None:
        """Выводит основную информацию о графе в формате таблицы."""
        d = self.data
        deg = degree(d.edge_index[0], d.num_nodes).cpu().numpy()

        metrics = {
            "Nodes": d.num_nodes,
            "Edges": d.num_edges,
            "Node features": d.num_node_features,
            "Classes": self.dataset.num_classes,
            "Train nodes": int(d.train_mask.sum()),
            "Validation nodes": int(d.val_mask.sum()),
            "Test nodes": int(d.test_mask.sum()),
            "Avg degree": float(deg.mean()),
            "Min degree": int(deg.min()),
            "Max degree": int(deg.max()),
        }

        df = pd.DataFrame.from_dict(metrics, orient="index", columns=["Value"])
        df.index.name = "Metric"
        print("\033[92m" + "Метрики графа:" + "\033[0m")
        print(df.to_markdown())

    def label_summary(self) -> None:
        """Выводит распределение классов в графе и строит столбчатую
        диаграмму."""
```

```

cora_label_names: Dict[int, str] = {
    0: "Case_Based",
    1: "Genetic_Algorithms",
    2: "Neural_Networks",
    3: "Probabilistic_Methods",
    4: "Reinforcement_Learning",
    5: "Rule_Learning",
    6: "Theory",
}

labels = self.data.y.cpu().numpy()
counts = pd.Series(labels).value_counts().sort_index()

df = pd.DataFrame(
    {
        "Label ID": counts.index,
        "Label Name": [cora_label_names[i] for i in counts.index],
        "Count": counts.values,
    }
)

print("\033[92m" + "Метки классов:" + "\033[0m")
print(df.to_markdown(index=False))

plt.figure(figsize=(8, 5))
plt.bar(df["Label Name"], df["Count"], color="skyblue")
plt.xticks(rotation=45, ha="right")
plt.xlabel("Label Name")
plt.ylabel("Number of Nodes")
plt.title("Distribution of Classes in Cora Dataset")
plt.tight_layout()
plt.show()

def show_sample_nodes(self, n: int = 10) -> None:
    """
    Показывает первые n вершин графа: метку и часть признаков.

    Параметры:
        n (int): Количество узлов для отображения.
    """
    d = self.data
    print("\033[92m" + f"Первые {min(n, d.num_nodes)} вершин графа:" +
          "\033[0m")
    for i in range(min(n, d.num_nodes)):
        feat = d.x[i].tolist()
        label = int(d.y[i])
        print(f"Node {i:4d} | label={label} | features={feat[:10]}...")

def degree_stats(self, bins: int = 50) -> None:
    """
    Строит гистограмму распределения степеней вершин.

    Параметры:
        bins (int): Количество столбцов в гистограмме.
    """
    deg = degree(self.data.edge_index[0], self.data.num_nodes)
    plt.figure(figsize=(6, 4))
    plt.hist(deg.cpu().numpy(), bins=bins)
    plt.title("Degree Distribution")
    plt.xlabel("Degree")
    plt.ylabel("Count")

```

```
plt.show()

def plot_graph(self, n_nodes: int = 200) -> None:
    """
    Строит визуализацию подграфа первых n_nodes вершин.

    Параметры:
        n_nodes (int): Количество узлов для визуализации.
    """
    sub_data = self.data.subgraph(torch.arange(min(n_nodes,
self.data.num_nodes)))
    G = to_networkx(sub_data, to_undirected=True)
    plt.figure(figsize=(6, 6))
    pos = nx.spring_layout(G, seed=42)
    nx.draw_networkx_nodes(G, pos, node_size=20)
    nx.draw_networkx_edges(G, pos, alpha=0.5)
    plt.title(f"Subgraph of first {n_nodes} nodes")
    plt.axis("off")
    plt.show()

class TrainingManager:
    def __init__(
        self,
        model: torch.nn.Module,
        data: Data,
        optimizer: torch.optim.Optimizer,
        criterion: torch.nn.Module,
        max_epochs: int = 200,
        patience: int = 20,
    ) -> None:
        """
        Инициализирует менеджер обучения для модели.

        Параметры:
            model (torch.nn.Module): Модель GNN.
            data (Data): Графовые данные.
            optimizer (torch.optim.Optimizer): Оптимизатор.
            criterion (torch.nn.Module): Функция потерь.
            max_epochs (int): Максимальное количество эпох обучения.
            patience (int): Число эпох без улучшения для ранней остановки.
        """
        self.model = model
        self.data = data
        self.opt = optimizer
        self.crit = criterion
        self.max_epochs = max_epochs
        self.patience = patience

        self.train_losses: List[float] = []
        self.train_accs: List[float] = []
        self.val_accs: List[float] = []
        self.test_accs: List[float] = []

    def train_epoch(self) -> float:
        """Выполняет одну эпоху обучения и возвращает loss."""
        self.model.train()
        self.opt.zero_grad()
        out = self.model(self.data.x, self.data.edge_index)
        loss = self.crit(out[self.data.train_mask],
self.data.y[self.data.train_mask])
        loss.backward()
```

### Продолжение Листинга В

```
self.opt.step()
return loss.item()

@torch.no_grad()
def evaluate(self, mask: torch.Tensor) -> float:
    """Оценивает точность модели на заданной маске узлов."""
    self.model.eval()
    out = self.model(self.data.x, self.data.edge_index)
    pred = out.argmax(dim=1)
    correct = (pred[mask] == self.data.y[mask]).sum().item()
    total = mask.sum().item()
    return correct / total

def train(self) -> None:
    """Запускает процесс обучения модели с ранней остановкой."""
    best_val = 0
    patience_ctr = 0

    print("\033[92m" + "Обучение модели..." + "\033[0m")
    for epoch in range(1, self.max_epochs + 1):
        loss = self.train_epoch()
        train_acc = self.evaluate(self.data.train_mask)
        val_acc = self.evaluate(self.data.val_mask)
        test_acc = self.evaluate(self.data.test_mask)

        self.train_losses.append(loss)
        self.train_accs.append(train_acc)
        self.val_accs.append(val_acc)
        self.test_accs.append(test_acc)

        print(
            f"Epoch {epoch:03d} | Loss: {loss:.4f} "
            f"| Train: {train_acc:.4f} "
            f"| Val: {val_acc:.4f} "
            f"| Test: {test_acc:.4f}"
        )

        if val_acc > best_val:
            best_val = val_acc
            patience_ctr = 0
        else:
            patience_ctr += 1

        if patience_ctr >= self.patience:
            print(f"Early stopping at epoch {epoch}")
            break

def plot_loss(self) -> None:
    """Строит график изменения loss от числа эпох."""
    plt.figure(figsize=(6, 4))
    plt.plot(self.train_losses)
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training Loss over Epochs")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

def plot_accuracy(self) -> None:
    """Строит график изменения accuracy для train, val и test."""
    plt.figure(figsize=(6, 4))
```



### Окончание Листинга В

```
plt.plot(self.train_accs, label="Train")
plt.plot(self.val_accs, label="Val")
plt.plot(self.test_accs, label="Test")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Accuracy over Epochs")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

class GCN(torch.nn.Module):
    def __init__(self, hidden_channels: int) -> None:
        """Определяет двухслойную GCN-сеть."""
        super().__init__()
        self.conv1 = GCNConv(manager.data.num_node_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, manager.dataset.num_classes)
        self.dropout = 0.5

    def forward(self, x: torch.Tensor, edge_index: torch.Tensor) ->
torch.Tensor:
    """Прямое распространение модели."""
    x = self.conv1(x, edge_index)
    x = F.relu(x)
    x = F.dropout(x, p=self.dropout, training=self.training)
    x = self.conv2(x, edge_index)
    return F.log_softmax(x, dim=1)

manager = DatasetManager(dataset_name="Cora", root_dir="data")
manager.summary()
manager.label_summary()
manager.show_sample_nodes(10)
manager.degree_stats()
manager.plot_graph(n_nodes=200)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = GCN(hidden_channels=16).to(device)
manager.data = manager.data.to(device)

print("\033[92m" + "Архитектура и число параметров:" + "\033[0m")
summary(
    model,
    input_data=(manager.data.x, manager.data.edge_index),
    col_names=("input_size", "output_size", "num_params"),
    depth=6,
)

optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)
criterion = torch.nn.CrossEntropyLoss()
trainer = TrainingManager(
    model, manager.data, optimizer, criterion, max_epochs=200, patience=20
)
trainer.train()

trainer.plot_loss()
trainer.plot_accuracy()

final_test_acc = trainer.test_accs[-1]
print("\033[92m" + f"Final Test Accuracy: {final_test_acc:.4f}" + "\033[0m")
```

## Приложение Г.1

### Использование предобученной модели

*Листинг Г.1 – Предобученная модель трансформера*

```
import torch
from transformers import pipeline, AutoTokenizer,
AutoModelForSequenceClassification
from datasets import load_dataset
from tqdm import tqdm

def main():
    dataset = load_dataset("sms_spam")["train"]

    model_name = "distilbert-base-uncased"
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForSequenceClassification.from_pretrained(model_name,
num_labels=2)

    classifier = pipeline(
        "text-classification",
        model=model,
        tokenizer=tokenizer,
        device=0 if torch.cuda.is_available() else -1,
        truncation=True,
        max_length=128,
    )

    correct = 0
    total = len(dataset)

    for example in tqdm(dataset, desc="Evaluating"):
        text = example["sms"]
        true_label = example["label"]

        prediction = classifier(text)[0]
        predicted_label = 1 if prediction["label"] == "LABEL_1" else 0
        correct += (predicted_label == true_label)

    accuracy = correct / total
    print(f"\n\033[92mТочность: {correct}/{total} = {accuracy:.4f}\033[0m")

if __name__ == "__main__":
    main()
```

## Приложение Г.2

### Реализация трансферного обучения

*Листинг Г.2 – Код реализации трансферного обучения*

```
import torch
from torch.utils.data import DataLoader
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    DataCollatorWithPadding,
    Trainer,
    TrainingArguments,
)
from datasets import load_dataset, DatasetDict
import matplotlib.pyplot as plt

from typing import Any, Dict, Optional, Tuple

from collections import Counter
import pandas as pd

class DatasetManager:
    """
    Менеджер для загрузки, анализа и подготовки датасета для задачи детекции спама.
    """

    def __init__(
        self,
        dataset_name: str = "sms_spam",
        text_column: str = "sms",
        validation_split: float = 0.2,
        test_split: float = 0.1,
        seed: int = 42,
    ) -> None:
        """
        Менеджер для загрузки, анализа и подготовки датасета для задачи детекции спама.

        Параметры:
            dataset_name (str): Название датасета.
            text_column (str): Колонка с текстом.
            validation_split (float): Доля валидационной выборки.
            test_split (float): Доля тестовой выборки.
            seed (int): Сид для воспроизводимости.
        """
        self.text_column: str = text_column
        raw = load_dataset(dataset_name)

        if "test" not in raw:
            train_temp = raw["train"].train_test_split(
                test_size=test_split + validation_split, seed=seed
            )

            temp = train_temp["train"].train_test_split(
                test_size=test_split / (test_split + validation_split),
                seed=seed
            )

            self.dataset = DatasetDict(
```

```

        {
            "train": train_temp["train"],
            "validation": temp["train"],
            "test": temp["test"],
        }
    )
    else:
        self.dataset = raw

    self.tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

    def summary(self) -> None:
        """
        Выводит статистику по датасету:
        1. Распределение примеров между train/validation/test
        2. Распределение классов в обучающей выборке
        """
        sizes = {split: len(ds) for split, ds in self.dataset.items()}
        labels = list(self.dataset["train"]["label"])
        dist = Counter(labels)

        df1 = pd.DataFrame.from_dict(sizes, orient="index", columns=["Samples"])
        df2 = pd.DataFrame({"Label": list(dist.keys()), "Count":
list(dist.values())})

        print("\033[92mDataset Splits:\033[0m")
        print(df1.to_markdown())
        print("\n\033[92mClass Distribution (train):\033[0m")
        print(df2.to_markdown(index=False))

    def show_samples(self, n: int = 5) -> None:
        """
        Отображает первые n записей с текстом и меткой.

        Параметры:
            n (int): число примеров.
        """
        ds = self.dataset["train"]
        print(f"\033[92mFirst {min(n, len(ds))} samples:\033[0m")
        for idx in range(min(n, len(ds))):
            example = ds[idx]
            label = example["label"]
            text = example[self.text_column]
            print(f"Sample {idx+1:2d}: [{label}] {text}")

    def preprocess(
        self,
        max_length: int = 128,
    ) -> DatasetDict:
        """
        Токенизация текстов и добавление поля 'labels'.

        Параметры:
            max_length (int): максимальная длина последовательности.

        Возвращает:
            DatasetDict: токенизированный датасет с 'input_ids',
            'attention_mask', 'labels'.
        """

```

## Продолжение Листинга Г.2

```
def tokenize_fn(example: Dict[str, Any]) -> Dict[str, Any]:
    tokens = self.tokenizer(
        example[self.text_column],
        truncation=True,
        max_length=max_length,
    )
    tokens["labels"] = example["label"]
    return tokens

tokenized = self.dataset.map(
    tokenize_fn,
    batched=True,
    remove_columns=self.dataset["train"].column_names,
)
return tokenized

class SpamClassifier:
    """
    Обёртка для DistilBERT, обучаемая на задаче классификации спама.
    """

    def __init__(
        self,
        num_labels: int,
        output_dir: str = "./spam_model",
        epochs: int = 3,
        batch_size: int = 8,
        learning_rate: float = 2e-5,
    ) -> None:
        """
        Инициализация модели и параметров тренировки.

        Параметры:
        num_labels (int): количество классов.
        output_dir (str): директория для результатов.
        epochs (int): число эпох.
        batch_size (int): размер батча.
        learning_rate (float): скорость обучения.
        """
        self.tokenizer: Optional[AutoTokenizer] = None
        self.model = AutoModelForSequenceClassification.from_pretrained(
            "distilbert-base-uncased", num_labels=num_labels
        )
        self.args = TrainingArguments(
            output_dir=output_dir,
            num_train_epochs=epochs,
            per_device_train_batch_size=batch_size,
            per_device_eval_batch_size=batch_size,
            learning_rate=learning_rate,
            evaluation_strategy="epoch",
            save_strategy="epoch",
            logging_strategy="epoch",
            logging_dir=f"{output_dir}/logs",
            load_best_model_at_end=True,
            metric_for_best_model="accuracy",
        )
        self.trainer: Optional[Trainer] = None

    def compute_metrics(
        self,
        eval_pred: Tuple[Any, Any],
```

## Продолжение Листинга Г.2

```
) -> Dict[str, float]:
    """
    Вычисляет accuracy по предсказаниям.

    Параметры:
        eval_pred: кортеж (логиты, метки).
    Возвращает:
        Dict[str, float]: {'accuracy': value}.
    """
    logits, labels = eval_pred
    preds = logits.argmax(axis=-1)
    accuracy = (preds == labels).astype(float).mean().item()
    return {"accuracy": accuracy}

def train(
    self,
    tokenized_dataset: DatasetDict,
    tokenizer: AutoTokenizer,
) -> Trainer:
    """
    Запускает тренировку и валидацию модели.

    Параметры:
        tokenized_dataset: токенизированные 'train' и 'validation'.
        tokenizer: токенизатор для паддинга.
    Возвращает:
        Trainer: объект Trainer.
    """
    self.tokenizer = tokenizer
    data_collator = DataCollatorWithPadding(tokenizer=self.tokenizer)
    trainer = Trainer(
        model=self.model,
        args=self.args,
        train_dataset=tokenized_dataset["train"],
        eval_dataset=tokenized_dataset["validation"],
        tokenizer=self.tokenizer,
        data_collator=data_collator,
        compute_metrics=self.compute_metrics,
    )
    trainer.train()
    trainer.save_model(self.args.output_dir)
    self.trainer = trainer
    return trainer

def test(self, tokenized_dataset: DatasetDict) -> Dict[str, float]:
    """
    Прогоняет модель на тестовом наборе и возвращает метрики.

    Параметры:
        tokenized_dataset (DatasetDict): Датасет с токенизированным сплитом
'test'.

    Возвращает:
        Dict[str, float]: Метрики оценки.
    """
    print("\n\033[92mTesting on test split...\033[0m")
    metrics = self.trainer.evaluate(eval_dataset=tokenized_dataset["test"])
    test_accuracy = metrics.get("eval_accuracy", 0.0)
    print(f"\033[92mTest Accuracy: {test_accuracy:.4f}\033[0m")
    return metrics
```

## Продолжение Листинга Г.2

```
def plot_loss(self) -> None:
    """
    Строит график train_loss, eval_loss и test_loss по эпохам.
    """
    train_epochs, train_losses = [], []
    eval_epochs, eval_losses = [], []
    for record in self.trainer.state.log_history:
        if "epoch" in record:
            if (
                "loss" in record
                and "eval_loss" not in record
                and "test_loss" not in record
            ):
                train_epochs.append(record["epoch"])
                train_losses.append(record["loss"])
            if "eval_loss" in record:
                eval_epochs.append(record["epoch"])
                eval_losses.append(record["eval_loss"])
    plt.figure(figsize=(6, 4))
    plt.plot(train_epochs, train_losses, marker="o", label="Train Loss")
    plt.plot(eval_epochs, eval_losses, marker="o", label="Eval Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Loss over Epochs")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

def plot_accuracy(self) -> None:
    """
    Строит график train_accuracy, eval_accuracy и test_accuracy по эпохам.
    """
    eval_epochs, eval_accs = [], []
    for record in self.trainer.state.log_history:
        if "epoch" in record:
            if "eval_accuracy" in record:
                eval_epochs.append(record["epoch"])
                eval_accs.append(record["eval_accuracy"])
    plt.figure(figsize=(6, 4))
    plt.plot(eval_epochs, eval_accs, marker="o", label="Eval Acc")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title("Accuracy over Epochs")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

manager = DatasetManager(
    dataset_name="sms_spam",
    text_column="sms",
    validation_split=0.2,
    test_split=0.1,
    seed=42,
)
manager.summary()
manager.show_samples(n=10)
tokenized = manager.preprocess(max_length=128)

classifier = SpamClassifier(num_labels=2)
```

### *Окончание Листинга Г.2*

```
dummy_input_ids = torch.zeros((1, 128), dtype=torch.long)
dummy_attention = torch.ones((1, 128), dtype=torch.long)

trainer = classifier.train(
    tokenized_dataset=tokenized,
    tokenizer=manager.tokenizer,
)

classifier.plot_loss()
classifier.plot_accuracy()
classifier.test(tokenized)

print(
    "\033[92mTraining complete. Model and metrics saved to:\033[0m",
    classifier.args.output_dir,
)
```