Санкт–Петербургский государственный университет

Павлова Александра Алексеевна

Выпускная квалификационная работа

Разработка мобильного приложения и применение машинного обучения для создания игрового персонажа

Уровень образования: бакалавриат
Направление 02.03.01 «Математика и компьютерные науки»
Основная образовательная программа СВ.5152.2019
«Математика, алгоритмы и анализ данных»

Научный руководитель:
доцент, Факультет математики и компьютерных наук СПбГУ, к. ф.-м. н.
Авдюшенко Александр Юрьевич

Рецензент:
старший инженер ООО «МПГ АйТи Солюшнз»
Гардер Антон Владимирович

Санкт-Петербург
2023 г.

Содержание

Введение

На сегодняшний день настольные ролевые игры занимают особое место в игровой индустрии, они предоставляют игрокам возможность воплотиться в разнообразных фантастических персонажей и погрузиться в захватывающие приключения. Подземелья и Драконы (Dungeons & Dragons, D&D) является одной из самых популярных и известных игр этого жанра. И хотя она была придумана более 45 лет назад, она стала набирать популярность только в последнее десятилетие – на данный момент по 5 редакции правил играет более 50 млн игроков [1].

В контексте D&D персонаж – это виртуальный аватар игрока, через которого он взаимодействует с игровым миром и другими участниками. Каждый персонаж обладает уникальными характеристиками, такими как раса, класс, способности и черты, что создает огромное количество уникальных комбинаций. Поэтому процесс создания и развития персонажа может быть сложным и трудоемким, особенно для новичков. В связи с этим, мы решили разработать мобильное приложение, которое поможет игрокам упростить процесс создания, хранения и управления своими персонажами в D&D.

Мы выбрали мобильное приложение как платформу для нашего проекта по нескольким причинам. Во-первых, мобильные устройства, такие как смартфоны, являются практически постоянным спутником большинства людей, что делает их удобной и доступной платформой для использования приложений. Во-вторых, мобильное приложение позволяет игрокам иметь доступ к своим персонажам в любое время и в любом месте, что особенно важно для ролевых игр, где игровые сессии могут происходить в разных местах и в разное время.

Одной из особенностей нашего приложения является возможность генерации текста и изображений при помощи нейросетей. Мы использовали методы машинного обучения, способные автоматически генерировать уникальные предыстории и визуальные представления персонажей. Генерация текста позволяет игрокам получить увлекательные истории, описывающие прошлое их персонажей, а генерация изображений предоставляет визуальное представление персонажей, отражающее их уникальные черты, без необходимости рисовать или находить подходящую картинку в интернете.

Хотя генерация текста и изображений не является основной целью нашего приложения, она придает ему уникальность и расширяет возможности игроков в создании и представлении своих персонажей. Мы стремились создать инструмент, который поможет игрокам D&D насладиться процессом игры и проявить свою творческую натуру, предоставляя им удобные и интуитивно понятные функции для создания и управления персонажами.

Постановка задачи

Несмотря на то, что в пятой редакции D&D была предпринята попытка упростить систему, она все еще обладает значительным количеством правил, которые могут стать причиной путаницы и ошибок даже для опытных игроков. Существует огромное количество уникальных механик и способностей, и многие из них имеют сложные зависимости и нюансы. Уже на стадии создания персонажа первого уровня игроку приходится принимать десятки решений, что приводит к огромному числу потенциальных вариаций.

Полностью охватить все крайности правил D&D до сих пор не смог ни один из аналогов, и даже для охвата большинства правил требуются нестандартные архитектурные решения. Кроме того, помимо реализации back-end части, которая должна обеспечивать поддержку правил, необходимо создать front-end интерфейс, который упростит процесс создания персонажа и его дальнейшего использования.

В свете этих сложностей, мы осознавали необходимость создания мобильного приложения, которое бы помогло игрокам D&D насладиться игровым процессом и облегчило бы процесс создания персонажей. Именно здесь вступает в игру часть приложения, основанная на генеративных моделях и предлагающая не встречавшийся у аналогичных приложений функционал для генерации визуальных представлений персонажей и их предысторий. Приложение сочетает в себе модели, способные генерировать реалистичные изображения и увлекательные тексты, связанные с предысторией персонажа D&D.

Можно разбить вышеописанное на три основных задачи, каждая из которых состоит из нескольких подзадач:

- **Front-end:**
  Простой интерфейс для взаимодействия приложения с пользователями – получение информации от пользователя, отображение и редактирование этой информации.
  - Создание и редактирование персонажа (4 экрана): настройка характеристик, предыстория и раса, класс, заклинания.

  - Отображение информации о персонаже: действия, инвентарь, доступные заклинания, модификаторы характеристик и навыков, etc.

- **Back-end:**
  Хранение, обработка и передача данных о персонаже.

  - Разработка и реализация системы хранения всевозможных данных о персонаже, далее базы данных.
  - Разметка и интеграция огромного количества данных по классам, расам, способностям, заклинаниям.
  - Разработка и реализация архитектуры взаимодействия front-end и базы данных.
  - Реализация охватывающей все правила системы способностей персонажа.

- **Генеративные модели машинного обучения:**
  Генерация изображения и предыстории персонажа.

  - Исследование существующих решений и моделей, выбор наиболее оптимальных в контексте D&D.
  - Сбор, подготовка данных и дообучение выбранных моделей.
  - Оценка и настройка гиперпараметров.
  - Интеграция лучших моделей в мобильное приложение.

Это был совместный проект трёх разработчиков, так что выделим подзадачи, затронутые в данной работе:

- **Front-end:** дизайн и реализация навигации, векторная графика.
- **Генеративные модели машинного обучения:** вся подзадача.

1. Front-end

1.1. Дизайн и реализация кнопок навигации

Кнопки навигации являются важным элементом пользовательского интерфейса и должны быть ясными и наглядными для обеспечения удобства использования приложения. Они должны быть легко обнаруживаемыми и доступными на каждом экране приложения, чтобы пользователи могли быстро переключаться между основными функциональными разделами.

Каждый экран приложения требует разработки уникального дизайна, разметки и наполнения элементами интерфейса. Переходы между экранами требуют разработки графа навигации, который определяет, какая кнопка ведет на какой экран. Одной из основных задач является обеспечение удобства использования приложения. Необходимо тщательно продумать расположение кнопок навигации, их визуальный стиль и размер, чтобы обеспечить легкий доступ и интуитивно понятное перемещение между экранами.

Разработка front-end для мобильного приложения может столкнуться с проблемой повторения кода, особенно при наличии множества экранов с похожими элементами интерфейса. В случае необходимости изменить внешний вид одной кнопки, требуется обеспечить согласованность с общим дизайном приложения. Это может потребовать пройтись по всем экранам и обновить соответствующую кнопку, чтобы гарантировать единообразие и целостность пользовательского интерфейса.

В контексте разработки под Android, “style” (стиль) [3] - это набор атрибутов, определяющих внешний вид и поведение элементов пользовательского интерфейса (UI). Стили в Android позволяют задавать общие свойства для различных компонентов интерфейса, таких как кнопки, текстовые поля, фрагменты и т.д.

С помощью стилей можно установить различные атрибуты, такие как цвет, шрифт, размер, фон, границы, отступы и другие свойства элементов UI. Однажды определенный стиль может быть применен к одному или нескольким элементам интерфейса, что позволяет достичь единообразного внешнего вида и упростить обновление дизайна приложения.

Стили определяются в файле ресурсов XML, который обычно называется “styles.xml”. В этом файле можно определить свои собственные стили или использовать готовые стили, предоставляемые Android Framework. Каждый стиль имеет уникальное имя, которое можно использовать для применения к конкретному элементу UI через атрибут “style”.

Таким простым образом определяется общий стиль кнопки навигации с картинкой, от которой потом наследуется стиль кнопки “назад”:

```
<style name="Widget.DnD.ImageButton"
    parent="Widget.AppCompat.ImageButton">
    <item name="android:layout_width">48dp</item>
    <item name="android:layout_height">48dp</item>
    <item name="android:layout_margin">4dp</item>
    <item name="android:background">?backgroundColor</item>
    <item name="android:tint">?colorPrimaryDark</item>
</style>

<style name="Widget.DnD.ImageButton.Back">
    <item name="android:src">
    @drawable/ic_baseline_arrow_back_36
    </item>
</style>
```

В нашем случае стили позволяют задать общие атрибуты для кнопок, такие как цвет, размер, форма и внешний вид, что обеспечивает консистентность и узнаваемость интерфейса. Благодаря стилям, изменения в дизайне кнопок навигации могут быть легко внесены в одном месте - в определении стиля. Это упрощает процесс обновления и обеспечивает единообразный внешний вид кнопок на всех экранах приложения. Кроме того, стили можно повторно использовать на различных экранах, что способствует снижению дублирования кода и облегчает его обслуживание.

Например, таких изменений можно добиться поменяв две строчки в файле “styles.xml” – обновился цвет всех кнопок и виджет настроек, рис. 1- 2.

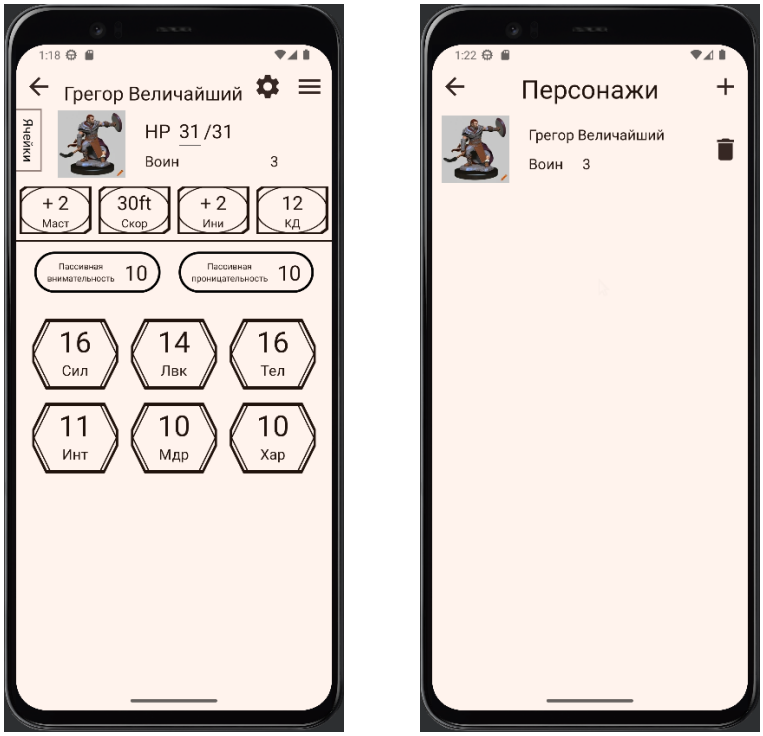

Рис. 1: Смена стиля. Оригинальный вариант.

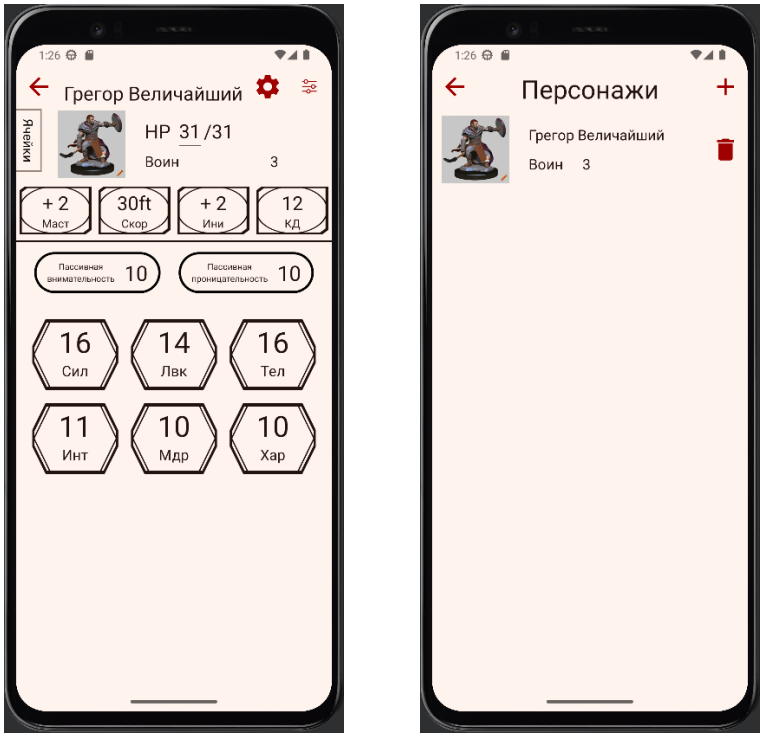

Рис. 2: Смена стиля. Изменили 2 строчики кода.

1.2. Векторная графика

Векторная графика в мобильных приложениях обычно реализуется с использованием специальных графических библиотек и технологий.

Основная идея векторной графики заключается в том, что изображение создается с помощью математических объектов, таких как линии, кривые и формы, определенных с использованием геометрических примитивов. В отличие от растровой графики, векторные изображения сохраняют свою четкость и качество независимо от размера или масштабирования, рис. 3.

В мобильных приложениях векторная графика может использоваться для создания различных элементов интерфейса, включая иконки, кнопки, линии и формы в границах текстовых полей. Она обычно реализуется с помощью векторных форматов файлов, таких как SVG (Scalable Vector Graphics) или используя графические API, такие как Canvas или OpenGL.

Преимущества использования векторной графики включают высокое качество и четкость изображений при любом масштабировании, возможность создания динамических и анимированных элементов, а также более эффективное использование памяти устройства.

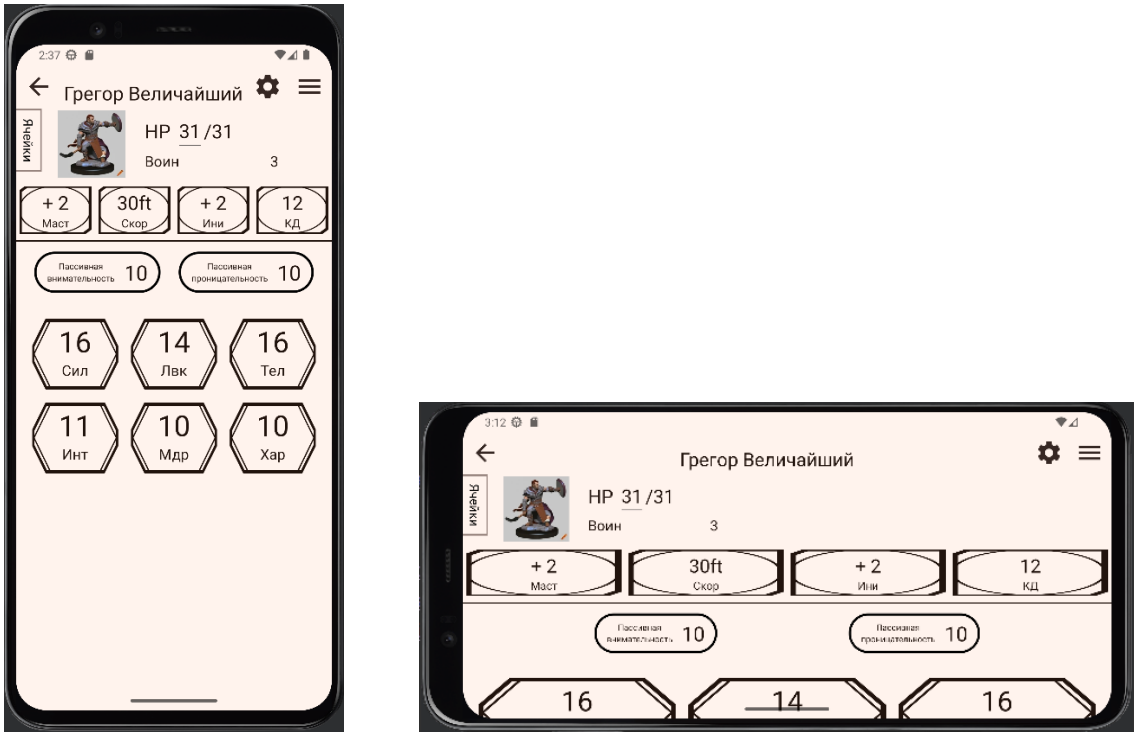

Рис. 3: Векторная графика подстраивается под размер экрана.

Векторная графика присустствует во многих элементах приложения. Например, она добавлена на экран загрузки, который реализован через новую удобную функциональность под названием “splash screen” [4], там сделана анимированнная векторная картинка, рис. 4.

Рис. 4: Векторная анимация.

1.3. Граф навигации

Граф навигации [2] в мобильном приложении представляет собой структуру, определяющую переходы между различными экранами или состояниями приложения. Граф навигации показывает связи между экранами и определяет, какие действия или события приводят к переходу с одного экрана на другой.

Граф навигации обычно представляется в виде дерева или ориентированного графа, где вершинами являются экраны, а ребра определяют возможные переходы между ними. Например, узел может быть домашним экраном, а ребра могут представлять кнопки или жесты, ведущие к другим экранам, таким как экраны настроек, экраны списка персонажей или экраны деталей персонажа.

Необходимо тщательно спроектировать структуру графа навигации, определить связи между экранами и их последовательность. Это требует учета функциональных и пользовательских требований приложения, чтобы обеспечить логическую и интуитивно понятную навигацию.

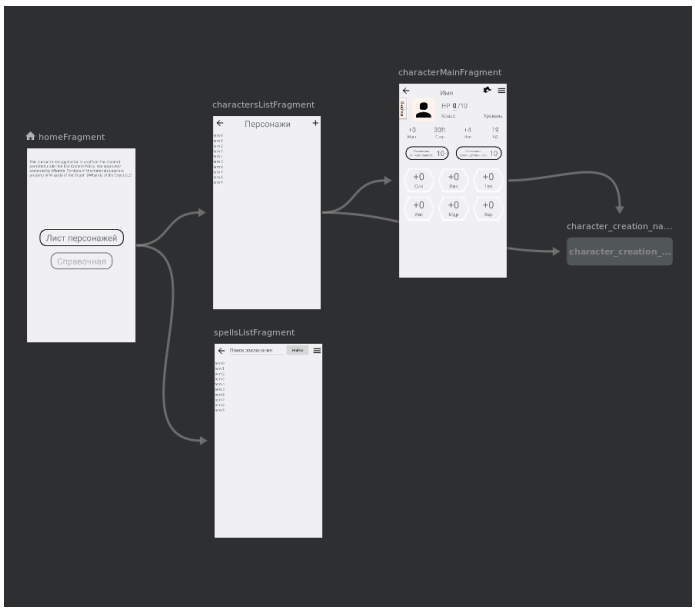

Рис. 5: Основной граф навигации

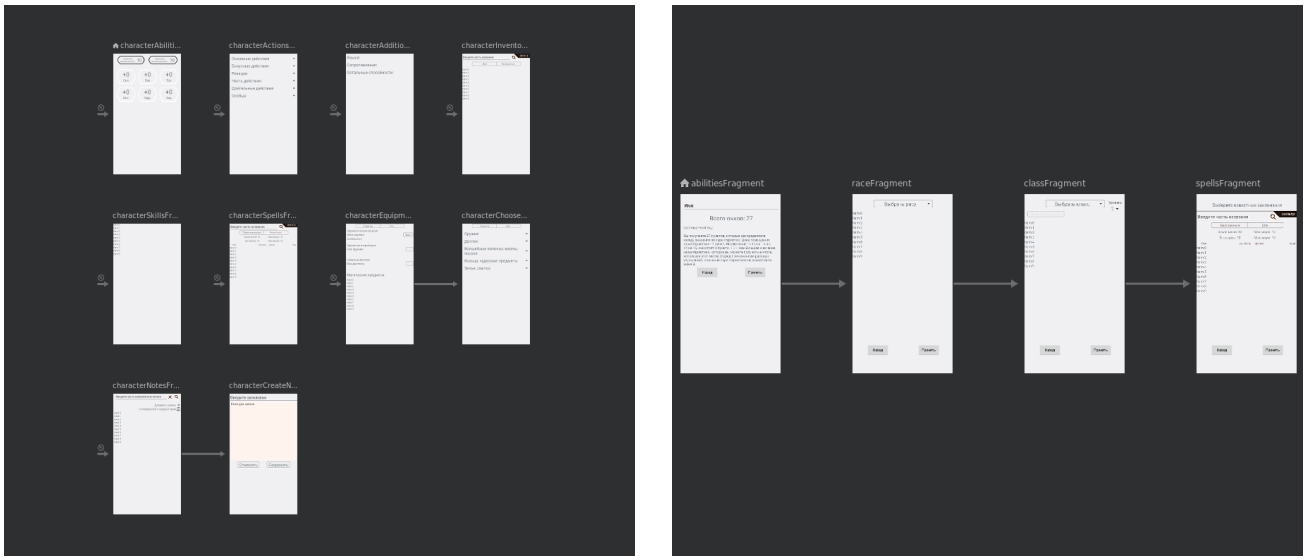

Рис. 6: Графы навигации отображения и создания/редактирования персонажа.

Наше приложение имеет сложную навигацию с различными состояниями или режимами, такими как режим создания персонажа, редактирования или просмотра, рис. 5- 6.

2. Генерация предысторий

Автоматическая генерация предысторий будет полезна игрокам в D&D, потому что поможет создавать интересных персонажей с уникальными историями, не тратя много времени на разработку их бэкграунда.

2.1. Выбор моделей

Очевидно, нужно начать с выбора наиболее подходящей модели для нашей задачи. Мы рассмотрели самые популярные на данный момент варианты нейронных сетей, сфокусированных на генерации текста.

- **LSTM (Long Short-Term Memory) [5]:**
  LSTM - это рекуррентная нейронная сеть (RNN), способная улавливать долгосрочные зависимости в последовательностях данных. Она может использоваться для генерации текста на основе заданных параметров персонажа.

  Устройство, рис. 7:

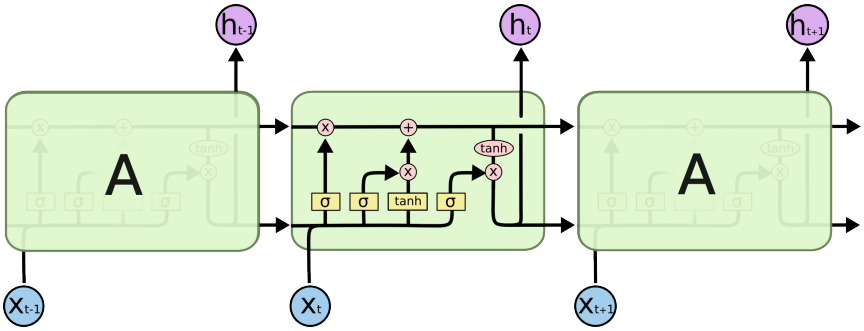

Рис. 7: Повторяемый модуль в модели LSTM.

  – *Входной слой:* Принимает входные данные, которые являются последовательными, например, последовательность слов или символов.
  – *Клеточное состояние (Cell State):* Клеточное состояние LSTM поддерживает и запоминает информацию в течение длительного времени. Оно пропускает или обновляет информацию, используя входные данные и предыдущие значения.

- *Фильтры ворот (Gates):* LSTM использует три типа ворот: входные ворота (Input Gate), ворота забывания (Forget Gate) и ворота вывода (Output Gate). Ворота регулируют поток информации, фильтруя, какая информация должна быть запомнена, забыта или передана на выход.
- *Скрытое состояние (Hidden State):* LSTM генерирует скрытое состояние, которое содержит информацию, передаваемую в следующий временной шаг или в качестве окончательного выхода модели.

Однако LSTM может иметь ограниченную способность улавливать более сложные контексты. Таким образом, более передовые модели предоставляют более мощные и точные результаты, способные улавливать сложные зависимости в тексте и создавать более семантически богатые и интересные предыстории.

- **GAN (Generative Adversarial Network) [6]:**
  GAN - это модель машинного обучения, состоящая из двух основных компонентов: генератора и дискриминатора. GAN использует конкурентную игру между этими двумя компонентами для генерации новых данных, которые максимально похожи на обучающий набор.

  Устройство, рис. 8:

  - *Генератор:* Генератор принимает на вход случайный шум или некоторый другой входной сигнал и генерирует новые данные, которые пытаются имитировать образцы из обучающего набора данных. Генератор является нейронной сетью, которая обучается создавать реалистичные выходные данные.
  - *Дискриминатор:* Дискриминатор принимает на вход данные, созданные генератором, а также реальные образцы из обучающего набора данных, и пытается классифицировать их как реальные или сгенерированные. Дискриминатор также является нейронной сетью, которая обучается различать реальные и сгенерированные данные.

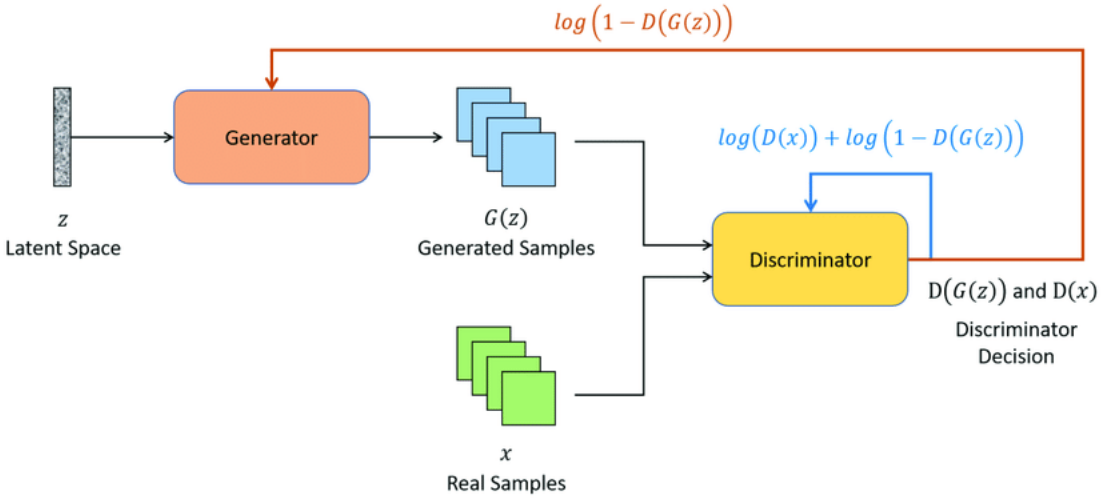

Рис. 8: Архитектура GAN.

Можно использовать GAN в совокупности с другой архитектурой (например, трансформеры, которые упомянуты далее). Но генерация текста с помощью GAN может быть сложной задачей, требующей большого объема данных и вычислительных ресурсов. Поэтому невыгодно тренировать их с нуля.

- **Transformers [7]:**
  Трансформеры - это тип архитектуры моделей глубокого обучения, который применяется для обработки последовательностей данных, таких как тексты. Они были представлены в работе “Attention is All You Need” [8] и стали важным прорывом в области обработки естественного языка.

  Устройство, рис. 9:

  – *Механизм внимания (Attention Mechanism):* Основная особенность трансформера - это механизм внимания, который позволяет модели фокусироваться на различных частях входных последовательностей при выполнении задачи. Механизм внимания позволяет модели учиться выделять и учитывать значимые контекстные зависимости в тексте.

– *Кодировщик и декодировщик (Encoder-Decoder):* Трансформер состоит из кодировщика и декодировщика. Кодировщик обрабатывает входные данные, а декодировщик генерирует соответствующие выходные данные. Каждый кодировщик и декодировщик состоит из множества слоев, называемых "трансформерными блоками которые выполняют операции внимания и преобразования данных.

– *Позиционное кодирование (Positional Encoding):* Поскольку трансформер не сохраняет порядок слов входной последовательности, позиционное кодирование добавляет информацию о позиции слов во входных данных, чтобы модель могла учитывать последовательность слов при обработке текста.

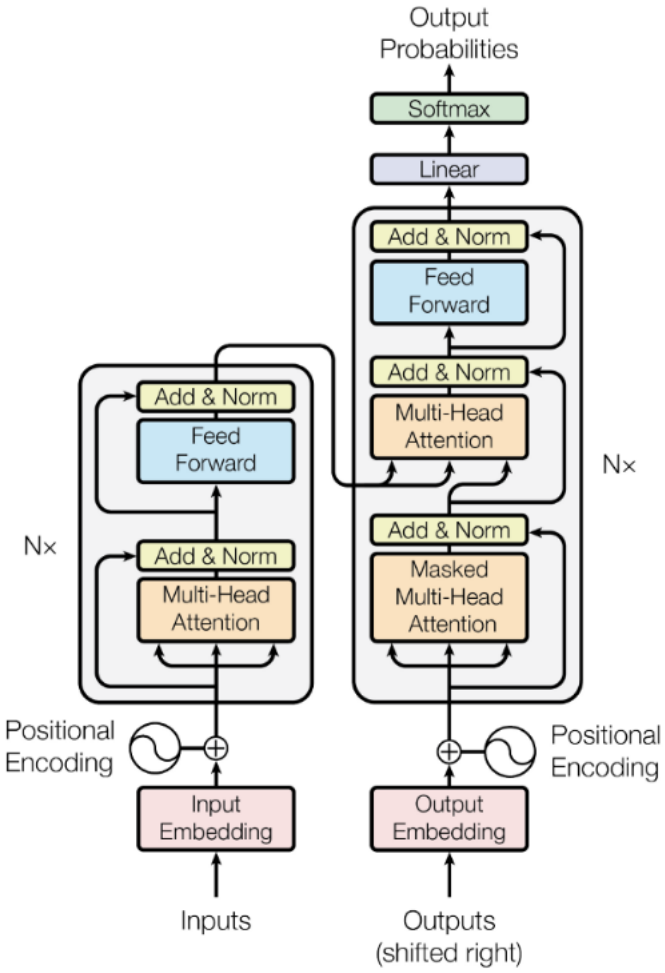

Рис. 9: Архитектура трансформера.

Трансформеры отлично подходят для генерации предысторий персонажей D&D на основе параметров, таких как раса, класс, имя и т.д. Они позволяют модели улавливать сложные зависимости в тексте, создавать более связные и семантически богатые предыстории.

Трансформеры также обучаются на больших объемах данных и предобучены на разнообразных данных, что дает им способность генерировать качественные и разнообразные тексты. Они способны учиться из общего знания и контекста, что делает их особенно полезными для генерации предысторий персонажей D&D, которые требуют согласованности и связности с игровым миром.

2.2. Генерация данных и дообучение

Трансформеры можно успешно дообучать на различных задачах и данных. Они обладают гибкой архитектурой, которая позволяет легко настраивать модель на новые задачи. В общем, трансформеры могут быть дообучены на различных текстовых задачах, требующих глубокого понимания контекста. Они позволяют адаптировать модель к специфическим требованиям и добиваться более точных и качественных результатов на новых данных.

Является ли в случае нашей задачи дообучение модели более выгодным подходом, чем обучение модели с нуля?

Предполагается, что у нас уже есть модель, которая была предобучена на больших объемах текстовых данных. Такие модели обладают обширными знаниями о языке и способностью генерировать качественные тексты. Дообучение модели позволяет учесть специфические требования нашей задачи, а именно генерацию предысторий персонажей D&D, используя параметры, такие как раса, класс, имя, etc.

Обучение модели с нуля может потребовать значительное количество времени, вычислительных ресурсов и большого объема данных. Вместо этого, дообучение позволяет использовать уже предобученные параметры модели и настроить их под нашу конкретную задачу. Дообучение модели дает нам больший контроль и возможность настройки модели на основе наших конкретных требований.

Таким образом, дообучение модели является предпочтительным в нашем случае, поскольку оно позволяет использовать существующие знания модели, экономить время и ресурсы, а также обеспечивает контроль и настройку под наши специфические задачи.

Следующий вопрос, с которым мы сталкиваемся – где взять данные для дообучения модели трансофрмера? Было рассмотрено несколько вариантов:

- **Существующие базы данных персонажей D&D:** Были изучены существующие базы данных персонажей D&D или веб-сайты, на которых представлены истории персонажей. На таких сайтах, как D&D Beyond, сообщества D&D на Reddit или различных форумах часто есть истории персонажей, предоставленные пользователями, которые можно взять или собрать вручную.

  У этого варианта есть как минимум два минуса. Во-первых, потребуется очень много времени, чтобы собрать базу предысторий персонажей, их сложно найти и привести к общему формату. Во-вторых, могут возникнуть проблемы с качеством текста, потому что мы не можем его гарантировать. Значит, потребуется ещё больше времени на вычитку. И наконец большинство этих баз данных англоязычные, а значит придётся переводить тексты вручную или вычитывать то, что выдал переводчик.

- **Исходные книги и модули D&D:** Книги правил D&D, исходники и модули приключений часто содержат примеры персонажей или NPC (неигровых персонажей) вместе с их историями. Можно было бы использовать их в качестве отправной точки или даже адаптировать их для создания разнообразной обучающей базы данных.

  Первый минус – ручная сборка и разметка данных, что опять же слишком трудоёмко. Второй минус – авторские права, не все существующие книги придерживаются то же политики, что Wizards of The Coast. И опять проблемы с переводом, потому что англоязычного контента по D&D в разы больше, и он настолько богаче и обширнее, что нельзя его пропустить.

- **Создание датасета вручную:** Можно потратить огромное количество ресурсов и времени, чтобы создать свой собственный набор данных, написав истории персонажей вручную самостоятельно, наняв писателей для создания разнообразных и увлекательных историй или же попросить русскоязычных игроков написать свои истории.

  Такой подход позволяет полностью контролировать набор данных и гарантирует, что он соответствует вашим конкретным требованиям. Но требует слишком много времени и ресурсов.

- **Генерация предысторий:** Допустим у нас получится собрать ограниченный набор реальных данных или не получится собрать ничего – есть опция расширить его синтетическими данными, сгенерированными одной из продвинутых языковых моделей (например GPT). Создание синтетических данных позволяет расширить обучающий набор и увеличить его разнообразие. Дополнительные данные могут помочь модели лучше обобщить и выявить более широкий спектр возможных вариантов и сценариев.

  Когда уже появится собственная дообученная модель трансформер, можно использовать ее собственные прогнозы для создания новых примеров. Это позволит модели углубить свое понимание задачи и уточнить свои предсказательные способности. Такое дообучение на сгенерированных данных может привести к улучшению результатов на целевой задаче.

  Важно отметить, что при генерации данных для дообучения следует учитывать, что качество и репрезентативность сгенерированных данных могут оказаться ниже, чем у реальных данных. Поэтому придётся потратить некоторое время, чтобы оценить и проверить результаты, а также применить соответствующие методы контроля качества данных и регуляризации для предотвращения переобучения модели на синтетических примерах.

Итого был выбран последний вариант – дополнение небольшого реального датасета при помощи синтетической генерации текстов. Далее приложена часть нашего кода на Python, который генерирует множество запросов к GPT-API и сохраняет их в локальный csv-файл. GPT-API это удобная форма запросов, сделанная OpneAI, которая добавила возможность для разработчиков интегрировать их генеративные модели в различные продукты.

```
for race in races:
  for class_type in classes:
    ... # Тут можно перебирать всевозможные парметры для генерации

    # Собираем запрос, включающий все эти параметры
    prompt = f"Раса: {race}\nКласс: {class_type}\n...Предыстория:"

    # Генерируем предысторию персонажа, используя ChatGPT
    response = openai.Completion.create(
      engine='text-davinci-003', # Тут можно поменять генеративную модел
      prompt=prompt,
      max_tokens=200, # Регулируем размер текста
      n=1,
      early_stopping=True,
      temperature=0.7
      # Тут были подобраны наиболее удобные для нас параметры
    )
    generated_backstory = response.choices[0].text.strip()

    # Добавляем в csv-файл
    ...
```

Далее следует выбор и дообучение модели-трансформера. Для этого процесса удобнее всего использовать готовую библиотеку *Transformers* [9], созданную сообществом Hugging Face.

Из большого списка разнообразных русскоязычных генеративных моделей, представленных в сообществе Hugging Face – лучше всего до дообучения себя показали:

- **rugpt3** – русскоязычная модель основанная на архитектуре GPT-2. Создана и опубликована в 2020 году командой разработчиков SberDevices, которая уже много лет разрабатывает умные устройства и виртуальных ассистентов.

- **mGPT** – мультилингвистическая модель основанная на архитектуре GPT-3. Тоже разработана SberDevices, но уже в 2022 году. На тот момент была первой в мире генеративной моделью, которая поддерживает такое количество языков (61, включая языки народов России и стран СНГ).

Был реализован скрипт на Python, который дообучает модель на наших собранных и сгенерированных данных. Рассмотрим его чуть подробнее:

Сначала нужно инициализировать выбранную модель и преобразовать данные в числовое представление, то есть токенизировать.

```
tokenizer = AutoTokenizer.from_pretrained('ai-forever/mGPT')
model = GPT2LMHeadModel.from_pretrained('ai-forever/mGPT')
model = model.to('cuda')
model.eval()

X = ... # Входные данные: имя, класс, раса и т.д.
y = ... # Соответствующие им предыстории

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
X_train_tokenized = tokenizer(X_train, *parameters*)
X_val_tokenized = tokenizer(X_val, *parameters*)
```

Далее нужна функция оценки метрик, чтобы было понятно, насколько мы смогли улучшить модель. Выбранные метрики:

- **Перплексия:**
  Перплексия является мерой "неожиданности"или "сложности"модели для исходных данных. Она рассчитывается на основе вероятности модели для исходных последовательностей. Модели с более низкой перплексией обычно считаются лучшими, так как они имеют более высокую вероятность для правильных предсказаний на тестовых данных. В контексте генеративных моделей, перплексия оценивает, насколько хорошо модель согласуется с обучающими данными и насколько она может генерировать последовательности, близкие к исходным.

- **BLEU (Bilingual Evaluation Understudy):**
  BLEU является метрикой оценки качества генеративной модели. Она оценивает сходство между сгенерированными последовательностями и эталонными последовательностями. BLEU вычисляет точность совпадения n-грамм (последовательностей из n токенов) между сгенерированными и эталонными последовательностями. Она принимает во внимание и точность совпадения отдельных слов, и точность совпадения последовательностей слов. Чем выше BLEU-метрика, тем лучше соответствие генерируемого текста и исходных эталонных данных.

```
def compute_metrics(p):
  references, predictions = p
  metrics = {}

  # Вычисление перплексии
  references_input_ids = tokenizer.batch_encode_plus(references,
    padding=True, truncation=True, return_tensors='pt')
  with torch.no_grad():
    loss = model(**references_input_ids).loss
    perplexity = torch.exp(loss)
  metrics['perplexity'] = perplexity.item()
```

```
    # Преобразование текстовых последовательностей в токены
    references_tokens=[reference.split() for reference in references]
    predictions_tokens=[prediction.split() for prediction in predictions]

    # Вычисление BLEU
    bleu_score = corpus_bleu(references_tokens, predictions_tokens)
    metrics['bleu'] = bleu_score

    return metrics
```

И наконец воспользуемся тренировочным циклом, встроенным в библиотеку *Transformers*. Из токенизированных данных сделаем PyTorch-датасет, чтобы передать его в класс *Trainer*, который потом произведёт несколько эпох дообучения модели.

```
args = TrainingArguments(
    output_dir="output",
    num_train_epochs=3,
    per_device_train_batch_size=8

)
trainer = Trainer(
    model=model,
    args=args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics
)
trainer.train()
```

Итоговая модель стала лучше понимать поставленную задачу и генерировать более связный текст.

2.3. Проблемы с размером

И в этот момент мы сталкиваемся с самой большой проблемой нейросетей, и ещё большей проблемой трансформеров – количество параметров модели.

Вес модели трансформера для генерации текста может значительно варьироваться в зависимости от различных факторов, таких как размерность модели, количество слоев, размер скрытого пространства, словарный запас и используемые параметры обучения. Обычно модели трансформера для генерации текста занимают от нескольких десятков мегабайт до нескольких гигабайт. Однако, более крупные модели, такие как GPT-3, могут весить несколько сотен гигабайт.

Обе рассмотренные ранее модели **ruGPT3** и **mGPT** – весят более 3Гб. Что это значит для нашего приложения?

Это значит, что перед разработчиками встаёт выбор. Наше приложение лучше своих аналогов по нескольким параметрам: ему не нужно подключение к интернету, оно занимает в разы меньше места.

- Первый вариант – это отказаться от изначального ограничения на вес приложения. Тогда можно будет загрузить веса нейросети при первоначальном скачивании приложения и пользоваться ей. Однако стоит ли делать шаг от 30 Мб к 3 Гб ради одной довольно специфической функциональности?

- Второй вариант – вместо включения полной модели в приложение, можно развернуть модель на удаленном сервере и обращаться к ней через API. В приложении можно реализовать логику отправки запросов к серверу для генерации предысторий персонажей. Это позволит уменьшить размер приложения, так как модель будет находиться на сервере, но потребуется подключение к интернету для взаимодействия с ней.

Очевидно, гораздо лучше было бы реализовать второй вариант, тем более основная функциональность приложения не зависит от подключения к интернету. А значит можно реализовать эту опцию так, чтобы она была доступна только, когда есть подключение к интернету.

Однако, одной из проблем второго подхода является сам сервер. На данный момент не существует всеобъемлющего бесплатного сервера, который бы предоставлял бесплатную и стабильную инфраструктуру для запуска модели трансформера весом более 3 Гб без ограничений. Однако, существуют различные облачные провайдеры, которые предоставляют услуги хостинга и вычислительные ресурсы для запуска моделей на удаленных серверах.

Некоторые из таких провайдеров предлагают бесплатные тарифы с ограниченными ресурсами, которые могут быть полезными для разработки и тестирования. Например, Google Cloud Platform (GCP) и Amazon Web Services (AWS) предлагают бесплатные тарифы для новых пользователей, где можно получить ограниченные вычислительные ресурсы для выполнения моделей. Однако, для работы с моделью размером более 3 Гб или для продукционного использования, возможно, потребуется платный тариф или более мощные вычислительные ресурсы.

Если нам потребуется стабильный сервер с большими вычислительными мощностями и возможностью запуска модели трансформера весом более 3 Гб без ограничений, нам, вероятно, придется рассмотреть коммерческие облачные услуги или аренду выделенного сервера.

Теперь же вернемся к начальному выбору модели. В конечном счёте у нас получилась относительно неплохо работающая генеративная модель, которую можно расположить на сервере и делать к ней запросы. Это похоже по функциональности на ChatGPT. И мы уже умеем работать с его API. Раз уж мы отказались от идеи полностью оффлайн приложения, рассмотрим идею использования ChatGPT.

Надо отметить, что использование GPT-API имеет несколько преимуществ по сравнению с запуском собственной модели на удалённом сервере:

- Использование GPT-API позволит нам избежать сложностей, связанных с настройкой, управлением и обслуживанием собственного сервера и инфраструктуры. OpenAI берет на себя ответственность за поддержку и обновление инфраструктуры.

- OpenAI постоянно работает над улучшением моделей и алгоритмов.

Значит получаем доступ к последним версиям модели без необходимости обновления и дообучения модели на собственном сервере.

- GPT 3.5-turbo является результатом многолетней работы множества высококвалифицированных специалистов в области NLP и машинного обучения, обученной на большом количестве данных и предназначенной для решения множества задач, что позволяет ему предоставлять качественные и точные ответы на широкий спектр вопросов. В том числе и на нашу задачу, на данный момент он работает лучше, чем наша дообученная модель.

- API предоставляет готовую инфраструктуру, готовую к масштабированию, и позволяет обрабатывать множество запросов параллельно. В будущем мы надеемся на большое количество пользователей, что потребует большой вычислительной мощности, API ChatGPT может легко масштабироваться для обработки этой нагрузки.

- Запуск и поддержка собственной модели на сервере может потребовать значительных вычислительных ресурсов, времени и затрат. Использование API позволит нам платить только за фактическое использование модели в виде оплачиваемых запросов, что может быть более экономически эффективным решением, особенно при небольшом или переменном объеме использования.

Получается, изначально мы планировали сделать полностью оффлайн приложение, обладающее функцией генерации предыстории персонажа. Поэтому мы рассмотрели множество различных архитектур нейронных сетей и остановили свой выбор на трансформерах. Мы попробовали использовать готовую предобученную модель, а после попытались её дообучить при помощи синтетически сгенерированных данных. Это привело к положительной динамике в генерации текста. Далее мы столкнулись с проблемой веса моделей, что привело нас к решению отказаться от полностью оффлайн приложения. Раз уже мы отказались от этой идеи – мы обратили внимание на ChatGPT API и приняли решение использовать его.

2.4. Интеграция в приложение

Теперь поговорим про интеграцию нейросети в мобильное приложение. Так как мы используем модель, расположенную на сервере – нужно подключить вызов API. В первую очередь нужно подключить разрешение на использование интернета. Это делается при помощи добавления двух строк в файл *AndroidManifest.xml*:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" /
```

Далее стоит обратить особое внимание на то, что доступ к продуктам OpenAI происходит при помощи уникального SSH-ключа, который не должен попасть в руки конечных пользователей. Ключ API предоставляет доступ к сервисам и ресурсам OpenAI от лица разработчика. Если ключ попадает в неправильные руки, злоумышленники могут использовать его для несанкционированного доступа к системам или проведения вредоносных действий.

Один из способов защиты – это маскировка ключа в переменной окружения при помощи библиотеки dotenv.

```
private val dotenv = dotenv {
    directory = "/assets"
    filename = "env"
}

private val key: String = dotenv["OPENAI_TOKEN"]
private val openai = OpenAI(key)
```

При использовании удаленного API возможны задержки при передаче запросов и получении ответов от сервера. Это может привести к проблемам с производительностью и использованием в реальном времени. Для улучшения отзывчивости приложения мы добавили использование асинхронных запросов/ответов и механизмы обработки ошибок и переподключения для обработки сетевых проблем.

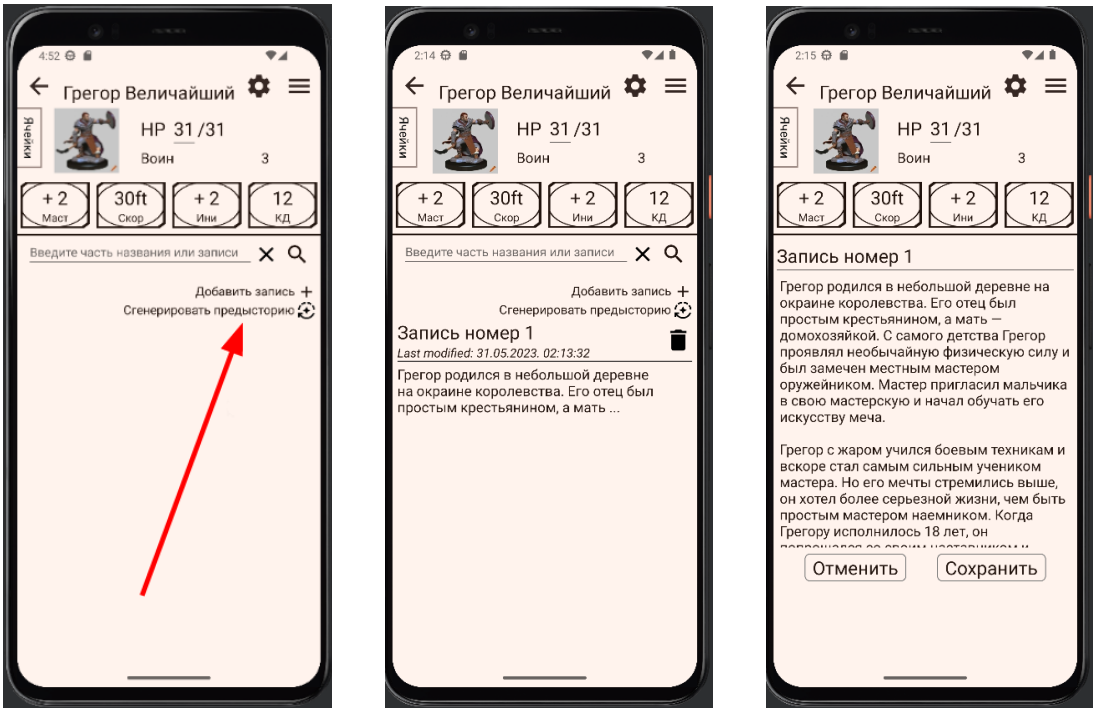

Рис. 10: Генерация предыстории.

В итоге мы добавили кнопку "сгенерировать предысторию"на экран с заметками о персонаже. При нажатии этой кнопки приложение формирует запрос, содержащий имя, расу, класс персонажа, а так же его основные навыки. Сгенерированный текст записывается в виде новой заметки и далее доступен для редактирования пользователем, рис. 10.

3. Генерация изображений

И последней уникальной функциональностью нашего приложения стала возможность сгенерировать изображение персонажа. Мы учитываем различные параметры персонажей, такие как раса, класс и черты, чтобы создать уникальные изображения. Таким образом, игроки могут легко визуализировать своих персонажей и делиться ими с другими игроками, добавляя более глубокий уровень вовлеченности в игровой процесс.

3.1. Поиск данных

До того как мы столкнулись с проблемой размера генеративных моделей при решении задачи генерации предыстории, мы занимались поиском данных для создания или дообучения модели для генерации изображений.

Один из подходов к созданию датасета для генерации картинок D&D персонажей - это использование уже существующих изображений, нарисованных художниками или созданных другими пользователями. В этом случае может потребоваться разрешение и согласие авторов на использование их работ в качестве части датасета.

Другой подход заключается в создании собственного датасета путем рисования или генерации персонажей с помощью инструментов редактирования изображений или специализированных генеративных моделей. Это может потребовать больше времени и усилий, но даст контроль над создаваемыми персонажами и гарантированное соблюдение авторских прав.

Оба подхода, к сожалению, ещё и включают в себя ручную разметку данных, что занимает слишком много времени.

Есть ли готовые датасеты? Единственное, что мы смогли найти – это небольшой (по меркам обучения новых моделей) датасет на HuggingFace. Он является частью большего проекта под названием Dungeons & Diffusion, в котором генеративная модель была дообчуена создавать изображения персонажей. Однако, эта модель весить более 2 Гб, а значит по вышеупомянутым причинам нам придётся использовать сервер или искать готовую модель с сервером.

3.2. Craiyon (DALL·E Mini)

Самым лучшим вариантом в данном случае является бесплатный сервис, предлагаемый библиотекой Craiyon. Единственный минус – у этого сервиса есть API только для Python, а приложение мы пишем на Kotlin. Можем ли мы решить эту проблему?

Существует несколько вариантов интеграции кода на Python в мобильное приложение. Самый популярный из них – это Chaquopy, которым мы и решили воспользоваться.

Его нужно добавить в *build.gradle* и запустить Python в MainActivity:

```
if (! Python.isStarted()) {
    Python.start(AndroidPlatform(this))
}
```

Далее остаётся только сделать front-end меню выбора сгенерированного фото и back-end связывающийся со скриптом на Python. Вот так выглядит взаимодействие с функцией *generate_images*:

```
...
private val module: PyObject
init {
    val py = Python.getInstance()
    module = py.getModule("generate")
}
private fun generatePhoto(s: String) {
    val fromPython: List<PyObject> =
                    module.callAttr("generate_images", s).asList()
    val list = fromPython.map { obj ->
        val bytes = obj.toJava(ByteArray::class.java)
        BitmapFactory.decodeByteArray(bytes, 0, bytes.size)
    }
    setupPhotoPopUpMenu(requireContext(), list)
}
...
```

Библиотека Craiyon предоставляет открытый исходный код и доступна для бесплатного использования. Она разработана для облегчения создания изображений с использованием нейронных сетей. Она обладает простым и понятным интерфейсом, что упрощает интеграцию с мобильным приложением. Основанная на передовых технологиях машинного обучения, таких как глубокие нейронные сети, Craiyon позволяет создавать высококачественные и реалистичные изображения персонажей, учитывая различные аспекты и детали, такие как лицо, одежда и атрибуты, рис. 11.

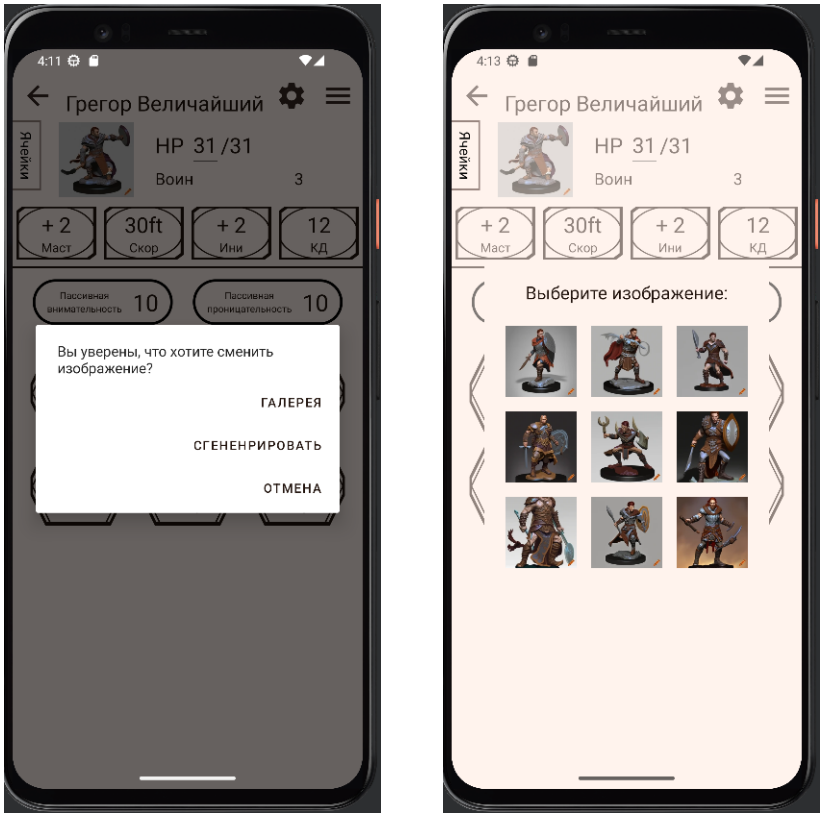

Рис. 11: Генерация изображения.

Заключение

Итогом совместной дипломной работы является готовое мобильное приложение. Текущая альфа-версия совсем немного отстаёт от аналогов по количеству опций, которые можно выбрать для персонажа. Однако, его выделяет поддержка русского языка, небольшой вес (до 30 Мб) и способность функционировать без подключения к интернету. А так же оно имеет две выделяющих его уникальных функциональности – генерация предыстории и изображения персонажа.

В процессе разработки front-end были реализованы и настроены кнопки навигации. Элементы интерфейса были продуманы с учетом удобства использования и наглядности, а граф навигации был создан для определения переходов между экранами. С помощью стилей была достигнута консистентность внешнего вида кнопок на всех экранах, обеспечивая единообразный пользовательский интерфейс. Также была использована векторная графика, которая позволила создать качественные и масштабируемые изображения для различных элементов интерфейса. В результате была создана удобная и интуитивно понятная навигация в приложении с учетом его функциональных и пользовательских требований.

Изначально планировалось создать полностью оффлайн приложение, которое бы отличалось от своих аналогов уникальной функциональностью – генерировать предыстории для персонажей. В процессе разработки мы рассмотрели множество различных архитектур нейронных сетей и, в конечном счете, выбрали трансформеры. Чтобы проверить их эффективность, мы протестировали готовую предобученную модель и провели дообучение, используя синтетически сгенерированные данные. Положительная динамика в генерации текста подтвердила успешность нашего подхода.

Однако, в ходе работы мы столкнулись с проблемой, связанной с весом моделей, что заставило нас переосмыслить нашу исходную идею о полностью оффлайн приложении. Можно было бы попробовать разместить нашу модель на удалённом сервере. Но вместо этого, мы решили обратить внимание на ChatGPT API и использовать его в нашем проекте. Это позволило нам сохранить функциональность генерации предыстории персонажа, но теперь с

возможностью онлайн-взаимодействия. Решение использовать ChatGPT API открывает новые перспективы для нашего приложения и дает нам гибкость в интеграции с другими сервисами и платформами.

И вторая уникальная функциональность, которую предоставляет наше приложение – это генерация изображения персонажа. Мы учитываем параметры персонажей, такие как раса, класс и черты, чтобы создать уникальные изображения. Это позволит игрокам визуализировать своих персонажей и делиться ими с другими игроками, повышая вовлеченность в игровой процесс. Мы исследовали различные подходы для создания датасета, включая использование существующих изображений и создание собственного датасета. Однако, мы нашли библиотеку Craiyon, которая позволяет создавать высококачественные изображения персонажей с использованием нейронных сетей. Мы интегрировали эту python библиотеку с нашим мобильным приложением с помощью Chaquopy, что позволяет нам предложить реалистичные и детализированные изображения персонажей.

Список литературы

[1] C. Corliss. Dungeons and Dragons Infographic Shows How Popular the Game Has Become. (дата обр. 29.05.23)

[2] Android Documentation. Navigation. (дата обр. 29.05.23)

[3] Android Documentation. Styles and themes. (дата обр. 29.05.23)

[4] Android Documentation. Splash screens. (дата обр. 29.05.23)

[5] Christopher Olah. Understanding LSTM Networks. (дата обр. 29.05.23)

[6] Wikipedia. Generative adversarial network. (дата обр. 29.05.23)

[7] Xavier Amatriain. Transformer models: an introduction and catalog.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. Attention Is All You Need.

[9] State-of-the-art Machine Learning for JAX, PyTorch and TensorFlow (дата обр. 30.05.23)