МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Институт № 8 «Компьютерные науки и прикладная математика» **Кафедра 805**
Направление подготовки 01.03.04 «Прикладная математика» **Группа М8О-403Б-18**
Профиль Математическое и программное обеспечение систем обработки информации и управления
Квалификация (степень) бакалавр

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

На тему: Построение системы генерации стилизованных текстов с использованием алгоритмов искусственного интеллекта и нейронных сетей.

Автор ВКРБ Ларькин Владимир Дмитриевич (_____)
(фамилия, имя, отчество)
Научный руководитель Пановский Валентин Николаевич (_____)
(фамилия, имя, отчество)

К защите допустить

Заведующий кафедрой № 805 Пантелеев Андрей Владимирович (_____)
(фамилия, имя, отчество)
« 24 » мая 2022 г.

Москва 2022 г.

РЕФЕРАТ

Отчёт содержит 51 стр., 25 рис., 1 табл., 14 источн., 1 прил.

ГЕНЕРАЦИЯ ТЕКСТА, НЕЙРОННЫЕ СЕТИ, NLP, TRANSFORMER, ДООБУЧЕНИЕ, RUGPT-3

В работе представлено решение задачи дообучения нейросетевой языковой модели ruGPT-3 Small архитектуры Transformer на сравнительно небольшом корпусе текстов, принадлежащих конкретной предметной области, для усвоения моделью стилистики текстов данной области и последующего её встраивания в графический веб-интерфейс пользователя, предоставляющий возможности по генерации новых текстов, принадлежащих той же предметной области.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире с ростом уровня образования и увеличением спроса на специалистов высокой квалификации для оптимизации производственных процессов требуется повышать уровень автоматизации предприятий, чтобы разгрузить работников и предоставить им возможность заниматься интеллектуальным трудом. А с увеличением вычислительных мощностей и развитием компьютерных наук всё больше рутинных задач поддаются автоматизации. Задача автоматизации написания разного рода текстов стоит особенно остро, так как она актуальна в самых разных сферах человеческой деятельности.

В настоящей работе предлагается система, которая позволяет автоматизировать большую часть действий, требуемых для создания базовой структуры документа и его частичного заполнения, предоставляя возможность коррекции и дополнения полученного текста пользователем «на лету».

Работа системы демонстрируется применительно к задаче генерации сценариев юмористических телешоу. Актуальность решения этой задачи обусловлена его применимостью в сферах психологии и психиатрии для тестирования уровня эмпатии способом, близким к методике А. Меграбяна и Н. Эпштейна [1]: пациенту предлагается прочитать несколько текстов и ответить на ряд вопросов, касающихся испытываемых им чувств, после чего на основании полученных ответов делается вывод по поводу уровня его эмпатии.

Но возможности данной системы не ограничены только этой предметной областью. Предлагаемое решение предоставляет гибкий механизм дообучения под генерацию текстов из той предметной области, которой принадлежит обучающая выборка текстов.

ОСНОВНАЯ ЧАСТЬ

1. Теоретическая часть

1.1 Определения

Корпус — множество подобранных и определённым образом обработанных текстов.

Токен — элементарная единица разбиения корпуса.

Токенизация — процесс разбиения корпуса на токены с присвоением им уникальных числовых идентификаторов.

Языковая модель — распределение $P(w_t|w_1,w_2,w_3,\ldots,w_n)$ вероятностей встретить токен w_t в корпусе сразу после n токенов w_i, $i \in [1,n]$, идущих подряд, где $w_i \in W\,\forall i$, W — множество всех токенов корпуса, n — длина контекста модели.

Длина контекста — количество n токенов w_i, $i \in [0,n]$, предшествующих токену w_t, от которых зависит вероятность появления в тексте токена w_t.

Дообучение — процесс обучения уже обученной на некоторых данных модели машинного обучения на новых данных. В случае языковой модели это означает подстройку модели под новое распределение токенов.

Перплексия — мера схожести двух вероятностных распределений, используемая для оценки качества генерации текста языковой моделью. Перплексия задаётся формулой 1.1.

$$\mathrm{PP}(W) = \sqrt[n]{\frac{1}{P(w_1,w_2,\ldots,w_n)}} \tag{1.1}$$

1.2 Постановка задачи

Дано:

а) корпус, состоящий из текстов, принадлежащих конкретной предметной области,

б) предобученная нейросетевая языковая модель, хорошо моделирующая вероятностное распределение слов в естественном языке.

Требуется:

а) дообучить данную языковую модель на данных из корпуса, получив новую языковую модель, моделирующую распределение вероятностей слов в данном корпусе,

б) реализовать возможность применения её для генерации новых текстов, принадлежащих предметной области данного корпуса,

в) создать графический интерфейс для взаимодействия пользователя с моделью,

г) подготовить получившееся приложение для дистрибуции.

1.3 Нейронные сети

1.3.1 Однослойный перцептрон

Одной из простейших моделей машинного обучения является однослойный перцептрон. Он позволяет, обучаясь на выборке данных, решать задачу линейной регрессии, то есть, устанавливать зависимость между зависимой переменной y и независимыми переменными x при условии, что между ними существует линейная зависимость, которую можно описать уравнением 1.2. В нём всегда $x_0 \equiv 1$, а w_0 называется смещением, так как изменение этой компоненты приводит к увеличению или уменьшению y на постоянное значение.

$$y = \mathbf{w} \cdot \mathbf{x} = w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n, \quad x_0 \equiv 1 \tag{1.2}$$

В этом уравнении неизвестными являются компоненты вектора **w**, найдя которые, можно получить взаимосвязь между x и y. Обучающая выборка данных представляет собой матрицу **X** размера $m \times (n+1)$ наблюдений вектора **x** и вектор **y** размера m, где $y_i = \mathbf{w} \cdot \mathbf{x}_i$, $\mathbf{x}_i$ — i-я строка матрицы **X**, как

показано в уравнении 1.3.

$$\mathbf{Xw} = \begin{pmatrix} x_0^1 & x_1^1 & x_2^1 & x_3^1 & \dots & x_n^1 \\ x_0^2 & x_1^2 & x_2^2 & x_3^2 & \dots & x_n^2 \\ x_0^3 & x_1^3 & x_2^3 & x_3^3 & \dots & x_n^3 \\ x_0^4 & x_1^4 & x_2^4 & x_3^4 & \dots & x_n^4 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & x_2^m & x_3^m & \dots & x_n^m \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_m \end{pmatrix} = \mathbf{y} \tag{1.3}$$

Уравнения 1.3 имеет одно решение относительно $\mathbf{w}$ тогда и только тогда, когда $\operatorname{rank}\mathbf{X} = n$. Если $\operatorname{rank}\mathbf{X} < n$, уравнение имеет бесконечное число решений, и если $\operatorname{rank}\mathbf{X} > n$, уравнение не имеет решений. Но на практике данных обычно больше, чем компонент в векторе $\mathbf{w}$, поэтому используется аппроксимация методом наименьших квадратов, суть которой состоит в том, чтобы путём решения задачи минимизации, показанной в уравнении 1.4, найти такую прямую, чтобы функция потерь $L(\mathbf{y},\hat{\mathbf{y}})$ была минимальна.

$$\begin{cases} \mathbf{w} = \arg\min\limits_{\mathbf{w}} L(\mathbf{y},\hat{\mathbf{y}}), \\ \hat{\mathbf{y}} = \mathbf{Xw} \end{cases} \tag{1.4}$$

Из курса математической статистики известно, что лучше всего в данной задаче подходит функция потерь MSE или средний квадрат ошибки (уравнение 1.5). При использовании этой функции потерь дисперсия ошибки получается наименьшей.

$$\text{MSE}(\mathbf{y},\hat{\mathbf{y}}) = \sum_{i=1}^{m} (\mathbf{x}_i^\mathsf{T}\mathbf{w} - y_i)^2 = (\mathbf{Xw} - \mathbf{y})^\mathsf{T}(\mathbf{Xw} - \mathbf{y}) \tag{1.5}$$

Минимум функции потерь можно найти градиентными методами, например, методом Adam, так как она дифференцируема, и её производную можно вычислить аналитически (уравнение 1.6). Процесс поиска минимума функции потерь модели называют обучением.

$$\frac{\partial\,\text{MSE}}{\partial\mathbf{w}}(\mathbf{w}) = 2\mathbf{X}^\mathsf{T}(\mathbf{Xw} - \mathbf{y}) \tag{1.6}$$

Если же от модели требуется предсказывать не численные характеристики объектов, а относить их к той или иной группе, то такая задача называется задачей классификации, а модель — логистической регрессией.

В этом случае требуется не аппроксимировать точки гиперплоскостью, а сделать так, чтобы гиперплоскость отделяла точки одного класса от точек другого (рисунок 1.1). Для этого к выходу линейной регрессии дополнительно применяют функцию с областью значений $[0, 1]$, чтобы выход модели можно было интерпретировать как вероятность принадлежности объекта заданному классу [2].

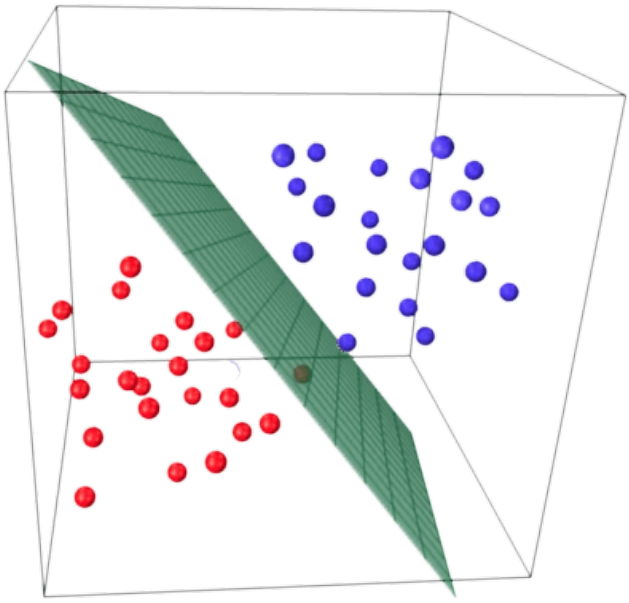

Рисунок 1.1 — Работа логистической регрессии в случае бинарной классификации

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.7}$$

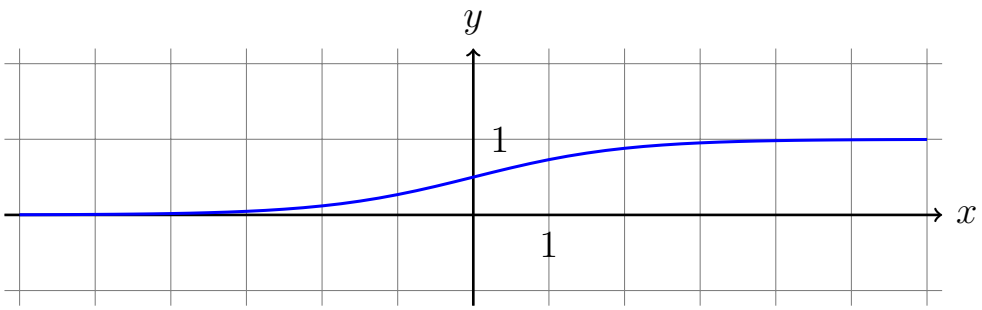

Рисунок 1.2 — График сигмоиды

$$\text{Softmax}(\mathbf{x}) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \begin{pmatrix} e^{x_1} \\ e^{x_2} \\ e^{x_3} \\ \vdots \\ e^{x_n} \end{pmatrix}, \quad \mathbf{x} \in \mathbb{R}^n \tag{1.8}$$

В случае бинарной классификации, то есть, когда класса два, в роли такой функции выступает сигмоида (уравнение 1.7, рисунок 1.2). Тогда выход модели трактуется как вероятность принадлежности объекта первому классу. Если же классов больше двух, то на выходе модели должен получиться вектор, содержащий столько компонент, сколько в задаче классов, и к нему применяется функция Softmax (уравнение 1.8), выход которой удовлетворяет аксиомам вероятности 1.9.

$$\begin{cases} \mathbf{y} = \begin{pmatrix} y_1 & y_2 & y_3 & \dots & y_n \end{pmatrix}^\mathsf{T} \\ y_i \in [0,1] \, \forall i \in [1,n], \\ \sum_{i=1}^{n} y_i = 1 \end{cases} \tag{1.9}$$

В задаче классификации используется функция потерь, называемая перекрёстной энтропией (уравнение 1.10). В случае бинарной классификации она представляется в виде 1.11.

$$\text{CE}(\mathbf{y},\hat{\mathbf{y}}) = -\sum_{i=1}^{n} y_i \ln \hat{y}_i \tag{1.10}$$

$$\text{BCE}(y,\hat{y}) = -y \ln \hat{y} - (1-y) \ln(1-\hat{y}) \tag{1.11}$$

1.3.2 Нелинейная аппроксимация

Модели линейной и логистической регрессии позволяют строить аппроксимации только гиперплоскостями. Чтобы иметь возможность аппроксимировать нелинейные зависимости в данных, их применяют последовательно, перемежая нелинейными функциями, например, как в уравнении 1.12. Тогда функция $f(x)$ называется нейронной сетью, а линей-

ные модели, из которых она состоит — её слоями. Параметры, или веса, нейронной сети, состоят из параметров всех её слоёв, а оптимизируются они так же, градиентными методами.

$$f(x) = \sigma(\dots\sigma(\sigma(xw_1)w_2)w_3\dots) \tag{1.12}$$

Функции, перемежающие применения линейных слоёв, могут быть любыми кусочно-гладкими нелинейными функциями, а на то, какие операции производятся над слоями внутри нейронной сети, накладывается только одно ограничение: функция $f(x)$ должна оставаться кусочно-гладкой, чтобы можно было искать минимум функции потерь градиентными методами.

1.4 Градиентные методы оптимизации

Минимум функции потерь при обучении нейронных сетей обычно ищется градиентными методами, которые хорошо себя зарекомендовали в решении задач на оптимизацию дифференцируемых функций.

Суть этой группы методов хорошо иллюстрировать на примере простейшего из них — градиентного спуска.

Пусть имеется задача, представленная в уравнении 1.13.

$$\begin{cases} L(\mathbf{w}) : \mathbb{R}^n \to \mathbb{R}, \\ \nabla L(\mathbf{w}) = \left(\frac{\partial L}{\partial w_1}(w_1) \quad \frac{\partial L}{\partial w_2}(w_2) \quad \frac{\partial L}{\partial w_3}(w_3) \quad \dots \quad \frac{\partial L}{\partial w_n}(w_n)\right)^\mathsf{T}, \\ L(\mathbf{w}) \to \min\limits_{\mathbf{w}} \end{cases} \tag{1.13}$$

Требуется найти минимум дифференцируемой функции n переменных. Как известно из курса математического анализа, градиент функции нескольких переменных, вычисленный в конкретной точке — это вектор, указывающий направление наискорейшего возрастания функции в этой точке. Значит, чтобы найти минимум, двигаясь из какой-либо начальной точки, нужно перемещаться в направлении, противоположном градиенту. Эту идею реализует градиентный спуск (уравнение 1.14).

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha\nabla L(\mathbf{w}) \tag{1.14}$$

Делая шаги длины α в направлении антиградиента функции, с увеличением t алгоритм приближается к минимуму.

Но у этого алгоритма есть ряд проблем, связанных со скоростью сходимости. Наглядный пример приведён на рисунке 1.3. В нём целевая функция представляет собой вытянутый параболоид, и градиентный спуск в этом случае делает шаги неоптимальным образом, двигаясь к точке минимума очень медленно.

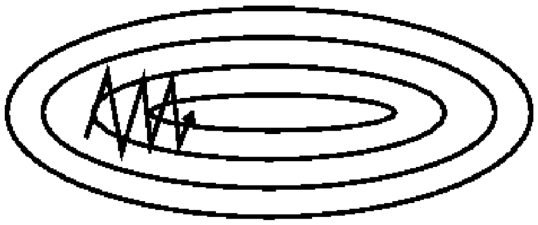

Рисунок 1.3 — Проблема долгой сходимости простого градиентного спуска

Одним из решений этой проблемы является введение физических характеристик для точки, двигающейся по поверхности целевой функции. Например, в методе Adam (англ. — Adaptive Moment Estimation) вводятся масса и скорость, моделируя движение шарика вниз по неровной поверхности.

Другая проблема состоит в том, что фиксированный шаг градиентного спуска может мешать оптимизации функции по некоторым координатам. В методе Adam это решается путём введения специальных весов, уменьшающих шаг по координатам, которые часто изменяются. Метод Adam описывается уравнениями 1.15.

$$\begin{cases} \mathbf{m}^t = \beta_1 \mathbf{m}^{t-1} + (1-\beta_1)\nabla L(\mathbf{w}) \\ \mathbf{v}^t = \beta_2 \mathbf{v}^{t-1} + (1-\beta_2)(\nabla L(\mathbf{w}))^2 \\ \hat{\mathbf{m}}^t = \frac{\mathbf{m}^t}{1-\beta_1^t} \\ \hat{\mathbf{v}}^t = \frac{\mathbf{v}^t}{1-\beta_2^t} \\ \mathbf{w}^{t+1} = \mathbf{t} - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}^t}+\varepsilon}\hat{\mathbf{m}}^t \\ \mathbf{m}^0 = \mathbf{0}, \quad \mathbf{v}^0 = \mathbf{0} \end{cases} \tag{1.15}$$

Параметры β_1, β_2, ε и α задаются по усмотрению пользователя в зависимости от решаемой задачи.

На сегодняшний день Adam является одним из лучших алгоритмов оптимизации, показывающих наименьшее время сходимости при обучении нейронных сетей на разнообразных данных [3].

1.5 Стохастический градиентный спуск

На практике бывает так, что данных настолько много, что одновременно они не помещаются в оперативную память компьютера или видеокарты, из-за чего становится невозможно вычислить градиент функции потерь на всей обучающей выборке. Поэтому обычно применяется стохастический градиентный спуск, смысл которого сводится к рассчёту функции потерь при использовании подвыборки обучающих данных.

Пусть $\mathbf{X}$ — обучающая выборка данных, содержащая n наблюдений. Она разбивается на m подвыборок одинакового размера, и функция потерь считается на каждой из них независимо, как в формуле 1.16.

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha \nabla L_i(\mathbf{w}^t) \tag{1.16}$$

Здесь $L_i(\mathbf{w}^t)$ — функция потерь, посчитанная на i-ой подвыборке $\mathbf{x}_i$. Важно, что при всех $i \in [1, m]$ $L_i(\mathbf{w}^t)$ считается в одной и той же точке $\mathbf{w}^t$. После завершения итераций по i подвыбоки случайным образом перемешиваются, и расчёт повторяется в новой точке $\mathbf{w}^{t+1}$.

1.6 Токенизация

Одним из важнейших понятий области обработки естественного языка является токенизация. Так как нейронные сети работают с векторами, то есть, с числами, для решения с их помощью задачи обработки текста требуется эти тексты представлять в виде векторов.

Суть токенизации сводится к дроблению текста на токены, некие элементарные единицы, с последующей их заменой на числа, соответствующие их номерам в списке уникальных токенов текста. Такой подход позволяет

однозначно отображать тексты в вектора, которые можно обрабатывать нейронными сетями, а также, наоборот, декодировать текст из векторов.

Токенизация бывает основана на словах, частях слов и символах. У каждого из её видов есть свои преимущества и недостатки.

Так, если брать за токены слова, то теряется информация о словообразовании и словоформах, даже одно слово в разных падежах будет для модели двумя совершенно разными токенами. Зато, если обрезать у слов окончания, то токены будут совпадать, и на таких данных уже можно обучить простую модель, решающую, например, задачу классификации текстов. Но, конечно, такой способ кодирования не подходит для генерации текста ввиду потери важной информации об окончаниях.

Если за токены брать отдельные символы текста, то в теории вся исходная информация из текста сохранится. Но на практике эта информация будет слишком разрежена, и для её использования понадобится слишком сложная модель и огромное количество данных.

В естественном, например, русском языке элементарными единицами словообразования являются морфемы, которые состоят из больше чем одного символа, и разделение текста на них сохранит всю информацию о возможных вариациях слов, которая при этом будет не слишком разреженной. Таким образом, для моделирования языка лучше всего подходит токенизация, берущая за основу части слов.

Но алгоритм деления слов на морфемы очень сложен и узко специализирован для работы с конкретным языком. А в обучающей выборке могут встретиться тексты на разных языках, и нужно, чтобы модель могла их все обработать. Для решения этой задачи существует алгоритм BPE (англ. — Byte-Pair Encoding), изначально разработанный для сжатия данных.

Суть его состоит в том, что для всех пар символов из обучающей выборки считается частота, с которой они встречаются вместе. Затем самые частотные пары остаются в виде самостоятельных токенов, токены с нулевой частотой отбрасываются, а остальные остаются как есть. Далее процесс

повторяется до тех пор, пока не останется возможности объединять токены в пары или пока не будет превышено максимальное число итераций.

Для больших корпусов алгоритм BPE позволяет создать оптимальный словарь токенов при минимальном их числе [4].

1.7 Языковые модели

Задача генерации текста на естественном языке сводится к задаче моделирования языка. Языковая модель — это условное распределение вероятности встретить в тексте токен w_{n+1} сразу после n токенов $\{w_i\}_{i=1}^{n}$, где n — длина контекста модели, то есть, максимально возможное количество токенов, влияющих на появление токена w_{n+1}.

Нейросетевые языковые модели принимают на вход вектор длины n, содержащий токены контекста, и выдают вектор длины, равной количеству токенов в словаре, к которому применяется функция Softmax, чтобы получить вектор, в коротом каждому токену сопоставлена вероятность, что он идёт после заданных n токенов контекста. Такая модель схематично изображена на рисунке 1.4 [5].

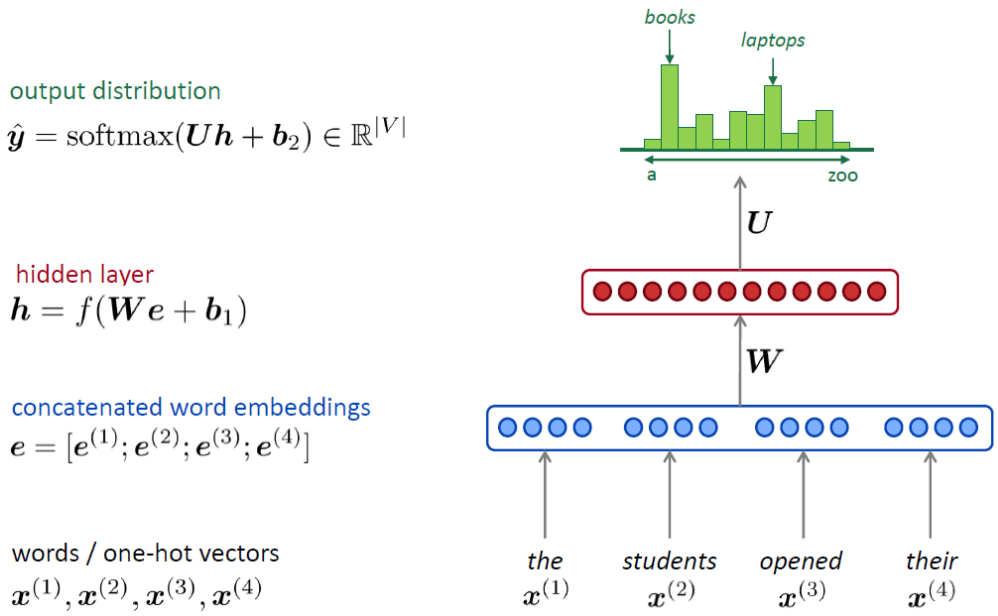

Рисунок 1.4 — Простая нейросетевая языковая модель

Как видно, задача языкового моделирования является частным случаем задачи многоклассовой классификации. Поэтому, при обучении в качестве функции потерь используется перекрёстная энтропия.

Для того, чтобы решить задачу генерации текста с помощью языковой модели, нужно просто генерировать случайные числа из распределения, получаемого на выходе функции Softmax, с проведением обратного преобразования токенов в текст, добавлять новый токен в конец вектора контекста с удалением первого и генерацией нового токена до тех пор, пока не будет сгенерирован текст желаемой длины.

1.8 Рекуррентные нейронные сети

Так как текст представляет собой последовательность токенов, а определение языковой модели подразумевает зависимость вероятности следующих токенов от нескольких предыдущих, естественно попробовать реализовать алгоритм его обработки, который бы учитывал эту рекуррентность.

Примером таких моделей являются рекуррентные нейронные сети (англ. — Recurrent Neural Networks, RNN). Их особенность состоит в наличии так называемого скрытого состояния — вектора, накапливающего в себе информацию, полученную от поэлементной обработки входной последовательности, как отражено в уравнении 1.17. Здесь x_t — вход слоя, w_{hh}, w_{xh}, w_{hy} — обучаемые параметры слоёв сети, y_t — выход слоя, h_t — скрытое состояние на шаге t, $\text{th}(\cdot)$ — гиперболический тангенс (уравнение 1.18, рисунок 1.6).

$$\begin{cases} h_t = \text{th}(w_{hh}h_{t-1} + w_{xh}x_t) \\ y_t = w_{hy}h_t \end{cases} \tag{1.17}$$

$$\text{th}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.18}$$

Схематическое изображение RNN представлено на рисунке 1.5.

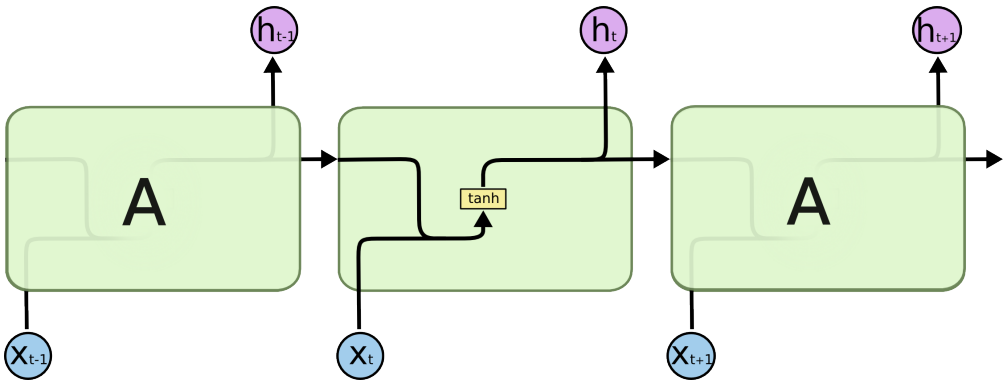

Рисунок 1.5 — Схема работы рекуррентной нейронной сети

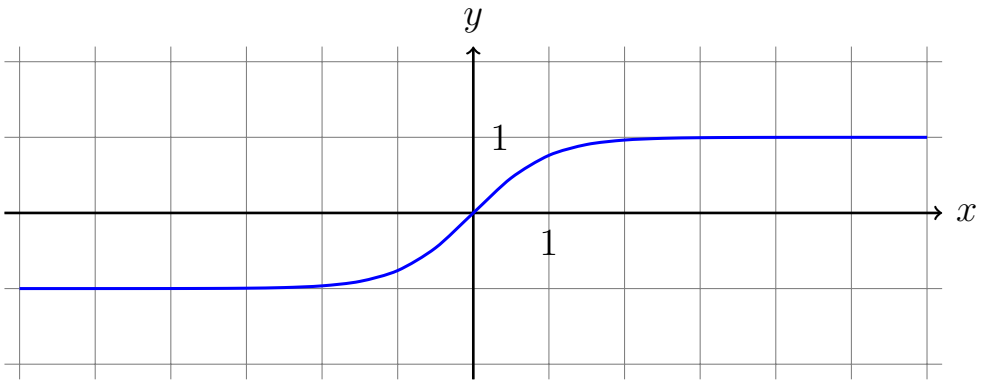

Рисунок 1.6 — График гиперболического тангенса

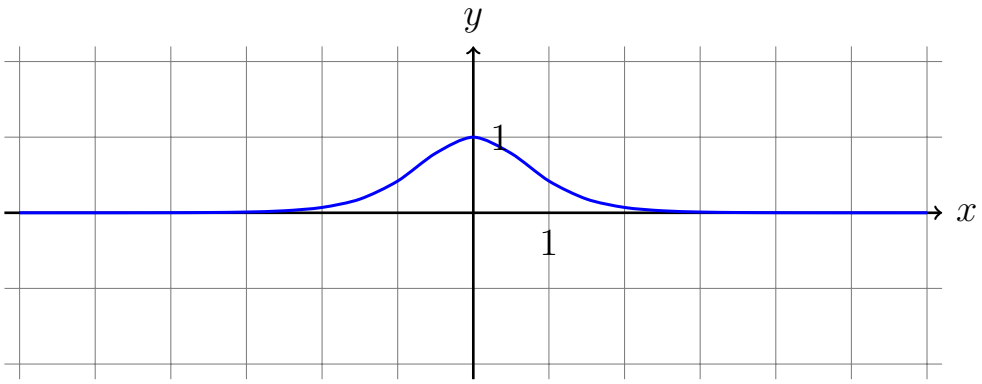

Рисунок 1.7 — График производной гиперболического тангенса

Рекуррентные нейронные сети способны улавливать зависимости в последовательных данных, но у них есть недостатки. Как видно на рисунке 1.7, производная гиперболического тангенса, используемого в качестве функции активации в RNN, почти на всей области определения близка к нулю. На практике это означает, что чем длиннее последовательность данных, тем больше в формуле производной функции потерь будет множителей, близких

к нулю, и тем ближе к нулю будут шаги градиентного спуска, что замедляет обучение, а иногда делает его и вовсе невозможным. С этим связана вторая проблема — новые данные в последовательности оказывают на результат большее влияние, чем старые за счёт многократного умножения последних на близкие к нулю числа. Иными словами можно сказать, что нейросеть «забывает» старые данные.

1.9 Модификации RNN

Для решения этой проблемы существует две основных модификации RNN, общая идея которых сводится к введению дополнительных блоков для предсказания степени важности пришедших на вход данных, обеспечивая более контролируемое забывание.

1.9.1 GRU

Архитектура GRU (англ. — Gated Recurrent Unit) предлагает решать проблему забывания с помощью ряда так называемых вентилей:

— вентиль обновления, определяющий количество информации, которое требуется сохранить с предыдущей итерации (уравнение 1.19),

— вентиль забывания, определяющий количество информации, которое требуется забыть (уравнение 1.20),

$$z_t = \sigma(w_{zx}x_t + w_{zh}h_{t-1}) \tag{1.19}$$

$$r_t = \sigma(w_{rx}x_t + w_{rh}h_{t-1}) \tag{1.20}$$

Информация, которую будет хранить текущая ячейка сети, рассчитывается по формуле 1.21, а итоговое значение скрытого состояния — по формуле 1.22.

$$c_t = \text{th}(w_{cx}x_t + w_{ch}r_t h_{t-1}) \tag{1.21}$$

$$h_t = (1 - z_t)h_{t-1} + z_t c_t \tag{1.22}$$

Здесь $\sigma(\cdot)$ — сигмоида, а w_{zx}, w_{zh}, w_{rx}, w_{rh}, w_{cx}, w_{ch} — обучаемые параметры слоя. Схема ячейки GRU изображена на рисунке 1.8.

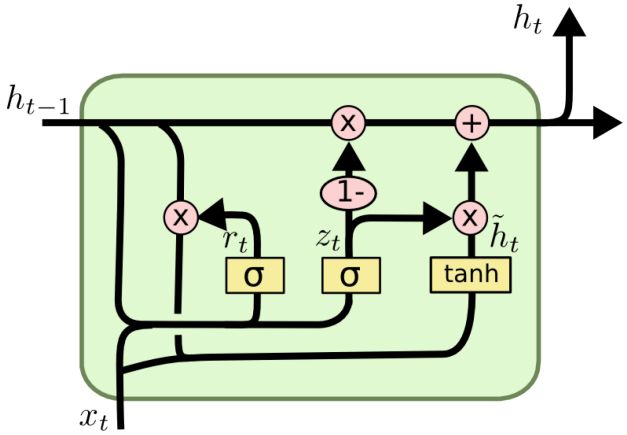

Рисунок 1.8 — Схема ячейки GRU

1.9.2 LSTM

Нейронная сеть с долгой краткосрочной памятью (англ. — Long Short-Term Memory, LSTM) предлагает решение для борьбы с забыванием, благодаря которому сети удаётся сохранять в 4 раза больше информации, чем обычной RNN [6].

Эта архитектура тоже оперирует понятием вентиля, их в ней три:

— входной вентиль, контролирующий поток приходящей информации (уравнение 1.23),

— вентиль забывания, управляющий количеством информации, которое требуется сохранить с предыдущей итерации (уравнение 1.24),

— выходной вентиль, контролирующий поток выходящей информации (уравнение 1.25).

$$i_t = \sigma(w_{ix}x_t + w_{ih}h_{t-1}) \tag{1.23}$$

$$f_t = \sigma(w_{fx}x_t + w_{fh}h_{t-1}) \tag{1.24}$$

$$o_t = \sigma(w_{ox}x_t + w_{oh}h_{t-1}) \tag{1.25}$$

Далее вычисляются промежуточные значения g_t и s_t (уравнения 1.26 и 1.27), а с их помощью считается результирующее скрытое состояние h_t по формуле 1.28.

$$g_t = \text{th}(w_{gx}x_t + w_{gh}h_{t-1}) \tag{1.26}$$

$$s_t = s_{t-1}f_t + g_t i_t \tag{1.27}$$

$$h_t = \text{th}(s_t)o_t \tag{1.28}$$

w_{ix}, w_{ih}, w_{fx}, w_{fh}, w_{ox}, w_{oh}, w_{gx}, w_{gh} — обучаемые параметры.

Схема ячейки LSTM изображена на рисунке 1.9.

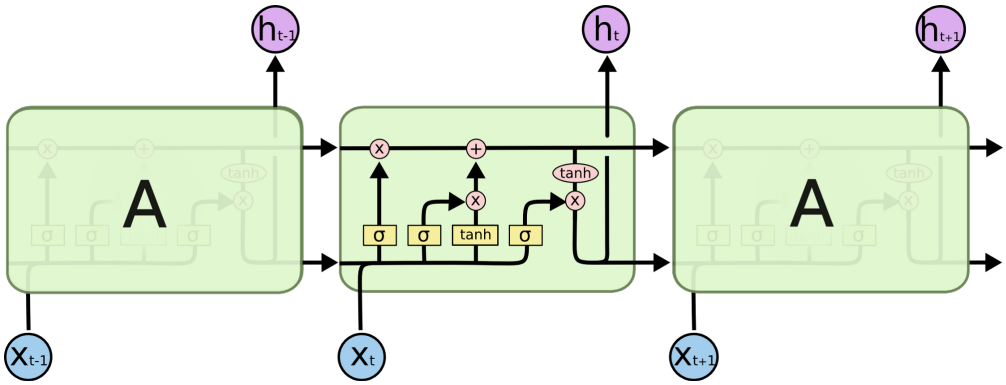

Рисунок 1.9 — Схема ячейки LSTM

LSTM позволяют частично решить проблему забывания, но за счёт большого числа обучаемых параметров они долго обучаются и занимают много места в памяти. Кроме того, остаётся нерешённой проблема затухающих градиентов.

1.10 Механизм внимания

В задачах преобразования одной последовательности в другую, например, в задаче машинного перевода или преобразования текста в речь, проблема затухающих градиентов может частично решаться с помощью механизма внимания (англ. — Attention).

Его суть состоит в добавлении в сеть дополнительных параметров, отвечающих за взаимосвязь между отдельными ячейками RNN.

Рассмотрим нейронную сеть, изображённую на рисунке 1.10.

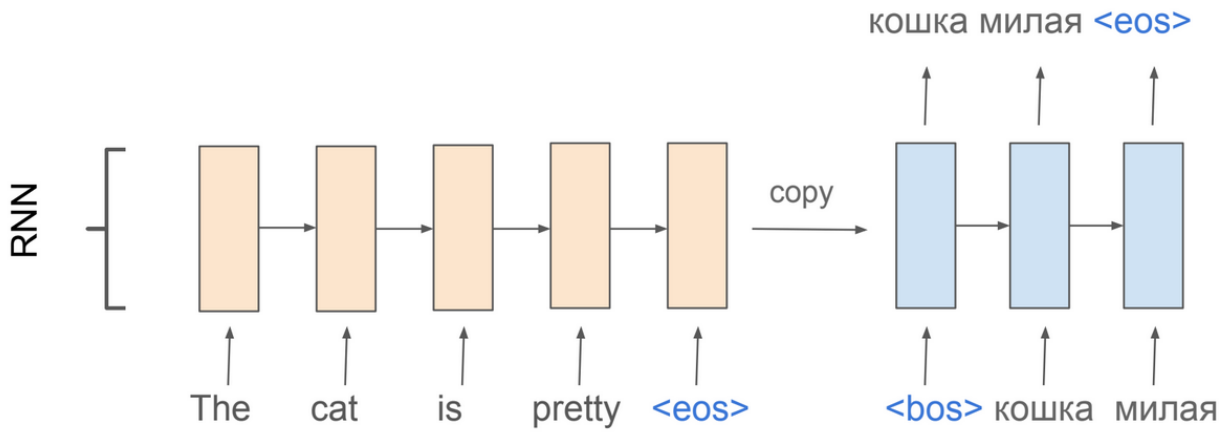

Рисунок 1.10 — Модель машинного перевода

Она состоит из двух рекуррентных сетей. Первая RNN принимает на вход предложение и токен за токеном накапливает информацию о нём, получая в скрытом состоянии последней ячейки вектор, кодирующий входное предложение. Вторая RNN принимает на вход этот вектор, а затем каждая её ячейка на основе своего скрытого состояния предсказывает токены слов на русском языке до тех пор, пока не будет предсказан токен конца предложения.

У этого подхода есть существенный недостаток: скрытое состояние последней ячейки первой RNN хранит больше информации о последнем слове предложения, чем об остальных, причём чем раньше слово стоит в предложении, тем меньше информации о нём будет сохранено в силу причин, описанных выше. Получается, что «развернуть» из этого вектора предложение,

действительно являющегося правильным переводом, маловероятно. Решение этой проблемы предлагает механизм внимания.

Рассмотрим рисунок 1.11. В отличие от рисунка 1.10, здесь выход ячейки второй RNN зависит не только от скрытого состояния первой сети, но и от скрытого состояния всех остальных ячеек первой RNN с разными весами. Это и есть внимание: в процессе обучения такая архитектура позволяет выводить зависимости выходных данных от разных частей входных, как бы обращая внимание на разные слова входного предложения.

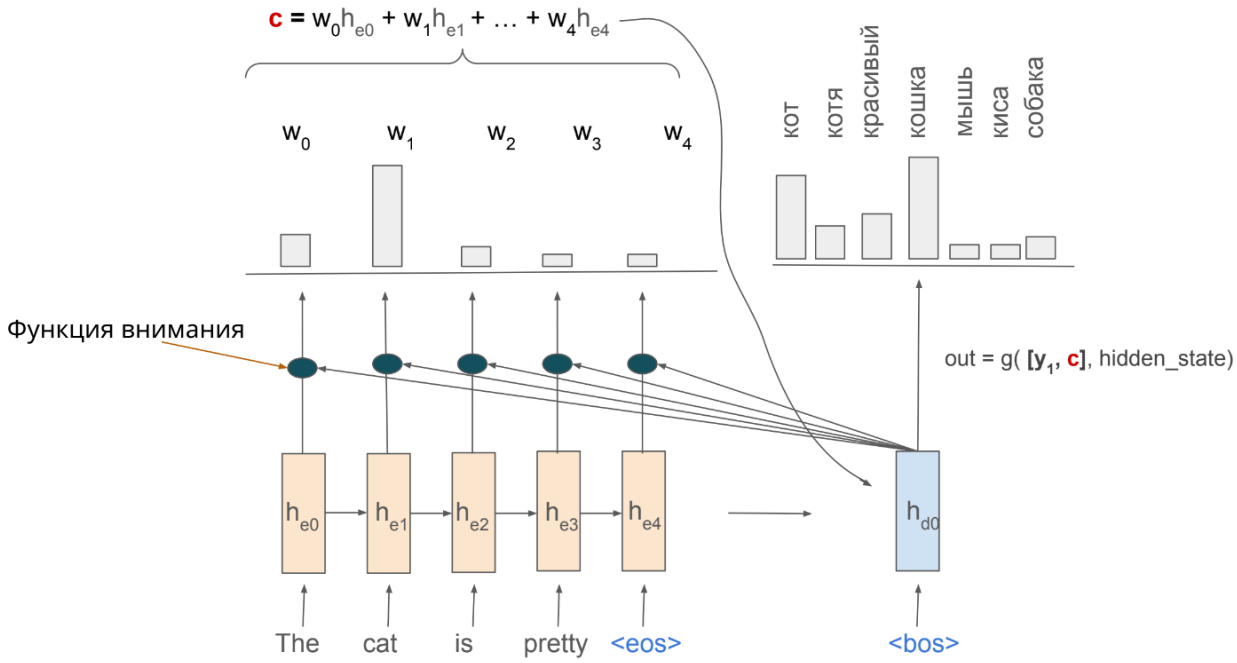

Рисунок 1.11 — Внимание

Благодаря тому, что каждая ячейка второй RNN зависит от каждой ячейки первой, информация о входных данных не зануляется, тем самым отчасти решается проблема затухающих градиентов.

1.11 Архитектура Transformer

Архитектура Transformer предлагает совершенно иной подход к нейросетевой обработке текстовых данных. Её разработчики в своей статье [7] показывают, что отказавшись от рекуррентных слоёв, можно получить гораздо лучшие результаты работы модели, а также заметно ускорить её обучение.

Вместо рекуррентных слоёв предлагается использовать механизм внутреннего внимания (англ. — Self-Attention), близкий по сути к обычному вниманию, описанному выше, но с одним отличием: оба аргумента функции внимания принадлежат одной и той же последовательности. Получается, веса слоёв внутреннего внимания кодируют внутренние зависимости между токенами входной последовательности.

Слоёв внутреннего внимания используется сразу несколько — это позволяет им выучивать разные зависимости, кодируя разного рода отношения между токенами.

В связи с отказом от рекуррентности Transformer не имеет возможности обрабатывать данные произвольной длины. Модели этой архитектуры принимают на вход заранее заданное число токенов — последовательность длины контекста. С другой стороны, в них нет проблемы затухания градиентов, а информация обо всех токенах учитывается в равной мере, и благодаря этому все лучшие современные языковые модели построены на основе архитектуры Transformer.

В том случае, если входная последовательность токенов короче, чем контекст модели, то её дополняют сначала специальными токенами, обозначающими отсутствие слова и представляемыми нулями.

Другой позитивный аспект использования множественных независимых слоёв внутреннего внимания — это возможность обучать их параллельно на видеокартах, что было невозможно в случае с RNN, где результат каждой ячейки зависел от результата предыдущей.

Схема модели архитектуры Transformer представленая на рисунке 1.12.

Модель состоит из двух смысловых блоков — кодировщик и декодировщик. Первый возвращает промежуточное векторное представление входной последовательности, а второй снова получает из него текст.

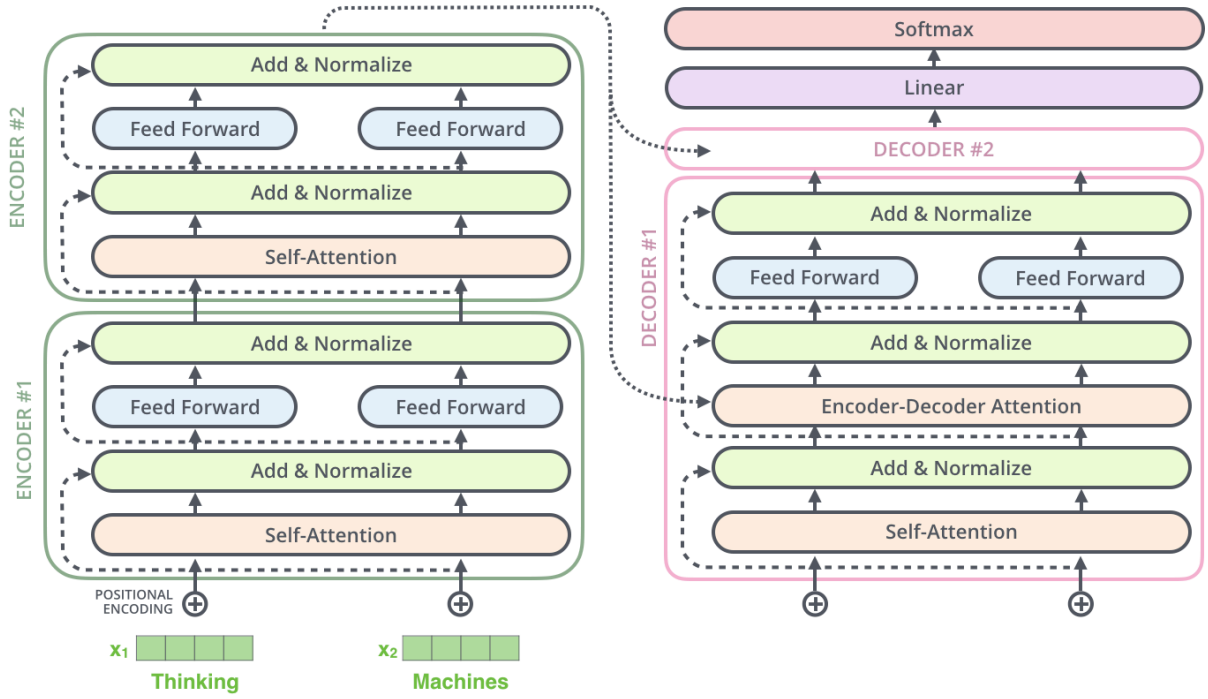

Рисунок 1.12 — Transformer

1.12 BERT

Архитектура Transformer позволяет строить очень большие модели, содержащие сотни миллионов вараметров, и обучать их на видеокартах, распараллеливая вычисления. Такие модели способны усваивать гораздо более глубокие закономерности в данных, чем могли RNN, и это можно использовать для построения векторного представления текста — эмбеддингов (англ. — Embedding), которые, в свою очередь, можно подать на вход модели классификации и классифицировать тексты.

Для решения этой задачи хорошо подходит первая часть модели Transformer — кодировщик, возвращающий вектор чисел, который можно использовать в качестве эмбеддинга [8].

Модель, за основу которой была взята эта идея, называется BERT (англ. — Bidirectional Encoder Representations from Transformers). Её схема изображена на рисунке 1.13.

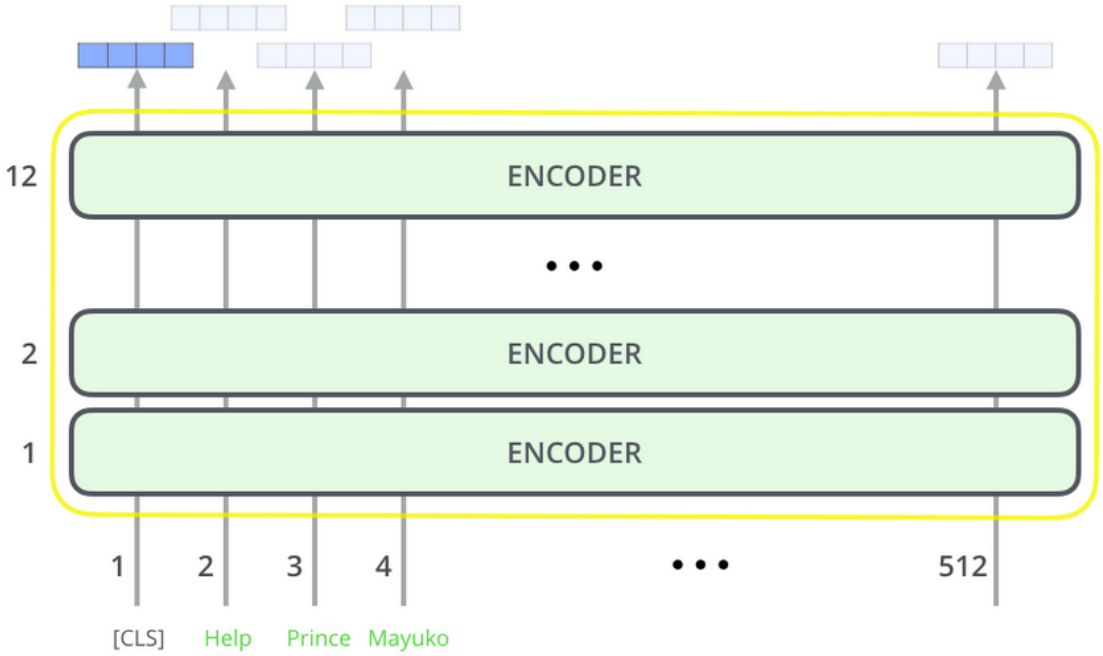

Рисунок 1.13 — BERT

BERT обучается для решения сразу двух задач:

а) определение по двум поданным на вход предложениям, с какой вероятностью одно идёт после другого в тексте,

б) предсказание слов, стоящих на месте пропусков, отмеченных токеном [MASK].

Такой процесс обучения позволяет BERT усваивать глубинные закономерности в устройстве языка и возвращать эмбеддинги, хорошо подходящие для классификации, то есть такие, что в их пространстве кластеры эмбеддингов эквивалентны группам текстов, имеющих общие признаки: тему, жанр, эмоциональную окраску и т. д.

1.13 GPT

Transformer хорошо подходит и для решения задачи генерации текста. Для этого была создана модель GPT (англ. — Generative Pre-trained Transformer).

Если BERT представляет собой первую часть модели Transformer — кодировщик, то GPT, наоборот, является второй частью, то есть, декодировщиком. Принимая на вход последовательность токенов, она их кодирует и предсказывает вероятности токенов, и чем какая-то из них выше, тем вероятнее, что соответствующий токен идёт сразу после заданной последовательности.

Здесь присутствует параллель с моделью BERT, которая тоже способна предсказывать токены на основании заданного контекста, но там используется информация как о тексте до предсказываемого токена, так и после. В GPT же учитывается только информация после, так как эта модель создавалась специально для решения задачи генерации текста, и такой подход в этом случае подходит лучше, так как делает модель авторегрессивной: генерируя один токен за раз, при каждой следующей генерации может учитываться информация с предыдущих шагов, и это позволяет генерировать тексты любой длины, хоть и с ограниченным размером окна контекста.

Схема модели GPT изображена на рисунке 1.14.

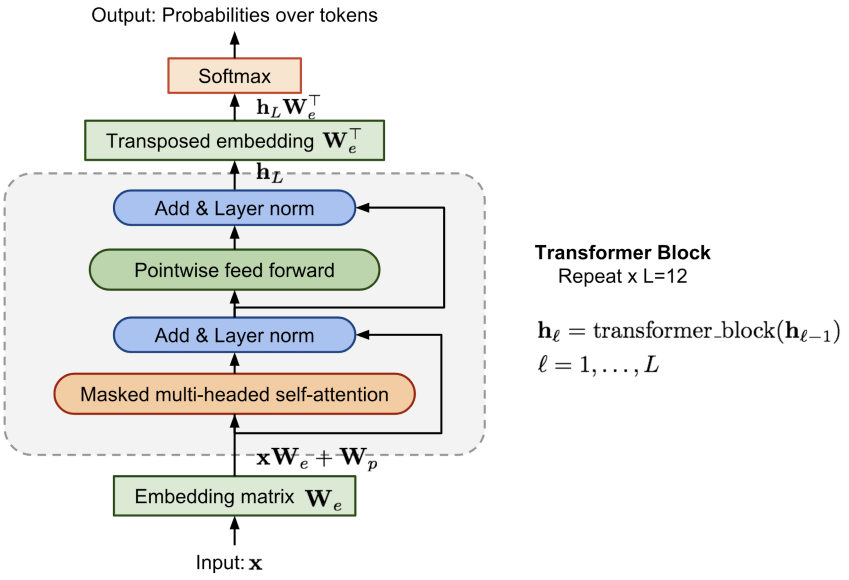

Рисунок 1.14 — GPT

1.13.1 Маскированное внутреннее внимание

В модели GPT, в отличие от описанной выше Transformer, используется модифицированная версия слоя внутреннего внимания: вместо того, чтобы маскировать токены токеном [MASK], как в BERT, слой маскированного

внутреннего внимания (англ. — Masked Self-Attention) использует маску, показанную в формуле 1.29. Маска представляет собой вектор, состоящий из k идущих подряд единиц, после которых идут нули.

$$M = \begin{pmatrix} m_1 = 1 & \dots & 1 & 1 & \dots & m_{k+1} = 0 & \dots & 0 & 0 \end{pmatrix}^{\mathsf{T}} \tag{1.29}$$

Здесь k — номер обрабатываемого токена. Маска поэлементно умножается на вектор контекста, что исключает возможность срабатывания внимания между k-м токеном и токенами, идущими после него.

Процесс обучения GPT с поэтапной работой механизма маскированного внутреннего внимания проиллюстрирован на рисунке 1.15.

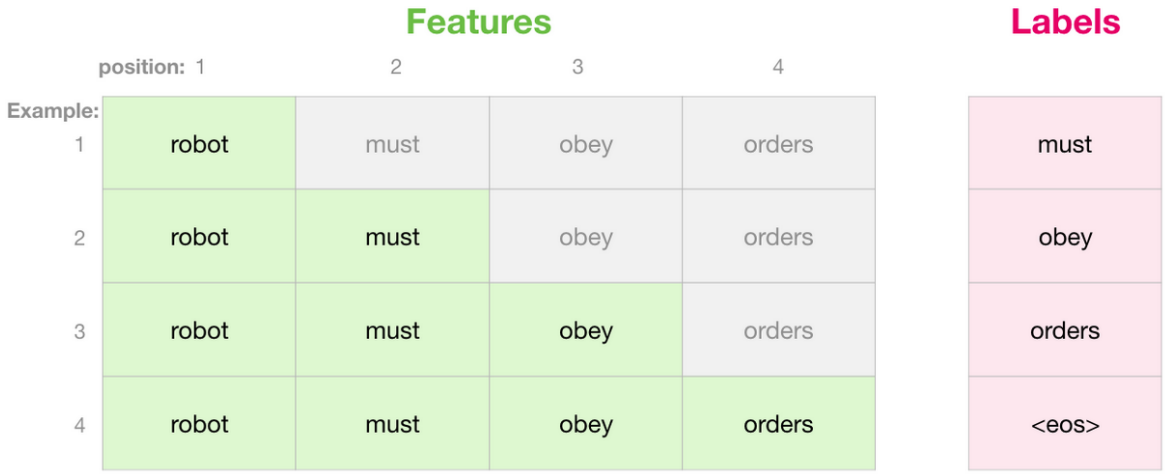

Рисунок 1.15 — Процесс обучения GPT

На нём слева изображён вектор контекста, а справа — слово, которое нужно предсказать на каждом этапе.

1.13.2 Развитие идей GPT

Модель GPT продемонстрировала беспрецедентное качество генерации на момент своего выхода, а модели GPT-2 и GPT-3 показали, что улучшения результата можно добиться с использованием той же самой архитектуры за счёт увеличения размера контекста, числа блоков декодировщика и объёма обучающей выборки [9].

Так, GPT-2 имеет длину контекста от 768-и токенов в версии Small до 1600-а в версии Extra Large и от 10-и до 48-и слоёв декодировщика соответственно.

GPT-3 же имеет уже 2048 токенов контекста и от 96 до 128 блоков декодировщика в зависимости от версии.

1.13.3 Другие приложения GPT

Оказалось, что модели GPT настолько хорошо моделируют язык, что их можно использовать для решения задач помимо генерации текста.

Например, GPT-3 способна без дополнительного дообучения давать ответы на некоторые вопросы, переводить текст с одного языка на другой, а также делать краткую выжимку из длинного текста (задача суммаризации), если подать ей на вход текст в определённой форме.

А если произвести дообучение, то модели GPT могут показывать очень хорошие результаты в этих задачах. Ниже изображены схемы дообучения GPT с примерами формата входных данных для решения задач машинного перевода (рисунок 1.16) и суммаризации (рисунок 1.17). Видно, что достаточно ввести специальный токен, обозначающий функцию, которую требуется применить к тексту [9].

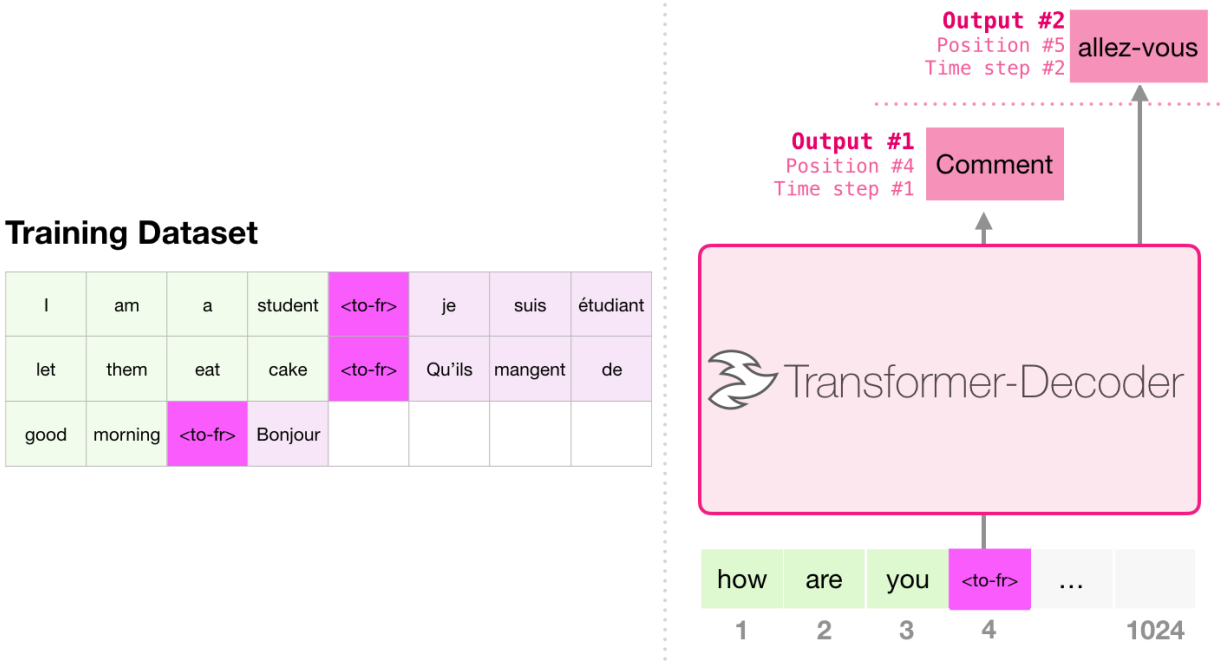

Рисунок 1.16 — использование GPT для машинного перевода с английского на французский

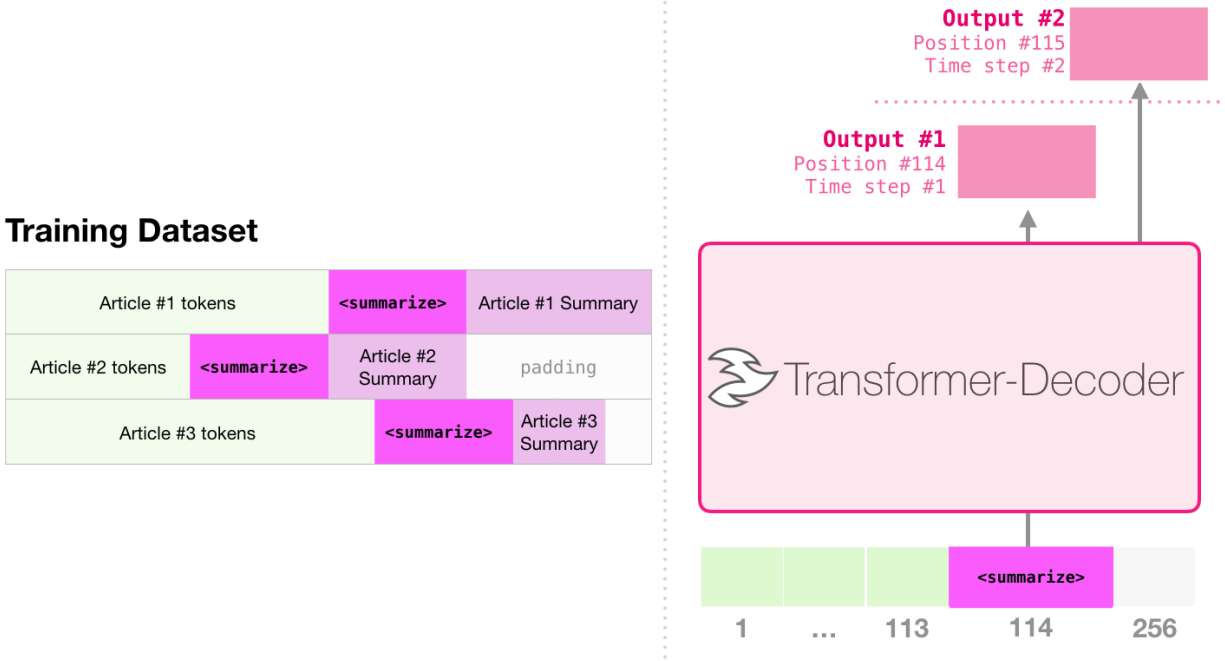

Рисунок 1.17 — использование GPT для суммаризации

2. Практическая часть

2.1 Процесс работы с системой

Процесс взаимодействия с программой выглядит следующим образом:

а) обученная на десятках гигабайт текста и хорошо моделирующая распределение слов в русском языке нейронная сеть дообучается на требуемом наборе данных, подстраиваясь под требуемую предметную область,

б) в дообученную языковую модель подаётся затравка — начало текста, которое модели необходимо продолжить,

в) пользователь оценивает результат генерации и может вручную отредактировать или отменить его,

г) процесс повторяется, начиная с пункта б, но в качестве затравки теперь выступает результат коррекции из пункта в.

Более наглядно процесс работы продемонстрирован на рисунке 2.1.

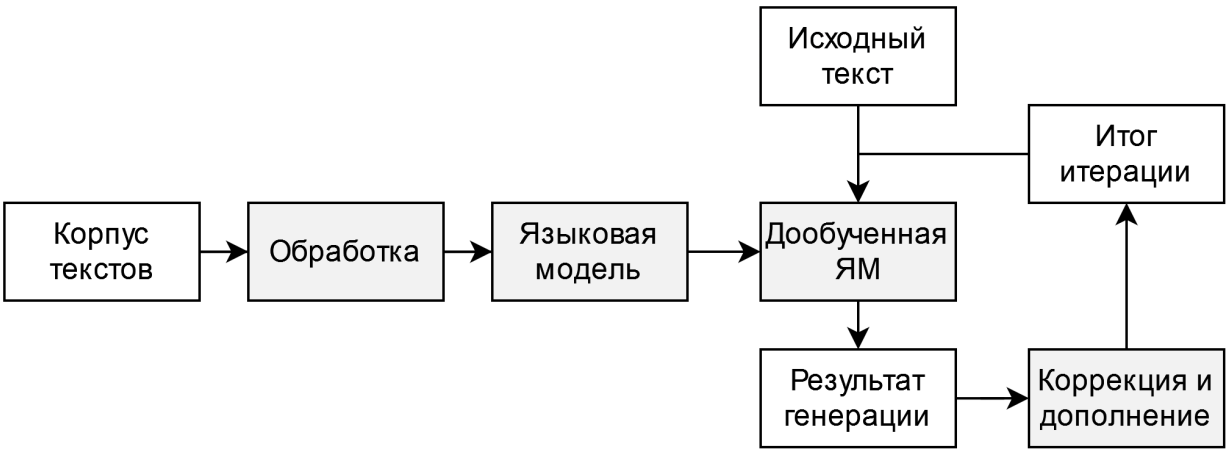

Рисунок 2.1 — Высокоуровневая схема системы

2.2 Выбор языковой модели

Среди доступных под свободными лицензиями языковых моделей рассматривались те, что приведены в таблице 2.1. Они построены на основе архитектуры Transformer, показывающей лучшие на данный момент результаты в задаче генерации текста [10], а длина контекста, который они учитывают,

достаточно большая, что важно для поддержания связности повествования в сгенерированном тексте. Названия моделей и их характеристики приведены в таблице 2.1.

Семейство модели	Название модели	Число параметров	Длина контекста
Russian GPT-3	ruGPT-3 Small	117 млн.	2048
	ruGPT-3 Large	760 млн.	2048
	ruGPT-3 XL	1,3 млрд.	2048
GPT-2	GPT-2 Small	124 млн.	1024
	GPT-2 Medium	355 млн.	1024
	GPT-2 XL	1,5 млрд.	1024

Таблица 2.1 — Некоторые нейросетевые языковые модели под свободными лицензиями

Из рассмотренных вариантов лучшие всего подошла модель ruGPT-3 Small от «Сбера» [11]. Её преимуществом является то, что она обучалась на русскоязычном корпусе и заточена под генерацию текста, в первую очередь, на русском языке, а малое относительно других моделей семейства ruGPT-3 число параметров позволяет дообучать её, располагая сравнительно небольшими мощностями.

2.3 Обработка текста

Перед обучением нейронной сети требуется привести данные к особому виду. В данном случае предварительная обработка корпуса заключается в выделении структурных блоков текста специальными синтаксическими конструкциями на естественном языке, формат которых определён заранее и сохраняется неизменным во всём корпусе. Выбор естественного языка обусловлен тем, что обученной на корпусе текстов на естественном языке языковой модели в этом случае не понадобится много данных для подстройки под новый синтаксис.

Код преобразования данных к нужному виду представлен в листинге А.2.

Пример обработанных данных показан в листинге 2.1.

Листинг 2.1 — Пример обработки входных данных

```
Место действия -- ПАВ. лобби/лифт.
Время действия -- день. день 1.
Действующие лица -- элеонора, Управляющий, массовка.
Ремарка -- Лифт открывается. Элеонора в лифте с букетом в руках, дочитывает
↪  записку. Смена плана. Перед лифтом стоит Управляющий.
Управляющий говорит: «Доброе утро, Элеонора Андреевна! Красивые цветы!»
Элеонора говорит: «Спасибо, я и сама заметила. Ты что-то хотел?»
Управляющий говорит: «Да. Лифт»
Ремарка -- Элеонора выходит из лифта. Управляющий, проводив её взглядом,
↪  входит. зк
Макс говорит: «И вот, спустя пару недель, она явно испытывает симпатию. Но
↪  пока к нему – незнакомцу, а не к тебе»
Ремарка -- Лифт закрывается.
```

2.4 Разбиение корпуса

При обучении данные разбиваются на части, способные поместиться в видеопамять, следовательно, важно производить разбиение определённым образом для лучшего результата. Все тексты обучающей выборки разделяются на части такого размера, чтобы:

а) в токенизированном виде их длина была не меньше длины контекста модели, чтобы при генерации учитывалось максимально возможное количество информации,

б) их длина была не слишком большой, чтобы при обучении иметь возможность подавать их в модель в случайном порядке для более эффективной оптимизации,

в) каждая часть была самостоятельным текстом, принадлежащим исходной предметной области.

Для подачи разбитых данных в модель был написан собственный класс **TextsDataset**, представленный в листинге 2.2. Он представляет собой коллекцию, которая при инициализации загружает с диска данные в виде длинных

текстовых файлов, разбивает их вышеописанным способом и предоставляет интерфейс для доступа к получившимся коротким фрагментам.

Листинг 2.2 — Класс датасета, хранящий данные в разбитом виде

```
class TextsDataset(Dataset):

    def __init__(self, tokenizer: PreTrainedTokenizer, path: str,
    ↪  block_size=2048):
        assert os.path.isdir(path)

        block_size = block_size - (tokenizer.max_len -
        ↪  tokenizer.max_len_single_sentence)

        self.examples = []
        try:
            for file in os.listdir(path):
                file_path = os.path.join(path, file)
                with open(file_path, encoding="utf-8") as f:
                    text = f.read()

                tokenized_text =
                ↪  tokenizer.convert_tokens_to_ids(tokenizer.tokenize(text))

                for i in range(0, len(tokenized_text) - block_size + 1,
                ↪  block_size):  # Truncate in block of block_size
                    self.examples.append(
                        tokenizer.build_inputs_with_special_tokens(
                            tokenized_text[i: i + block_size]
                        )
                    )
        except Exception as e:
            logger.exception(e)

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

    def __getitem__(self, item):
        return torch.tensor(self.examples[item], dtype=torch.long)
```

2.5 Процесс обучения

Исходная модель ruGPT-3 Small была загружена из библиотеки Transformers для языка Python. Обучение производилось с помощью ориги-

нального программного кода от «Сбера» [12], в котором был изменён механизм подачи данных в модель так, чтобы это происходило с использованием собственного класса **TextsDataset** из листинга 2.2.

Обучение происходило с помошью метода оптимизации Adam с параметрами $\beta_1 = 0{,}99$, $\beta_2 = 0{,}999$, $\varepsilon = 10^{-8}$, $\alpha = 5 \cdot 10^{-5}$ в течение 100-а эпох. В качестве набора данных были взяты сценарии юмористических телешоу. Итоговое значение перплексии составило 10,7.

2.6 Пользовательский интерфейс

Для создания графического интерфейса пользователя был использован фреймворк Streamlit. Он позволяет средствами языка Python создавать веб-приложения, которые открываются прямо в браузере на любой операционной системе [13].

Пользователь взаимодействует с программой через следующие элементы управления:

— поле ввода текста,

— кнопка «Дополнить»,

— кнопка «Отменить»,

— кнопка «Скачать результат».

Поле ввода текста позволяет вводить затравку для генерации, в нём же появляется сгенерированное дополнение, которое сразу можно отредактировать. Кнопка «Дополнить» запускает генерацию продолжения текста, находящегося в поле ввода; кнопка «Отменить» отменяет результат одной генерации, пользователь может отменять их сколько угодно вплоть до самого начала; кнопка «Скачать результат» нужна, чтобы загрузить на компьютер текстовый файл, содержащий весь текст, находящийся в поле ввода.

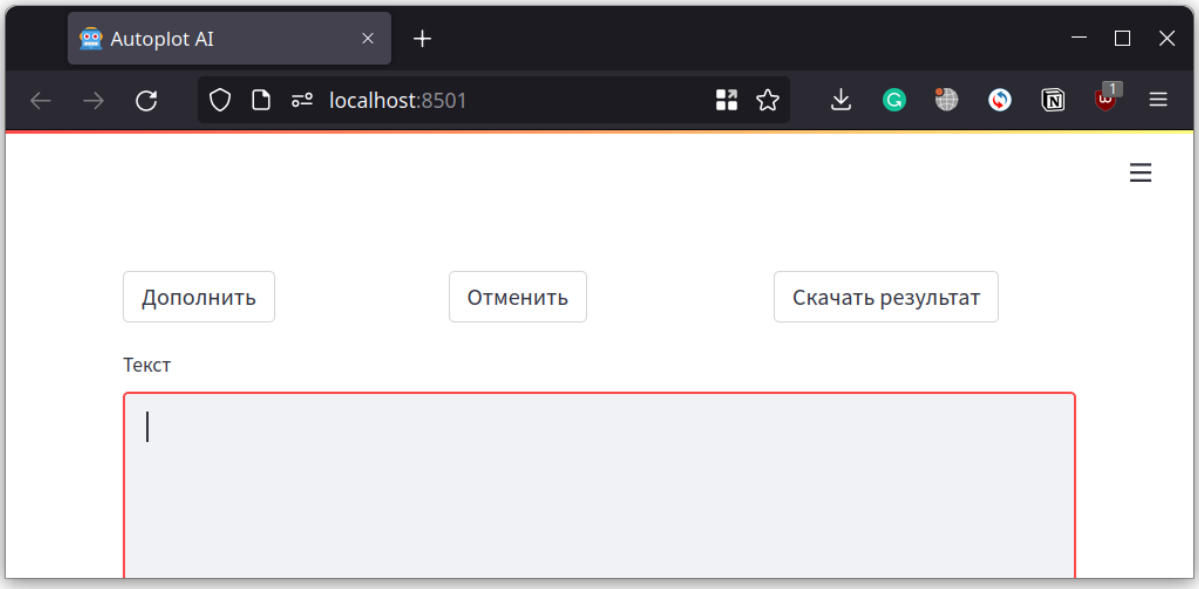

Рисунок 2.2 — Графический пользовательский интерфейс

2.7 Процесс работы с приложением

Рассмотрим сценарий работы с приложением:

а) пользователь вводит затравку (рисунок 2.3),

б) модель её продолжает (рисунок 2.4),

в) пользователь вводит дополнительную информацию, чтобы напрвить ход повествования (рисунок 2.5),

г) модель продолжает текст (рисунок 2.6),

д) пользователь остаётся неудовлетворён результатом, исправляет его и добавляет новые сведения (рисунок 2.7).

И в результате ещё нескольких итераций подобного процесса получается текст, показанный на рисунке 2.8.

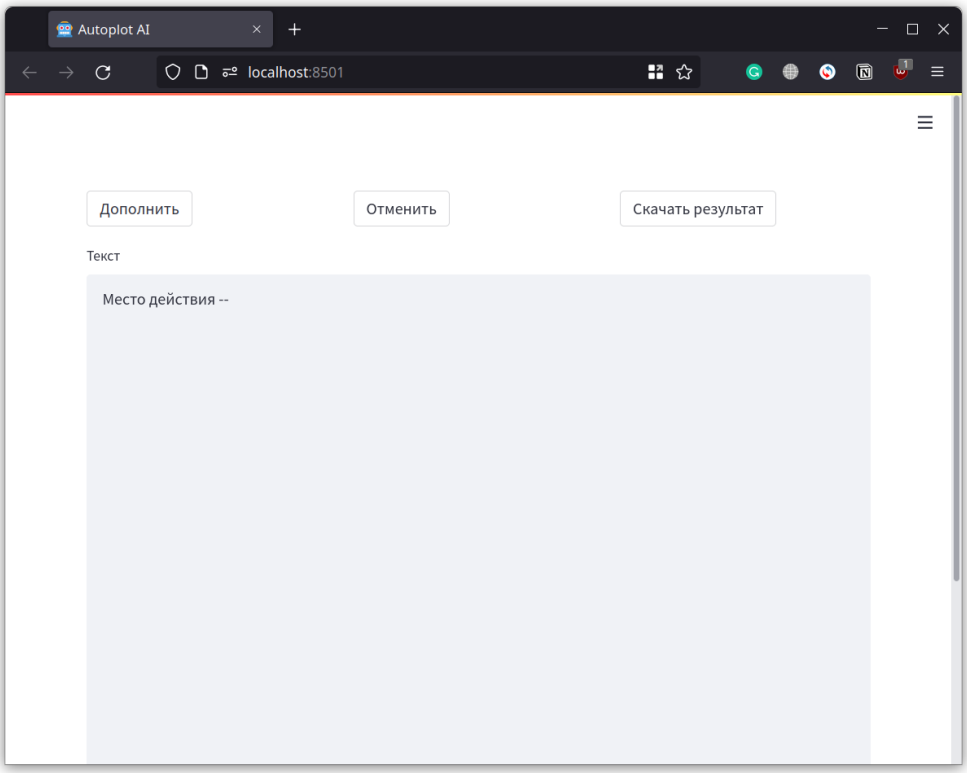

Рисунок 2.3 — Ввод затравки

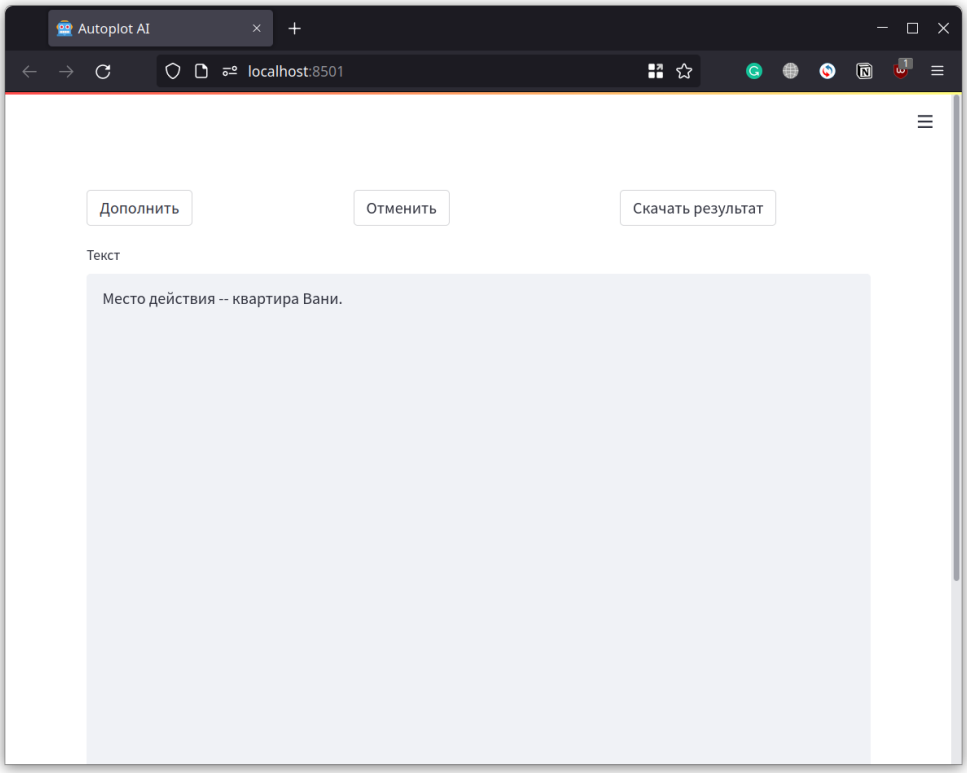

Рисунок 2.4 — Дополнение затравки

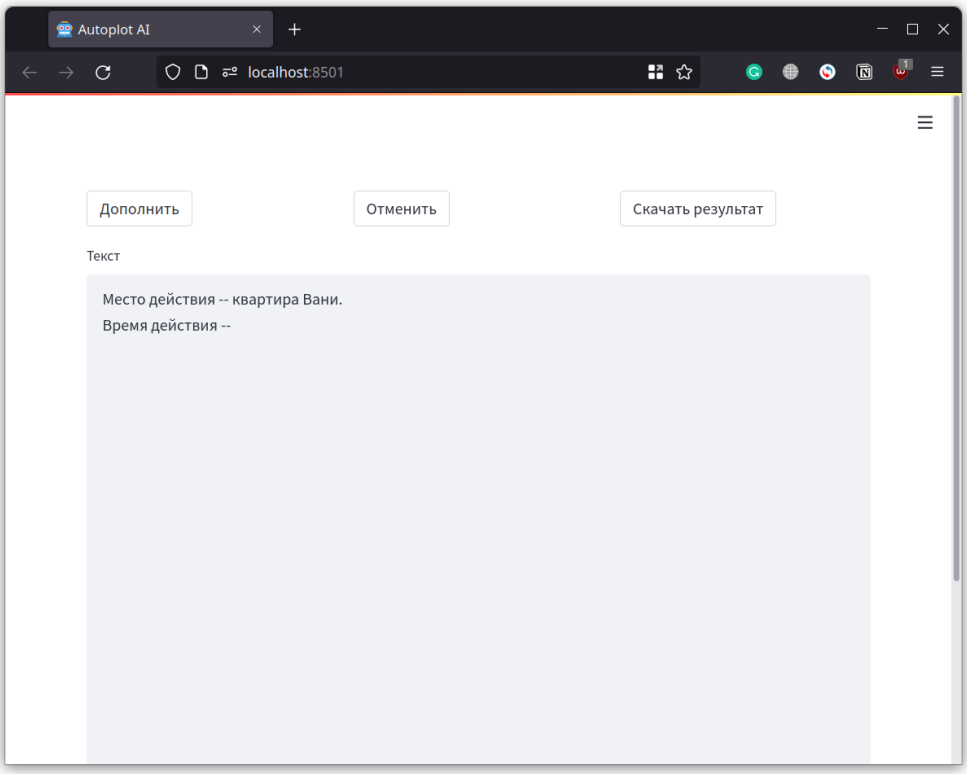

Рисунок 2.5 — Ввод дополнительной информации

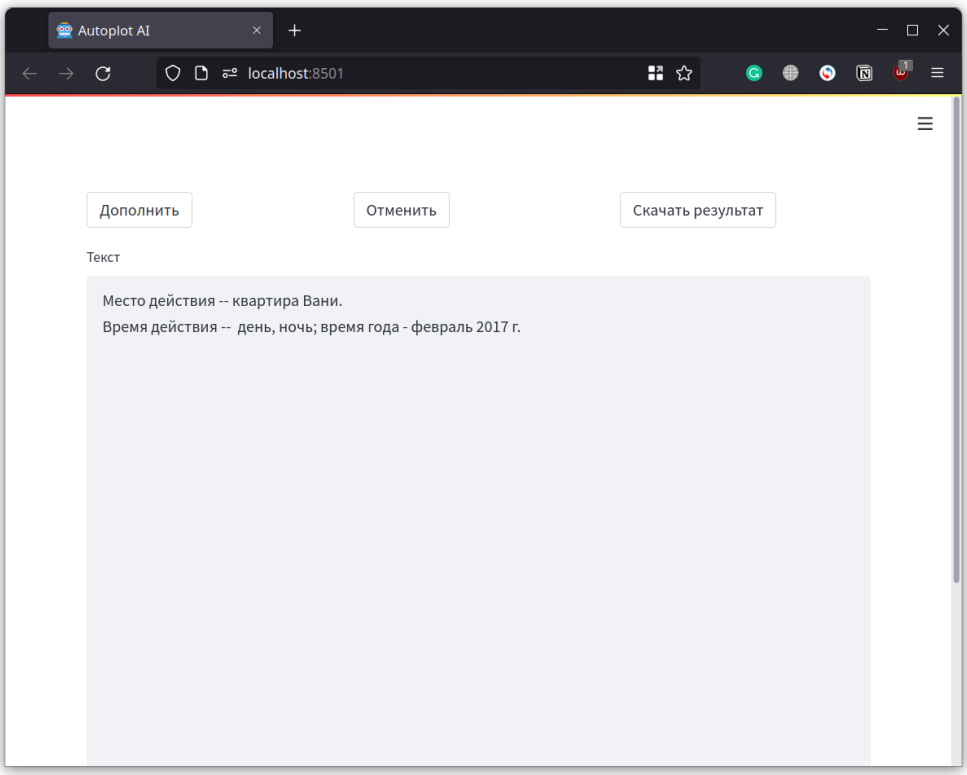

Рисунок 2.6 — Генерация продолжения

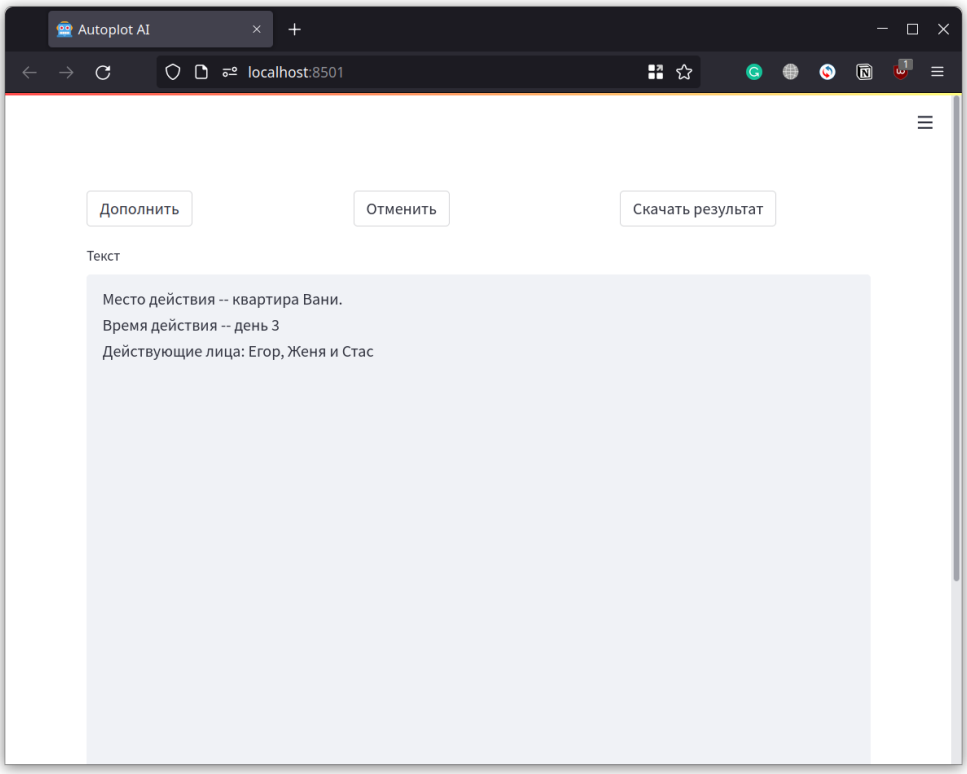

Рисунок 2.7 — Исправление и ввод новой информации

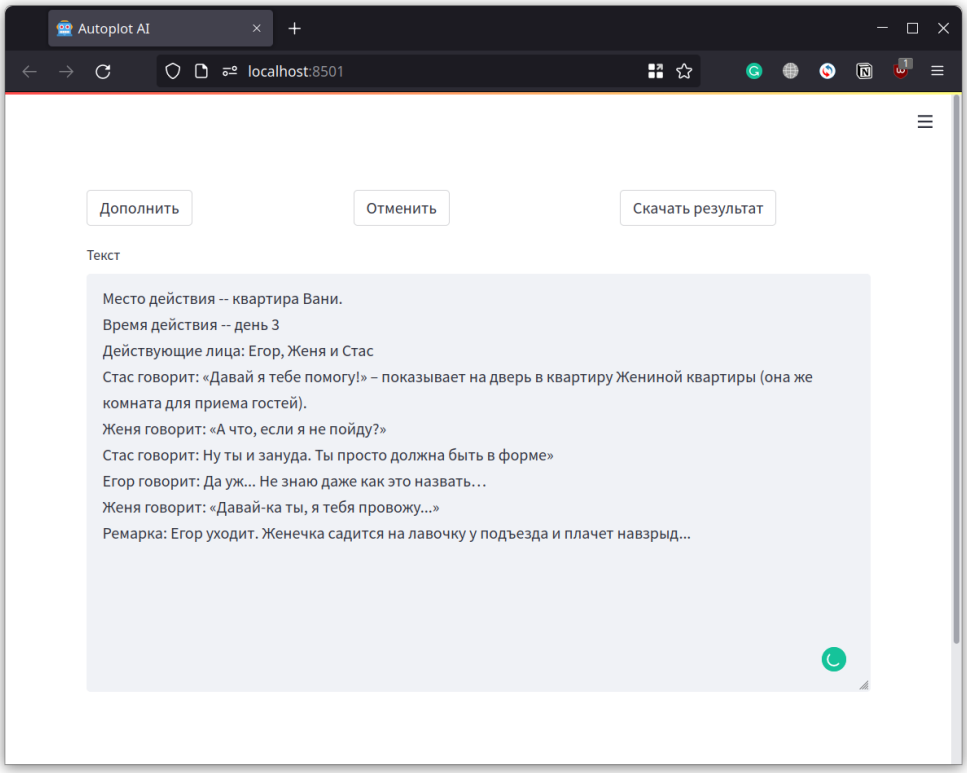

Рисунок 2.8 — Результат генерации

2.8 Дистрибутив

Для обеспечения переносимости и удобства распространения приложения все его файлы требуется поместить в один пакет и прописать инструкции по настройке и запуску программы.

Для выполнения этих требований была выбрана технология Docker. С её помощью всё, что необходимо программе для работы, а именно: исходный код, файлы модели, интерпретатор Python и библиотеки, помещаются в изолированное окружение, называемое контейнером и представляющее собой виртуальную машину с облегчённой операционной системой Debian [14].

В таком виде для передачи программы пользователю достаточно предоставить один файл — Docker-образ, который он сможет запустить с помощью всего одной команды `docker-compose up --build`.

Листинг 2.3 — Compose-файл

```
version: "3.0"

services:
  ctc_autoplotter:
    image: ctc
    restart: always
    build: .
    ports:
      - 8501:8501
```

Dockerfile (листинг 2.4) отвечает за сборку образа. Он настраивает Python-окружение, скачивает зависимости и файл модели. Compose-файл (листинг 2.3) нужен для упрощения процедуры запуска. Так как это веб-приложение, нужно знать адрес и порт, чтобы его открыть, и compose-файл производит связывание внутреннего порта Docker-контейнера с внешним на компьютере пользователя. Таким образом, страница с веб-приложением всегда располагается по адресу http://localhost:8501.

Листинг 2.4 — Главный скрипт **Dockerfile**

```
FROM python:3.7.12-slim

ENV aptDeps="wget unzip" \
    pipDeps="poetry gdown" \
    driveID="1L03nroUBlX7Q8K286eKapo8kk0Nmuiia"

WORKDIR /ctc

RUN apt-get update && \
    apt-get install -y --no-install-recommends ${aptDeps} && \
    ln -snf /usr/share/zoneinfo/Europe/Moscow /etc/localtime && echo
    ↪  Europe/Moscow > /etc/timezone && \
    pip install ${pipDeps} && \
    gdown --id ${driveID} -O model_cache.zip && \
    unzip model_cache.zip -d model_cache && \
    rm model_cache.zip

COPY pyproject.toml poetry.lock /ctc/

RUN poetry config virtualenvs.create false && \
    poetry install --no-dev --no-interaction --no-ansi && \
    pip install torch==1.4.0+cpu -f
    ↪  https://download.pytorch.org/whl/cpu/torch_stable.html && \
    python -m pip uninstall -y ${pipDeps} && \
    rm -rf /var/lib/apt/lists/* /var/cache/apt/archives /tmp/* /var/tmp/*
    ↪  /root/.cache/pip/*

RUN apt-get remove -y ${aptDeps} && \
    apt-get autoremove -y && \
    apt-get clean

COPY .streamlit /ctc/.streamlit
COPY demo /ctc/demo

EXPOSE 8501

CMD streamlit run /ctc/demo/demo.py --server.port 8501
```

ЗАКЛЮЧЕНИЕ

В результате выполнения описанной работы были решены следующие подзадачи:

— найден и предобработан корпус — набор сценариев юмористических телешоу,

— выбрана и дообучена предобученная нейросетевая языковая модель ruGPT-3 Small архитектуры Transformer с использованием библиотек для глубокого обучения PyTorch и Transformers для языка Python,

— разработан графический веб-интерфейс с использованием фреймворка Streamlit на Python, предоставляющий следующие возможности:

- дополнение введённого пользователем текста сгенерированным нейронной сетью,
- отмена результатов генерации,
- свободное редактирование получившегося документа,
- сохранение результата на компьютере,

— приложение для упрощения дистрибуции упаковано в изолированное окружение — контейнер, созданный при помощи технологии Docker.

По итогу проделанной работы можно заключить, что все изначально поставленные задачи были успешно выполнены:

— сгенерированные моделью тексты по форме действительно являются сценариями юмористических телешоу: они имеют аналогичную структуру, а при прочтении человеком могут вызвать у него смех,

— графический интерфейс получился удобным для взаимодействия с моделью, предоставляет достаточно гибкие возможности для экспериментов,

— технология Docker как средство упаковки приложения оказала положительное влияние на простоту развёртывания приложения на локальном компьютере, позволив прописать чёткие и универсальные инструкции для пользователя.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Диагностика эмпатии по А. Меграбяну и Н. Эпштейну. — Режим доступа: https://hrliga.com/index.php?module=profession&op=view&id=847 (дата обращения: 23.05.2022).

2. Открытый курс машинного обучения. Тема 4. Линейные модели классификации и регрессии. — Режим доступа: https://habr.com/ru/company/ods/blog/323890/ (дата обращения: 26.05.2022).

3. Ruder Sebastian. An overview of gradient descent optimization algorithms. — Режим доступа: https://arxiv.org/abs/1609.04747 (дата обращения: 25.05.2022).

4. Rico Sennrich Barry Haddow Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units. — Режим доступа: https://aclanthology.org/P16-1162.pdf (дата обращения: 25.05.2022).

5. Kapronczay Mor. A beginner's guide to language models. — Режим доступа: https://towardsdatascience.com/the-beginners-guide-to-language-models-aa47165b57f9 (дата обращения: 25.05.2022).

6. Mebsout Ismail. Recurrent Neural Networks. — Режим доступа: https://towardsdatascience.com/recurrent-neural-networks-b7719b362c65 (дата обращения: 27.05.2022).

7. Vaswani Ashish, Shazeer Noam, Parmar Niki и др. Attention Is All You Need. — Режим доступа: https://arxiv.org/abs/1706.03762 (дата обращения: 29.05.2022).

8. Devlin Jacob, Chang Ming-Wei, Lee Kenton, Toutanova Kristina. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. — Режим доступа: https://arxiv.org/abs/1810.04805 (дата обращения: 29.05.2022).

9. Brown Tom B., Mann Benjamin, Ryder Nick и др. Language Models are Few-Shot Learners. — Режим доступа: https://arxiv.org/abs/2005.14165 (дата обращения: 29.05.2022).

10. Yao Mariya. 10 Leading Language Models For NLP In 2021. — Режим доступа: https://www.topbots.com/leading-nlp-language-models-2020/ (дата обращения: 17.04.2022).

11. RuGPT-3 – AI-модель для написания текстов для разработчиков, обработка естественного языка. — Режим доступа: https://developers.sber.ru/portal/products/rugpt-3ysclid=l249jkw241&attempt=1 (дата обращения: 18.04.2022).

12. ru-gpts. — Режим доступа: https://github.com/ai-forever/ru-gpts (дата обращения: 17.04.2022).

13. Streamlit documentation. — Режим доступа: https://docs.streamlit.io/ (дата обращения: 18.04.2022).

14. Docker overview. — Режим доступа: https://docs.docker.com/get-started/overview/ (дата обращения: 25.05.2022).

ПРИЛОЖЕНИЕ А

ЛИСТИНГИ ИСХОДНОГО КОДА

Листинг А.1 — Графический интерфейс пользователя

```
import streamlit as st

from generate import load_tokenizer_and_model, generate, CACHE_DIR


def initialize() -> None:
    """Initialize session state and set page config"""

    st.set_page_config(
        page_title="Autoplot AI",
        page_icon="▫",
        layout="wide",
        initial_sidebar_state="collapsed"
    )

    if "model" not in st.session_state or "tokenizer" not in st.session_state:
        with st.spinner("Loading model"):
            tokenizer, model = load_tokenizer_and_model(CACHE_DIR)
            st.session_state["tokenizer"] = tokenizer
            st.session_state["model"] = model

    if "text_versions" not in st.session_state:
        st.session_state["text_versions"] = [""]


def main() -> None:
    """User interface logic"""

    text_versions = st.session_state["text_versions"]
    tokenizer = st.session_state["tokenizer"]
    model = st.session_state["model"]

    button_cols = st.columns(3)
    with button_cols[0]:
        continue_btn = st.button("Дополнить")

    with button_cols[1]:
        undo_btn = st.button("Отменить")

    with button_cols[2]:
        st.download_button("Скачать результат", text_versions[-1], "result.txt")
```

```
42
43      text_container = st.empty()
44      text_area_attrs = {"label": "Текст", "height": 500}
45
46      with text_container:
47          working_text = st.text_area(value=text_versions[-1], **text_area_attrs)
48
49      if continue_btn:
50          if len(working_text) == 0:
51              working_text = "Место действия -- "
52
53          working_text = working_text[:-100] + generate(model, tokenizer, working_text[-100:])[0]
54
55          with text_container:
56              st.text_area(value=working_text, **text_area_attrs)
57
58      if text_versions[-1] != working_text:
59          text_versions.append(working_text)
60          st.experimental_rerun()
61
62      if undo_btn and len(text_versions) > 1:
63          text_versions.pop()
64          working_text = text_versions[-1]
65          with text_container:
66              st.text_area(value=working_text, **text_area_attrs)
67
68
69  if __name__ == "__main__":
70      initialize()
71      main()
```

Листинг А.2 — Модуль генерации текста

```
1   import time
2   import os
3   import sys
4   import random
5
6   from zipfile import ZipFile
7
8   import numpy as np
9   import torch
10
11  from transformers import GPT2LMHeadModel, GPT2Tokenizer
12
13
14  USE_CUDA = True
```

```
CACHE_DIR = os.path.join(os.curdir, "model_cache")
SEED = random.randint(0, 1000)

if not os.path.isdir(CACHE_DIR):
    print("Extracting model...")
    with ZipFile("model.zip") as f:
        f.extractall(CACHE_DIR)

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

print(f"Running on {device}")


def load_tokenizer_and_model(model_name_or_path):
    print("Loading tokenizer and model from " + CACHE_DIR)
    tokenizer = GPT2Tokenizer.from_pretrained(model_name_or_path)
    model = GPT2LMHeadModel.from_pretrained(model_name_or_path).to(device)
    return tokenizer, model


def generate(
    model, tok, text,
    do_sample=True, max_length=50, repetition_penalty=5.0,
    top_k=5, top_p=0.95, temperature=1,
    num_beams=None,
    no_repeat_ngram_size=3
    ):
    input_ids = tok.encode(text, return_tensors="pt").to(device)
    out = model.generate(
        input_ids.to(device),
        max_length=max_length,
        repetition_penalty=repetition_penalty,
        do_sample=do_sample,
        top_k=top_k, top_p=top_p, temperature=temperature,
        num_beams=num_beams, no_repeat_ngram_size=no_repeat_ngram_size
    )
    return list(map(tok.decode, out))


def main(beginning):
    np.random.seed(SEED)
    torch.manual_seed(SEED)

    tok, model = load_tokenizer_and_model(CACHE_DIR)

    print("Generating")
```

```
    prev_timestamp = time.time()
    generated = generate(model, tok, beginning, max_length=200, top_p=0.95, temperature=0.7)
    time_spent = time.time() - prev_timestamp

    print(generated[0])

    print(f"Elapsed time: {time_spent} s.")


if __name__ == "__main__":
    main(sys.argv[1])
```

Листинг А.3 — Скрипт для валидации и обработки данных

```
import os
import re
import shutil

from collections import namedtuple
from sys import argv
from typing import List, Union


DATA_PATH = argv[1]
if DATA_PATH[-1] != "/":
    DATA_PATH += "/"

OUT_PATH = "humanized"

Block = namedtuple("Block", ["tag", "content"])


def parse(text: str) -> List[Union[Block, str]]:

    text = re.sub("<<", "«", text)
    text = re.sub(">>", "»", text)
    text = re.sub(r"\s+|\n", " ", text)
    s = re.sub(r"(</?\w+>)", r"[CUT]\1[CUT]", text)
    cut = list(filter(lambda t: len(t) > 0, map(str.strip, s.split("[CUT]"))))

    open_tag_pat = re.compile(r"<\w+>")

    cur_errors = []

    def parse_list(l: List[str]) -> List[Union[Block, str]]:
        it = iter(l)
        out = []
```

```
        while True:
            try:
                el = next(it)
            except StopIteration:
                break

            if re.match(open_tag_pat, el):
                tag = el[1:-1]
                next_it = []
                while not re.match(f"<\W{tag}>", el):
                    try:
                        el = next(it)
                    except StopIteration:
                        # raise SyntaxError(f"<{tag}> was not closed: {out[-1]}; {'
                            '.join(next_it)}")
                        cur_errors.append(f"<{tag}> was not closed: {out[-1] if len(out) > 0
                            else ''}; {' '.join(next_it)}")
                        break
                    else:
                        next_it.append(el)
                if len(next_it) > 0:
                    out.append(Block(tag, parse_list(next_it[:-1])))
            else:
                out.append(el)

        for e in out:
            if isinstance(e, str) and re.match(r"</?.+>", e):
                i = out.index(e)
                cur_errors.append(f"Found tag in processed data: {e}; {out[max(i - 2, 0):i +
                    1]}")
        return out

    return parse_list(cut), cur_errors


cur_name = ""
def humanize(s: List[Union[Block, str]]) -> str:
    sentences = []

    global cur_name

    for el in s:
        if isinstance(el, str):
            sentences.append(el.strip())
        else:
            if el.tag == "header":
```

```
77                  continue
78              elif el.tag == ”footer”:
79                  continue
80              elif el.tag == ”remark”:
81                  sentences += [”\n” + ”Ремарка --”, humanize(el.content)]
82              elif el.tag == ”author”:
83                  sentences += [”\n” + ”Слова автора --”, humanize(el.content)]
84              elif el.tag == ”title”:
85                  sentences += [”\n\n” + ”Заголовок --”, humanize(el.content), ”\n”]
86              elif el.tag == ”place”:
87                  sentences += [”\n” + ”Место действия --”, humanize(el.content).strip(”.”) +
                        ”.”]
88              elif el.tag == ”time”:
89                  sentences += [”\n” + ”Время действия --”,
                        humanize(el.content).strip(”.”).lower() + ”.”]
90              elif el.tag == ”chars”:
91                  sentences += [”\n” + ”Действующие лица --”, humanize(el.content).strip(”.”) +
                        ”.”]
92              elif el.tag == ”name”:
93                  cur_name = humanize(el.content).strip().capitalize()
94              elif el.tag == ”line”:
95                  sentences += [”\n” + cur_name, ”говорит:”, ”«” +
                        humanize(el.content).strip(”.”) + ”»”]
96              elif el.tag == ”how”:
97                  sentences.append(humanize(el.content).lower())
98
99      sentences = filter(lambda t: not re.match(r”^\W*$”, t) or t == ”\n”, sentences)
100     sentences = ” ”.join(sentences)
101     if sentences[0] == ”\n”:
102         sentences = sentences[1:]
103     return sentences
104
105
106 if __name__ == ”__main__”:
107     paths = []
108     for root, _, files in os.walk(DATA_PATH):
109         for file in files:
110             paths.append(os.path.join(root, file))
111
112     parsed = []
113     errors = []
114     for path in paths:
115         with open(path) as file:
116             try:
117                 content = file.read()
118             except Exception as e:
```

```
                print(e, path)
                raise

        try:
            t, e = parse(content)
            if len(e) > 0:
                raise SyntaxError("\n\n\n".join(e))
            parsed.append((path, t))
        except SyntaxError as e:
            errors.append((path, e))
            continue

    print(f"Errors occured in {len(errors)} files")
    print(f"Successfully parsed {len(parsed)} files")

    if os.path.isdir(os.path.join("errors", "data")):
        for f in os.listdir(os.path.join("errors", "data")):
            os.remove(os.path.join(os.path.join("errors", "data"), f))
    for p, e in errors:
        os.makedirs(os.path.join("errors", "data"), exist_ok=True)
        path = os.path.join("errors", "data", os.path.split(p)[-1])
        with open(path + ".log", "w") as f:
            f.write(str(e))

        os.system(f"cp '{p}' '{path}'")
        # break

    if os.path.isdir(OUT_PATH):
        for f in os.listdir(OUT_PATH):
            os.remove(os.path.join(OUT_PATH, f))
    os.makedirs(OUT_PATH, exist_ok=True)

    for i, (path, script) in enumerate(parsed):
        text = humanize(script)

        new_path = os.path.join(OUT_PATH, re.sub(DATA_PATH, "", path))
        os.makedirs(os.path.split(new_path)[0], exist_ok=True)
        with open(new_path, "w") as f:
            f.write(text)

    if os.path.isdir("invalid_files"):
        for f in os.listdir("invalid_files"):
            os.remove(os.path.join("invalid_files", f))
    for path, _ in errors:
        os.makedirs("invalid_files", exist_ok=True)
        shutil.copy(path, "invalid_files")
```